MORTEN TENGESDAL

Frå transistor til datamaskin

NOTAT NR. 39, UNIVERSITETET I STAVANGER – APRIL 2018

Frå transistor til datamaskin

Morten Tengesdal,
Institutt for data- og elektroteknologi,
Universitetet i Stavanger

Dato: 4. april 2018.

Universitetet i Stavanger
N-4036 Stavanger
NOREG
www.uis.no

ISSN 1504-4939
ISBN 978-82-7644-767-5
Notater fra Universitetet i Stavanger, nr. 39

Forord

Dette skrivet prøver å gi ei forståing av korleis datamaskinar er oppbygde og virkar. Ein ser også på korleis ein kan programmera mindre datamaskinar brukte til styring og overvaking av prosessar[1].
Skrivet er laga for bruk i emnet Datamaskinarkitektur ved Universitetet i Stavanger (UiS). Det kan også brukast til å gi ei innføring i digitalteknikk generelt og som ein del av dette, mikroprosessoren sin struktur og virkemåte. Ein får også ei innføring i grunnlaget for konstruksjon av datamaskinar.

Ein har prøvd å gjera framstillinga kort og klar. Hovudfokus er på dei ulike byggjesteinane i ein datamaskin og korleis digitalteknikk blir praktisert i dag. Mykje av den klassiske digitalteknikken som manuell logikkonstruksjon med enkle portar av ein type, manuell optimalisering av logikk vhja. Karnaugh-diagram og konstruksjon av tilstandsmaskinar basert på J-K-vipper er tillagt redusert vekt samanlikna med vanlege lærebøker. Utviklingsprosjekt i dag tar stort sett utgangspunkt i programmerbar elektronikk. Mange klassiske metodar er her innbakt i utviklingsverktøyet ein bruker, eller blir realisert på andre måtar. Tema som logikkoptimalisering er likevel tatt med til ein viss grad der dette høyrer heime.

Emnet Datamaskinarkitektur kom inn som ein obligatorisk del av bachelorstudia i data og elektro ved UiS hausten 2013. Læringsmåla for desse bachelorstudia kan formulerast som vist i det følgjande.
Studentane skal som ferdige ingeniørar i **data** vera i stand til å:
- Forstå ulike typar operativsystem.
- Spesifisera, utvikla og testa objektorienterte datasystem.
- Utnytta kjente algoritmer og datastrukturar til å løysa konkrete datatekniske problem.
- Utvikla og bruka databaser.
- Planleggja og driva datasystem.
- Vurdera og utvikla nettbaserte dataløysinger.

Studentane skal som ferdige ingeniører i **elektro** vera i stand til å:
- Analysera og konstruera mikroprosessorbaserte system og programvare for desse.
- Vurdera måletekniske løysingar for ei gitt oppgåve.
- Vurdera komponentar og metodar for overvaking og automatisering av prosessar og produksjonslinjer.
- Konstruera og programmera PLS- baserte styringssystem.
- Utvikla diskrete algoritmer for regulerings- og signalbehandlingsformål.

Ei solid forståing av korleis ein datamaskin er oppbygd og virkar er eit grunnleggjande vilkår for dei fleste av desse måla.

[1] Ein prosess kan f.eks. vera ein vaskemaskin, kvadrokopter, kollisjonspute eller ein robotgrasklyppar.

er det i skrivet brei bruk av referansar til utdjupande stoff på internett.

Ein del av stoffet på mikroprosessorar og datamaskinkonstruksjon er basert på skrivet [1], som blei laga for eit tidlegare datamaskinemne ved UiS.
I tillegg er vedlegg A.1 basert på forelesingsnotat i digitalteknikk frå 2009 av Trygve Eftestøl, UiS, [12]. Desse notata innheldt også handteikna skisser av Tom Ryen, UiS.
Elles har Jon Fidjeland, UiS, kome med mange verdifulle kommentarar undervegs i arbeidet.
Sidan oppstarten i 2013 har rundt 400 studentar tatt emnet Datamaskinarkitektur, og spørsmåla og kommentarane frå dei har påverka utviklinga av skrivet gjennom desse åra.

Alle tilbakemeldingar[2] blir mottekne med takk.

Når det gjeld referansar til Wikimedia, er dette gjort ved å oppgi strengen "wiki filnamn" og til Wikipedia[3] ved å oppgi "landkode.wiki søkjeord1 søkjeord2 ..".
Begge desse strengtypane kan då leggjast rett inn i ein passande søkjemotor.

Skrivet er norsk. Engelske omgrep blir sett i hermeteikn viss dei blir nytta åleine og i parentes når dei blir viste saman med norsk utgåve av nemninga.

[2]F.eks. til e-postadressa `morten.tengesdal@uis.no`.

[3]Det er her ofte referert til det omfattande dugnadsleksikonet Wikipedia, vel vitande om at det kan vera feil her og der, og kan hende i større grad enn i andre leksikon. Her kjem likevel nye ting raskare ut til lesaren, og med mange kompetente bidragsytarar som også rettar kvarandre. Ved i tillegg å lesa om eit tema i andre kjelder også, vil ein kunne danna seg eit godt og så korrekt som mogleg bilete av dette. Dette er bl.a. omtalt på følgjande side: "https://sites.google.com/site/digitaltkompetent/fagstoff/finne/wikipedia-som-kilde" (Lasta ned sist 6/2-18.)
Sida "https://sites.google.com/site/digitaltkompetent" låg under Høgskolen i Bergen og blir ikkje oppdatert lenger. Opphavsmannen Jon Hoem driv nå ei liknande side,
"https://sites.google.com/site/digitaleferdigheter" ved den nye Høgskolen på Vestlandet.

Innhald

Nokre vanlege forkortingar innan digital elektronikk

Alle engelske omgrep med unnatak av produktnamn er sett i hermeteikn.
Forkortingar med små bokstavar er forfattaren sine.

A/D, ADC	"Analog-to-Digital Converter", AD-omformar, dvs. omformar frå analog til digital verdi.
ASIC	"Application Spesific Integrated Circuit", kundespesifisert integrert krets.
ALS	"Advanced Low-Power Schottky TTL", familie av digitale kretsar, sjå også TTL.
ALU	"Arithmetic Logic Unit", utreknaren inne i mikroprosessoren.
ARM	"Advanced RISC Machine", mikroprosessor lisensiert av ARM Holdings.
BJT	"Bipolar Junction Transistor", bipolar transistor.
BPS	"Bits Per Second", bitar per sekund (bps), bitrate.
BRAM	"Block RAM", SRAM-blokk i FPGA. Minne i FPGA blir sett opp av mange BRAM-blokker.
BSB	Base System Builder, del av utviklingsverktøyet EDK.
BSP	"Board Support Package", plattformstøtte, dvs. systemfiler for ein plattform.
BWF	"ButterWorth Filter", Butterworth-filter (vanleg som anf).
CISC	"Complex Instruction Set Computer", tradisjonelle prosessorar, f.eks. 80xxx frå Intel.
CMOS	"Complementary MOSFET", familie av digitale kretsar.
CMSIS	Cortex Micro-controller Software Interface Standard, programmeringsstandard for ARM-baserte mikrokontrollerar.
CPLD	"Complex PLD", ligg mellom PLD og FPGA i kompleksitet.
C/T	"Counter/timer", teljar/taimer-modul.
D/A, DAC	"DA Converter", DA-omformar, dvs. omformar frå digital til analog verdi.
DRAM	Dynamisk RAM, flyktig ("volatile") minneteknologi.
EDK	Embedded Development Kit, Xilinx sitt utviklingsverktøyet for mjukC.
EMI	"Electro Magnetic Interference", elektromagnetisk støy/interferens.
FLASH	Ikkje-flyktig ("non-volatile"), dvs. permanent, minneteknologi.
FPGA	"Field Programmable Gate Array", portmatrise.
GAL	"Generic Array Logic", reprogrammerbar logisk krets.
GPIO	"General Purpose Input Output port", parallellport.
ibs	Innebygd system, "embedded system".
IC	"Integrated Circuit", Integrert krets.
INTC	"Interrupt Controller", kontrollermodul for avbrot.
IP-modul	"Intellectual Property", logikkmodul, perifermodul.
ISR	"Interrupt Service Rutine", avbrotsmetode, same som "Handler".
LCD	"Liquid Crystal Display", skjerm basert på flytande krystallar.
LED	"Light Emitting Diode", lysdiode.

LMB	”Local Memory Bus”, ein av bussane i MB-baserte mikrokontrollerar.
LSb,B,H,W	”Least Significant..”, minst signifikante bit, byte, halvord, ord.
LUT	”Look-Up Table”, oppslagstabell, måte å realisera logiske funksjonar på i FPGA, alternativ: SOP.
LVTTL	”Low Voltage TTL”, familie av digitale kretsar, sjå også TTL.
MB	MicroBlaze, 32-bits mjukprosessor frå Xilinx.
MChEMC	”Multi Channel External Memory Controller”, kontrollermodul for eksternt minne.
MHA	”Modified Harvard Architecture”, fellesnamn på ulike Harvard-arkitekturar.
mjukC	mjuk mikrokontroller, ”soft microcontroller”.
mjukP	mjukprosessor, ”soft microprocessor”.
MHA	Modifisert Harvard-arkitektur.
MOSFET	”Metal Oxide Semiconductor Field Effect Transistor”, felteffekttransistor.
Ms	Maskinsyklus, 1Ms = n klokkesyklar, der n = 1 for MicroBlaze, ”machine cycle”.
MSb,B,H,W	”Most Significant..”, mest signifikante bit, byte, halvord, ord.
MUX	Multipleksar, dvs. kanalveljar. Finst både for digitale og analoge signal.
NML,H	”Noise Margin..”, støymargin for lågt nivå, høgt nivå.
NVIC	Nested Vectorized Interrupt Controller, kontrollermodul for avbrot.
OPB	”On-chip Peripheral Bus”, ein av bussane i MB-baserte mikrokontrollerar. F.o.m. EDK-versjon 10.1 er denne erstatta av PLB, sjå denne.
OTP	”One-Time Programmable”, 1-gongs-programmerbar.
PLA	”Programmable Logic Array”, vanlegvis 1-gongsprogrammerbar logisk krets.
PAL	”Programmable Array Logic”, som PLA.
PC	”Program Counter” eller ”Personal Computer”, programteljar eller personleg datamaskin.
PCB	”Printed Circuit Board”, kretskort.
PD	”Photo Diode”, fotodiode.
PLB	”Processor Local Bus”, ein av bussane i MB-baserte mikrokontrollerar.
PLD	”Programmable Logic Device”, samlenamn for programmerbare kretsar.
PMOSFET	”Power MOSFET”, ”KraftMOS” / krafttransistor (av felteffekt-typen).
PWM	”Pulse Width Modulation”, pulsbreiddemodulering.
RAM	”Random Access Memory”, eks. Statisk RAM, Dynamisk RAM.
RISC	”Reduced Instruction Set Computer”, nyare og effektive μP-ar, f.eks. MicroBlaze, AVR- og ARM-familien.
ROM	”Read Only Memory”, eks. Electrical Erasable Programmable ROM, Flash.
RS	”Recommended Standard”, utforma av Electronic Industries Association.
RTOS	”Real-Time Operating System”, sanntids operativsystem.
SAR	”Successive Approximation Register”, vanleg AD-omformingsmetode.
SCF	”Switched Capacitance Filter”, filterkrets basert på svitsja kapasitansnettverk.
SDK	”Software Development Kit”, del av utviklingsverktøyet EDK.
SFR	”Special Function Register”, registra inne i ein perifermodul.
S/H	”Sample/Hold”, analog haldekrets, også kalla T/H.
SoC	”System on a Chip”, større system enn ein vanleg μC på ei brikke.

SOP	”Sum Of Products”, samlenamn for eit ”OG - ELLER”-nettverk som ein finn i PLD-kretsar.
SPI	”Serial Peripheral Interface”, synkron seriekommunikasjonsmetode.
SPLD	”Simple PLD”, vanleg PLD, dvs. i form av PAL elller GAL.
SR	”Slew Rate”, stigningsrate.
SRAM	Statisk RAM, flyktig (”volatile”) minneteknologi.
SSG	Matlab Simulink med Xilinx System Generator. Verktøy for realisering av digital logikk i FPGA.
ST	STMicroelectronics, produsent av bl.a. ARM-baserte μC-ar.
stm	Stegmotor, ”step motor”.
taimer	Elektronikkrets som gir et varsel når et visst tidsintervall er utløpt, ”timer”. Det å taima er nå lovleg å skriva på norsk. Det fell derfor naturleg å bruka nylaginga taimer her.
tbk	Tilbakekobling, ”feedback”.
T/H	”Track/Hold”, analog haldekrets.
TTL	”Transistor Transistor Logic”, familie av digitale kretsar.
UART	”Universal Asyncronous Receiver and Transmitter”, asynkron seriekrets.
USART	”Universal Syncronous and Asyncronous Receiver and Transmitter”, seriekrets som kan begge typane overføring.
UiS1	Øvingsmaskin basert på hovudkortet Spartan 3 Starter Board frå Xilinx og det UiS-konstruerte grensesnittkortet UiS_AD.
UiS2	Øvingsmaskin basert på hovudkortet STM32VLDiscovery frå STMicroelectronics og FPGA-kortet ZedBoard frå Digilent.
VHDL	”VHSIC Hardware Description Language”, språk for å spesifisera logikken i programmerbare kretsar.
VHSIC	”Very High Speed Integrated Circuits”, vanlegvis nyare og store integrerte kretsar, dvs. VLSIC.
VLSIC	”Very Large Scale Integrated Circuits”, komplekse integrerte kretsar. Desse innheld meir enn 10000 transistorar.
VNA	Von Neumann-arkitektur.
vt	Vektortabell, ”interrupt vector table”.
XMD	”Xilinx Microprocessor Debugger”, avlusingsverktøy i EDK.
XPS	Xilinx Platform Studio, del av utviklingsverktøyet EDK.
μP, μC	Mikroprosessor, mikrokontroller.

Kapittel 1

Innleiing

Dette kapitlet vil handla om følgjande:

- Litt frå historia til datamaskinen.
- Hovudgrupper og utviklingstrekk.
- Litt om emnet Datamaskinarkitektur.

1.1 Kvifor datamaskinar?

1.1.1 Litt historie

Historia om livet på jorda er ei historie om utvikling. Det er eit innebygt driv i naturen mot større mangfald. Mykje endar opp i blindspor, men jamnt over har dette drivet gitt stadig meir avanserte livsformer.
Mennesket har også dette drivet innebygt i genene. Heile tida er ein på jakt etter meir **effektive** og innbringande måtar å gjera ting på. Dette, som eigentleg er ein overlevingsmekanisme, får oss til å prøva å innretta oss slik at me kan ha det betre enn før og betre enn våre forgjengarar.

Eit stikkord i menneskja si utvikling er aukande grad av **automatisering**. Ved å automatisera arbeidsoppgåver som er tidkrevjande, einsformige eller farlege, eller kombinasjonar av dette, kan ein bruka tida på kjekkare oppgåver, og auka produksjonen av tenester og materielle gode[1]. Kort sagt aukar ideelt sett levestandarden som følgje av automatisering og effektivisering, men som me ser rundt oss kvar dag, så får langt frå alle del i dette. Her

[1] Det er sjølvsagt minussider også med dette som aukande materialisme og ressursbruk.

ligg nok ei av dei største utfordringa i dag i tillegg til at det ennå er nok av oppgåver som kan automatiserast.

Dei første anlegga med ein viss grad av automatisering som ein kjenner til, var vatningsanlegg i Egypt.
Dei første reine automatane som er dokumenterte, blei funne opp av grekarar for over to tusen år sidan. Ein av dei største oppfinnarane var Heron[2] som mellom anna demonstrerte bruk av vindkraft[3] og damp.
Vindmøller blei som namnet seier, ofte brukte til maling av korn. I 1745 blei det patentert eit haleror[4] som gjorde at mølla automatisk stilte seg opp mot vinden.

I kjølvatnet av opplysningstida kom for eksempel dampmaskinar med sentrifugalregulator for å halda kjeletrykket konstant. Desse maskinane var bokstaveleg talt sjølve drivkrafta bak den industrielle revolusjonen.

Dei første **datamaskinane** var **mekaniske kalkulatorar** av ulike slag. Dette var f.eks. klokker[5] som kunne vera drivne av vatn, eller manuelt drivne såkalla astrolabar[6] som viste astronomiske bevegelsar.

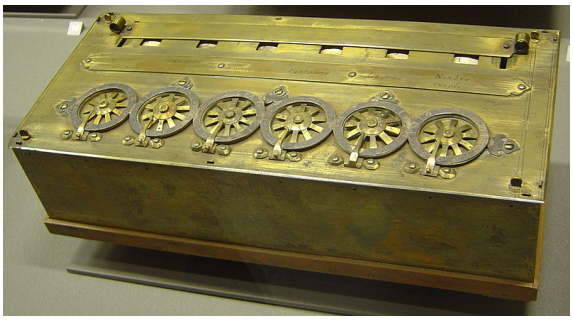
Ein av dei første som er kjent for oss, er eit eksemplar av ein imponerande gresk astrolab frå det 1. århundret fvt. Eitt av 82 funne fragment av denne såkalla Antikythera-mekanismen[7] er vist i figur 1.1.

Figur 1.1: Eit fragment av Antikythera-maskinen.
(Ref.: "Wiki NAMA Machine d'Anticythère 1.jpg")

[2] Sjå "en.wiki Hero of Alexandria".
[3] Sjølve bruken av vindkraft i f.eks. vatningssystem kan vera meir enn 1500 år eldre enn dette igjen, sjå "en.wiki Windmill".
[4] Sjå "en.wiki Windmill fantail".
[5] Sjå "en.wiki Clock".
[6] Sjå "en.wiki Astrolabe".
[7] Sjå "en.wiki Antikythera mechanism".

Ein kunne programmmera dato ved å dreia på eit eige hjul for dette. Ved å sveiva maskinen vidare, ville denne kalkulera og visa komande sol- og måneformørkingar og sannsynlegvis framtidige posisjonar til dei fem planetane som var kjende då. Kalkulasjonane kan ha vore baserte på babylonske oppskrifter og realiserte vhja. i alle fall 30 gir.

Det er kjent at folk heilt tilbake i sumerriket[8], dvs. for meir enn fire tusen år sidan, brukte enkle former for kulerammer, abakussar, som hjelpemiddel ved rekning, men alle kulene blei her flytta manuelt.
Den første kjente maskinen som utførte aritmetiske operasjonar, var Blaise Pascal sin mekaniske kalkulator[9] frå 1600-talet. Eit eksemplar av desse såkalla Pascalinane er vist i figur 1.2.



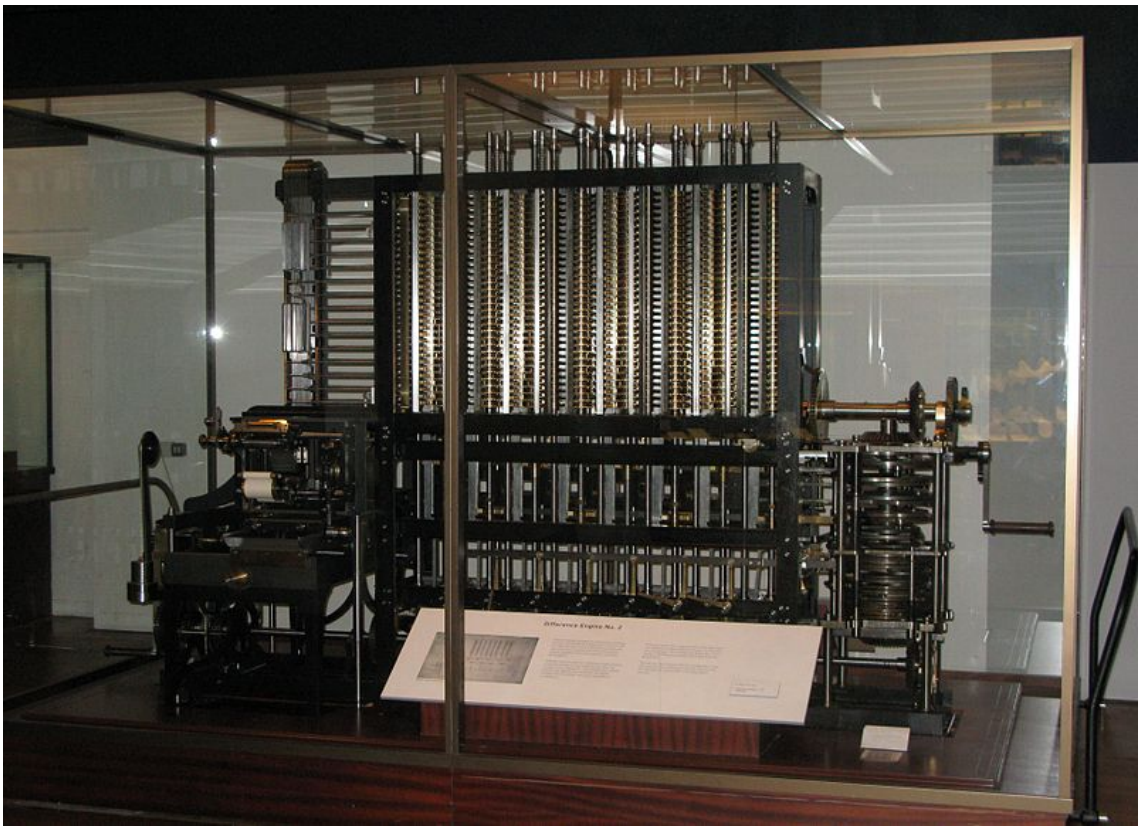
Figur 1.2: Ein Pascaline frå 1652.
(Ref.: "Wiki Arts_et_Metiers_Pascaline_dsc03869.jpg")

Her programmerte ein inn tal, siffer for siffer vhja. hjul. Eigne vindu viste summen av alle tal som var lagde inn sidan sist maskinen blei nullstilt. Summasjonar blei utførte med mente. Maskinane kunne også utføra subtraksjon.

Tradisjonelle kalkulatorar utfører ein eller fleire **faste** operasjonar. Brukaren kan leggja inn data og evt. velja kva operasjon som skal køyrast.
Med ordet datamaskin tenkjer ein ofte på ein meir generell maskin der brukaren i tillegg kan laga og så leggja inn eit **program** som maskinen skal utføra.
Den fyrste maskinen som hadde ein slik datamaskinliknande oppførsel, var vevstolen til Joseph-Marie Jackuard[10]. Arbeidsoperasjonane til denne blei styrte vhja. holkort. Ein kunne altså programmera vevinga. Vevstolen blei demonstrert i 1801.

Charles Babbage[11] er av fleire sett på som datamaskinen sin far. Han blei særleg berømt for sin såkalla differerensemaskin. Maskinen hans var som Pascalinane ein mekanisk og manuell kalkulator, men aritmetikken var her mykje meir avansert. Maskinen kunne rekna ut polynomfunksjonar av høg orden og presisjon. Babbage rakk aldri å fullføra ein prototype av maskinen, men eit eksemplar blei laga ved London Science museum i 1989-91, sjå figur 1.3. Denne prototypen kan gjera utrekningar av sjuandegradspolynom med ei oppløysing på 31 siffer!

[8] Sjå "no.wiki Sumer".
[9] "en.wiki Mechanical calculator".
[10] Sjå "en.wiki Jacquard loom".
[11] "Sjå en.wiki Charles Babbage".



Figur 1.3: Charles Babbage sin differansemaskin. Legg merke til sveiva.
(Ref.: "Wiki babbage_difference_engine.jpg")

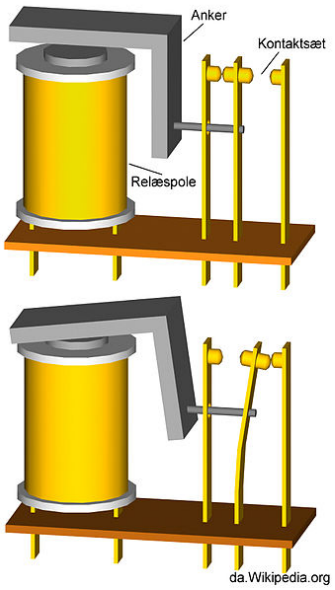
Motivasjonen til Babbage låg i mellom anna dei tidkrevjande og einsformige manuelle utrekningane av logaritmetabellar[12] som blei gjort på den tida, og med mange feil som resultat.

Med oppdagingane til Hans Christian Ørsted[13] på vekselverknadane mellom elektrisitet og magnetisme som igjen la eit viktig grunnlag for forskinga til mellom anna Ampere, Faraday og Maxwell, dukka elektriske og elektromekaniske komponentar opp utover på 1800-talet.

Med telefonen sitt inntog kom etter kvart store automatiserte telefonsentralar[14] på starten av 1900-talet. Desse var baserte på mellom anna **brytarar** og **elektromekaniske rele**. Formålet var å erstatta alle operatørane som sat og ruta samtalar manuelt frå sendarar til mottakarar.
Eit rele med to kontaktsett er vist i figur 1.4.

Releet blir styrt ved å la ein straum gå i spolen eller ikkje. Når straumen er slått på, vil spolen bli ein elektromagnet som trekkjer ankeret til seg som vist nedst i figuren. Det

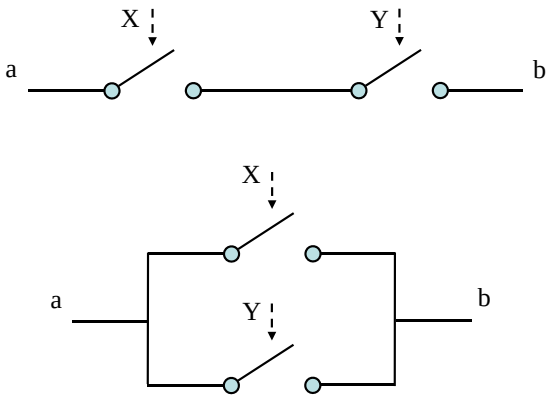
[12]Som kjent kan ein matematisk funksjon tilnærmast av ei Taylorrekkje, som jo er ein polynomfunksjon.
[13]Sjå "da.wiki H.C. Ørsted".
[14]"Sjå en.wiki telephone exchange".



Figur 1.4: Elektromekanisk rele.(Ref.: "Wiki relæ3.jpg")

innerste kontaktsettet vil då opnast og det andre lukkast. Når straumen blir slått av, vil det motsette skje som vist øvst.

I kapittel 3.6 skal ein sjå på eit verktøy som etterkvart blei brukt som grunnlag for analyse og konstruksjon av slike telefonsentralar, nemleg Boolsk algebra. Det var **Claude Shannon**[15], som i masteroppgåva si frå 1937, [9], viste at brytar- og relekoblingar kunne modellerast slik.
To basiskoblingar er viste i figur 1.5.



Figur 1.5: To logiske basiskoblingar.

Shannon kalla tilstanden til ein brytar eller rele[16] for **hindring** ("hindrance"). Ein brytar som er lukka, har då hindringa $X = 0$, dvs. null hindring.

[15] Sjå "en.wiki Claude Shannon".
[16] Ein brytar blir altså opna og lukka manuelt mens eit rele blir opna og lukka med eit styresignal.

Tilsvarande har ein open brytar full hindring, som her er gitt av at $X = 1$.

Ein kan basert på desse definisjonane formulera hindringa X_{ab} for ei kobling mellom terminalane a og b. For den øvste koblinga i figuren blir då den Boolske likninga:

$$X_{ab} = X + Y$$

Viss den første brytaren er open **ELLER** eller den andre brytaren er open, vil koblinga mellom a og b vera open, dvs. ha full hindring.
Ein Boolsk ELLER-operasjon blir symbolisert med eit summasjonsteikn i Boolsk algebra.
Koblinga øvst i figuren blir kalla ei **seriekobling**.

Den nedste koblinga i figur 1.5 er ei **parallellkobling**. For denne får ein følgjande Boolske likning:

$$X_{ab} = X \cdot Y$$

Her har ein full hindring mellom a og b bare viss det er full hindring i eine brytaren **OG** samtidig full hindring i den andre brytaren.
Ein Boolsk OG-operasjon blir symbolisert med eit multiplikasjonsteikn i Boolsk algebra.

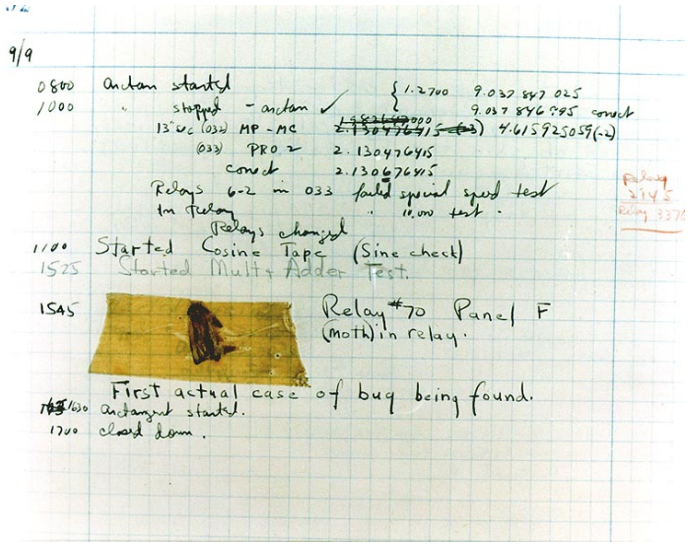
Shannon viste vidare korleis ein med utgangspunkt i slike basisoperasjonar kunne realisera algebraiske likningar for vilkårlege brytar- og relekoblingar.
Han viste også med basis i denne algebraen måtar å **forenkla** slike likningar. Basert på desse kan ein då laga meir optimale koblingar.
Denne oppgåva la med dette grunnlaget for konstruksjon av datamaskinar, og er seinare blitt kåra av nokre til den viktigaste masteroppgåva gjennom tidene!
Claude Shannon blei seinare vel så kjent for samplingsteoremet[17].

Me skal sjå meir på Boolsk algebra i kapittel 3.6.

Utpå 1900-talet kom dei første elektromekaniske datamaskinane. Dei første var eigentleg reine kalkulatorar. Eit berømt eksempel var Harvard Mark I[18]. Han blei bygd ved IBM og flytt til Harvard University i 1944. Maskinen vog 4.5 tonn og trekte 4.5 kW[19]!
Datamaskinen tok inn data via brytarpanel og instruksjonar via holband. Denne "parallellkøyringa" av data og instruksjonar blei opphavet til **Harvard-arkitekturen**[20]. Meir om denne og den andre hovudtypen arkitektur, nemleg **Von Neumann**, står i kapittel 2.1.
Seinare kom det raskare utgåver av maskinen Harvard Mark.

[17] Sjå "en.wiki Nyquist-Shannon sampling theorem".
[18] Sjå "en.wiki Harvard Mark I".
[19] Mykje av denne effekten gjekk til ein elektromotor som var kobla til eit drivverk tilsvarande det som blei sveiva rundt i Babbage-maskinen.
[20] Sjå "en.wiki Harvard architecture".

Ordet **"bug"** brukt om tekniske feil stammar i følgje [10] frå Edison. Når det gjeld ordet **"debugging"** [21], dukka dette opp første gong i samband med Harvard-maskinen. Ordet blei ytra av den seinare berømte programmeringspioneren Grace Murray Hopper[22] i samband med at feilsøking avslørte ein møll mellom to relekontaktar, sjå figur 1.6.



Figur 1.6: Årsaka til ein historisk maskinvarefeil ("hardware bug").
(Ref.: "Wiki H96566k.jpg")

Etter kvart gjorde dei såkalla radiorøyra sitt inntog, og ei rekkje datamaskinar blei baserte på desse. Radiorøyr kunne lagast for likeretting, svitsjing og forsterking. Nokre utgåver er viste i figur 1.7.

Figur 1.7: Radiorøyr. (Ref.: "Wiki Elektronenroehren-auswahl.jpg")

Eit berømt eksempel her var datamaskinen ENIAC[23], som stod klar i 1946. Røyrbaserte kalkulatorar var mykje raskare enn elektromekaniske. Ei samanlikning av dei to eksempelmaskinane her er vist i tabell 1.1.

[21] Det norske ordet **avlusing** eller lysking dekkjer strengt tatt ikkje møll, men er like kort og vel etablert i landet.

[22] Sjå "en.wiki Grace Hopper".

[23] Sjå "en.wiki ENIAC".

	Harvard Mark I (sek)	ENIAC (sek)
Multiplikasjon	6	0.0026
Divisjon	15	0.025
Logaritme	> 60	(Har ikkje data)

Tabell 1.1: Reknefarten til to historiske datamaskinar.

Den første transistoren blei utvikla i 1947[24], og den første kommersielt tilgjengelege integrerte kretsen ("Integrated Circuit", IC), kom i 1961. Her var altså fleire transistorar samla på ei brikke.
Den første transistorbaserte datamaskinen skal ha blitt bygd ved Universitetet i Manchester i 1953[25].
Alt dette danna starten på ei kolossal utvikling.

Det er her bare gitt ein liten smakebit av datamaskinhistoria, og ein skal nå stort sett forlata denne.
Vidare skal ein nå sjå litt på integrerte kretsar og så på korleis datamaskinar er oppbygde.

[24]Sjå "en.wiki Transistor".
[25]Sjå "en.wiki Transistor computer".

1.1.2 Litt om klassifisering av digitale kretsar

Den første kommersielt tilgjengelege IC-en blei produsert av Fairchild Semiconductor og hadde 15 transistorar[26].
IC-ar blir ofte klassifiserte som vist i tabell 1.2.

Kom første gong	**Kategori**	**Portar ("Gates")**	**Transistorar**
1961	Small Scale IC (SSIC)	1 - 25	< 100
Slutten av 1960-åra	Medium Scale IC (MSIC)	25 - 250	< 1000
ca.1975	Large Scale IC (LSIC)	250 - 2500	< 10.000
ca.1980	Very Large Scale IC (VLSIC)	Over 2500	> 10.000
ca.1986	Ultra Large Scale IC (ULSIC)		$> 1.000.000$

Tabell 1.2: Klassifisering av integrerte kretsar, IC. (Ref.: "en.wiki integrated circuit".)

Ein typisk **port** som det blir vist til i tabellen, inneheld pr. definisjon fire transistorar. Med fire transistorar kan ein f.eks. realisera ein NOG-port ("NAND")[27] med to inngangar.

Graden av integrasjon på IC-ar har faktisk heilt fram til nå utvikla seg i rimeleg godt samsvar med ei modifisert utgåve av **"Moore si lov"** frå 1965. Den modifiserte lova sa at talet på transistorar pr. arealeining ville doblast i løpet av 18 månader[28]. Opprinneleg spådde Moore ei dobling for kvar 12 månader, men dette blei altså modifisert ei stund seinare av ein kollega av Gordon Moore i Intel.

Nokre av dei tettaste kretsane nå (2017) er DRAM[29]-minne produsert med 0.020μm-teknologi, der dei minste transistordelane då har breidder ned til $0.020\mu\text{m} = 20$ nm,
dvs. 200 Ångstrøm eller He-atomdiametrar.
På desse minnekretsane kan det vera meir enn 17 milliardar transistorar på typisk $100mm^2$ areal.
Fleire mikroprosessorar i dag inneheld også meir enn 1 milliard transistorar. Intel skal nå (2017) investera meir enn 7 milliardar dollar i ein fabrikk for **7nm**-produksjon. Intel seier at dei med slike inversteringar "sikrer at vi følger takten i Moores lov" [30].

[26] IC-en frå 1961 inneheldt ein enkel logisk funksjon, nemleg ei såkalla J/K-vippe, realisert vhja. såkalla resistor-transistor-logikk (RTL). Seinare kom "transistor-transistor-logikk" (TTL) som var mykje raskare og i tillegg mindre effektkrevjande. TTL-teknologien var dominerande i lang tid, men er nå heilt utkonkurrert av CMOS-teknologien, som er like rask og brukar mykje mindre effekt. Meir om dette kjem i kapittel 3.4.

[27] Meir om logiske operasjonar og portar som realiserer desse kjem i kapittel 3.5.

[28] **Prøv å rekna ut** kva transistortal pr. $100mm^2$ ein skulle hatt i 2018 i følgje den modifiserte Moore-lova viss det i IC-en produsert i 1961 var 15 transistorar pr. $100mm^2$.

[29] I dynamisk RAM er kvar minnecelle bygd opp av ein transistor og ein kondensator. DRAM er såleis mykje tettare enn SRAM, der kvar av minnecellene normalt krev 6 transistorar. Meir om ulike minne står i kapittel 3.10.9 og i bolk C9 i [8]. Eit eksempel er 16Gigabit-kretsen *MT40A1G16* frå Micron.

[30] Sjå "http://elektronikknett.no/Elektronikk-eMagasin", utgåve 02/2017. Dette er eit interessant blad og med nyttige nettsider i tillegg.

1.1.3 Datamaskinen i eit nøtteskal

Me er kan hende ennå bare i startfasen av den utviklinga som starta med inntoget av integrerte kretsar i 1960-åra. Det er eit aukande og umetteleg behov for datamaskinkraft. Men kva er det eigentleg datamaskinen hjelper oss med?
Svaret er følgjande to ting:

- **Datalagring:**
  Her er det eit sterkt aukande behov i form av både såkalla **flyktige minne** og ikkje-flyktige eller **permanente minne**. Førstnemnde er ulike typar RAM, mens permanente minne hovudsakleg er diskbaserte eller av typen Flash. Meir om dette kjem i kapittel 3.10.9.
- **Databehandling:**
  Dette er alle typar **aritmetiske operasjonar** og **logiske operasjonar** samt all **overføring** og **presentasjon** av data.

1.2 Datamaskinar i dag

1.1.1 Litt om grunnlaget for datamaskinen

Datamaskinen er basert på **totalssystemet**, også kalla det **binære** talsystemet. Den grunnleggjande byggjesteinen, nemleg transistoren, kan ved bruk i digitale kretsar[31] bare vera i ein av to **tilstandar**, nemleg **høg**/"**1**" eller **låg**/"**0**" [32].
Både datalagring og databehandling blir altså realisert vhja desse to tilstandane.

Eksempel 1.1. *Talsystem*

Talet *193* i titalssystemet er det same som *11000001* i totalssystemet.
Eit siffer i det binære talet blir kalla ein **bit**. Kvar bit kan altså ha verdien 0 eller 1.
Desse talsystema er såkalla **posisjonstalsystem**. Dette kan illustrerast med tala over:

$$
\begin{aligned}
193_{10} &= 1 \cdot 10^2 + 9 \cdot 10^1 + 3 \cdot 10^0 \\
11000001_2 &= 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
&= 128 + 64 + 1 = 193
\end{aligned}
\tag{1.1}
$$

[31] Ved bruk i analoge kretsar som f.eks. tradisjonelle audioforsterkarar, bruker ein heile spenningsområdet til transistoren.
[32] Meir om sjølve transistoren kjem i kapittel 3.4.

Ved **datalagring** blir kvar bit lagra i kvar si **minnecelle**.
Ved **aritmetiske operasjonar** arbeider datamaskinen på same måte som me kjenner dette frå titalssystemet. Ein kan finna meir om totalssystemet og totalsmatematikk i vedlegg A.1.

1.2.2 Nokre datamaskineksempel

1.2.2.1 Innleiing

Sidan den første mikroprosessoren kom i 1971[33], har det vore ei rivande utvikling. Datamaskinar av ulike slag treff ein på fleire gonger dagleg, ofte utan å vera klar over det. Mikroprosessoren er sjølve hjernen i datamaskinen. Ein eller fleire slike hjernar finst mellom anna i mobiltelefonen, nettbrettet, bilen, vaskemaskinen og PC-en, der dei utrøytteleg utfører sine program, dvs. lister av instruksjonar.

Figur 1.8: Eksempel på eit nettbrett.
(Ref.: "Wiki IPad_2_Smart_Cover_at_unveiling_crop.jpg".)

Eit eksempel på ein vanleg datamaskin i dag er vist i figur 1.8. Ei populær utgåve av slike nettbrett var Ipad3[34], som kom i 2012. Dette har fleire prosessorar, nemleg:

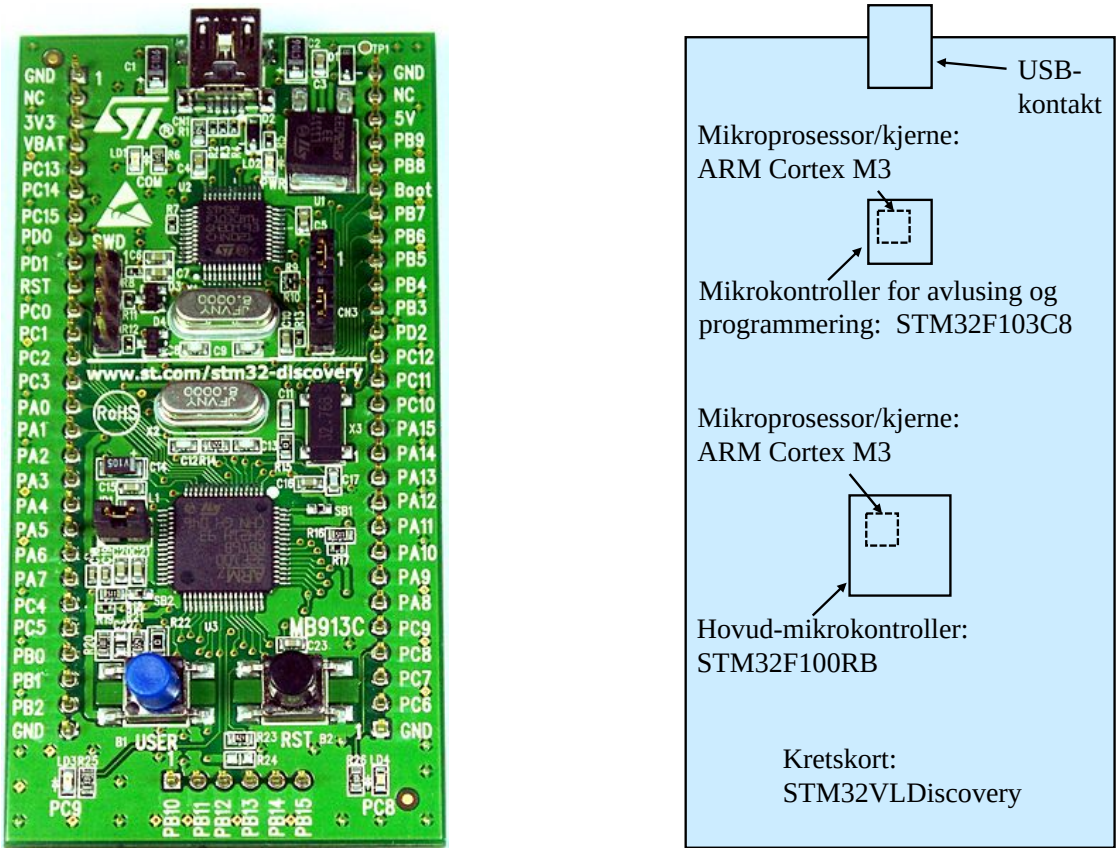
- Hovudprosessor ARM Cortex A9 som eigentleg er dobbel, dvs. at han har to mikroprosessorar eller kjernar ("core").
- Grafisk prosessor Power VR, som også har to kjernar. Denne arbeider mellom anna med skyving (translasjon) og dreiing (rotasjon) av skjermbilete.
- Bildesignalprosessor for automatisk gjenkjenning av fjes mm. Denne er integrert saman med mellom anna hovudprosessoren og hurtigminne ("cache") på systembrikka[35] Apple A5.

ARM-prosessorane finst i mange utgåver og er i brei og aukande bruk verda over. Kvifor akkurat denne prosessoren nå er dominerande utanfor PC-verda, er ei interessant historie. Meir om dette står mellom anna i kapittel 1 hjå Yiu, [3] og i kapittel 1 hjå Toulson og Wilmshurst, [6].

[33] Sjå "no.wiki Intel 4004".
[34] Sjå f.eks. "en.wiki list of ios devices".
[35] På engelsk "System on Chip", SoC.

1.2.2.2 Maskinvareplattform for emnet Datamaskinarkitektur

Emnet Datamaskinarkitektur blei undervist første gong hausten 2013, og har ARM som maskinvareplattform. Basis er her mellom anna det vesle **kretskortet** ("printed circuit board", PCB) som er vist i figur 1.9.



Figur 1.9: Den ARM-baserte maskinvareplattformen *STM32VLDiscovery*.
(Ref.: "Wiki STM32_LV_Discovery_board.jpg")

Blokkskjemaet til høgre i figuren viser litt av innhaldet på kretskortet. Den største integrerte kretsen i figuren og sjølve hovudbrikka på dette kortet er ein ARM-basert **mikrokontroller** av typen STM32F100RB frå STMicroelectronics. Denne kan programmerast via USB frå eit utviklingsverktøy på PC.

Ein mikrokontroller er ein liten datamaskin integrert på ei brikke, noko ein skal sjå meir på i kapittel 2.
Inni ein mikrokontroller er det ein mikroprosessar eller kjerne, som utfører programmet som skal køyrast. I tillegg er det inne i mikrokontrolleren minne og modular for seriell og parallell kommunikasjon mm.

Mikrokontrolleren STM32F100RB inneheld mellom anna følgjande:

- Mikroprosessor/kjerne: ARM Cortex-M3.
- Databreidde: 32 bit.
- Klokkefrekvens: 24 MHz.
- Programminne (Flash): 128 KB (kilobyte).
- Dataminne (SRAM): 8 KB.
- Digitale linjer inn/ut: 80[36].

Samanlikna med klokkefrekvens og minnekapasitet til ein PC er dette ein liten datamaskin. I vanleg forbrukarelektronikk og svært mange industrielle produkt for styring og overvaking er ein mikrokontroller likevel meir enn kraftig nok. Han har i tillegg svært lågt effektforbruk, noko som er nyttig i batteridrivne produkt.

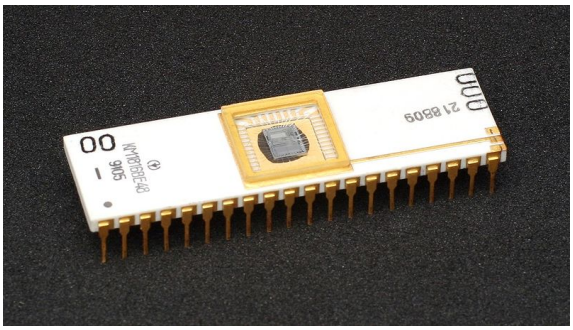
På kretskortet i figur 1.9 er det ein mikrokontroller i tillegg. Denne har same kjerne som den andre mikrontrolleren, men skil seg litt ut i oppbygginga elles. STM32F103C8 har bl.a. ein USB-modul og står på plattformkortet som ein grensesnittsmodul mellom PC-en og hovudmikrokontrolleren.
Oppgåva er følgjande:

- Administrera overføring av nytt program til Flash-minnet på hovud-mikrokontrolleren når brukaren lastar dette ned frå utviklingsverktøyet på PC-en.
- Administrera overføring av interne data frå hovud-mikrokontrolleren til PC-en når brukaren køyrer avlusing ("debugging") av program.
- Administrera overføring av avlusingskommandoar til hovud-mikrokontrolleren frå utviklingsverktøyet på PC-en. Viss brukaren f.eks. vil køyra eit program steg for steg, skal hovud-mikrokontrolleren stoppast etter kvart steg og interne data overførast til PC-en.

[36] Av desse linjene er 51 tilgjengelege som pinnar på kortet STM32VL Discovery. Som vist i figuren, står pinnane langs kantane og er namngitte.

1.2.2.3 Litt om historia til mikrokontrolleren

Den første mikrokontrolleren kom i 1976. Han blei produsert av Intel og heitte 8748[37]. Databreidda var 8 bit, dataminnet var på bare 64 byte og programminnet
på 1024 byte = 1kbyte!
Figur 1.10 viser ein flott sovjetisk klon av denne i kvit keramikk og gull[38].



Figur 1.10: Sovjetisk klon av 8748 i gull og keramikk.
("Ref.: "Wiki KL_USSR_KM1816BE_i8748_Black_Background.jpg".)

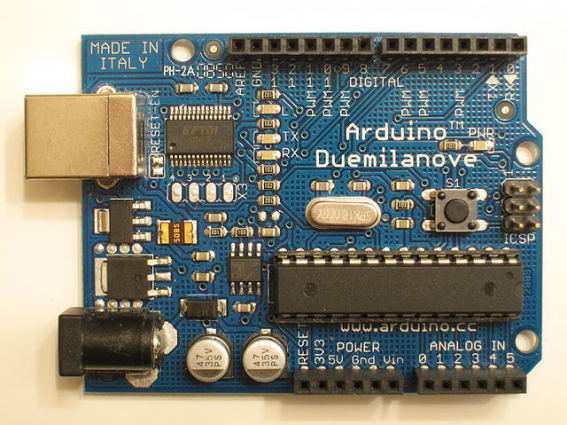
I 1980 kom oppfølgjaren 8051 som blei veldig populær. Han lever den dag i dag som IP-modul[39] i programmerbar elektronikk.

1.2.2.4 Fleire datamaskineksempel

Med inntoget av ARM-prosessoren har det dukka opp mange og rimelege kretskort tilsvarande det i figur 1.9 på kort tid. Eit som raskt blei svært populært hjå hobbyprogrammerarar, var f.eks. det kraftige Raspberry Pi[40].

Arduino-standarden har også vore veldig populær, og eit mykje brukt hovudkort er vist i figur 1.11.

[37] Sjå "en.wiki Intel-MCS48".
[38] Under den kalde krigen hadde ikkje austblokka tilgang på kretsar utvikla og produsert i Vesten, men lukkast likevel i å få tak i konstruksjonsunderlag slik at mange kretstypar kunne produserast. Gull blei mykje brukt då dette ikkje var mangelvare i aust. Sjå "http://www.cpushack.com/soviet-cpus.html".
[39] "Intellectual Property", sjå kapittel 3.7.1.3 på side 160.
[40] Sjå "en.wiki Raspberry PI".



Figur 1.11: Eit vanleg Arduino-kort. (Ref.: "Wiki Arduino_Duemilanove_0509.jpg".)

På Arduino-kort dominerer Atmel sine mikrokontrollerar av typen AVR[41]. Desse er som fotnota viser, faktisk utvikla i Trondheim.
ARM gjorde sitt første inntog på kort av denne standarden i 2012, noko mange hadde venta på.

1.2.3 Hovudgrupper av datamaskinar

Ein kan dela datamaskinar opp i to hovudgrupper, nemleg **innebygde** og **generelle** datamaskinar.

I produkt som mobiltelefon, nettbrett, bil og vaskemaskin er det sjølve funksjonane, f.eks. vaskeprogramma, som er dei viktige, og ikkje datamaskinen i seg sjølv. Denne blir ofte usynleg, og ein kallar difor ofte datamaskinen for eit **innebygd** ("embedded") system[42], (ibs).

Det motsette er f.eks. ein personleg datamaskin, PC, som har eit breitt bruksområde utan å vera tilknytta anna utstyr. Dette er ein **generell** datamaskin som kan brukast til f.eks. utvikling av innebygde system.
Ein PC kan sjølvsagt også byggjast inn og då bli til f.eks. ein minibank, ein fotoautomat for passbilete eller eit kassaapparat.
Det er derfor bruken av datamaskinen eller produktet som datamaskinen er ein del av, som avgjer klassifiseringa. Ein robust industridatamaskin av typen PLS[43] vil vera ein generell datamaskin for systemutviklaren, men del av eit innebygd system for brukarane av f.eks. ein PLS-styrt bilvaskemaskin.

[41] Namnet står truleg for **A**lf (Erik Bogen) og **V**egard (Wollan) sin **R**isc-prosessor, sjå "en.wiki Atmel AVR".
[42] Sjå "en.wiki Embedded system".
[43] Programmerbar logisk styring, jfr. "en.wiki Programmable logic controller".

1.2.4 Utviklingstrekk

Når ein ser litt på den utviklinga som har vore til nå, er det nokre hovudtrekk som trer fram:

- **Aukande integrasjon:**
  Elektronikk på silisium har blitt mindre og mindre, og fleire funksjonar har då fått plass på ei brikke. Små datamaskinar på **ei** brikke, dvs. mikrokontrollerar, kom ganske tidleg, og er i dag så kraftige at dei et seg innpå PC-marknaden. ARM er ein effektiv prosessorarkitektur som nå dominerer som kjerne i slike mikrokontrollerar.
  I datamaskinar så kraftige som PC-ar ser ein likevel at det er ei grense for integrasjon. Dette er pga. av all effekten som går over til varme og må leiast vekk under køyring. Ein må difor når ein kjem opp i slike ytingar, basera datamaskinen på eit sett av brikker.

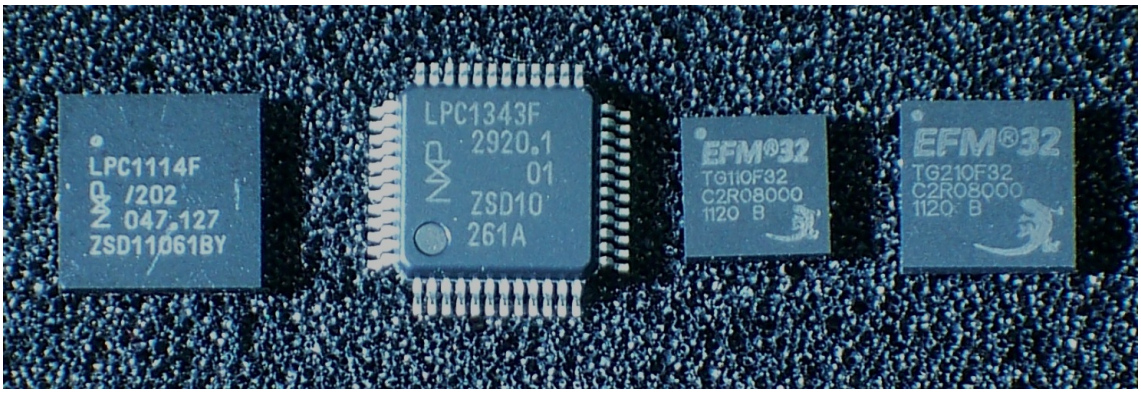
  Det vil også vera ei grense for kor små transistorane kan bli, og det blir stadig meir krevjande å redusera størrelsen etter kvart som ein nærmar seg denne nedre grensa. Derfor lagar ein nå i aukande grad prosessorar med mange kjernar i staden for å auka klokkefrekvensen og talet på transistorar.

- **Prosesseringskapasiteten aukar:** Dette er eit resultat av aukande integrasjon og meir effektiv prosessorarkitektur. Bitbreidda i prosessorane kan aukast og er nå 64 i PC-verda.
  Klokkefrekvensen i PC-verda stongar nå hovudet i taket litt over 3 GHz, men utviklinga går vidare ved at ein som nemnt legg inn fleire kjernar, dvs. prosessorar.

- **Gjerrige mikrokontrollerar:**
  Med meir effektive prosessorar, aukande integrasjon og avanserte sovefunksjonar, har effektforbruket i mikrokontrollerar gått kraftig ned.
  Det er eit veldig fokus på dette då ein samtidig som mobile og batteridrivne produkt blir stadig kraftigare, ønskjer at tida mellom ladingar skal bli lenger.

  ARM-baserte mikrokontrollerar er i ei særstilling når det gjeld effektforbruk. Og blant desse er det faktisk ein mikrokontrollerserie utvikla av det norske selskapet Energy Micro[44] som framstår som dei gjerrigaste. To kretseksempel frå Energy Micro, nå Silicon Labs[45], er vist til høgre i figur 1.12.

[44] Sjå "en.wiki EFM32".
[45] Energy Micro blei kjøpt opp av Silicon Labs i 2013, sjå "en.wiki Silicon Labs".



Figur 1.12: Nokre mikrokontrollerar frå NXP og Energy Micro.
(Ref.: "Wiki ARM_Cortex-M0_and_M3_ICs_in_SMD_Packages".)

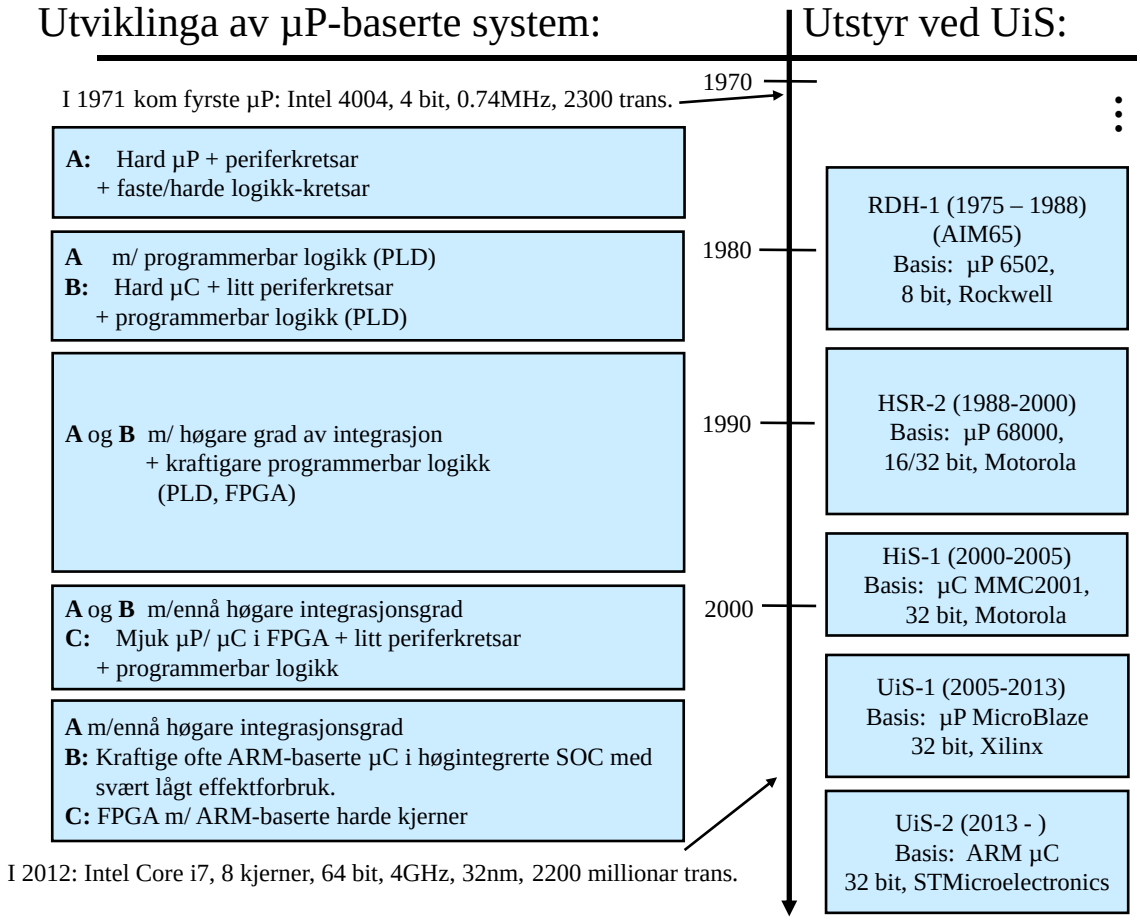
- **Sjølvforsynte komponentar:**
  Med gjerrig elektronikk kan f.eks. trådlause sensornodar i eit nettverk greia seg utan batteri. Ved å hausta energi[46] frå omgivnadane i form av lys, varme eller vibrasjonar mm. og omforma dette til elektrisk energi, kan desse vera sjølvforsynte.
  Det har vore stor aktivitet på dette feltet dei siste åra.

- **Prisreduksjon:**
  Med den veldige prisreduksjonen ein har sett over tid, kan avansert elektronikk leggjast inn i alle slags produkt, og i tillegg koblast til Internett. Begrepet Internet of Things (IoT) blei lansert i 1999[47], og det er bare fantasien som set grenser for kva dette kan brukast til.

[46] Stikkordet er "energy harvesting", sjå en.wikipedia.
[47] Sjå "en.wiki Internet of Things".

1.2.5 Ein illustrasjon av utviklinga

Hovudtrekka i utviklinga av mikroprosessorbaserte system er forsøkt illustrert i figur 1.13. Her er også vist dei ulike øvingsmaskinane brukt i ingeniørutdanninga på Ullandhaug og i kva tidsrom dei blei brukte. Første maskinen her var ein standard datamaskin av type AIM65, mens etterfølgjarane fram til UiS-2 var heilt eller delvis eigenkonstruerte. UiS-2 er ein standardmaskin, sjå figur 1.9.



Figur 1.13: Ulike steg i utviklinga av mikroprosessorbaserte system.
(* RDH - Rogaland distriktshøgskole, 1969 - 86,
HSR - Høgskolesenteret i Rogaland, 1986 - 94,
HiS - Høgskolen i Stavanger, 1994 - 2004,
UiS - Universitetet i Stavanger, 2005 -)

Utviklinga har som nemnt i førre delkapittel gått mot aukande integrasjon, dvs. stadig færre, men kraftigare kretsar og aukande bruk av programmerbare logiske kretsar. Dette gjeld også dei viste øvingsmaskinane på Ullandhaug.
Eit viktig steg i utviklinga kom heilt på slutten av 90-talet som vist i figuren. Ein kunne då laga sin eigen mikroprosessor basert på ei oppskrift som ein så lasta ned i ei programmerbar elektronisk brikke. Dette gir ein meir fleksibel datamaskin, og kortare produktutviklingstid.

Arkitekturen til mikroprosessoren er her altså spesifisert i ei oppskrift, og mikroprosessoren blir då kalla ein mjukprosessor. Øvingsmaskinen UiS-1 brukt fram til 2013, var basert på mjukprosessoren MicroBlaze frå Xilinx Inc[48]. Det er gitt ut et kompendium her på konstruksjon av små innebygde system med utgangspunkt i MicroBlaze, sjå [1].

Ein ulempe med mjukprosessorar er relativt høgt effektforbruk. I batteridrivne innebygde system vil derfor harde mikrokontrollerar ennå dominera, og med ARM som kjerne i dei aller fleste som nemnt før.
Andre ulemper med mjukprosessorar er mangel på billege utviklingsverktøy og plattformkort. Bruken av denne teknologien i undervisninga er i tillegg noko tyngre for studentane enn med bruk av harde mikrokontrollerar. Dette og ARM sin etterkvart dominerande stilling er årsaker til at ein nå har gått over til ARM som plattform i emnet Datamaskinarkitektur.
Det må likevel nemnast her at Xilinx sin nyaste familie av programmerbare kretsar, Zynq-familien, inneheld ARM-kjernar som eigne modular i kretsane. Eit plattformkort basert på nettopp Zynq-familien vil bli brukt i digitalteknikkdelen av undervisninga i Datamaskinarkitektur. Meir om dette kjem i kapittel 3.7.1.3.

Første PC-en kom i 1980. Merk at slike maskinar ennå er av type A i figuren. Grunnen til at ein ikkje integrerer det meste på ei brikke i så kraftige datamaskinar og dermed får eit system av type B, er som nemnt effektforbruket. Ei slik brikke ville sjølv med teknologien i dag havarert pga. for høg kjernetemperatur.
I tillegg ville kompleksiteten bli så stor at brikkene ville bli vanskelege å produsera med tilstrekkjeleg økonomisk margin.

I tida framover vil auken i bruk av kraftige programmerbare kretsar halda fram, mens ein på prosessorsida som nemnt har byrja å gå nye vegar. Eit relativt tidleg eksempel på prosessorar med mange kjernar er Intel Core i7 vist nedst i figur 1.13. Utfordringa er her å laga kompilatorar som greier å utnytta slike parallelle prosessorar.

[48] Xilinx er ein av verda sine største produsentar av programmerbare logiske kretsar, sjå "xilinx.com".

1.2.6 Litt om opplegget i emnet Datamaskinarkitektur

Emnet Datamaskinarkitektur er tenkt å gi følgjande **læringsutbytte**:

- **Kunnskap**: Vita korleis ein datamaskin generelt og eit innebygd system (IBS) spesielt er oppbygd. Vita korleis grunnleggjande programstrukturar er oppbygde. Vita kva som ligg i omgrepet sanntidsoppførsel.
- **Dugleik** (ferdigheit): Kunne realisera grensesnitt mellom parallell- og serieportar i ein mikrokontroller og eksterne komponentar. Kunne både utvikla overordna programvarestruktur og kode for eit enkelt IBS basert på ein spesifikasjon av ønskte funksjonar. Kunne realisera enkle PC-baserte brukargrensesnitt mot IBS. Kunne realisera enkel digital støtte-elektronikk til ein mikrokontroller.
- **Generell kompetanse**: Fundamental forståing av struktur, eigenskapar og potensiale til datamaskinbaserte system.

For å kunne oppnå dette, er **innhaldet** i emnet som vist under:

- **Innleiing:** Overordna datamaskinstruktur, innebygde system.
- **Grunnleggjande digitalteknikk:** Transistor, logisk krets, talsystem, sentrale logiske funksjonar, programmerbar elektronikk.
- **Maskinvare:** Mikroprosessorar og mikrokontrollerar med vekt på ARM, minne, timer, avbrot, seriell og parallell kommunikasjon.
- **Programvare:** Lågnivå- og høgnivåprogrammering, sanntidssystem, operatørgrensesnitt.

Emnet er basert på forelesingar, teoriøvingar og laboratorieøvingar.
Så gjeld det og å bruka referansane, søkja opp nye referansar, bruka forelesingsnotata, gjera øvingar, og gjerne utvida systemet ein arbeider med på laben utover det som står i oppgåvetekstane.

Utviklingsverktøyet er gratis og kan installerast på eigen PC slik at ein kan arbeida med plattformkortet STM32VLDiscovery andre stader også.
Dette kortet kan greitt skaffast f.eks. hjå leverandøren Elfa[49] og kostar litt over 100 kr.

[49] Ein finn dette ved å søkja på "Elfa Evalueringskort STM32".

1.3 Litt om resten av skrivet

Vidare inneheld dette skrivet følgjande:

- I kapittel 2 ser ein på mikroprosessorbaserte system generelt, men avgrensar dette til ein-prosessorsystem ("single processor systems"). Ein ser som ein del av dette på høgnivå- kontra lågnivåprogrammering og på korleis ein kan byggja opp programvaren for slike system. Eksempel er baserte på mikroprosessoren ARM Cortex-M3.
- Kapittel 3 tar for seg digitalteknikk frå botnen av og viser oppbygging og virkemåte til sentrale byggjesteinar i ein datamaskin. Ein vil her arbeida seg opp frå transistoren og opp til ein enkel mikroprosessor.
  I kapittel 3 ser ein også på programmerbar elektronikk generelt og litt om korleis ein konstruerer digital logikk i dag.
- I siste kapittel ser ein generelt på korleis ein kan laga grensesnitt mot omverda. Dette blir eksemplifisert med eit grensesnitt mellom ein ekstern skjermmodul og ein parallellport (GPIO-modul) i mikrokontrolleren vår.
- Vedlegg A inneheld som nemnt tidlegare meir om totalssystemet og totalsmatematikk samt noko om binærkoding.

Kapittel 2

Mikroprosessorbaserte system

Dette kapitlet vil handla om følgjande:

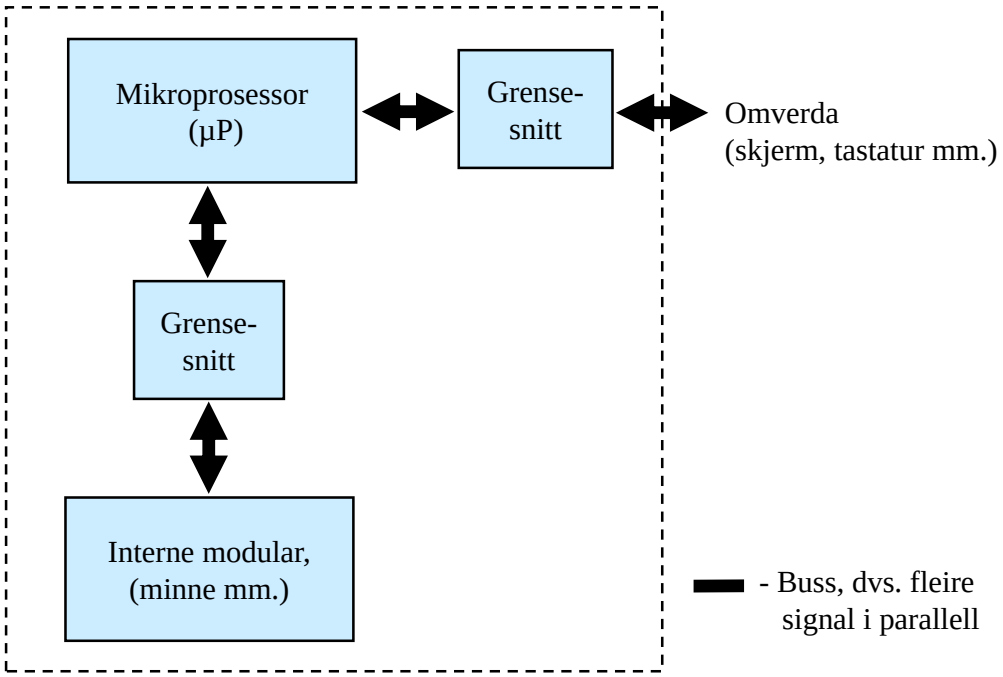
- Oppbygginga av mikroprosessorbaserte system generelt.
- Oppbygging og virkemåte til mikroprosessoren.
- Mikroprosessoren ARM Cortex-M3 (CM3).
- Den CM3-baserte mikrokontrolleren STM32F100.
- Høg- og lågnivåprogrammering.
- Programvarestandarden CMSIS.
- Ulike programstrukturar for eit system.

2.1 Generelt oppsett

Oppbygginga av eit generelt mikroprosessorbasert system, dvs. ein datamaskin, er vist i figur 2.1.

Eit **grensesnitt**[1] er her ein elektronikkmodul som gjer at mikroprosessoren, μP-en, kan snakka med andre modular i eller utanfor systemet.
Eit mikroprosessorbasert system blir ofte også kalla eit ”smart” eller ”intelligent” system.

[1] Omgrepet grensesnitt har forskjellig tyding alt etter samanhengen ordet blir brukt i. Nedanfor er nokre eksempel på bruk av ordet grensesnitt:
1) Ein har grensesnitt mellom menneske og maskin, også kalt brukargrensesnitt.
2) Innanfor programmering snakkar ein om programmeringsgrensesnitt (”application programming interface”, API) når ein for eksempel bruker metodar frå eit bibliotek.
3) Ved kobling av ein laserskrivar til ein PC med kabel, ser ein ofte på sjølve kontaktane som fysiske grensesnitt.



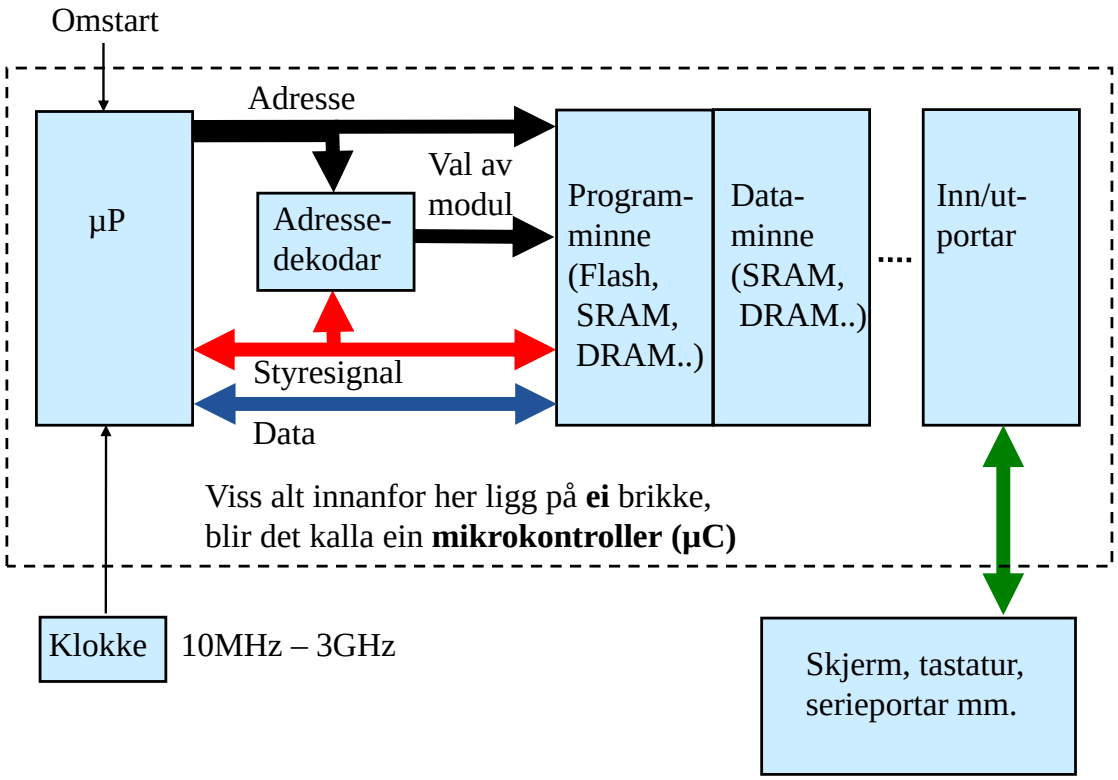
Figur 2.1: Generelt mikroprosessorbasert system.

Det **å konstruera ein datamaskin** dreier seg om å utforma grensesnitta i systemet. Dette kjem ein tilbake til i kapittel 4.
I tillegg til å utforma grensesnitta, må ein sjølvsagt ha med dei rette modulane i systemet. Typiske modular i eit mikroprosessorbasert system er viste i figur 2.2.

I **programminnet** ligg sjølve **programmet**, dvs. ei liste av **instruksjonar** som mikroprosessoren skal utføra.
Data, dvs. variablar, som skal handterast vhja. desse instruksjonane, ligg i **dataminnet**. Når det gjeld dei minnetypane som figuren viser til, kan ein dela desse inn i to hovudtypar, nemleg permanente og flyktige (”volatile”) minne. I ein datamaskin er minstekravet at programmet er lagra permanent, dvs. at det ikkje blir sletta viss kraftforsyninga går av. I større datamaskinar som f.eks. PC-ar brukar ein ennå harddiskar, men med såkalla Flash-diskar som eit robust alternativ. I mindre system er Flash-minne den dominerande teknologien.
Av flyktige minne dominerer dynamisk RAM, DRAM, i større system mens statisk RAM dominerer dei mindre systema. SRAM-teknologien er meir plasskrevjande på silisiumsbrikka, men SRAM-brikker (IC-er) har et greitt grensesnitt og er enklare å kobla seg til.
Grunnen til at ein ikkje bruker Flash-minne til all lagring, er at det er grenser for kor mange gonger ein kan skriva til minnet samt at skrivinga går tregt samanlikna med RAM. Meir om desse minnetypane kjem i kapittel 3.10.9.

Modulane i eit mikroprosessorbasert system er knytta saman vhja. ein eller fleire **systembussar**[2]. Figuren viser det aller enklaste oppsettet, ein mikroprosessor som bare har ein

[2]Ein buss er ei samling digitale linjer mellom komponentar i ein datamaskin. Sjå og ”Wiki Bus (com-



Figur 2.2: Typisk oppsett av eit mikroprosessorbasert system.

systembuss.
Ein systembuss inneheld tre underbussar:

- **Databuss**
  Over databussen hentar μP-en inn programinstruksjonar eller data[3], dvs. **lesing**, eller μP-en overfører data til ein annan modul, dvs. **skriving**. Databussen er altså **bidireksjonal** då overføringar kan gå begge vegar.
  Breidda på bussen er avhengig av kva slags prosessor som blir brukt, og er vanlegvis på 8, 16, 32 eller 64 bit.

- **Adressebuss**
  Alle modulane i systemet har kvart sitt adresseområde. Dataminnet har f.eks. eit adresseområde gitt av kor stort minnet er, og inne i minnet ligg variablar på kvar sine adresser.
  På adressebussen legg μP-en ut informasjon om **kor** data skal hentast frå[4] eller overførast til.

puting)".

[3]Databussen kan også vera delt i to; ein databuss for programinstruksjonar, og ein databuss bare for data.

[4]Eigentleg kopierast frå.

Ein spesiell modul i systemet er **adressedekodaren**. Denne ser kva adresseområde den løpande adressa tilhøyrer, og vel/aktiverer rett modul.

- **Styrebuss** ("control bus")
  Denne bussen innheld signal som blir styrt av μP-en slik at overføringane går føre seg på rett måte.
  Mellom anna har ein her signal som fortel ein modul om overføringa skal vera ei lesing eller skriving.
  Styrebussen kan også ha signal som μP-en les. Feks. kan ein modul gi ut eit opptattsignal ("busy") som får μP-en til å utsetja ei dataoverføring.

Merk at systemet blir kalla ein **mikrokontroller**, μC, viss alt innanfor den stipla ramma i figur 2.2 er plassert på **ei** brikke, ("chip"). Mikrokontrolleren er altså ein datamaskin på ei brikke. I kapittel 1.2.2 blei det vist nokre eksempel på mikrokontrollerar.

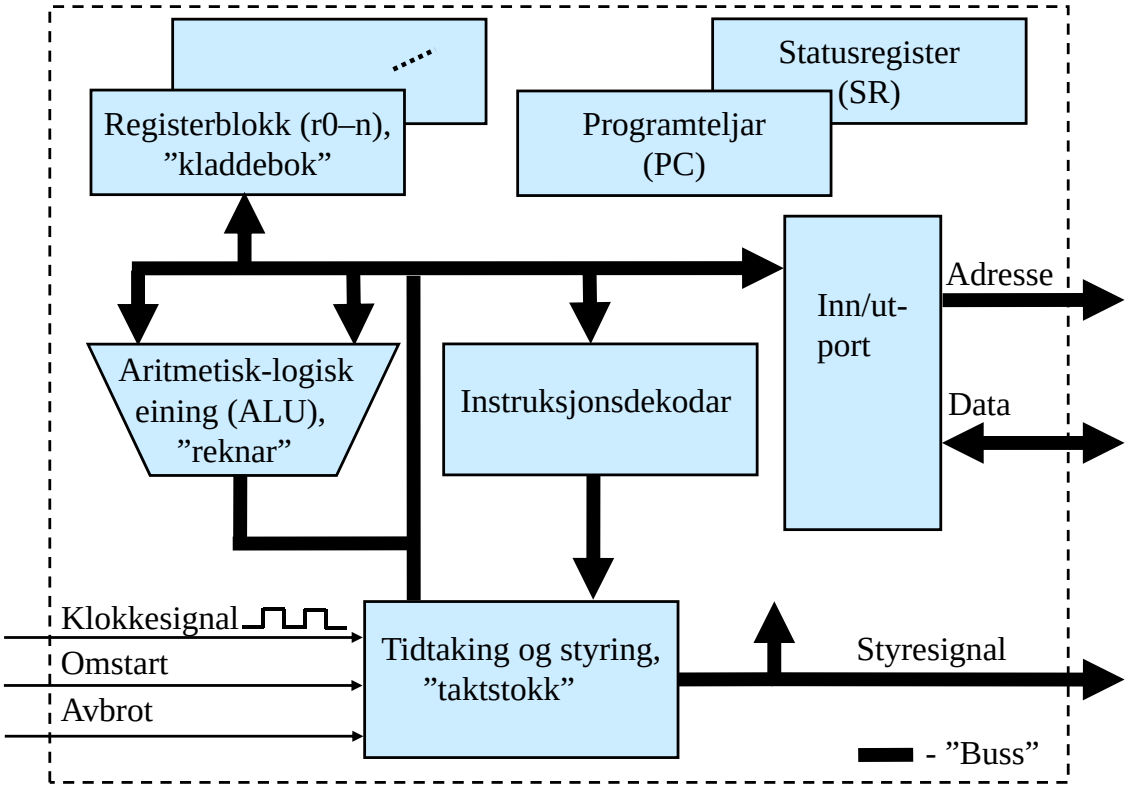
Systemet i figur 2.2 har ein såkalla **Von Neumann**-arkitektur (VNA). Mikroprosessoren både hentar instruksjonar og overfører data over ein og same systembuss. Dette gir ein enkel arkitektur, men bussystemet kan lett bli ein flaskehals, den såkalla Von Neumann-flaskehalsen.
Von Neumann-arkitekturen var vanleg i dei første elektroniske datamaskinane. Ein alternativ arkitektur er som nemnt i kapittel 1.1.1, **Harvard**-arkitekturen. Her har ein **separate systembussar** for instruksjons- og dataoverføring. Dette gir ein meir omfattande, men også meir effektiv arkitektur. Mikroprosessoren kan då kontinuerleg henta inn nye programinstruksjonar mens han parallelt hentar inn eller sender ut data som blir prosesserte av programmet.
Ein har fleire variantar av denne arkitekturen, og desse går ofte under namnet modifisert Harvard-arkitektur[5] (MHA). Dei første MHA-baserte mikrokontrollerane kom utpå 1980-talet. Denne arkitekturen er dominerande i dag, men opptrer som nemnt i mange variantar. Arkitekturen mogleggjer mellom anna parallellkøyring ("pipelining") av instruksjonar, noko som gir kraftig redusert utføringstid for eit program. Meir om dette kjem i kapittel 2.2.4.

[5] Sjå "en.wiki Modified Harvard architecture".

2.2 Litt om mikroprosessoren

2.2.1 Generelt oppsett

Sjølve **mikroprosessoren** er grovt sett bygd opp som vist i figur 2.3.



Figur 2.3: Generelt oppsett av ein mikroprosessor.

Den sentrale modulen er den **aritmetisk-logiske eininga**, dvs. reknemodulen. I denne kan ein få gjort addisjonar og subtraksjonar samt logiske samanlikningar og operasjonar av mange slag. For å kunne bruka denne på ein effektiv måte, treng μP-en ei **registerblokk**, dvs. eit lite minne, for kortvarig lagring av data og adresser.
Her har f.eks. mikroprosessoren vår, ARM Cortex-M3, 16 register med breidde på 32 bit. Meir om dette kjem i kapittel 2.4.4.2.
For å kunne henta **inn** data og levera **ut** resultat av ALU-operasjonane, treng μP-en ein **port** mot dei andre modulane i systemet.
Funksjonane til bussane inn og ut av mikroprosessoren er som vist i kapittel 2.1.

Styring av operasjonane inne i μP-en skjer vhja. ein eigen **styremodul**. Eit **klokkesignal** gjer at μP-en kan halda ein fast takt i arbeidet. Klokkefrekvensen for ulike mikroprosessorar ligg vanlegvis i området vist i figur 2.2.

Her har f.eks. mikrokontrolleren vår, STM32F100RB[6], **klokkefrekvensen** $f_{clk} = 24$ MHz[7].
Dette gir følgjande **klokkesyklustid** eller periodetid:

$$t_{clk} = \frac{1}{f_{clk}} = \frac{1}{24 \cdot 10^6}\ sek \approx 41.7 \cdot 10^{-9}\ sek = 41.7\ nsek$$

Mikroprosessoren finn ikkje opp arbeidsoppgåvene sine sjølv, men les inn **instruksjonar** frå **programminnet**. Etter **innlesing** ("fetch") blir instruksjonen **dekoda**, dvs. tolka, i ein eigen modul vist i figuren, og så blir instruksjonen **utført** ("executed"). Mange av instruksjonane til ein moderne prosessor blir utførte i løpet av 1 klokkesyklus, mens andre instruksjonar i **instruksjonssettet** til prosessoren kan ha ei utføringstid på fleire klokkesyklar.

Eit eige register i mikroprosessoren, nemleg **programteljaren** ("program counter", PC), viser til ei kvar tid kor langt ein er kome i programmet. PC inneheld altså **adressa** til instruksjonen som nå blir køyrt.

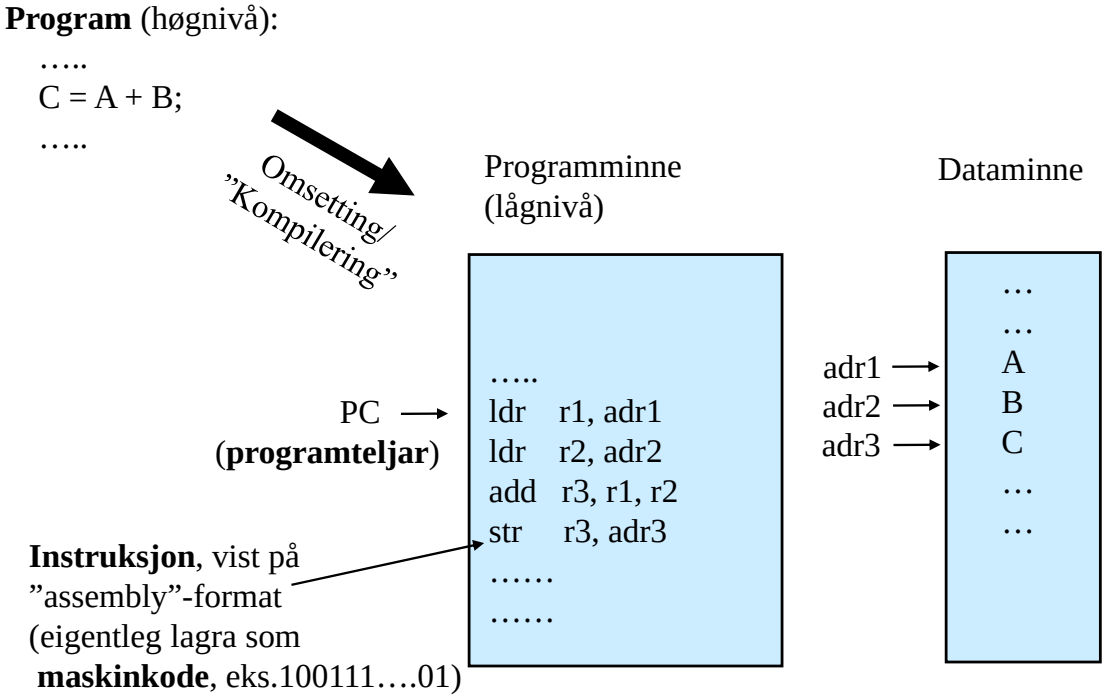
I tillegg kan ein som vist i figuren, gi mikroprosessoren signala **omstart** ("reset") og **avbrot** ("interrupt"). Førstnemnde signal vil føra til at μP-en mellom anna vil nullstilla programteljaren, dvs. hoppa til starten av programmet og køyra vidare derifrå. I alle datamaskinar finst det elektronikk som genererer omstartssignal f.eks. når ein slår på systemet. På ein PC er vanlegvis av/på-knappen kobla til "reset"-elektronikken. Ved å trykka på denne vil altså PC-en byrja frå start på ein kontrollert måte.

Avbrotssignalet vil føra til at μP-en vil avbryta køyringa og hoppa til eit spesielt program, avbrotsprogrammet ("interrupt handler" eller "interrupt service routine", ISR), og køyra dette. Etterpå vil μP-en gå tilbake til der han var før avbrotet og halda fram køyringa. Ved å bruka avbrot kan det for oss sjå ut som om μP-en køyrer to eller fleire uavhengige program samtidig. Dette blir utnytta av operativsystem som f.eks. Linux, Unix og Windows.

[6] Kjernen eller prosessoren i denne er altså ARM Cortex-M3.
[7] Sjå kap. 2 i databladet, [16].

2.2.2 Køyring av program

Måten mikroprosessoren arbeider på, kan illustrerast med programeksemplet vist i figur 2.4.



Figur 2.4: Programeksempel.

Dei fleste innebygde system blir programmerte vhja. høgnivåspråk som f.eks. C eller C++, og ein høgnivåinstruksjon kan vera som vist i figuren. For at μP-en skal forstå programmet, må det omsetjast/kompilerast til **maskinkode** som altså er kode på lågaste nivå. For å gjera slik kode lesbar for oss, vil kompilatoren mellom anna laga ei fil der programmet er i såkalla **assemblyformat**.
Figur 2.4 viser korleis assemblyutgåva av ein høgnivåinstruksjon kan sjå ut. Ein ser at denne nå er blitt til mange assemblyinstruksjonar som fortel μP-en i detalj korleis addisjonen skal utførast.

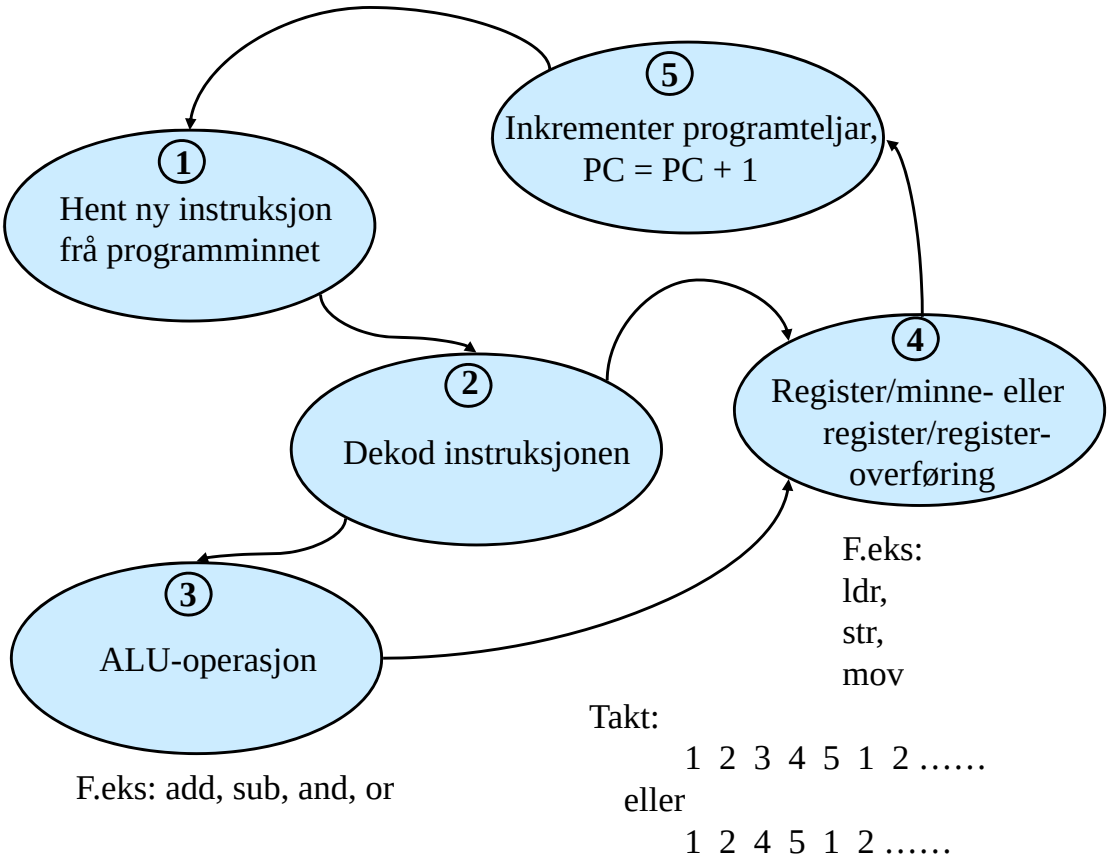
Først skal verdiane til dei 2 variablane hentast (*ldr*, "load (into) register") frå plassane sine i dataminnet og plasserast i kvar sitt register inne i μP-en, sjå figur 2.3. Så skal resultatet av addisjonen leggjast i eit tredje register før μP-en til slutt vhja. ein eigen instruksjon lagrar (*str*, "store register") ein kopi av resultatet i dataminnet.

Både programinstruksjonane og variablane ligg på sine tilordna adresser.
På same måte som
eit *brev* ligg i Geir Ryge sin **postkasse** i Rygjavegen 2, ligg
ein *verdi* i variabel A si **minnecelle** på adresse adr1 som vist i figur 2.4. Meir om addres-

sering kjem i kapittel 2.2.5.

2.2.3 Tilstandsmaskin

Køyringa av eit program, dvs. ei liste av instruksjonar, vil grovt sett gå føre seg som vist i figur 2.5.

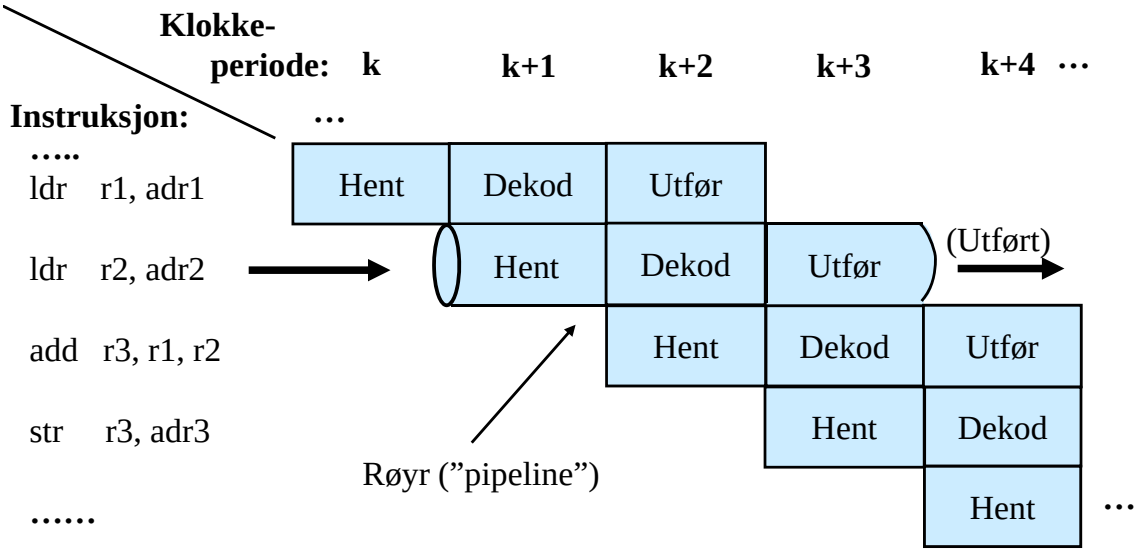


Figur 2.5: Mikroprosessoren sin tilstandsmaskin.

Køyringa blir styrt av ein såkalla **tilstandsmaskin** inne i μP-en som går med fast takt gitt av klokkefrekvensen som nemnt før. Dei nummererte boblene er ulike tilstandar som prosessoren er i ved handtering av instruksjonar. Tilstand 3 og 4 har med sjølve utføringa å gjera.
Avhengig av kva instruksjon som skal utførast, blir sekvensane som vist i figuren. Viss det f.eks. er ein overførings-instruksjon, går ein rett frå tilstand 2 til tilstand 4.

2.2.4 Parallellkøyring

I praksis går ikkje alle deloperasjonane i figur 2.5 sekvensielt, dvs. etter kvarandre, men parallelt. Prosessorane blir laga slik at dei ikkje treng å venta til ein instruksjon er utført, før dei hentar neste instruksjon, men gjer dette altså parallelt. Dette blir kalla parallellkøyring ("pipelining")[8].
Prosessoren ARM Cortex-M3 som er basis her, har eit **3-stegsrøyr**[9] ("pipeline") som vist i figur 2.6.



Figur 2.6: Tre-stegs parallellkøyring.

I si ferd gjennom **røyret**, går ein instruksjon som vist her gjennom 3 steg eller fasar. Instruksjonen blir først **henta**, på neste klokkesteg eller periode blir han **dekoda**, og på siste klokkesteget blir han **utført**.
Som figuren også viser, vil μP-en på eitt og same klokkesteg gjera tre ting i parallell, nemleg henta ein instruksjon, dekoda den førre instruksjonen og utføra instruksjonen før denne igjen.
Dette er ei vanleg oppdeling, men det finst mange mikroprosessorar som har eit røyr delt opp i fleire steg. Nokre har også bare to steg[10].

Parallellkøyring gir ei mykje meir effektiv programkøyring. Programeksemplet i figur 2.6 er henta frå figur 2.4.

[8] Sjå "en.wiki Instruction pipeline".
[9] Sjå kap. 3.2.1 hjå Yiu, [3].
[10] Eksempel her er AVR frå Atmel, sjå kapittel 1.2.2.4, og PIC frå Microchip.

Merk at ein har redusert effekt av slike røyr på stader i programmet der ein gjer hopp framover eller bakover. Me kan ta som eit eksempel ei enkel løkkje der ein dekrementerer variabelen A i kvar gjennomgang av løkkja og ikkje går vidare før A er blitt negativ. Høgnivåkoden kan sjå slik ut:

```
.....
A = 100;                // Startverdi = 100
while ( --A >= 0 ) {    // Dekrementer og test så A.
    ;
}
.....                   // Gå vidare.
```

Poenget med ei slik løkkje er ofte å realisera ein pause i programkøyringa og blir ofte kalla ei **venteløkkje**. Startverdien bestemmer pauselengda. Assemblykoden for løkkja kan bli slik:

```
   .....
   mov r1, #100    ; Går her ut frå at A er blitt plassert i register r1
L1 add r1, r1, #-1 ; Dekrementering: r1 = r1 - 1
   bge r1, L1      ; Test og hopp tilbake viss r1 >= 0 (''branch if greater
                                                          or equal'')
   neste1          ; Gå vidare med dei neste instruksjonane
   neste2
   .....
```

Når prosessoren utfører sjølve hoppinstruksjonen, vil han samtidig dekoda neste instruksjon, *neste1*, og henta instruksjonen etter det igjen, *neste2*.
Viss testen av register *r1* viser at han skal ut av løkkja, er det bare å køyra på slik som vist i figur 2.6. Viss han skal hoppa tilbake, må instruksjonane *neste1* og *neste2* **spylast ut** av røyra ("flush") og røyra fyllast på nytt på programadressa *L1* som ein hoppar til. Ein kan visa at det samla sett vil gå like mange klokkesyklar på hoppinga som det er steg i røyret.

Hoppinstruksjonen i eksemplet her vil altså i utgangspunktet ta tre klokkesyklar. Dei fleste mikroprosessorane i dag bruker likevel færre klokkesyklar enn dette på eit hopp. Når det gjeld mikroprosessoren vår, ARM Cortex-M3, så vil ein enkel test på laboratoriet[11] visa at eit hopp kan bli utført i løpet av bare to syklar.
Parallellkøyringa er altså meir avansert enn som vist i figur 2.6. Mikroprosessoren utfører i tillegg noko som blir kalla **spekulativ henting** ("speculative fetch"). For å få til dette, må han kunne både predikera kor hoppet går og om det skal hoppast eller ikkje. Dette heiter **hopp-prediksjon** ("branch prediction")[12].

[11] Dette ser ein på i laboratorieøving 2 i Datamaskinarkitektur.

[12] Viss ein guglar "arm cortex-m3 branch prediction" hamnar ein fort hjå ARM sitt informasjonssenter, [7], men det er lite dei vil dela med seg om dette.

Prediksjon av om ein skal hoppa eller ikkje, blir gjort ved å byggja opp enkel statistikk på kva som har skjedd i dei føregåande gjennomgangane av løkkja. Det spekulative ligg at prediksjonen kan vera feil, og at ein då har henta ein instruksjon det ikkje blir bruk for. Viss ein skal hoppa til *L1* i eksemplet over, vil ein pga. den spekulative hentinga kunne ha *add*-instruksjonen klar til dekoding når ein kjem til *L1*. Ein vil då spara ein klokkesyklus. Dette kan høyrast lite ut, men kan bidra til ei merkbar betring av ytinga til eit større program.

Andre prosessorar handterer hoppsituasjonen på andre måtar slik at ikkje begge instruksjonane *neste1* og *neste2* går i vasken, sjå f.eks. kapittel 3.7.4.2 om prosessoren MicroBlaze.

2.2.5 Litt meir om adressering

2.2.5.1 Minneorganisering

Som me såg i programeksemplet i figur 2.4, så ligg programkode og data lagra i minnet til datamaskinen. Minnet er organisert i **bytar**[13]. Ein byte inneheld ei informasjonsmengde på 8 **bit**[14] og er den minste informasjonsmengda som kan **adresserast**. *Kvar byte har altså ei unik adresse.*

Eit minne er generelt sett saman av ulike delar som kan vera av ulik størrelse og type. Ein del kan f.eks. vera realisert i Flash-teknologi og lagra programkode, mens ein annan del kan vera eit SRAM-minne som lagrar data, dvs. variabelverdiar mm.
Eit minne består av **minneceller** der ei celle lagrar **eitt bit**. Ein hovudforskjell mellom ulike **minneteknologiar** er korleis minnecellene er bygd opp, og dette avgjer eigenskapane til minnet. Meir om minneteknologi kjem i kapittel 3.10.9.

[13] Sjå også "no.wiki Byte".
[14] Meir om det binære talsystemet finst i kapittel A.1.

2.2.5.2 Adresseringskapasitet og adresseområde

Det er ei grense for kor stort minne ein mikroprosessor kan handtera.
Kor mange bytar ein mikroprosessor kan adressera, er gitt av talet på adresselinjer.

Eksempel 2.1.

Mikroprosessoren vår, ARM Cortex-M3, er som nemnt ein 32-bitsprosessor. Både data-[15] og adressebussen har her ei breidde på $n = 32$ bit.
Adresseringskapasiteten er då gitt av[16]:

$$\begin{aligned} 2^n = 2^{32} &= 2^2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10} = 4 \cdot 1\ K \cdot 1\ K \cdot 1\ KB \\ &= 4 \cdot 1\ K \cdot 1\ MB = 4\ GB = 4.294.967.296\ B \approx 4.3 \cdot 10^9\ bytar \end{aligned} \tag{2.1}$$

der *KB*, *MB* og *GB* står for Kilo-, Mega- og Gigabyte.

Adresseområdet er samlinga av alle dei ulike adresseverdiane som kan brukast ved adressering.
Det er vanleg å framstilla dette vhja. det **heksadesimale** talsystemet. Ein skal då setja adressebitane i **4-grupper** der kvar gruppe blir representert vhja. eitt av dei heksadesimale sifra **0 - F**, jfr. kapittel A.1.5.

Den **minste** adressa er då gitt av at alle adressebitane har verdien 0.
Den **maksimale** adressa er gitt av at alle bitane er 1.
For ein adressebuss på 32 bit vil den maksimale adresseverdien på heksadesimal form[17] bli `0xFFFFFFFF` $= 2^{32} - 1$.

Adresseområdet for ein mikroprosessor med adressebus på 32 bit blir då `0x00000000 - FFFFFFFF`.

[15] Det er som nemnt før, databussbreidda som avgjer om ein prosessor blir kalla ein 4-, 8-, 16-, 32- eller 64-bitsprosessor.
[16] Sjå kapittel A.1.3.
[17] Notasjonen "0x" blir brukt i språket C for å visa at ein talverdi er på heksadesimal form.

2.2.5.3 Utrekning av minnestørrelse

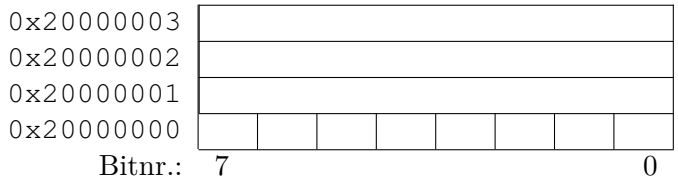
Størrelsen til ei minneblokk er gitt av følgjande uttrykk:

	Maksimumsadresse
−	Minimumsadresse
+	1
=	Størrelse

Eit enkelt eksempel viser kvifor ein må leggja til talet 1.

Eksempel 2.2.

Ei lita minneblokk i ein datamaskin med adressebusbreidde på 32 bitar er vist i tabell 2.1.



Tabell 2.1: Ei lita minneblokk.

Kvar **byte** i eit minne har som kjent si eiga adresse. Bytane er delte opp i 8 **bitar** som har kvart sitt nummer. I tabellen over er bitnummereringa vist for den byten som ligg på lågaste adresse.
Den minst signifikante biten, "Least Significant bit" (**LSb**), står heilt til høgre. For dei aller fleste mikroprosessorar er det bit 0 som er LSb[18].

Minimumsadressa til ei minneblokk blir også kalla **baseadressa** eller **start-adresssa** til blokka.

Reknestykkjet for minnestørrelse blir her:

	Maksimumsadresse	0x20000003
−	Baseadresse	0x20000000
+	1	1
=		4

Me ser lett av minneblokkoppsettet i tabell 2.1 at reknestykkjet er rett. Blokka inneheld 4 bytar.

I neste kapittel blir dette brukt vidare.

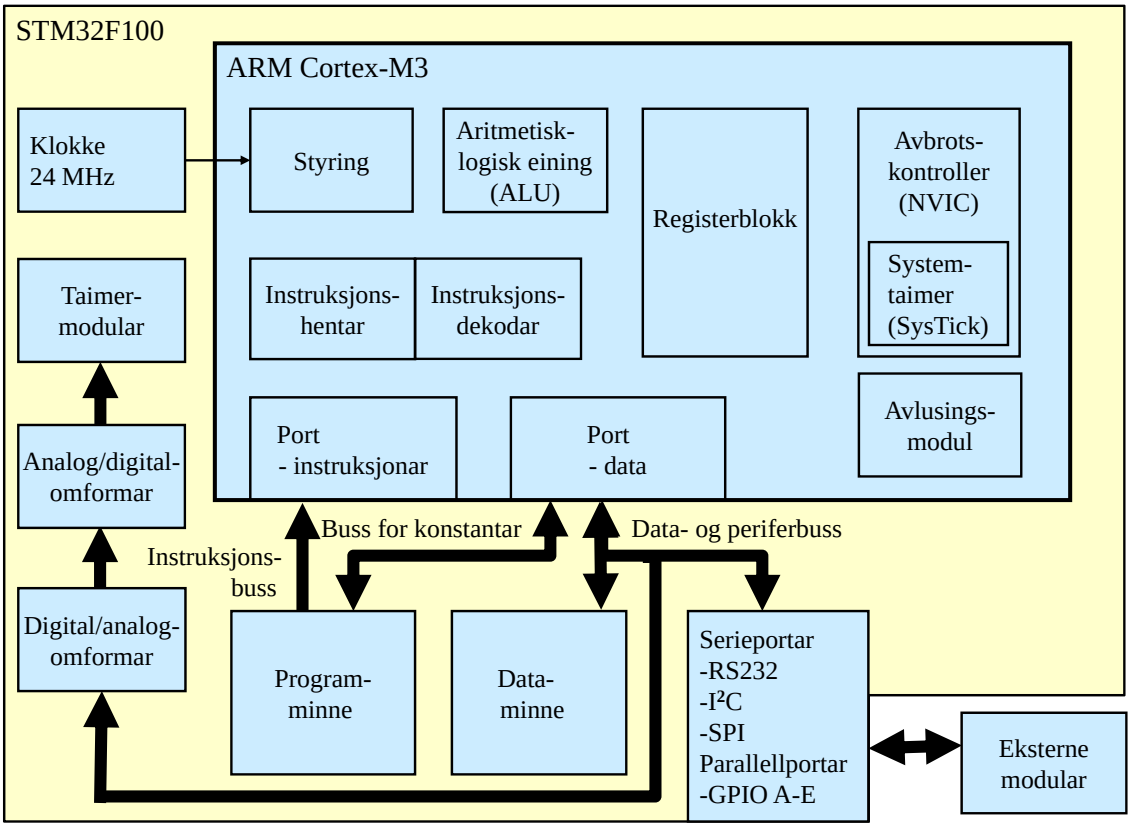
[18] I nokre få mikroprosessorar som f.eks. mjukprosessoren MicroBlaze i kapittel 3.7.4, er bit 0 den mest signifikante biten, "Most Significant bit". Slike system blir kalla **bit-reverserte**.

2.3 Litt om den ARM-baserte mikrokontrolleren vår

I figur 1.9 på side 12 blei det vist eit bilete av mikrokontrollerkortet vårt, *STM32VLDiscovery*, samt eit grovt blokkskjema av kortet.
Det vil nå bli gitt ei innføring i mikrokontrolleren STM32F100RB frå STMicroelectronics[19]. Dette er altså hovudmikrokontrolleren på kretskortet vårt.
Brown gir også ei innføring i oppbygging og eigenskapar til denne mikrokontrolleren samt kjernen ARM Cortex-M3 i kap. 2 i [2]. Det kan med fordel lesast i tillegg.

2.3.1 Oppbygging

Figur 2.7 viser viktige blokker i sjølve mikrokontrolleren der også hovudblokkene i sjølve mikroprosessoren eller kjernen er med.



Figur 2.7: Forenkla blokkdiagram av mikrokontrolleren STM32F100.

Som ein ser, så har mikroprosessoren Harvard-struktur i og med at det er separate bussar, dvs. vegsystem, for **instruksjonar** og **data**[20].

[19] Heimesida er *www.st.com*.

[20] Dette er ein veldig forenkla illustrasjon av bussystemet. Du kan finna mange fleire detaljar i figur 6 i brukarmanualen på mikrokontrollerkortet, [15], samt i figur 1 i referansemanualen på mikrokontrolleren,

Fleire detaljar om STM32F100RB er gitt under:

- Kjerne: ARM Cortex-M3
- Klokkefrekvens: 24 MHz.
- Programminne (Flash): 128 KByte.
- Dataminne (SRAM): 8 KByte.
- Kontrollermodul for handtering av avbrot: "Nested Vectorized Interrupt Controller", NVIC.
- Avlusingsmodul ("Debug system"): Modul for å sjekka og finna feil i program. Eksempel på funksjonar som er implementerte i modulen, er avbrekk ("breakpoint") og overvaking ("watchpoint") mm.
- Taimerar for tidsmåling og styring (f.eks. basert på pulsbreiddemodulasjon, "Pulse Width Modulation", PWM)[21].
- Taimerar for overvaking og programstyring (f.eks. SysTick-taimeren inne i kjernen).
- Serieportar: Modular for SPI-, I2C-, RS232-kommunikasjon mm.
- Parallellportar[22]: GPIO-modulane ("General Purpose In Out") A-E, til saman 80 parallelle digitale linjer. Kvar av desse kan programmerast til å vera inn- eller utgangslinje.
- Analog/Digital-omformar: 12 bit, 16 kanalar.
- Digital/Analog-omformar: 12 bit, 2 kanalar.

Dei av blokkene som ligg utanfor kjernen og som ikkje er minnemodular, kallar ein ofte for **perifermodular**, då dei omgir eller ligg i "periferien" rundt kjernen.

---

[17].

[21] Meir om dette kjem i kapittel 3.10.6.

[22] Kvar **parallellport** har her 16 digitale linjer som kan styrast parallelt. Ein kan f.eks. leggja ut alle bitverdiane til ein 16-bitsvariabel samtidig.
Ein **serieport** har **ei** digital datalinje ut og **ei** linje inn. Skal ein overføra den same variabelverdien her, må dei 16 bitverdiane sendast etter kvarandre.

2.3.2 Minnekart

2.3.2.1 Innleiing

Alle minneblokkene i ein mikrokontroller har sitt **fysiske**[23]**adresseområde**. Dette blir fastlagt under konstruksjon av mikrokontrolleren. Eit **minnekart** ("memory map") for ein mikrokontroller viser dei ulike adresseområda.
Minnekartet er sentralt når ein skal laga program[24] for mikrokontrolleren. Viss ein f.eks. skal overføra data via ein serieport, må ein bruka adressa til serieporten i programmet for å kunne kommunisera med denne.

2.3.2.2 Minnekart for mikrokontrollerar baserte på ARM Cortex-M3

Konstruktørane av mikroprosessoren ARM Cortex-M3 har laga eit overordna minnekart for alle mikrokontrollerar med denne kjernen. Minnekartet er vist i tabell 2.2.
Ved å bruka framgangsmåten i eksempel 2.2 skal me nå rekna ut størrelsen på heile dette adresseområdet.

Eksempel 2.3.

Total adresseringskapasitet for ARM Cortex-M3 er gitt av:

	Maksimumsadresse	`0xFFFFFFFF`
−	Baseadresse	`0x00000000`
+	`1`	`1`
=	Størrelse	`0x100000000`

Dette kan også skrivast slik:

$\texttt{0x100000000} = 1 \cdot 16^8 = 1 \cdot (2^4)^8 = 2^{32} = 4$ GB

Dette er altså som funne i eksempel 2.1 i kapittel 2.2.5.2.

[23]Ein brukar ofte ordet **fysisk** for å visa at dette er adresseverdiar med direkte kobling til fysiske modular. Det motsette er **virtuell** adressering, som blir brukt av eit operativsystem ved handtering av store minnesystem. Her opererer ein med eit såkalla virtuelt minne, og dette er større enn det fysiske adresseområdet. Meir om dette kan finnast hjå "en.wiki memory management (operating systems)".

[24]I alle fall gjeld dette ofte for utviklarar av program for innebygde system og for utviklarar av drivarprogram og grensesnitt for ulike perifermodular i ein datamaskin. Slike program og grensesnitt blir pakka inn og ofte kalla applikasjonsgrensesnitt ("Application Program Interface", API), og brukt av andre programmerarar som då ikkje treng vita noko om f.eks. fysiske adresser.

Adresse	Minneblokk for
0xFFFFFFFF 0xFFFFFFFE 0xE0000000	Modular i kjernen (NVIC, SysTick-taimer,..).
0xDFFFFFFF 0xA0000000	Eksterne komponentar.
0x9FFFFFFF 0x60000000	Ekstern RAM.
0x5FFFFFFF 0x40000000	Perifermodular i mikrokontrolleren.
0x3FFFFFFF 0x20000000	Data (SRAM).
0x1FFFFFFF 0x00000000	Programkode, konstantar og vektortabell. (Flash) ← 1 byte →

Tabell 2.2: Minnekart ("memory map") for ARM Cortex-M3. (Ref.: Kap. 6.2 i [3].)

2.3.2.3 Minnekart for mikrokontrolleren STM32F100RB

Minnekartet ("memory map") for mikrokontrolleren vår, STM32F100RB, er vist i tabell 2.3. Alle blokkene er som i minnekartet for kjernen ARM Cortex-M3, men kor stor del av blokkene som er realiserte og innhaldet i desse varierer mellom ulike mikrokontrollerar.

Adresse	Minneblokk for
0xFFFFFFFF .... 0xE00FFFFF 0xE0000000	Kjernemodular (NVIC, SysTick-taimer,..).
.... 0x400233FF 0x40000000	Perifermodular i mikrokontrolleren.
.... 0x20001FFF 0x20000000	Data (SRAM)
.... 0x0801FFFF 0x08000000	Programkode, konstantar og vektortabell. (Flash)
.... 0x00000000	← 1 byte →

Tabell 2.3: Minnekart ("memory map") for STM32F100RB. (Ref.: Kap. 4 i [16].)

2.3.3 Litt om oppbygginga til ein perifermodul

2.3.3.1 Eit utdrag av perifermodulblokka i minnekartet

I kap. 4 i databladet for mikrokontrolleren vår, [16], finn me ei detaljert oversikt over **baseadressene** til dei ulike **perifermodulane** i mikrokontrolleren.

Me skal nå sjå litt nærare på dei såkalla parallellportane eller GPIO-modulane til mikrokontrolleren. Desse blei omtalte i kapittel 2.3.1 og er viste figur 2.7 der.
I tabell 2.4 er det vist eit utdrag av minnekartet for perifermodulblokka. Utdraget viser adresseområdet til dei fem GPIO-modulane.

....	
0x40011800	Port E
0x40011400	Port D
0x40011000	Port C
0x40010C00	Port B
0x40010800	Port A
....	

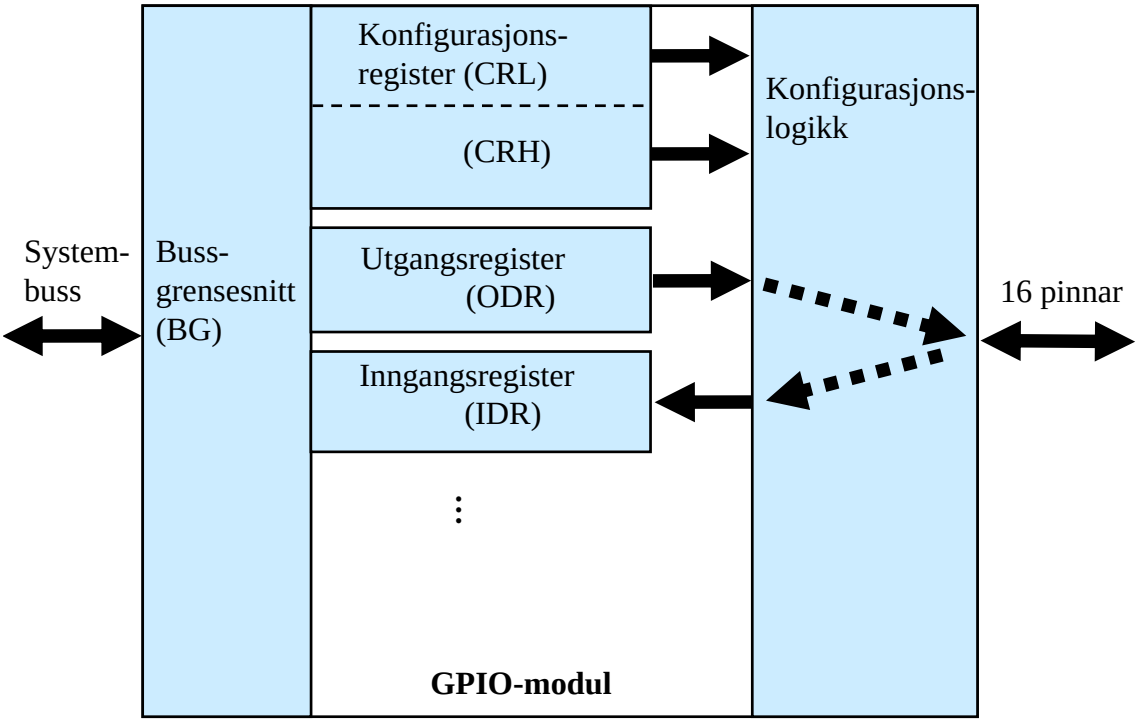
Tabell 2.4: Utdrag av minneblokka avsett til perifermodular.
(Ref.: Kap. 4 i [16].)

Ein ser at baseadressa eller startadressa til f.eks. GPIO-modul C (*GPIOC*) er `0x40011000`.
Me skal nå sjå nærare kva som ligg inne i sjølve parallellportblokkene i tabellen.

2.3.3.2 Registerstruktur for ein perifermodul

Viss ein kikkar nøye i figur 1.9 på side 12, så ser ein at det langs kantane til plattformkortet vårt står mange pinnar. Pinnane har namn etter kva GPIO-modul dei høyrer til i mikrokontrolleren STM32F100RB og kva bitnummer dei er kobla til i eit **dataregister** inne i modulen.
Pinnen med namn *PA7* er f.eks. kobla til *bit nr. 7* i dataregisteret til *GPIOA*.

Viss ein skal skriva/setja ut eit høgt eller lågt logisk nivå på ein pinne eller lesa kva logisk nivå ein pinne har, må ein først **konfigurera** porten. Kvar GPIO-modul i mikrokontrolleren vår har to konfigurasjonsregister. Her set ein mellom anna dei ulike pinnane til å vera **utgang** eller **inngang**. Dette er illustrert med stipla piler i figur 2.8.



Figur 2.8: Blokkskjema av ein GPIO-modul.

Skriving til utgangspinnar kan gjerast ved å leggja rett innhald i eit dataregister kobla til utgangane i GPIO-modulen. Dette blir kalla eit **utgangsregister** ("Output Data Register", ODR).
Viss ein pinne er sett opp som inngang, kan ein lesa nivået til pinnen frå eit dataregister kobla til inngangane. Dette blir kalla eit **inngangsregister** ("Input Data Register", IDR).

Konfigurasjonsregistera, inngangsdata- og utgangsdataregisteret er såkalla **funksjonsregister** ("Special Function Register", **SFR**) i GPIO-modulen.
På tilsvarande måte som ein mikroprosessor har eit sett av register, har alle perifermodular eit sett av funksjonsregister. Eit register er altså eit lite minne.

Ein GPIO-modul i mikrokontrolleren vår har fleire funksjonsregister enn dei som er viste i figur 2.8. Alle funksjonsregistra med adresser, namn og litt om innhaldet er viste i tabell 2.5.

	Bare bit 16 i bruk	Port Configuration Lock Register
	For pinnenr. 15 - 8	
+ 0x18	7 - 0	(GPIOn_LCKR)
	Ikkje i bruk	Port Bit Reset Register
	For resetting av pinne 15 - 8	
+ 0x14	7 - 0	(GPIOn_BRR)
	For resetting av pinne 15 - 8	
	7 - 0	Port Bit
	For setting av pinne 15 - 8	Set/Reset Register
+ 0x10	7 - 0	(GPIOn_BSSR)
	Ikkje i bruk.	Port Output
	For pinnenr. 15 - 8	Data Register
+ 0x0C	7 - 0	(GPIOn_ODR)
	Ikkje i bruk.	Port Input
	For pinnenr. 15 - 8	Data Register
+ 0x08	7 - 0	(GPIOn_IDR)
	For pinnenr. 15 - 14	
	13 - 12	Port Configuration
	11 - 10	Register High
+ 0x04	9 - 8	(GPIOn_CRH)
	For pinnenr. 7 - 6	
	5 - 4	Port Configuration
	3 - 2	Register Low
Baseadresse	1 - 0	(GPIOn_CRL)

Tabell 2.5: Alle funksjonsregistra (SFR) som høyrer til ein parallellport GPIOn, n = A,B,C,D eller E. (Ref.: Kap. 7.2 i [17].)

Funksjonsregistra til ein GPIO-modul i mikrokontrolleren vår er 32 bit breie. Eitt register vil då som vist i tabellen, bestå av 4 bytar og leggja beslag på 4 adresser i minnekartet. Sjølve GPIO-porten er 16 bit brei der 0 - 7 er dei låge bitane (L) og 8 - 15 er dei høge bitane (H).

Litt meir om registra:

CRL og CRH: Konfigurasjonsregister

Det er 4 bit for kvar pinne, og bitkombinasjonen bestemmer pinnefunksjonen. F.eks. vil koden $0011 = 0x3$ setja ein pinne som utgang og koden $0100 = 0x4$ setja ein pinne

som inngang. Ein pinne kan også brukast til ein såkalla **alternativ funksjon**[25] f.eks. som sende- eller mottakslinje for ein serieport.

IDR: Inngangsregister
Her kan ein lesa nivået på inngangane.

ODR: Utgangsregister
Her kan setja nivået på utgangane.

BSRR og BRR: Bitsettings- og resettingsregister
Alternative register for å setja og resetja nivået på enkeltpinnar. Bruk av desse kan gi raskare kodekøyring.

LCKR: Konfigurasjonslås
Ved å setja bit her vil porten ikkje kunne rekonfigurerast før heile mikrokontrolleren blir resett.

Meir om desse registra kan finnast i kapittel 7.2 i referansemanualen for mikrokontrolleren, [17]. I tillegg kan ein sjølvsagt i referansemanualen også finna funksjonsregistra til alle dei andre perifermodulane i mikrokontrolleren.

2.4 Programmering av ein ARM-basert mikrokontroller

Ein skal her sjå litt på programmering av mikrokontrolleren vår. Det desidert mest vanlege språket for programmering av innebygde system, som i svært stor grad er basert på mikrokontrollerar, er høgnivåspråket C[26]. Eit fortrinn er at dette språket gir ein effektiv maskinkode. Koden blir kompakt og vil utførast/eksekverast raskt. Eit anna hovudfortrinn er at ein **lett kan programmera overføringar til og frå fysiske adresser**. Det å kunne skriva til og lesa frå fysiske adresser i eit minne eller funksjonsregister i perifermodular er sentralt ved programmering av innebygde system.
Konstruktørane av ARM-prosessorane har utvikla standarden **CMSIS**, "Cortex Microcontroller Software Interface Standard", som gjer at ein mellom anna kan programmera slike overføringar på ein lesbar og effektiv måte.

Her skal ein sjå på nokre viktige datatypar i C som byggjer opp under fysisk adressering generelt og den nemnte standarden spesielt. Så vil CMSIS bli presentert, og etterpå vil eit eksempel på overføring til ein GPIO-modul bli vist.

For å forstå kva som eigentleg skjer ved slike overføringar, er det viktig å læra seg litt om **instruksjonssettet** til mikroprosessoren ARM Cortex-M3. Som me såg i figur 2.4 på side 29, vil høgnivåkoden vår bli kompilert til maskinkodeinstruksjonar. Ein skal her sjå

[25] I kapittel 3.10.5 går ein meir detaljert inn på korleis GPIO-porten er bygd opp, og ser då også på desse alternative funksjonane.

[26] Ikkje alle er samde i at C er eit høgnivåspråk, sjå "en.wiki High Level Language". Språket ligg i alle fall på eit høgt (abstraksjons-)nivå i forhold til maskin- eller assemblyspråk.

på nokre sentrale instruksjonar for å forstå korleis ARM-prosessoren arbeider.
Dette er også viktig når ein skal forstå korleis ein prosessor er oppbygd, noko som blir vist meir detaljert i kapittel 3.10.2.

Heilt til slutt i dette kapitlet vil me sjå litt på korleis den overordna oppbygginga av programvaren, dvs. programstrukturen, kan vera i eit system.
Før ein går laus på alt dette, kan det vera greitt å ha i bakhand nokre reglar for sikker programmering, sjå "en.wiki The Power of 10: Rules for Developing Safety-Critical Code".

2.4.1 Generelt om språket C

Ei oppsummering av viktige sider ved språket C er for eksempel å finna i Appendix B hjå Toulson og Wilmshurst, [6]. Her skal me først gi nokre eksempel på bruk av **bitvise operatorar**. Dette er sentralt ved utvikling av kode for innebygde system.

Så skal det fokuserast på ein viktig datatype som lar oss realisera mellom anna fysisk adressering, nemleg **peikaren**.

Ein annan viktig datatype er **strukturdatatypen**, *struct*, som lar oss deklarera samansette variablar. Me skal sjå på registersett for ein perifermodul deklarert som *struct*. Denne datatypen blir presentert saman med programvarestandarden CMSIS.

Aller først vil ein sjå litt på **heiltalsdatatypar**.

2.4.1.1 Litt om datatypar for representasjon av heiltal

I utviklingsverktøyet som blir brukt i emnet Datamaskinarkitektur, CoIDE[27], har ein fylgjande deklarasjonar for heiltal:

int8_t, *uint8_t* - heiltal på 8 bit, også kalla *byte*.
int16_t, *uint16_t* - heiltal på 16 bit, også kalla *halvord* ("half word").
int32_t, *uint32_t* - heiltal på 32 bit, også kalla *ord* ("word").

Når det gjeld heiltalsdeklarasjonen *..N_t*, så spesifiserer indeksen *t* at denne datatypen skal bruka eksakt N bitar[28]. Det er viktig ved overføring til eit register eller ein minnelokasjon at alle bitane får ein definert verdi. Ved stabling av data i eit minne er det også viktig å vite kor mange bytar ulike data tar. Deklarasjonen *_t* kom med C-standarden C99[29].

Når det gjeld ***i****ntN_t* kontra ***ui****ntN_t* ("unsigned integer"), så spesifiserer sistnemnte at alle

[27] Litt meir om dette verktøyet kjem i kapittel 2.4.2.
[28] Størrelsen til dei opprinnelege deklarasjonane *int* og *short int* mfl. var prosessoravhengige. Ein *int* kunne få tildelt 16 bit i ein prosessor og 32 bit i ein annan.
[29] Det står meir om alt dette hjå "en.wiki C data types".

bitane skal brukast som databitar. Talområdet blir då positivt eller meir presist **unipolart**. Ved å bruka datatypen *intN_t* kan ein representera både positive og negative talverdiar, dvs. eit **bipolart** talområde. Det står meir om representasjon av tal i vedlegg A.

2.4.1.2 Bitvise operasjonar i C

I språket C finn ein seks bitvise operatorar[30], sjå tabell 2.6.

Operator	Merknad	Norsk namn	Engelsk namn
~	Bitvis invertering	IKKJE	NOT
|	Bitvis eller-operasjon	ELLER	OR
&	Bitvis og-operasjon	OG	AND
^	Bitvis eksklusiv eller-operasjon	XELLER	XOR
>>	Flytt eit bitmønster til høgre	Skift høgre	Shift right
<<	Flytt eit bitmønster til venstre	Skift venstre	Shift left

Tabell 2.6: Dei bitvise operatorane i C.
(A og B er 1bits-variablar.)

Funksjonstabellen for dei fire øverste bitvise operatorane er vist i tabell 2.7.

A	B	~A	$A\&B$	$A\|B$	$A\hat{}B$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Tabell 2.7: Funksjonstabell for nokre bitvise operatorar.

Desse operasjonane er også sentrale ved konstruksjon av maskinvare. Det står meir om dette i digitalteknikkdelen av skrivet og spesielt i kapitla 3.5.1.2 - 3.5.1.3.

Ved bruk av ein *eller-*, *og-* eller *eksklusiv eller*-operator i eit C-program, kan ein endra enkeltbitar i eit bitmønster. Dei andre bitane vil då ikkje endra verdi.
Ved bitvis invertering av eit bitmønster vil kvart bit i mønsteret bli invertert, dvs. få motsett verdi[31] av det som bitane hadde før.
Og under skiftoperasjonar flyttar ein eit heilt bitmønster til venstre eller høgre.

Eksempel på alle operasjonane blir viste i det følgjande, og me startar med bitvis invertering.

[30] Sjå "en.wiki Bitwise operations in C".
[31] Denne operasjonen blir derfor også kalla einar-komplement, sjå kapittel A.2.2.2.

Bitvis invertering
Ein bitvis IKKJE-operator blir altså brukt til å invertera bitane i eit bitmønster.

Eksempel 2.4. *Invertering av eit bitmønster*

Variabelen *moenster* er **deklarert** som *uint16_t* og har **verdien** `0x8041`.

Ein C-instruksjon som inverterer variabelen bitvis er gitt av:

```
moenster = ~moenster;
```

For å sjå korleis denne instruksjonen virkar, kan det vera lurt å teikna opp variabelen.

Bit nr.	15 12	11 8	7 4	3 0
moenster	1000	0000	0100	0001
~ gir	0111	1111	1011	1110

Som ein ser, er dei ulike bytane skilde med dobbeltstrek, mens dei ulike **firargruppene** er skilde med enkeltstrek.

Tabellen viser at resultatet av C-instruksjonen blir `0x7FBE` på hexadesimal form.

Bitvis ELLER-operasjon
Ein bitvis ELLER-operator kan brukast til å setja bit, dvs. setja bit til verdien 1.

Eksempel 2.5. *Setjing av bit*

Variabelen *testbyte* er deklarert som *uint16_t*.
Ein skal her laga ein C-instruksjon som set bit 0, 6 og 15 **utan å påverka dei andre bitane**.

Først kan det igjen vera lurt å teikna opp variabelen.

Bit nr.	15 12	11 8	7 4	3 0
testbyte	xxxx	xxxx	xxxx	xxxx
ELLER med	1000	0000	0100	0001
gir	1xxx	xxxx	x1xx	xxx1

Dei ulike bitane i variabelen har eit eller anna innhald markert med "x".
For å utføra denne bitvise operasjonen, må ein altså bruka følgjande **bitmønster** på heksadesimal form: `0x8041`.
(Dette kjem greitt fram av firargruppene i tabellen.)

Ein ser av tabellen at dei spesifiserte bitane blir sette til 1 og at dei andre er urørte.

C-instruksjonen for denne bitsetjinga blir: `testbyte = testbyte | 0x8041;`

(Denne kan også skrivast slik: `testbyte | = 0x8041;`)

Bitvis OG-operasjon
Ein bitvis OG-operator kan brukast til å nullstilla/sletta bitar ("clear"), dvs. setja bitar til verdien 0.

Eksempel 2.6. *Sletting av bit*

Variabelen *data* er deklarert som *uint16_t*.
Ein skal nå laga ein C-instruksjon som slettar bit 0, 6 og 15 i variabelen **utan å påverka dei andre bitane**.

Først kan det som før vera lurt å teikna opp variabelen.

Bit nr.	15 12	11 8	7 4	3 0
data	xxxx	xxxx	xxxx	xxxx
OG med	0111	1111	1011	1110
gir	0xxx	xxxx	x0xx	xxx0

Dei ulike bitane i variabelen har som i førre eksempel eit eller anna innhald markert med "x".
For å utføra denne operasjonen, må ein altså bruka følgjande **bitmønster** på heksadesimal form: `0x7FBE`.

Ein ser av tabellen at dei spesifiserte bitane blir nullstilte og at dei andre er urørte.

C-instruksjonen blir då: `data = data & 0x7FBE;`

(Denne kan også skrivast slik: `data & = 0x7FBE;`)

Merk: Viss me bruker resultatet i eksempel 2.4, kan C-instruksjonen også skrivast slik:

```
data = data & (~0x8041);
```

Dette er ein meir **lesbar** variant då det er lettare her å sjå direkte av hex-koden kva bit som det blir operert på. I kapittel 2.4.2 om programvarestandarden CMSIS skal me sjå korleis dette kan utnyttast til å gjera koden ennå meir lesbar.

Eksempel 2.7. *Testing av bit*

Ein bitvis OG-operator kan også brukast til å sila ut informasjon slik at ein kan utføra bittesting.

Utgangspunktet i dette eksemplet er at innhaldet i eitt av funksjonsregistra til ein serieport er lese inn og ligg i variabelen *status*. Variabelen *status* er deklarert som *uint32_t*.
Innhaldet i dette funksjonsregisteret seier noko om tilstanden til serieporten. Mellom anna blir *bit 5* sett til 1 når serieporten har mottatt nye data.

Ein skal i dette eksemplet laga C-kode som testar om bit 5 er sett, og i så fall utfører **metoden**[32] *les_data()*.
Testinga skal gjerast utan å påverka bitane i variabelen.

Først må ein **sila ut** den informasjonen som er relevant her, nemleg verdien til *bit 5*. Å sila ut noko er det same som å **maskera** vekk all den andre informasjonen. Dette kan ein gjera med ein OG-operasjon.

Tabellen under viser variabelen *status* etter leseoperasjonen. Dei ulike bitane i denne har eit eller anna innhald markert med "x".

Bit nr.	31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
status	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
OG med	0000	0000	0000	0000	0000	0000	0010	0000
gir	0000	0000	0000	0000	0000	0000	00x0	0000

For å maskera vekk all irrelevant informasjon, må ein altså bruka følgjande **bitmønster** på heksadesimal form: `0x00000020 = 0x20`

Ein ser at etter maskering er det bare verdien til bit 5 som slepp igjennom nålauget og som kan vera 1 eller 0.

C-koden blir då:

```
if ( status & 0x20 ) {    // Viss (status & 0x20) > 0,
                          // må det vera bit 5 som er 1.
                          // Då er det nye data som skal lesast.
     les_data();
}
```

Bitvis XELLER-operasjon
Ein bitvis eksklusiv ELLER-operator, dvs. XELLER-operator kan brukast til å **snu** bit ("toggle"), dvs. **invertera** bitverdiar. Viss ein bit er 1, blir han 0 og omvendt.

Eksempel 2.8. *Snuing av bit*

Variabelen *blink_var* er deklarert som *uint8_t*.
Ein skal laga ein C-instruksjon som snur bit 0 og 6 **utan å påverka dei andre bitane**.

Tabellen under viser variabelen.

Bit nr.	7	6	5	4	3	2	1	0
blink_var	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
XELLER med	0	1	0	0	0	0	0	1
gir	0/1	1/0	0/1	0/1	0/1	0/1	0/1	1/0

[32] **Metode** er ofte brukt som namn i objektorienterte språk, mens **funksjon** og **subrutine** er vanlege ord i språket C. Ein funksjon i C returnerer eit resultat, mens ei subrutine ikkje returnerer noko. Her vil me bruka ordet metode.

Dei ulike bitane i denne har eit eller anna innhald markert med "`0/1`".
Dei ulike bitane er her skilde med enkeltstrek, mens **firargruppene** er skilde med dobbeltstrek.
Ein ser her, jfr. også funksjonstabellen for ein XELLER-operasjon, at når ein gjer ein XELLER med *1*, så blir bitverdien snudd/invertert.
For å få til den spesifiserte snu-operasjonen, må ein altså bruka følgjande **bitmønster** på heksadesimal form: `0x41`.

C-instruksjonen blir då: `blink_var = blink_var ^ 0x41;`

(Denne kan også skrivast slik: `blink_var ^ = 0x41;`)

Viss ein i eit program skal realisera f.eks. **blinking** av ein diode kobla til ein GPIO-pinne på ein mikrokontroller, vil XELLER-operasjonen vera effektiv.

Bitvise skiftoperasjonar
Ein bitvis skiftoperator kan brukast til å **flytta** bitmønster til venstre eller høgre i eit register eller ein variabel.

Eksempel 2.9. *Nokre høgreskift*

Følgjande variabeldeklarasjonar er gitt:

```
uint16_t  data_2B = 0x0201; // Binært: 0010 0001
uint8_t   data_1B;

int8_t    tal_1 = 0x90; // Binært: 1001 0000
uint8_t   tal_2 = 0x90; // Binært: 1001 0000

uint8_t   tal_3 = 128;  // Binært: 1000 0000
uint8_t   tal_4;
```

Følgjande C-instruksjonar blir så køyrde:

```
data_1B = data_2B >> 8;  // Legg MSByte i data_2B inn i data_1B
                         // Resultat: data_1B = 0x02

tal_1   = tal_1 >> 4;    // Aritmetisk skift.
                         // Resultat på binær form:   1111 1001
tal_2   = tal_2 >> 4;    // Logisk skift.
                         // Resultat på binær form:   0000 1001
```

```
tal_4   = tal_3 >> 1;    // Resultat på binær form:   0100 0000
                         // Resultat på desimal form: 64 = 128/2
tal_4   = tal_3 >> 3;    // Resultat på binær form:   0001 0000
                         // Resultat på desimal form: 16 = 128/8
tal_4   = tal_3 >> 7;    // Resultat på binær form:   0000 0001
                         // Resultat på desimal form: 1 = 128/128
```

Den første instruksjonen kan vera nyttig viss ein skal dela opp ein variabelverdi i enkelt-bytar for overføring mm.

Elles bør ein merka seg forskjellen på **aritmetisk** og **logisk** skiftoperasjon. Viss ein skiftar ein *int* til høgre, vil forteiknet bli kopiert ved skifting. MSbit som er forteiknsbiten, vil derfor ha same verdi etter skiftoperasjonen. Dette er eit såkalla aritmetisk skift.
Ein *uint* er per definisjon utan forteikn. Skifting til høgre av slike datatypar vil føra til at dei øvre bitane blir fylte med nullar. Dette er logisk skifting.

Dei siste tre instruksjonane viser eksempel på at **skifting** av ein variabelverdi N bitposisjonar **til høgre** er det same som å **dividera** variabelverdien med 2^N. Divisjonar er tunge operasjonar for prosessoren mens skifting er veldig lett. Ein bør derfor utnytta dette ved programmering[33] .

Eksempel 2.10. *Nokre venstreskift*

Følgjande variabeldeklarasjonar er gitt:

```
uint16_t  data_2B;
uint8_t   data_LSB = 0x01;
uint8_t   data_MSB = 0x02;

int8_t    tal_1 = 0x82; // Binært: 1000 0010
uint8_t   tal_2 = 0x82; // Binært: 1000 0010

uint8_t   tal_3 = 3;  // Binært: 0000 0011
uint8_t   tal_4;
```

Følgjande C-instruksjonar blir så køyrde:

```
data_2B  = data_MSB << 8; // Legg data_MSB i øvre del av data_2B
                          // Resultat: data_2B = 0x0200
data_2B += data_LSB;      // Legg til data_LSB i nedre del av data_2B
                          // Resultat: data_2B = 0x0201
```

[33]Nokre kompilatorar vil oppdaga og effektivisera noko av dette automatisk, men truleg langt frå alt.

```
tal_1    = tal_1 << 4;    // Aritmetisk skift dvs. skift av eit tal
                          // med forteikn.
                          // Resultat på binær form:  0010 0000
tal_2    = tal_2 << 4;    // Logisk skift.
                          // Resultat på binær form:  0010 0000

tal_4    = tal_3 << 1;    // Resultat på binær form:   0000 0110
                          // Resultat på desimal form:   6 = 3 * 2
tal_4    = tal_3 << 3;    // Resultat på binær form:   0001 1000
                          // Resultat på desimal form:  24 = 3 * 8
tal_4    = tal_3 << 6;    // Resultat på binær form:   1100 0000
                          // Resultat på desimal form: 192 = 3 * 64
```

Den første instruksjonen kan vera nyttig viss ein f.eks. les av ein variabelverdi byte for byte og skal kombinera dette til ein fleir-byteverdi etterpå.

Elles bør ein merka seg at **aritmetisk** og **logisk** skiftoperasjon mot **venstre** gir same resultat. Forteiknet blir ikkje tatt vare på ved skifting mot venstre. Mikroprosessoren vår, ARM Cortex-M3, har ingen maskininstruksjon for aritmetisk skift til venstre, bare logisk skift.
Ein *uint* er som nemnt før, per definisjon utan forteikn. Skifting til venstre av slike datatypar vil gi ei rein skifting av bitverdiar ut til venstre. Dette er logisk skifting.

Dei siste tre instruksjonane viser eksempel på at **skifting** av ein variabelverdi N bitposisjonar **til venstre** er det same som å **multiplisera** variabelverdien med 2^N.
Skifting er algoritmemessig ein mykje raskare operasjon enn multiplikasjon, viss ein tenkjer tradisjonelt. Som me hugsar frå grunnskulen, er ein multiplikasjon av to fleirsifra tal bygd opp av ei rekkje addisjons- og flytt-, dvs. skiftoperasjonar, og i dei første mikroprosessorane blei multiplikasjon utført i programvare etter dette prinsippet.
Seinare kom mikroprosessorar med eigne multiplikasjonsinstruksjonar som blei utført i maskinvare og vhja. meir effektive algoritmar[34].
Prosessorar som ARM Cortex-M3 har ein innebygd maskinvaremodul som utfører heiltalsmultiplikasjonar i løpet av ein klokkesyklus!
Ein bør likevel ha denne 2^N-samanhengen i mente ved programmering. Særleg gjeld dette for å spara divisjonar, som er mykje tyngre operasjonar enn multiplikasjon.

[34]Sjå f.eks. "en.wiki Binary multiplier", som viser meir om både historie og metodar.

2.4.1.3 Datatypen peikar

Ein peikar ("pointer") er ein datatype i C som **peikar på** ei adresse i eit minnekart. Når ein deklarerer peikaren, viser deklarasjonen også kva denne peikar på, dvs. kva **innhald** den bestemte adressa har. Dette blir vist med eit eksempel.

Eksempel 2.11. *Bruk av peikar*

```
uint8_t tabell[5];  // Tabell med 5 element av typen Byte (8 bit).

uint8_t *p;         // ''Innhaldet i'' det p peikar på,
                    //  er ein uint8_t, dvs. ein byte.
uint8_t *pr;        // Endå ein peikar.

 p   = tabell;      // p peikar nå på tabellstarten.
*p++ = 'A';         // ASCII-koden for teiknet A, sjå vedlegg A.3.3,
                    // blir lagt inn på førsteplass i tabellen, tabell[0].
                    // p++ er ''post-inkrementering'' av p.
*p++ = 0xA;
*p   =   10;        // Dvs. same innhald som i cella på adressa under.

 pr  = p + 2;
*pr  = '1';         // ASCII-koden til teiknet 1.

 pr  = &tabell[3];  // & - ''adressa til''.
*pr  = 0x7B;
```

Innhaldet i minnet der tabellen er lagra, ser slik ut etter at koden over er utført:

Adresse	Minneinnhald
....	
+4	0x31
	0x7B
	0x0A
+1	0x0A
tabell	0x41
....	

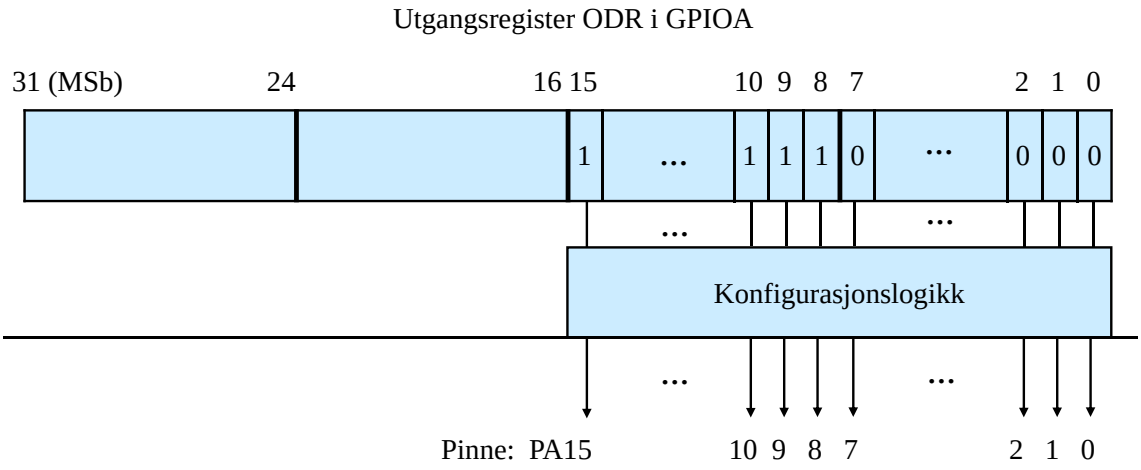
2.4.1.4 Eksempel på fysisk adressering av GPIO-modul vhja. peikar

Me tenkjer oss at alle dei 15 pinnane til perifermodulen GPIO_A er sett opp til å vera utgangar. Me ynskjer å setja dei 8 minst signifikante pinnane, dvs. PA0-7, til lågt nivå og resten til høgt nivå.
Pinne 0 - 7 skal altså setjast til 0 og pinne 8 - 15 skal setjast til 1.

Utgangsdataregisteret **ODR** til GPIOA er 32 bit breitt, sjå tabell 2.5 på side 43. Registeret skal då ha fylgjande innhald:

```
ODR = 0x0000FF00;    // Bit 16 - 31 er ikkje i bruk og
                     // blir her sett til 0.
```

Dette er illustrert i figur 2.9.



Figur 2.9: Oppsett av GPIOA

For å kunne leggja inn denne verdien i det fysiske registeret, dvs. ODR, i perifermodulen GPIO_A, må me finna den **fysiske adressa** til registeret. Dette kan me gjera i følgjande steg:

1. Ut frå minnekartet i tabell 2.3 på side 40 finn me ut at mikrokontrolleren si **perifermodulblokk** har startadresse eller **baseadresse** `0x40000000`.
2. Ut frå minnekartet i tabell 2.4, som er eit utdrag frå kap. 4 i [16], finn ein at **baseadressa** til GPIO_A er `0x40010800`.
3. Vhja. tabell 2.5 på side 43, som er henta frå kap. 7.2 i [17], ser me at adressa til registeret ODR er **baseadressa + 0x0C**.

Adressa til 32bits-registeret ODR i GPIOA er altså `0x4001080C`.

Me skal nå steg for steg visa korleis ein i C kan laga programkode for å leggja ein verdi inn på ei slik adresse og tar utgangspunkt i ein peikar som me kallar p:

1. Deklarasjon av peikaren p:

```
uint32_t  *p;     // p peikar på ei generell minneblokk
                  //   på 32 bit (4 bytar), i dette tilfellet
                  //   eit register på 32 bit.
```

2. Initialisering av peikaren:

```
p = (uint32_t  *)0x4001080C; // p peikar nå på registeret ODR
                             //   som ligg på denne adressa.
```

   Denne C-instruksjonen kan tolkast slik[35]:

   "Talet 0x4001080C skal oppfattast[36]som ein peikar, og innhaldet i det denne peikar på, skal vera ein heiltalsverdi på 32 bit utan forteikn ("unsigned")."

3. Skriving av verdi til registeret:

```
*p = 0x0000FF00;    // Verdien vil nå hamna i registeret ODR
                    // i GPIO-modul A.
```

4. Ved å setja uttrykket for p i punkt 2 inn i utrykket i punkt 3, får ein følgjande:

```
*(uint32_t  *)0x4001080C = 0x0000FF00;  // Kompakt instruksjon
                                           // for skriving.
```

5. Den kompakte skrivemåten i punkt 4 blir mykje brukt. Men for å gjera programkoden meir lesbar, bruker ein det såkalla preprosessor-direktivet `#define` til å definera ein såkalla **makro**[37], her illustrert med eksemplet vårt:

```
#define ODR_A  *(uint32_t  *)0x4001080C  // Innhaldet i
               // registeret ODR i GPIO-modul A definert
               // som makro med namn ODR_A.
```

[35]Jfr. deklarasjonane i eksempel 2.11.

[36]Ein døyper eller "støyper" om talet frå å vera ein vanleg heiltalsverdi og til å vera ein annan datatype. Dette blir kalla ein **CAST**-operasjon, av "cast" - støypa.

[37]Første fasen under bygging ("build") av programkode er preprosessering, sjå f.eks. "en.wiki C preprocessor". Då blir mellom anna den definerte makrokoden lagt inn alle plassar i programmet der makronamnet opptrer.

Viss me i programmet vårt nå legg inn følgjande C-instruksjon,

```
ODR_A = 0x0000FF00;
```

vil det fysiske utgangsregisteret til GPIOA, ODR, og pinnane som dette registeret er knytt til, få dei rette verdiane som vist i figur 2.9.

Eksempel 2.12. *Fysisk adressering av ein innport*

I dette eksemplet skal det visast korleis ein kan lesa nivå, dvs. verdiar, til GPIO-pinnar. Her skal ein sjå på GPIO-modul B.

Ein vil først definera *IDR_B* til å vera innhaldet i inngangsdataregisteret for GPIO-modul B.

Det skal så lagast ein C-instruksjon som les nivået på alle pinnane inn i variabelen *inndata*. (Variabelen er av type *uint32_t*.)
Ein går her ut frå at GPIO-modulen er sett opp som ein rein inngangsmodul.

Me ser frå minnekartet i tabell 2.4 på side 41, at baseadressa til GPIO_B er `0x40010C00`.
Vhja. tabell 2.5 på side 43, ser me at adressa til registeret IDR er
baseadressa + 0x08.

Definisjonen blir derfor som følgjer:

```
#define IDR_B  *(uint32_t  *)0x40010C08
```

C-instruksjonen blir då slik:

```
inndata = IDR_B;
```

2.4.2 Litt om programvarestandarden CMSIS

2.4.2.1 Kvifor standardisering?

Opplegget som blei vist i forrige kapittel, realiserer overføringar til eller frå fysiske adresser på ein enkel måte. I eit større system vil likevel denne direkte bruken av adresseverdiar og registerverdiar bli altfor uryddig og lite lesbar. Det vil vera tungvint å verifisera koden og feilsøkja i denne utan veldig gode kommentarar.
I tillegg vil alle desse viktige talverdiane rundt i koden gjera det vanskeleg å flytta koden over på ein annan plattform, sjølv til ein annan ARM-basert mikrokontroller. Koden er altså lite **portabel**. Minnekartet kan her vera annleis som gjer at svært mange verdiar må endrast rundt i koden.

Å oppnå godt lesbar, verifiserbar og testbar kode, samt å mogleggjera lett og effektiv gjenbruk i nye prosjekt på same og andre plattformer er nokre gode argument for standardisering. Konstruktørane av ARM-prosessorane har som nemnt utvikla standarden **CMSIS**, "Cortex Micro-controller Software Interface Standard", som skal gjera at ein oppnår dette.

2.4.2.2 Prosjekthierarki og viktige filer

Når ein opprettar eit prosjekt i eit utviklingsverktøy som f.eks. **CoIDE** frå **CooCox**[38], vil ein automatisk få eit prosjektoppsett som vist til venstre i figur 2.10.

Alt utanom metodemappa *Metodar* er ein del av CMSIS-standarden[39]. Prosjekthierarkiet inneheld som ein ser, filer med følgjander halar:

".h" - hjelpefiler ("header") som inneheld definisjonar samt deklarasjonar av *globale variablar*[40] mm.
".c" - kjeldefiler ("source") som inneheld deklarasjonar av *metodar*[41] og lokale variablar.

Prosjekthierarkiet inneheld eigne grupper av **filer** for

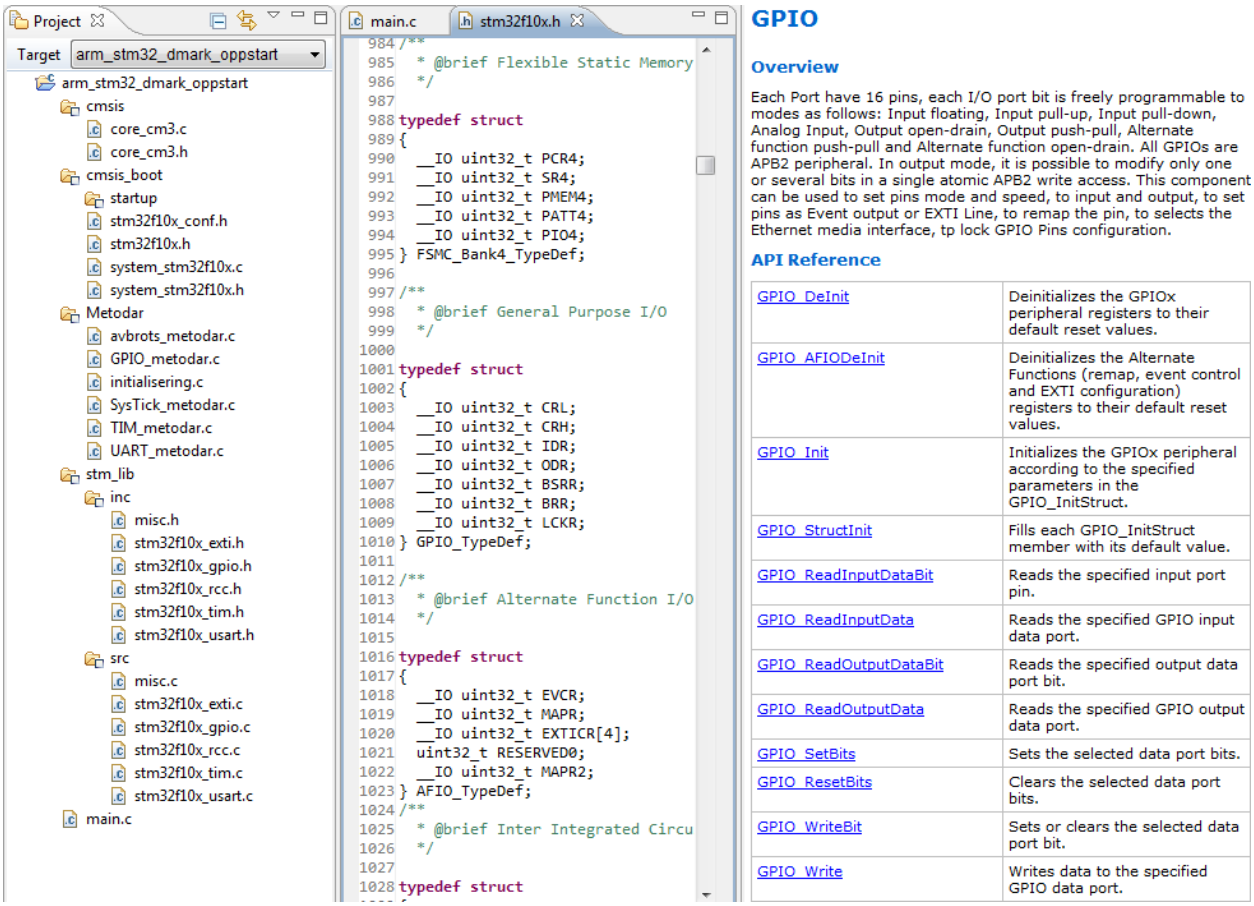
- **kjernen**, dvs. *ARM Cortex-M3*, sjå mappa *cmsis* i figuren.
- **mikrokontrolleren** *STM32F100RB* generelt, sjå mappa *cmsis_boot*.
- kvar **perifermodul** spesielt, sjå mappa *stm_lib*.

[38] CoIDE er eit fritt tilgjengeleg utviklingsverktøy for mikrokontrollerar basert på mikroprosessoren ARM Cortex, sjå *www.coocox.org*. Nokre av laboratorieøvingane i emnet Datamaskinarkitektur ved UiS blir utførte vhja. dette verktøyet.

[39] Hovudprogrammet *main()* kjem også opp automatisk, men innhaldet er det sjølvsagt programmeraren som bestemmer.

[40] Det kjem litt meir om variablar i kapittel 2.4.3.

[41] Ein har som nemnt før valt å henta namn frå objektorientert programmering sjølv om ein altså her fokuserer på rein C. Dette gjer ein fordi metodeomgrepet er kjent frå før. Andre namn på dette er *funksjon*, som returnerer eit resultat og *subrutine* som ikkje returnerer noko.



Figur 2.10: CMSIS-basert prosjekthierarki mm. i CoIDE.

Desse filene utgjer det ein kan kalla **plattformstøtte** (”Board Support Package”, BSP). Det vil bli vist ein del sentrale filer og utdrag frå desse under.
Ein kan også lesa meir om CMSIS i kapittel 2 hjå Yiu, [3], samt i kapittel 2 hjå Brown, [2].

2.4.2.3 Sentrale filer for kjernen ARM Cortex-M3

Sentrale CMSIS-filer for kjernen ("core") er her:

Fila *core_cm3.h* :
a) Her ligg **definisjonar av fysiske adresser**.
Me skal her ta den viktige taimeren *SysTick*[42] som eksempel, sjå figur 2.11. Eit utdrag av fila som viser definisjonane av denne, er vist under:

```
/* Memory mapping of Cortex-M3 Hardware */
#define SCS_BASE  (0xE000E000)  /* System Control Space Base Address */
....
#define SysTick_BASE (SCS_BASE + 0x0010)      /* SysTick Base Address*/
....
#define SysTick   ((SysTick_Type *) SysTick_BASE) /* SysTick config struct*/
```

SysTick er altså definert som ein **peikar** til ein struktur av typen *SysTick_Type* som er plassert i minnet på baseadresse *SysTick_Base*. Deklarasjonen av strukturen *SysTick_Type* blir vist i b).

b) I fila *core_cm3.h* ligg også deklarasjon av variablar. Ein viktig datatype er som nemnt *struct*[43] som mellom anna blir brukt til å deklarera **funksjonsregistersett** til modular i kjernen ARM Cortex-M3.
Med taimeren *SysTick* som eksempel, viser følgjande utdrag av fila korleis denne strukturdeklarasjonen kan gjerast:

```
typedef struct
{
  __IO uint32_t CTRL;  /* Offset: 0x00  SysTick Control and Status Register */
  __IO uint32_t LOAD;  /* Offset: 0x04  SysTick Reload Value Register       */
  __IO uint32_t VAL;   /* Offset: 0x08  SysTick Current Value Register      */
  __I  uint32_t CALIB; /* Offset: 0x0C  SysTick Calibration Register        */
} SysTick_Type;
```

Det registeret som ligg på lågaste adresse, dvs. på baseadressa, skal her deklarerast øvst[44] i strukturen.

[42] SysTick-taimeren er altså ein del av kjernen ARM Cortex-M3.
[43] Datatypen *struct* lar oss deklarera samansette variablar. Sjå "en.wiki struct (C programming language)" som har fleire gode eksempel.
[44] **Merk:** Dette er ein detalj som kan variera mellom ulike prosessorplattformar.

Dei såkalla **identifikatorane** heilt til venstre i strukturdeklarasjonen er gitt av følgjande definisjon i C:

```
#define __IO  volatile
#define __I   volatile const
```

Med ein `volatile`-deklarasjon viser me kompilatoren at variabelen/registeret kan bli endra av andre mekanismar enn sjølve programkoden, dvs. av maskinvaren[45].
Poenget er at me då hindrar kompilatoren frå å optimalisera vekk kode som **tilsynelatande** ikkje er godt for noko.

F.eks. kan ein ha kode som med jamne mellomrom les SysTick-registeret *CTRL* utan at det er kode nokon stad i programmet som skriv til dette registeret. For kompilatoren vil dette sjå meiningslaust ut, men når me veit at maskinvaren kan endra bitverdiane i slike register, er det viktig å få sagt til kompilatoren at slik kode ikkje skal fjernast.

Definisjonane `__IO` og `__I` fortel kva aksessar som er lovlege mot dei deklarerte variablane/registra. Ein kan altså i programkoden både skriva til og lesa frå SysTick-registeret *CTRL*, mens registeret *CALIB* bare er meint for lesing[46]. Dette registeret blir då i programkoden handtert som ein konstant (`const`) sidan ein ikkje skriv til dette.

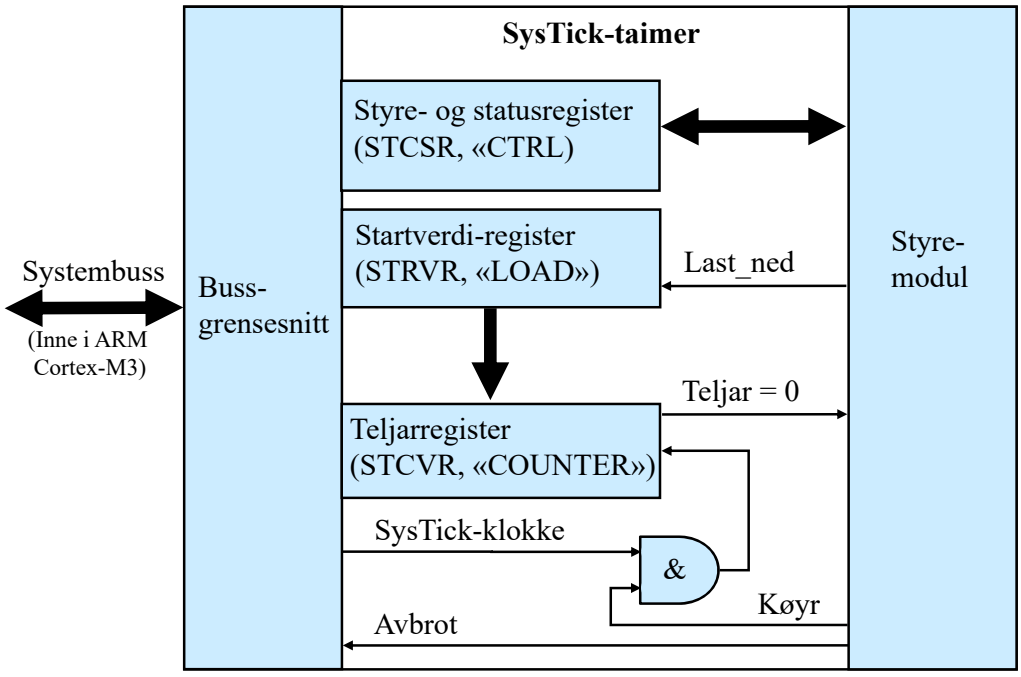
Tilsvarande har ein definisjonen
```
#define __O   volatile
```

Denne blir brukt til deklarasjon av variablar/register som bare er meint for skriving.

[45] Ordet "volatile" betyr som kjent flyktig, kortvarig eller ustadig på norsk.

[46] Ein C-instruksjon i koden som skriv til *CALIB* vil då gi feilmelding ved kompilering/bygging.
CALIB er eit såkalla kalibreringsregister og bør i følgje ARM Ltd. sine retningslinjer innehalda talet på teljingar som skal til for å gi eit taimerintervall på eksakt 10 msek. Meir om virkemåten til denne taimeren kjem ganske snart i dette underkapitlet.
Skriving til eit slikt register gir inga endring av innhaldet.

Taimeren har altså fire funksjonsregister (SFR). Figur 2.11 viser eit blokkskjema av denne der dei tre for oss viktigaste registra er med.



Figur 2.11: Blokkskjema av taimeren SysTick.

Me skal nå som eit eksempel laga ein instruksjon i C som legg verdien 24000 inn i *LOAD*-registeret, dvs. startverdiregisteret. Dette kan gjerast enkelt og greitt slik:

```
SysTick->LOAD = 24000;
```[47]

Ved å dra nytte av standarden CMSIS er ein altså ikkje avhengig av å vita fysiske adresser når ein lagar vanleg kode for ein applikasjon.

[47] *SysTick* er altså ein peikar og pila viser kva element ein peikar på. Alternativ formulering er *(*SysTick).LOAD = 24000;*. Dette tilsvarer spesifikasjon av eit **objektelement** i språket Java.

For å starta taimeren på rett måte, må ein mellom anna skriva eit passande bitmønster til *CTRL*-registeret, dvs. styre- og statusregisteret. Dette kan sjå slik ut:

```
SysTick->CTRL =
      (SysTick_CTRL_ENABLE | SysTick_CTRL_TICKINT | SysTick_CTRL_CLKSOURCE);
            // Start teljaren, gi avbrot og bruk intern klokke (24 MHz).
```

Desse symbolske bitmønstra er også definerte i fila *core_cm3.h*.

Dei to C-instruksjonane over er del av ein konfigurasjonssekvens[48]. Etter at denne er utført, vil taimeren telja ned frå startverdien med prosessorklokkefrekvensen 24 MHz. Når teljaren er null, får styremodulen eit signal om det.
Styremodulen sender då dette vidare som eit **avbrotssignal** ("interrupt") viss me har spesifisert dette i koden som vist over. Så vil han automatisk lasta ned startverdien på nytt i teljaren og igjen telja nedover.
Med startverdien 24000 vil han gi avbrotssignal til prosessoren med intervall på 1 millisekund. Dette kan brukast av eit operativsystem til f.eks. å hoppa mellom parallelle trådar. Meir om dette står i kapittel 2.4.5.8 samt i kapittel 11 hjå Brown, [2].

Kva som skjer når prosessoren får eit avbrotssignal, blir tatt opp i kapittel 2.4.4.11. Her blir også SysTick-taimeren brukt som eksempel.

Fila *core_cm3.c* :
Denne inneheld deklarasjon av viktige metodar koda i assemblyspråk. Desse blir brukt av overordna metodar som er deklarerte i CMSIS-filene for mikrokontrolleren. Metodane i *core_cm3.c* er vanlegvis ikkje aktuelle ved vanleg applikasjonsprogrammering.

[48] Heile denne sekvensen for å setja opp taimeren er vist i kapittel 3.10.6.

2.4.2.4 Sentrale filer for mikrokontrolleren *STM32F100RB*

Sentrale CMSIS-filer er her:

Fila *stm32f10x.h* :
a) Her ligg **definisjonar av fysiske adresser**.
Ein skal her som i kapittel 2.4.1.4 ta GPIO-modul A som eksempel, og tar med eit utdrag av fila som viser korleis definisjonane er gjort:

```
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
#define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
```

APB2 er eitt av to bussystem som knyttar perifermodulane saman med kjernen ARM Cortex-M3, sjå kapittel 2.1 i referansemanualen til mikrokontrolleren vår, [17].

b) I fila *stm32f10x.h* ligg også deklarasjon av **funksjonsregistersett** til perifermodulane i mikrokontrolleren.
Eksemplet under viser korleis denne deklarasjonen er gjort for GPIO-modulane. Dette er også vist i midtfeltet i figur 2.10 på side 58.

```
typedef struct
{
  __IO uint32_t CRL;  //Ligg på lågaste adressa, dvs. baseadressa.
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

Dette kan ein samanlikna med minnekartoppsettet i tabell 2.5 på side 43.
Viss ein nå som i eksemplet på fysisk adressering i kapittel 2.4.1.4, vil leggja verdien `0x0000FF00` i utgangsregisteret, blir C-koden slik:

```
GPIOA->ODR = 0x000FF00;
```

Fila *stm32f10x.c* :
Denne fila inneheld mellom anna deklarasjon av metoden *system_init()* som blir kalla av **oppstartsmetoden** *startup_stm32f10x_md_vl.c*[49].

[49]Metoden ligg under mappa *cmsis_boot -> startup* i prosjektoppsettet. (Ordet "boot" betyr å ta på seg støvlane, dvs. starta opp.)

Etter oppstart av mikrokontrolleren blir denne metoden starta automatisk. Oppstartsmetoden vil køyra initialisering før han kallar opp hovudprogrammet *main()*.

2.4.2.5 Sentrale filer for perifermodulane i mikrokontrolleren

Ein skal igjen bruka GPIO-modulane som eksempel. Sentrale CMSIS-filer for denne perifermodulen er:

Fila *stm32f10x_gpio.h* :
Her ligg mellom anna **definisjonar av standard bitmønster** som kan vera nyttige ved programmering. Eit utdrag er vist under.

```
#define GPIO_Pin_0                 ((uint16_t)0x0001)  /*!< Pin 0 selected */
#define GPIO_Pin_1                 ((uint16_t)0x0002)  /*!< Pin 1 selected */
#define GPIO_Pin_2                 ((uint16_t)0x0004)  /*!< Pin 2 selected */
#define GPIO_Pin_3                 ((uint16_t)0x0008)  /*!< Pin 3 selected */
#define GPIO_Pin_4                 ((uint16_t)0x0010)  /*!< Pin 4 selected */
#define GPIO_Pin_5                 ((uint16_t)0x0020)  /*!< Pin 5 selected */
#define GPIO_Pin_6                 ((uint16_t)0x0040)  /*!< Pin 6 selected */
#define GPIO_Pin_7                 ((uint16_t)0x0080)  /*!< Pin 7 selected */
#define GPIO_Pin_8                 ((uint16_t)0x0100)  /*!< Pin 8 selected */
#define GPIO_Pin_9                 ((uint16_t)0x0200)  /*!< Pin 9 selected */
#define GPIO_Pin_10                ((uint16_t)0x0400)  /*!< Pin 10 selected */
#define GPIO_Pin_11                ((uint16_t)0x0800)  /*!< Pin 11 selected */
#define GPIO_Pin_12                ((uint16_t)0x1000)  /*!< Pin 12 selected */
#define GPIO_Pin_13                ((uint16_t)0x2000)  /*!< Pin 13 selected */
#define GPIO_Pin_14                ((uint16_t)0x4000)  /*!< Pin 14 selected */
#define GPIO_Pin_15                ((uint16_t)0x8000)  /*!< Pin 15 selected */
#define GPIO_Pin_All               ((uint16_t)0xFFFF)  /*!< All pins selected */
```

At dei heksadesimale verdiane over er rette, kan ein greitt verifisera ved å teikna opp registeret som vist i eksempel 2.4 på side 47.

Eksempel 2.13. *Bruk av bitmønsterdefinisjonar til å styra GPIO-pinnar*

a) Me skal her først **setja** pinnane PA0, 6 og 15 **høge** og dei andre PA-ane låge på mikrokontrollerkortet vårt. For å få til det, må me setja dei rette bitane i utgangsregisteret til GPIOA. Me skal altså setja dei same bitane som i eksempel 2.5 på side 47.
Me går ut utfrå at heile GPIOA er sett opp som ein utport.
Styring av pinnane kan då gjerast slik:

```
GPIOA->ODR = (GPIO_Pin_0 | GPIO_Pin_6 | GPIO_Pin_15);
```

Dette gir ein mykje meir lesbar kode enn C-instruksjonen i eksempel 2.5 der ein bare viser ein hex-kode for bitsetjinga.

b) Nå skal i tillegg pinnane PA7 og 8 setjast høge **utan å røra** dei andre pinnane. Det er ofte at **ulike metodar** i eit system styrer kvar sine pinnar i ein GPIO-modul, og då må dei ikkje øydeleggja for kvarandre. Løysinga blir her følgjande C-instruksjon:

```
GPIOA->ODR = GPIOA->ODR | (GPIO_Pin_7 | GPIO_Pin_8);
```

Som instruksjonen viser, må nå prosessoren først lesa den nåverande ODR-verdien. Han vil så setja bit 7 og 8 og så skriva den nye ODR-verdien ut til GPIOA. Denne C-instruksjonen vil derfor ta lengre tid å utføra enn direkte-skrivinga i pkt. 1.

c) Til slutt her skal nå pinnane PA0, 6 og 15 **setjast låge** igjen utan å røra dei andre pinnane. Basert på resultatet i eksempel 2.6 på side 48 kan dette gjerast slik for å få mest mogleg lesbar kode:

```
GPIOA->ODR = GPIOA->ODR & ( ~(GPIO_Pin_0 | GPIO_Pin_6 | GPIO_Pin_15));
```

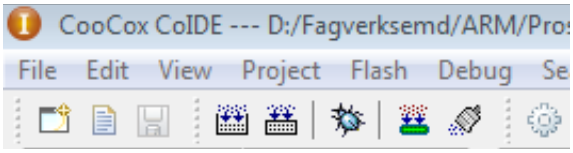
Lesaren kan på eiga hand visa at bitmønsteret som ein OG-ar med her, er det same som i eksempel 2.6.

Fila *stm32f10x_gpio.c* :
Her ligg deklarasjon av alle dei ferdige drivarmetodane for perifermodulen. Namnet på dei fleste av desse er viste til høgre i figur 2.10.

2.4.3 Frå kjeldekode til køyrbart program

2.4.3.1 Litt om verktøyinnstillingar

Som vist i kapittel 2.4.2, så bruker ein ved UiS utviklingsverktøyet CoIDE. Dette kan ein ta i bruk utan å endra på standardinnstillingane til verktøyet.
Ein skal likevel vera klar over at ein viktig ting som **optimalisering** er sett til nivå null. Dette gir lite effektiv kode. For å endra optimaliseringa, kan ein høgreklikka på prosjektnamnet i prosjektvinduet som vist til venstre i figur 2.10 på side 58 og så velja *Configuration*. Alternativt kan ein trykkja på konfigureringsknappen vist lengst til høgre i figur 2.12. Eit passande optimaliseringsval kan vera nivå 2.



Figur 2.12: Viktige knappar i CooCox.

2.4.3.2 Bygging av kjeldekoden

Eit trykk på bygg-symbolet (*Build*), som er nr. 4 frå venstre i figur 2.12, vil føra til følgjande:

1. Dei inkluderte *.h*-filene blir lagde inn i kjeldefilene (*.c*) som så blir kompilerte, dvs. gjort om til kvar sine objektfiler (*.o*). **Merk** at objektfiler er **relokerbare**, dvs. at det i objektkoden ikkje er nytta absolutte hopp- eller variabeladresser. Det er **lenkaren**, sjå kapittel 2.4.3.4, som gir koden desse absolutte adressane, dvs. fastset kor kode og data skal vera i minnet til mikrokontrollen.

2. Objektkoden, som kan vera fordelt på fleire filer, blir lenka saman med eventuelle ferdige biblioteksmodular, f.eks. drivarar for perifermodulane[50]. I tillegg får kode, variablar og stakk[51] sine absolutte plasseringar i minneområdet.
   Lenkaren utfører dette ut frå konfigurasjonsinnstillingane[52] og produserer så den køyrbare **absoluttfila** *executable.elf*.
   Denne fila inneheld maskinkode, som er det språket mikroprosessoren forstår og kan eksekvera.

3. I nokre tilfelle har programmeraren behov for å kikka på maskinkoden som er laga. Under bygging blir det difor utført ei såkalla **disassemblering** av maskinkoden. Mens maskinkoden er ei lang rekkje av instruksjonar gitt som heksadesimale tal,

[50] Sjå f.eks. filene under mappa *stm_lib* i prosjektvinduet i figur 2.10.
[51] Eit minne for midlertidig lagring av f.eks. lokale variablar. Meir om stakk kjem i kapittel 2.4.4.9.
[52] Desse kan ein finna ved å trykkja på konfigureringsknappen i figur 2.12 og sjå under *LINK*.

er assembly-kode den same rekkja av instruksjonar i (menneskjeleg) lesbart format. Meir om assemblyspråket eller det såkalla **instruksjonssettet** for ARM Cortex-M3 kjem i kapittel 2.4.4.

Etter bygging kan fila *executable.elf* lastast ned i programminnet i mikrokontrolleren ved å trykkja på nedlastingsknappen, som er nr. 3 frå høgre i figur 2.12.

2.4.3.3 Litt om variablar i C

Ein kan dela opp variabler i to typar, automatiske og statiske.

Automatiske variablar

Ein **automatisk** variabel er ein **lokal** variabel. Dei automatiske variablane blir lagde i nokre av dei faste prosessorregistra, evt. på stakken[53], og eksisterar berre ei tid.
Dersom ein metode har ein lokal variabel og kallar seg sjølv, dvs. er **rekursiv**, vil det ei tid finnast like mange lokale variable med same namn, men gjerne med ulik verdi, som talet på gonger metoden har blitt kalla. Då automatiske variable kjem og går med metodekall, held dei ikkje på verdien sin mellom metodekall. Dette gjer ein må gi dei verdi for kvar gong metoden blir kalla, ellers vil innhaldet av variabelen innehalda ein tilfeldig verdi.

Eksempel 2.14.

```
int32_t metode_1(...) {
  uint16_t i;          // Lokal variabel.

  ......
  return .. ;
}
```

Statiske variablar

Ein **global** variabel vil vera **statisk**. Statiske variablar blir lagde på kvar sin faste plass i minnet, og ligg der heile tida. Ein lokal variabel kan også deklarerast til å vera statisk. Denne vil då halda på verdien sin sjølv om metoden som eig han, har returnert.
Statiske variablar må ikkje forvekslast med konstantar, då konstantar ikkje kan endrast av metodar.

[53] Sjå fotnote til punkt 2 på førre side.

Eksempel 2.15.

```
uint32_t verdi;                 // Globale
uint8_t teikn  = 'A';           // variablar.

void main(void) {

   ......
}
```

eller

```
void metode_2(...) {
  static int x;  // Lokal, men statisk variabel.

  ......
}
```

Initialiserte variablar

Initialiserte **automatiske** variablar er automatiske variablar som får verdi under start av ein metode. Når metoden som eig variabelen blir kalla, gir den variabelen plass på stakken, og legg verdien på denne plassen.
Initialiserte **statiske** variablar er statiske variablar som får verdi før *main()* har starta. Når programmet blir lenka, blir det sett av plass til dei statiske variablane i minnet. Det blir også lagra ein konstant for kvar av alle dei initialiserte statiske variablane på ein annan stad i minnet. Ein konstant vil altså vera lik initial- eller startsverdien for variabelen han tilhøyrer. Vanlegvis blir konstantane lagra i programminnet, som er permanent, dvs. at det ikkje mistar innhaldet viss kraftforsyninga blir slått av.
Når systemet blir slått på, vil oppstartsmetoden, i dette tilfellet *startup_stm32f10x_md_vl.c*, kopiera konstantane inn i dei respektive variablane i dataminnet slik at desse får rett start-verdi. Denne metoden er som nemnt ferdiglaga og vil etter kopiering m.m., kalla opp hovudprogrammet, *main()*.

Eksempel 2.16.

```
int8_t y = 2;   // Initialisert global variabel, dvs. statisk.

void main(void) {
  uint8_t  j = 10;  // Initialisert automatisk, dvs. lokal variabel.
  ......
}
```

2.4.3.4 Litt om lenking og seksjonar

Under bygging er det som nemnt i kapittel 2.4.3.2, lenkaren som bestemmer kor kode og variablar mm. skal plasserast i minnet. Lenkaren plasserer dei ulike programdelene i kvar sine **seksjonar**.
Ein seksjon er eit område i minnet som blir avsett til å innehalda ein type data. I hovudsak har ein fylgjande seksjonar:

1. ***text***, der **programkoden** blir lagt.
2. ***rodata*** ("read only data"), seksjon for **konstantar** som f.eks. tekststrengar. Her vil også **startverdiane** for dei initialiserte statiske variablane liggja.
3. ***data***, der **initialiserte statiske variablar** blir lagt, som f.eks. variabelen y i eksempel 2.16.
4. ***bss*** ("block started by symbol")[54] , seksjon for **ikkje-initialiserte statiske variablar** som f.eks. variabelen *verdi* i eksempel 2.15.
5. ***bss_stack***, eigen seksjon for kladdeområde ("heap")[55] og stakk[56].
   Ein kan merka seg at begge desse områda er **dynamiske**, dvs. det løpande behovet for midlertidig lagring i desse områda avgjer størrelsen. Stakkområdet aukar nedover i minnet mens kladdeområdet aukar oppover.

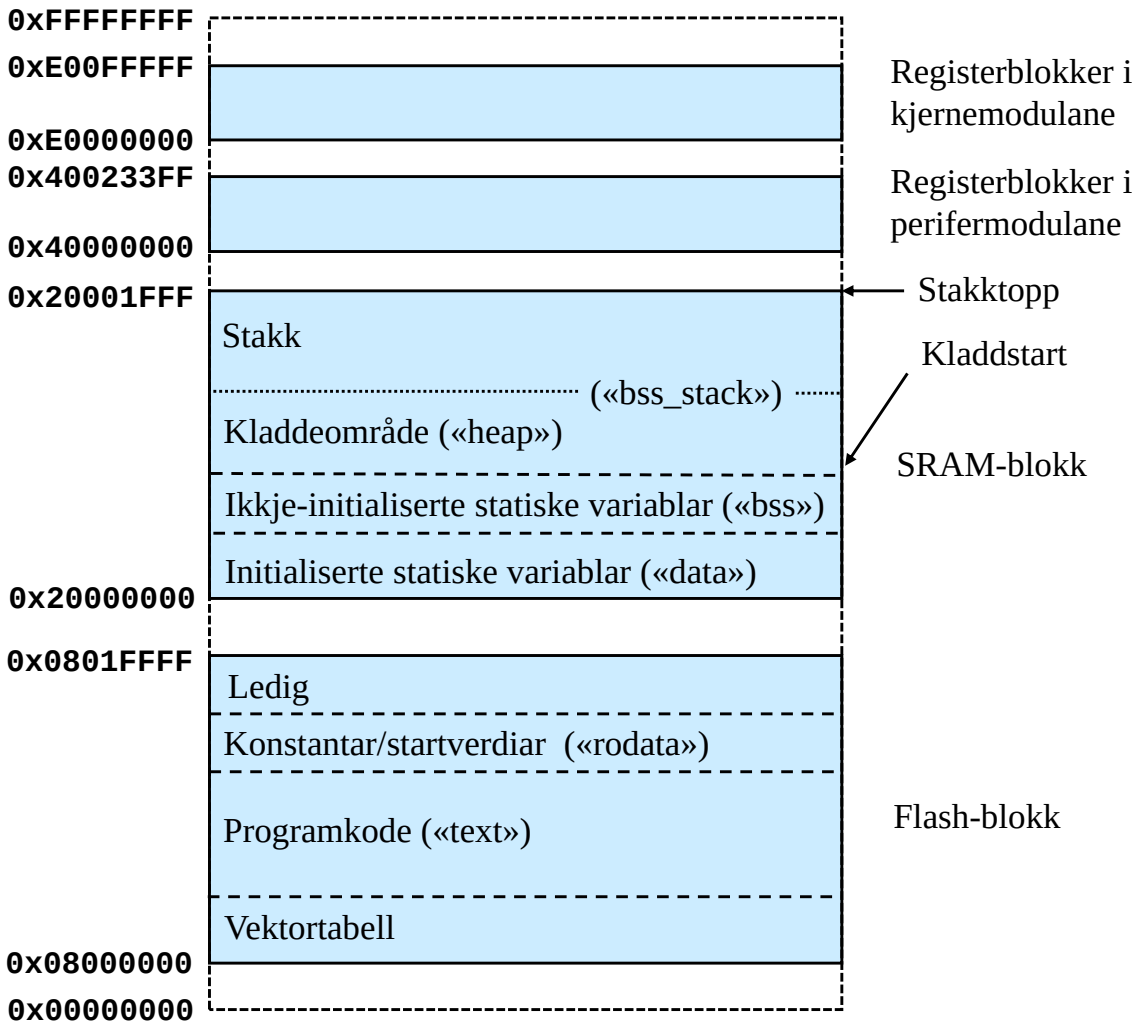
Korleis desse seksjonane blir plasserte i mikrokontrolleren vår, *STM32F100RB*, er vist i figur 2.13. Minnekartet er det same som i tabell 2.3 på side 40.

Den såkalla vektortabellen i minnekartet inneheld startadressene til metodane som skal køyrast når mikroprosessoren mottar eit avbrot. Det er ein slik avbrotsmetode ("interrupt

[54]Namnet dukka opp tidleg i datamaskinhistoria, sjå "en.wiki .bss". Der står det litt meir om denne seksjonen.

[55]Ved bruk av funksjonen *malloc* ("m(emory) alloc(ate)") i C vil det bli sett av eit minneområde i her. Viss mikrokontrolleren f.eks. har ein *ethernett*-modul som mottar ein større datapakke, vil kladdeområdet bli brukt som buffer.

[56]Meir om stakk kjem i kapittel 2.4.4.9.



Figur 2.13: Plassering av ulike seksjonar i minnet til mikrokontrolleren STM32F100RB.

handler") for kvar avbrotskjelde i systemet. Meir om avbrotshandtering kjem i kapittel 2.4.4.11.

2.4.4 Om ARM Cortex-M3 sitt instruksjonssett

2.4.4.1 Innleiing

I figur 2.4 på side 29 såg me eit eksempel på at **høgnivåinstruksjonane** som me lagar, blir omsette til **maskinkode**, som er det språket prosessoren forstår. Som nemnt tidlegare, så kan maskinkoden visast som såkalla **assemblykode** for å gjera maskinkoden lesbar for oss.

Me skal nå sjå på eit lite, men detaljert programeksempel henta frå kapittel 2.1 hjå Brown, [2].
Høgnivåkoden er vist først:

```
int counter;

int counterInc(void) {
  return counter++;
}
```

Som ein ser, er dette ein metode som inkrementerer den globale variabelen *counter*.
Figur 2.14 viser kva maskin- og assemblykode denne C-stubben gir.

```
counterInc:
  0: f240 0300  movw    r3, #:lower16:counter  // r3 = &counter
  4: f2c0 0300  movt    r3, #:upper16:counter
  8: 6818       ldr     r0, [r3, #0]           // r0 = *r3
  a: 1c42       adds    r2, r0, #1             // r2 = r0 + 1
  c: 601a       str     r2, [r3, #0]           // *r3 = r2
  e: 4740       bx      lr                     // return r0
```

Figur 2.14: Assembly- og maskinkode for ein inkrementeringsmetode.
(Frå Brown, [2], kapittel 2.1, under lisensen *Creative Common BY-NC-SA 3.0.*)

Ein skal i dette kapitlet bli litt kjent med desse og andre av ARM Cortex-M3 instruksjonar, men skal først sjå litt på kva maskinkoden til ein instruksjon fortel oss.
Maskininstruksjonen som ligg på adresse `0xA` (= `0xa`) relativt til startadressa for metoden *counterInc* i eksemplet over, har følgjande maskinkode:

```
0x1C42
```

Me forstår ingenting av dette utanom at instruksjonen er 16 bit brei, men prosessoren forstår denne og kan utføra han direkte.

Som me ser i figur 2.14, er instruksjonen `0x1C42` på assemblyform slik:

```
adds r2, r0, #1 //Altså: Adder talet 1 til innhaldet i prosessorregister r0
                //       og legg resultatet i prosessorregister r2.
```

Korleis talverdien `0x1C42` får transistorane i mikroprosessoren til å arbeida slik at akkurat denne addisjonen skjer, skal me sjå litt på i slutten av kapittel 3 om digitalteknikk.
Me skal likevel nå prøva å nærma oss denne digitalteknikken ved å sjå litt meir detaljert på addisjonsinstruksjonen over. Viss ein går til det ein kan kalla arkitekturreferansemanualen for mikroprosessorar av typen ARM Cortex-M3, [21], så vil ein i kapittel A7.7.3 finna ut meir om korleis dei 16 instruksjonsbitane er sett opp. Oppsettet er som vist i tabell 2.8.

Bit nr.	15 9	8 6	5 3	2 0
Felt	Opkode	imm	rN	rD
Verdi	0001110	001	000	010

Tabell 2.8: Instruksjonen *adds r2, r0, #1* plukka frå hvarandre.

Her er **operasjonskoden** *opkode* den delen som spesifiserer kva slags instruksjon dette er. I eksemplet her vil operasjonskoden `0001110` velja ut **addisjonsmodulen** i ALU-en som mottakar for inndata.
Ved bruk av denne instruksjonen kan ein spesifisera ein **talverdi** *imm* som skal adderast direkte (”immediate”) til innhaldet i det spesifiserte **kjelderegisteret** *rN*. Summen skal plasserast i mål-/resultat-/**destinasjonsregisteret** *rD*.
I eksemplet over er talverdien 1. Vidare er som vist *r0* kjelderegister og *r2* skal brukast som resultatregister.

Korleis instruksjonen blir handtert inne i mikroprosessoren er forsøkt illustrert i figur 2.15.

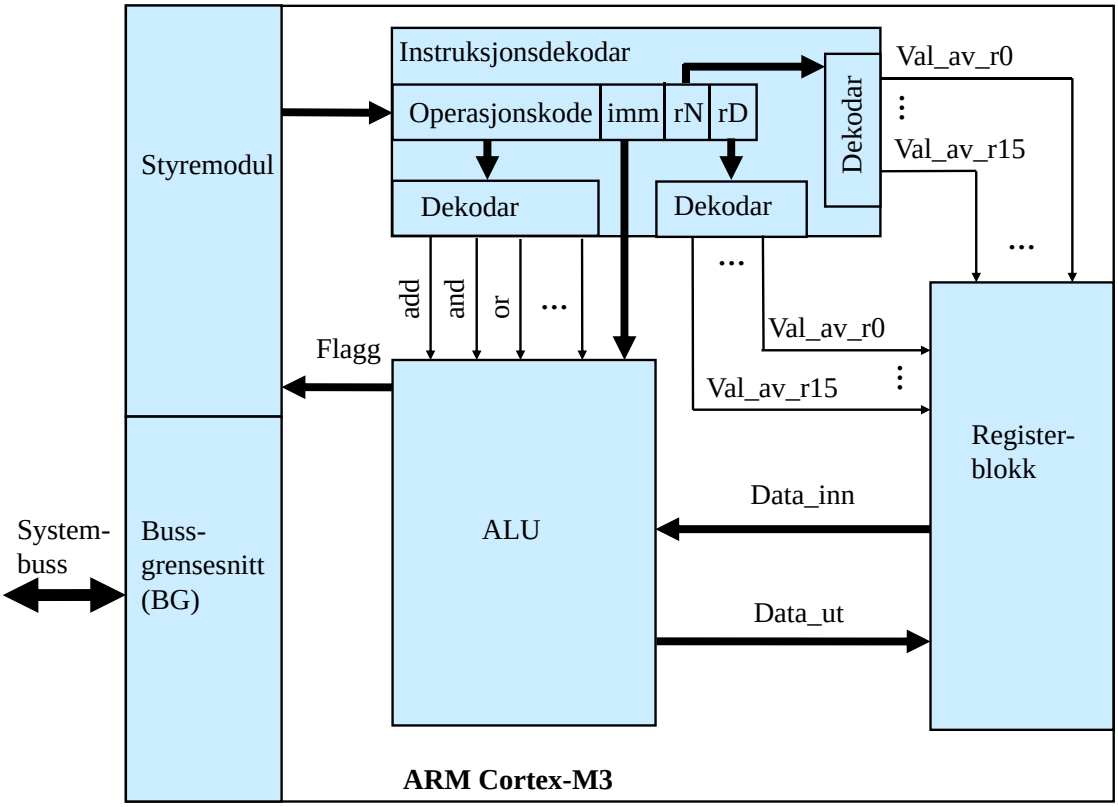
Styremodulen i mikroprosessoren les inn instruksjonar via systembussgrensesnittet og sender desse vidare til instruksjonsdekodaren.
Det er mange typar instruksjonar, og her er bare dekoding av instruksjonstypen over vist. Dei ulike delane av instruksjonen blir dekoda[57] kvar for seg og fører til aktivering av visse signal som går til registerblokka eller ALU-en. Alt dette går i ein gitt sekvens som ikkje framgår av figuren.

Viss me går tilbake til tabell 2.8 og samlar dei ulike maskinkodedelane i firar-grupper, får me verifisert at hex-koden for denne instruksjonen er følgjande:

```
0001 1100 0100 0010  =  0x1C42
```

Ein kallar ofte system som inneheld ein mikroprosessor, **smarte**. Det smarte ligg i at me som programmerarar kan spesifisera oppførselen til systemet vhja. eit program. Køyringa av programmet er ein **sekvens** der millionvis av transistorar blir slått av og på i ulike mønster.

[57] Meir om korleis ein dekodar er bygd opp, kjem i kapittel 3.8.2.



Figur 2.15: Enkelt blokkskjema av ARM Cortex-M3 med vekt på instruksjonsdekoding.

2.4.4.2 Registerstrukturen til ARM Cortex-M3

Før ein går vidare inn på instruksjonsdetaljar, kan det vera lurt å sjå på registerblokka inne i mikroprosessoren. Den inneheld registra vist i tabell 2.9.

Ein vil sjå eksempel på bruk av nokre av registra i det følgjande.
Når det gjeld køyremodus, så kan ein vhja. registeret *CONTROL* f.eks. la eit operativsystem køyra i **priviligert** modus og la trådar køyra i såkalla **brukarmodus** (”user mode”). I mindre og enkle innebygde system kan derimot heile programvaren køyra i priviligert modus, ein modus som systemet alltid vil starta opp i etter resetjing.

Ein gjennomgang av registra er gjort av Yiu i kapittel 4.2 i [3].

Register	Kommentar
r0	Generelt prosessorregister.
r1	-||-
r2	-||-
r3	-||-
r4	-||-
r5	-||-
r6	-||-
r7	-||-
r8	-||-
r9	-||-
r10	-||-
r11	-||-
r12	-||-
r13 (MSP)	Hovud-stakkpeikar ("main").
r13 (PSP)	Prosess-stakkpeikar.
r14 (LR)	Lenkeregister.
r15 (PC)	Programteljar.
PSR	Prosessorstatusregister (eigentleg tre register).
PRIMASK	Register for maskering av
FAULTMASK	avbrots- og feilkjelder.
BASEPRI	Register for å setja avbrotsprioritet.
CONTROL	Register for setting av køyremodus.

Tabell 2.9: Registerstruktur for ARM Cortex-M3. (Ref.: Kapittel 4.2 i [3].)

2.4.4.3 RISC og CISC

Mikroprosessorar kan delast i to hovudkategoriar etter kva type instruksjonssett dei har:

- Complex Instruction Set Computer (CISC)
- Reduced Instruction Set Computer (RISC)

ARM er som nemnt nå ei forkorting for *Advanced RISC Machine*[58] og er altså som dei fleste mikroproprosessorane i verda nå av typen RISC.
Eksempel på CISC er rekkja av Intel-prosessorar med nokre få unnatak som *i960*.

Toulson og Wilmhurst gir i kapittel 1 i [6] ei god oversikt over forskjellen på desse to typane instruksjonssett.

[58]Namnet var opprinneleg *Acorn RISC Machine* etter firmaet som utvikla denne, men etter nokre fusjonar der bl.a. Apple var involvert, blei namnet endra.

2.4.4.4 Litt om Thumb-2

Instruksjonssettet for ARM Cortex-M3 heiter **Thumb-2**. Dette instruksjonssettet er, som sjølve prosessoren også, resultatet av ei lang utvikling. Ein kan lesa meir om dette i kapittel 1.5 hjå Yiu, [3].
Som ein ser av eksemplet i figur 2.14, så inneheld Thumb-2 instruksjonar med breidde både på 16 og 32 bit. I dei neste kapitla skal ein visa nokre få sentrale instruksjonar for å kunne forstå mikroprosessoren betre.

2.4.4.5 Instruksjonskategoriar

Instruksjonane til ein mikroprosessor kan delast inn i følgjande kategoriar:

- Dataflytting
- Dataprosessering
- Hopping
- Andre instruksjonar

Det vil i det følgjande bli presentert instruksjonseksempel frå alle desse kategoriane.
Stoffet er basert på kapittel 5 hjå Yiu, [3], og kapittel A7 i arkitekturreferansemanualen for ARM Cortex-M3, [21].

2.4.4.6 Eit utdrag av instruksjonssettet

I tabell 2.10 er det vist eit utdrag av instruksjonssettet til ARM Cortex-M3 med korte forklaringar. Merk at i tabellen er kategorien **dataprosessering** delt opp i underkategoriane **aritmetikk**, **editering** og **logikk**.

Dei ulike instruksjonskategoriane blir tatt opp i kvart sitt kapittel i det følgjande.

Merk: Instruksjonane i tabellen under er viste utan tilleggspesifikasjonar. F.eks. kan dei fleste instruksjonane ofte ha tilleggsbokstaven *s* som spesifiserer at resultatet av denne instruksjonsvarianten skal påverka visse **statusbit** (flagg) i prosessorstaturregisteret, PSR. Viss det nede i eit program skal testast på om f.eks. resultatet av ein *OG*-operasjon blei null, vil kompilatoren velja instruksjonsvarianten *ands* i staden for bare *and*.
I eksemplet i figur 2.14 er det brukt *adds* i staden for *add* ut frå slike vurderingar.
Meir om om statusflagg og testing kjem i kapittel 2.4.4.9.

Instruksjon	Merknad	Type
Flytting		
`ldrb rD, [rN, #imm]`	Adresse = `rN + imm`	Les byte.
`ldrsb rD, [rN, #imm]`	Adresse = `rN + imm`	Les byte med forteikn.
`ldrh rD, [rN, #imm]`	Adresse = `rN + imm`	Les halvord.
`ldrsh rD, [rN, #imm]`	Adresse = `rN + imm`	Les halvord med forteikn.
`ldr rD, [rN, #imm]`	Adresse = `rN + imm`	Les ord.
`movw.w rD, #imm16`	`LSH av rD = imm16`	Flytt 16-bitsverdi direkte inn
`movt.w rD, #imm16`	`MSH av rD = imm16`	i botn/topp av register.
`push registerliste`	Adresse = r13	Kopier register til stakk.
`pop registerliste`	Adresse = r13	Hent registerverdiar frå stakk.
`strb rD, [rN, #imm]`	Adresse = `rN + imm`	Skriv byte.
`strh rD, [rN, #imm]`	Adresse = `rN + imm`	Skriv halvord.
`str rD, [rN, #imm]`	Adresse = `rN + imm`	Skriv ord.
Aritmetikk		
`add rD, rD, rM`	`rD = rN + rM`	Addisjon.
`add rD, rN, #imm`	`rD = rN + imm`	Addisjon med direkteverdi.
`mul rD, rN, rM`	`rD = rN * rM`	Multiplikasjon.
`udiv rD, rN, rM`	`rD = rN/rM`	Divisjon, unipolar.
Editering		
`asr rD, rN, rM`	`rD = rN » rM`	Aritmetisk høgreskift.
`asr rD, rN, #imm`	`rD = rN » imm`	Aritmetisk høgreskift.
`lsr rD, rN, rM`	`rD = rN » rM`	Logisk høgreskift.
`lsr rD, rN, #imm`	`rD = rN » imm`	Logisk høgreskift.
`lsl rD, rN, rM`	`rD = rN « rM`	Logisk venstreskift.
`lsl rD, rN, #imm`	`rD = rN « imm`	Logisk venstreskift.
Logikk		
`mvn rD, rM`	`rD =  rM`	Bitvis invertering.
`and rD, rD, rM`	`rD = rN & rM`	Bitvis OG.
`and rD, rN, #imm`	`rD = rN & imm`	Bitvis OG med direkteverdi.
`orr rD, rN, rM`	`rD = rN | rM`	Bitvis ELLER.
`orr rD, rN, #imm`	`rD = rN | imm`	Bitvis ELLER med direkteverdi.
`eor rD, rN, rM`	`rD = rN ^ rM`	Bitvis eksklusiv ELLER.
`eor rD, rN, #imm`	`rD = rN ^ imm`	Bitvis XELLER med direkteverdi.
`cmp rN, rM`	Flagg = `f(rN - rM)`	Samanlikning.
`cmp rN, #imm`	Flagg = `f(rN - imm`	Samanlikning med direkteverdi.
Hopping		
`b L`	Hoppadresse = L	Hopp til merket L ("label").
`bx rM`	Hoppadresse = rM	Hopp til adresse.
`bl L`	Hoppadresse = L LR = returadresse	Hopp til L med retur.
`b<nn> L`	Hoppadresse = L	Hopp til L viss vilkåret *nn* er oppfylt. Ulike vilkår *nn* er viste i tabell 2.12.

Tabell 2.10: Eit utdrag av instruksjonssettet til ARM Cortex-M3.
(Ref.: Kap.5 hjå Yiu, [3])

2.4.4.7 Flytting

Desse instruksjonane realiserer flytting av data mellom minnet og prosessorregistra *r0 - 15*, såkalla lasting (”load”) og lagring (”store”).
Her skal ein fokusera på nokre vanlege flyttinstruksjonar vist øvst i tabell 2.10, og vil illustrera desse med to eksempel.

Eksempel 2.17. *Lagring av data i minnet.*

Me tenkjer oss følgjande registerinnhald i ARM cortex-M3:

```
r0 =  0x12345678
r1 =  0x20003000
r2 =  0x20003004
```

Eit program inneheld følgjande maskinkodesekvens vist i assemblyformat:

```
str   r0, [r1, 0] ; Lagra eit ord     frå r0 på adresse = r1 + 0
strh  r0, [r2, 0] ; Lagra eit halvord frå r0 på adresse = r2 + 0
strb  r0, [r2, 3] ; Lagra ein byte    frå r0 på adresse = r2 + 3
```

Etter køyring av denne sekvensen vil den aktuelle delen av minnet sjå ut som vist under:

Adresse	Innhald	Kommentar
..	..	
0x20003007	0x78	LSByte av r0
6	-	
5	0x56	LSHalvord
0x20003004	0x78	av register r0
3	0x12	MSB på høgaste adresse
2	0x34	
1	0x56	
0x20003000	0x78	LSB på lågaste adresse
..	..	

Her er det viktig å merka seg følgjande:

1. Når ein bare skal henta ut ein byte eller eit halvord frå eit register, er det den minst signifikante byten, **LSB**, eller det minst signifikante halvordet, **LSH**, som blir tatt. Viss andre delar av registeret skal hentast ut, må me eller kompilatoren leggja inn ein **skiftinstruksjon** før sjølve lagringsinstruksjonen. Skiftinstruksjonen skiftar, dvs. flyttar den aktuelle delen av registeret sidevegs ned til botnen av registeret.
2. ARM Cortex-M3 er ein såkalla **”little endian”**-prosessor[59]. LSB av eit ord eller halvord vil då som vist alltid bli plassert på lagåste adresse av

[59] Sjå ”en.wiki Endianness”.

bytane i ordet eller halvordet.
Det motsette er **"big endian"**, dvs. **byte-reversering**. Det er viktig å vita dette når ein f.eks. skal setja opp ei kommunikasjonslinje mellom to **ulike plattformer**. Motsett "endian"-type i dei to endane vil gi feiltolking av data.

Eksempel 2.18. *Lasting av data frå minnet.*

Me tenkjer oss følgjande registerinnhald i ARM cortex-M3:

```
r4 =  0x20001000
```

Ein del av minnet har følgjande innhald:

Adresse	Innhald
..	..
0x20001006	0xDE
5	0x9A
0x20001004	0xBC
3	0x12
2	0x34
1	0x56
0x20001000	0x78
..	..

Eit program inneheld følgjande maskinkodesekvens vist i assemblyformat:

```
ldr   r0, [r4, 0] ; Last eit ord     inn i r0 frå adresse = r4 + 0
ldrh  r1, [r4, 4] ; Last eit halvord inn i r1 frå adresse = r4 + 4
ldrb  r2, [r4, 6] ; Last ein byte    inn i r2 frå adresse = r4 + 6
```

Etter køyring av denne sekvensen vil aktuelle prosessorregister ha følgjande innhald:

```
r0 =  0x12345678
r1 =  0x00009ABC
r2 =  0x000000DE
```

Her er det viktig å merka seg følgjande:

1. Når ein bare lastar ein byte eller eit halvord inn i eit register frå minnet vil dette bli plassert som vist i **botnen**, dvs. i nedre delen av registeret. Resten av registeret vil bli **null-utfylt** ("zero extended"). Ved bruk av desse instruksjonane oppfattar ein altså minneinnhaldet som ein positiv, eller som nemnt før, unipolar talverdi ("unsigned").
2. Viss kompilatoren skal laga maskinkode for å henta eit minneinnhald som kan vera både negativt og positivt, dvs. med forteikn ("signed"), vil han bruka instruksjonane *ldrsb* og *ldrsh*. Den øvre delen av registeret blir då **forteikns-utvida** ("sign extended"). Det står meir om representasjon av positive og negative tal i kapittel A.2.

I tillegg har ein mellom anna instruksjonane

- *movw* ("move wide") og
- *movt* ("move top").

Desse kan brukast til å leggja inn talverdiar på 16 bit inn i eit register, dvs. å **initialisera** registeret. Her treng ein ikkje bruka skiftoperasjonar då *movt* plasserer talverdien rett inn i toppen eller øvre halvdel, dvs. MSH, av registeret.

I eksemplet i figur 2.14 på side 71 blei desse instruksjonane brukte til å plassera **adressa** til variabelen *counter* inn i register *r3*.
Eit eksempel på bruk blir også vist i neste underkapittel.

Til slutt skal ein her nemna instruksjonane

- *push* og
- *pop*.

Desse er sentrale ved kall av metodar samt ved køyring av avbrotsprogram og fleirtrådssystem. Vhja. *push* kan ein lagra ein kopi av innhaldet i spesifiserte register på **stakken**, som er eit minneområde for midlertidig lagring. Ved retur frå f.eks. ein metode blir det gamle innhaldet av registra henta tilbake igjen vhja. av instruksjonen *pop*. Hovudprogrammet kan så arbeida vidare med dette.
Bruk av stakk er viktig då prosessoren bruker den same registerblokka ved køyring av hovudprogram og metodar eller ved køyring av ulike trådar i eit fleirtrådssystem.
I kapittel 2.4.4.9 om hoppinstruksjonar vil det bli gitt eit eksempel på bruk av desse stakkinstruksjonane.

2.4.4.8 Dataprosessering

Dataprosessering kan som nemnt delast opp i **aritmetiske** operasjonar, **editering** og **logiske** operasjonar. Her har me alt sett på addisjonsinstruksjonen *adds*[60]. Meir om dei aritmetiske instruksjonane utover det som står i tabell 2.10 kan finnast i kapittel 5 hjå Yiu, [3].

I tillegg til dei aritmetiske instruksjonane har ein ulike **skiftoperasjonar** som er viste i tabell 2.10. Skifting høyrer til underkategorien editering. Sjølve operasjonane blei forklarte i kapittel 2.4.1.2 frå side 50.

[60] I tabell 2.10 er det ikkje vist såkalla kvalifikatorar. Ein *s* vil som nemnt føra til at statusflagga blir oppdaterte etter operasjonen. Det er kompilatoren som vel både instruksjonar og med rette kvalifikatorar under bygging av programvaren vår. Det fører for langt å ta opp slikt i full breidde i ei lita innføring som dette.

Når det gjeld logiske operasjonar, skal me her nøya oss med å visa korleis assemblykoden for C-instruksjonen i eksempel 2.6 på side 48 kan bli.

Eksempel 2.19.

Den globale variabelen *testbyte* er deklarert som *uint8_t*.
Me tenkjer oss at adressa til variabelen er `0x20003000`.

Eit program inneheld følgjande C-instruksjon : `testbyte = testbyte | 0xC5;`

Ved bygging og disassemblering vil denne høgnivåinstruksjonen typisk gi følgjande assemblykode:

```
movw  r4, #0x3000     ; Legg inn nedre 16 adressebit
movt  r4, #0x2000     ; Legg inn øvre adressedel (''top'')
ldrb  r1, [r4, 0]     ; Last inn variabelverdi frå adresse = r4 + 0
orr.w r1, r1, #0xC5   ; W for ''wide'', dvs. 32bits-instruksjon
strb  r1, [r4, 0]     ; Oppdater variabelverdi i minnet
```

Dei valde registernummera er bare eksempel. Ein kompilator vil gjera sine val. Instruksjonssettet Thumb-2 inneheld som nemnt både 16- og 32-bitsinstruksjonar. Instruksjonar på 16 bit som kjem etter kvarandre i programmet, kan lesast to og to samtidig over ein databuss på 32 bit. Køyringa av slike programsekvensar vil då gå raskare enn sekvensar av 32-bitsinstruksjonar. Kompilatoren vil derfor prøva å velja registernummer og laga maskinkode slik at instruksjonsformata blir minst mogleg. Fleire av formata må likevel vera på 32 bit, som f.eks. *movw* og *movt* der adressedelen åleine okkuperer 16 bit.

Det kan til slutt her nemnast at samanlikningsinstruksjonen *cmp* er omtalt nærare i kapitlet om digitalteknikk, nemleg i kapittel 3.8.4.3.

2.4.4.9 Hopping

Vhja. hoppinstruksjonar (”branch”) kan ein styra programflyten, dvs. gjera avgreiningar i programmet. Det er to hovudtypar hopp:

- Hopp på vilkår (”conditioned”)
- Hopp utan vilkår (”unconditioned”)

Hopp på vilkår

Når prosessoren f.eks. har utført ein ALU-operasjon, vil det bli sett nokre bit i prosessor-statusregister *PSR* avhengig av kva resultatet av operasjonen blei. Slike statusbit blir også ofte kalla **flagg**.
Viss f.eks. resultatet av ein subtraksjon var null, vil Z-flagget ("Zero") bli sett til '1'. Neste instruksjon kan då testa på dette flagget. Testen avgjer om mikroprosessoren skal hoppa eller ikkje.

Eksempel 2.20.

Utgangspunktet er ei enkel venteløkkje i C der ein tel ned ein variabel frå ein startverdi. Ei slik løkkje kan brukast til å realisera ein pause i programmet[61].

```
#define startverdi 1000  // Gir ventetida.
...
int main(void) {
  ....
  uint32_t i;
  ....
  i = startverdi;
  while(--i >= 0) {    // Venteløkkje med dekrementering.
    ;
  }
  ...
  return;
}
```

Den resulterande assemblykoden viser vilkårshopp med ein *bpl*-instruksjon ("branch if pluss").

```
main
      ....
      movw r1, ..        ; Legg inn startverdi.
L1    adds r1, r1, -1    ; Dekrementering.
      bpl  L1            ; Viss positiv, dvs. større eller lik 0,
                         ; hopp til merket (''label'') L1.
      ....
```

[61] Meir fleksibelt og vanleg er å realisera dette vhja. ein taimer.

Ei liste over flagg i ARM-prosessorar er vist i tabell 2.11.

Flagg	**Engelsk namn**	**Viss flagg == 1, så ga siste operasjon**
N	Negative	eit negativt resultat.
Z	Zero	null som resultat.
C	Carry	eit mente.
V	Overflow	overrulling som resultat.

Tabell 2.11: Statusflagg i ARM Cortex-M3.
(Ref.: Tabell 5.33 hjå Yiu, [3].)

Dei ulike flagga er representerte med kvar sine bit i prosessorstatusregisteret PSR[62].

Når det gjeld V-flagget, så kan mange ulike operasjonar føra til at dette blir sett.
Viss f.eks. ein addisjon av to tal som begge har MSb lik 0 gir eit resultat med MSb lik 1, blir *V = 1*.
Det at to positive tal gir eit resultat med negativt forteiknsbit, er altså noko som blir varsla av dette flagget. Viss resultatet er deklarert som unipolart ("unsigned"), treng ein ikkje bry seg om det varselet.
I motsett fall er det resultatet som er for stort til å få plass i den datatypen det er deklarert som. Når resultatet altså renn over, må ein reagera på varselet[63].

I tabell 2.12 er det vist eit utdrag av vilkår ein kan bruka ved hopping. Prosessoren bruker eitt eller fleire av statusflagga over ved test av om eit vilkår er oppfylt.

Vilkår <nn>	**Engelsk namn**	**Statusflaggverdiar**
eq	equal	Z = 1
ne	not equal	Z = 0
mi	minus	N = 1
pl	plus	N = 0
hi	unsigned higher	C = 1 og Z = 0
ge	signed greater or equal	N = V
gt	signed greater than	Z = 0 og N = V

Tabell 2.12: Nokre testvilkår brukt av hoppinstruksjonar.
(Ref.: Tabell 5.35 hjå Yiu, [3].)
Bruk av koden **<nn>** er vist i eksempel 2.20 og nedst i tabell 2.10.

Ved utføring av instruksjonen *bpl L1* i eksempel 2.20 er det altså bare verdien av *N*-flagget som avgjer om ein skal hoppa eller ikkje. Andre vilkår kan vera baserte på meir komplekse kombinasjonar av flagg, og det fører for langt å gå inn på dette her.

[62] Sjå tabell 2.9. Bitnummera er vist i tabell 5.33 hjå Yiu, [3].
[63] Det står meir om V-biten hjå "en.wiki overflow flag".

Hopp utan vilkår

Sentrale instruksjonar blir illustrerte med eksemplet under.

Eksempel 2.21.

```
main                    ; Assemblykode for hovudprogrammet.
        .....
merke1  .....

        .....
        bx r5           ; Hopp til adressa som r5 viser til.
        .....
        bl metode1      ; Hopp til metode1 og legg returadressa i
                        ; lenkeregisteret LR (r14).
        .....
        b  merke1       ; Hopp til merke1.

metode1                 ; Assemblykode for metode1.
        .....
        bx LR           ; Retur til hovudprogrammet (main).
```

Ved å bruka hopp med l-index (”link”), får ein plassert returadressa i LR. Når prosessoren utfører instruksjonen *bx LR* i *metode1* over, vil han hoppa tilbake til der han slapp i hovudprogrammet.
Det står meir om dette og hopping generelt i kapittel 5.6.9 hjå Yiu, [3].

I neste kapittel møter ein desse instruksjonane igjen når ein skal sjå på bruk av stakk ved metodekall.

2.4.4.10 Bruk av stakk ved metodekall

Ved kall av metodar, køyring av avbrotsprogram og fleirtrådssystem er som nemnt instruksjonane *push* og *pop* sentrale i tillegg til sjølve hoppinstruksjonane.

Vhja. *push* kan ein lagra ein kopi av innhaldet i spesifiserte register på **stakken**, som altså er eit minneområde for midlertidig lagring.
Det følgjande eksemplet på eit program som inneheld **nesta** metodekall kan visa korleis stakken blir brukt. Nesting vil her seia at hovudprogrammet kallar ein metode som igjen kallar ein annan metode, noko som er heilt vanleg i eit program.

Eksempel 2.22.

Eit utdrag av programmet for eit system med nesta metodekall er vist i det følgjande.

```
int main(void) {
  uint32 i=1; //lokal variabel som f.eks. blir lagt i register r3.
  ....
  i++;
  ....
  metode_A();
  ....
  return;
}

void metode_A(void) {
  uint32 j=1; //lokal variabel som f.eks. også blir lagt i register r3.
  ....      //(Det er kompilatoren som vel kva register som skal brukast.)
  j++;
  metode_B();
  ....
}
```

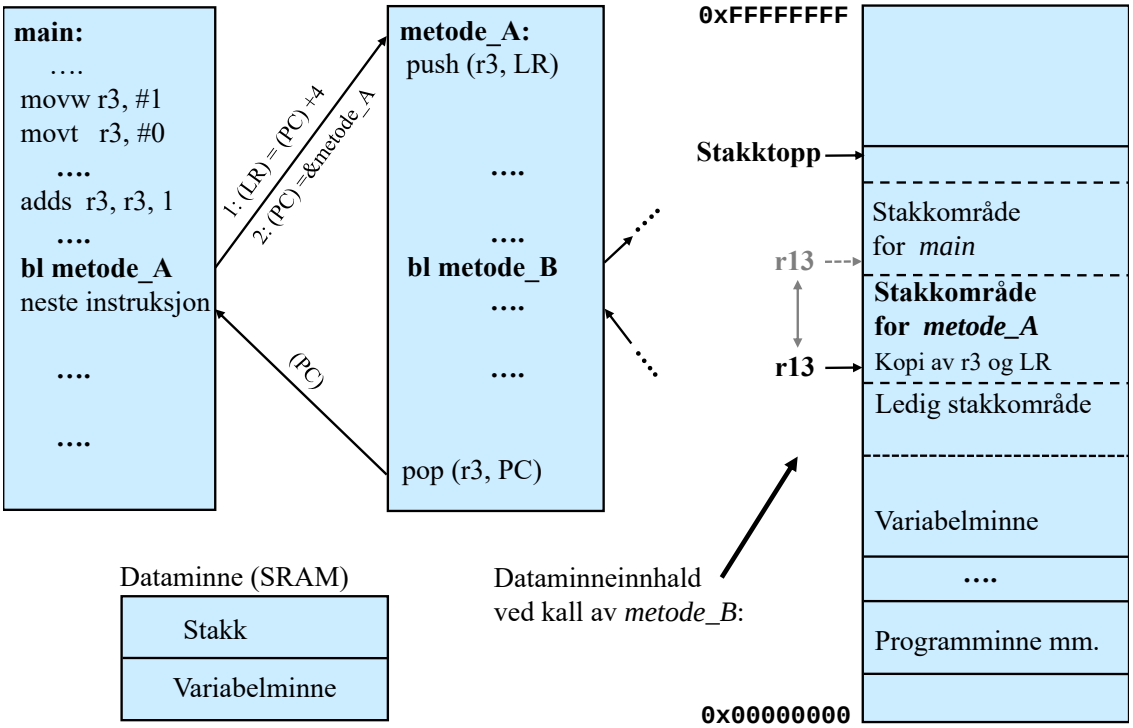
Metodekallet er altså nesta då *metode_A* vil kalla *metode_B*.
Ein tenkjer seg også som vist at både hovudprogrammet *main* og metoden *metode_A* gjer bruk av register *r3*. Dette er ein vanleg situasjon viss eit program f.eks. inneheld ein del variablar.

Alt dette er illustrert i figur 2.16.

Nestinga gjer det nødvendig å bruka stakk slik at ein ikkje mistar det "gamle" innhaldet i lenkeregisteret LR ved kall av *metode_B*.
I tillegg bruker ein i hovudproggrammet *r3* til lagring av ein lokal variabel. Ved bruk av dette registeret i ein metode, må ein kopi av den "gamle" verdien også **lagrast** ("push") på stakken.
Det siste som blir gjort før retur frå ein metode, er **henting** ("pop") av dei gamle verdiane frå stakken.

Prosessorregister *r13*[64] er stakkpeikar.
Kvar metode får tildelt eit stakkområde under køyring, og *r13* peikar på starten av dette området. Som vist i figuren, er det første som skjer ved køyring av *metode_A*, at stakkpeikaren blir flytt eit visst stykkje nedover[65] i minnet. I det området som då oppstår, blir dei viktige registerverdiane lagt inn, som forklart over.
Det siste som skjer før retur frå metoden, er altså henting av registerverdiar

[64] Viss ein køyrer i priviligert modus, er dette MSP. I brukarmodus arbeider ein med PSP, jfr.tabell 2.9 på side 74.
[65] Mot ei lågare minneadresse.



Figur 2.16: Register- og stakkhandtering ved metodekall.

og så flytting av stakkpeikaren tilbake til stakkområdet for den metoden ein returnerer til, som her er *main*. Ein seier då at stakkområdet for *metode_A* er tømt, dvs. at det ikkje eksisterer lenger.

I figuren er det også vist kva som skjer med programteljaren *r15 (PC)* ved kall av metodar. Ved kall av *metode_A* blir først adressa til hoppinstruksjonen plassert i lenkeregisteret *r14 (LR)*. LR får så eit tillegg på 4 slik at mikroprosessoren etterpå kan returnera til *neste instruksjon* i hovudprogrammet[66].

Etter at returadressa er plassert i LR, blir startadressa[67] for *metode_A* lagt inn i PC, og det fører til at mikroprosessoren går i gong med å køyra denne metoden.
Det siste som skjer i *metode_A* er køyring av instruksjonen
`pop (r3, PC)`.
Denne vil både henta dei gamle registerverdiane frå stakken, samt plassera verdien som låg på LR-plassen på stakken, inn i programteljaren. Dette er eit alternativ til bruk av instruksjonen `bx LR`[68] etter henting, som blir mindre effektivt i dette tilfellet[69].

[66] Hoppinstruksjonen **bl metode_A** er på 32 bit, dvs. 4 bytar, så returadressa ligg då 4 bytar lengre ned i minnet.

[67] Teiknet "&" framfor eit variabel- eller rutinenamn betyr "adressa til" i C

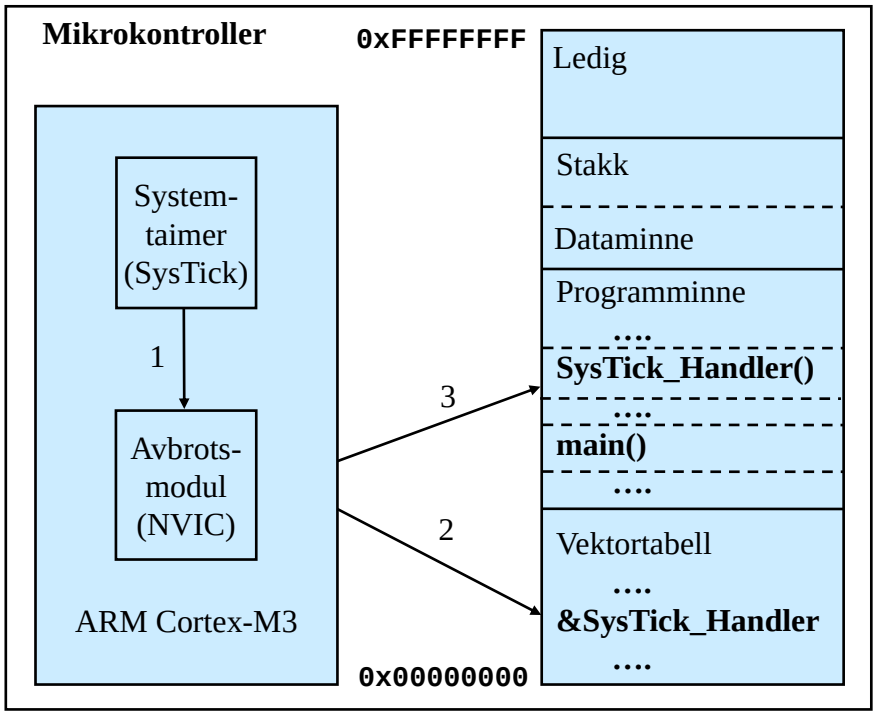
[68] Sjå eksempel 2.21.

[69] Viss det bare er eit enkelt metodekall og ingen felles registerbruk, treng ein ikkje bruka stakken. Då vil kompilatoren bruka instruksjonen **bx LR** for å realisera retur frå metoden.

2.4.4.11 Litt om avbrotshandtering med SysTick-taimeren som eksempel

Me har nå sett på korleis mikroprosessoren handterer kall av ein metode.
Ein nokså tilsvarande situasjon oppstår viss mikroprosessoren får eit **avbrotssignal** ("interrupt"). Det kan f.eks. vera ein serieport som gir avbrotssignal når han har mottatt eit nytt teikn, eller ein taimer ("timer"), dvs. tidtakarmodul, som gir avbrotssignal når han har talt seg ned til null[70].
Når μP-en mottar eit avbrotssignal, vil han gå over i ein såkalla **unntakstilstand** ("exception"). Han må her køyra ein såkalla **avbrotsmetode** ("handler")[71], som inneheld alt som må gjerast når eit slikt avbrot kjem.

For at ein avbrotsmetode skal bli køyrt etter at ei kjelde har gitt eit avbrotssignal, må mikrokontrolleren vera konfigurert eller sett opp på rett måte. Figur 2.17 viser handteringsvegen for eit avbrot frå taimermodulen SysTick.



Figur 2.17: Handtering av eit avbrot frå SysTick-taimeren.

[70] Eit avbrotssignal er bare ein av dei tinga som kan få mikroprosessoren over i ein såkalla **unntakstilstand** ("exception"). Feks. vil ei deling på null i programmet eller eit programstyrt avbrot ("software break") også krevja køyring av ein eigen metode for dette.
[71] Eit anna vanleg namn er "Interrupt Service Routine", ISR.

Viss alle dei 3 vegane i figuren skal vera opne, må følgjande vera oppfylt:

1. Kontoll- og statusregisteret i taimeren, **SysTick Control and Status Register**, må som vist tidlegare[72] vera sett opp til å gi avbrot ("enable interrupt").
   I tillegg må avbrotsmodulen NVIC ("Nested Vectorized Interrupt Controller") vera sett opp til å sleppa inn avbrotet. Viss ikkje, vil dette ikkje bli behandla. Såkalla **maskering** og **avmaskering** av avbrot blir gjort ved å skriva eit bitmønster til eit bestemt register i NVIC.
   I tillegg må ein skriva til NVIC for å setja avbrotsprioriteten som vist under:

   ```
   NVIC_SetPriority(SysTick_IRQn, 1); // 0-31 der 0 er høgast
   ```

   NVIC er som SysTick-taimeren ein modul inne i prosessoren, og NVIC si oppgåve er å handtera alle avbrota i systemet. Prioriteten fortel NVIC kven som er viktigast, dvs. som skal behandlast først viss fleire kjelder gir avbrot på ein gong.

2. For å få oppstart av ein avbrotsmetode, må **startverdien** for denne vera plassert på rett plass i den såkalla **vektortabellen**.
   Vektortabellen er ei blokk som ligg heilt nederst i programminnet (Flash). Under bygging av programvaren vil lenkaren generera dei startadressene som trengst. Vektortabellen kan derfor fyllast ut som ein tabell under bygging av programvaren. Tabellen blir overført saman med maskinkode og initialverdiar for variablar mm. til Flash-minnet ved programmering av dette.

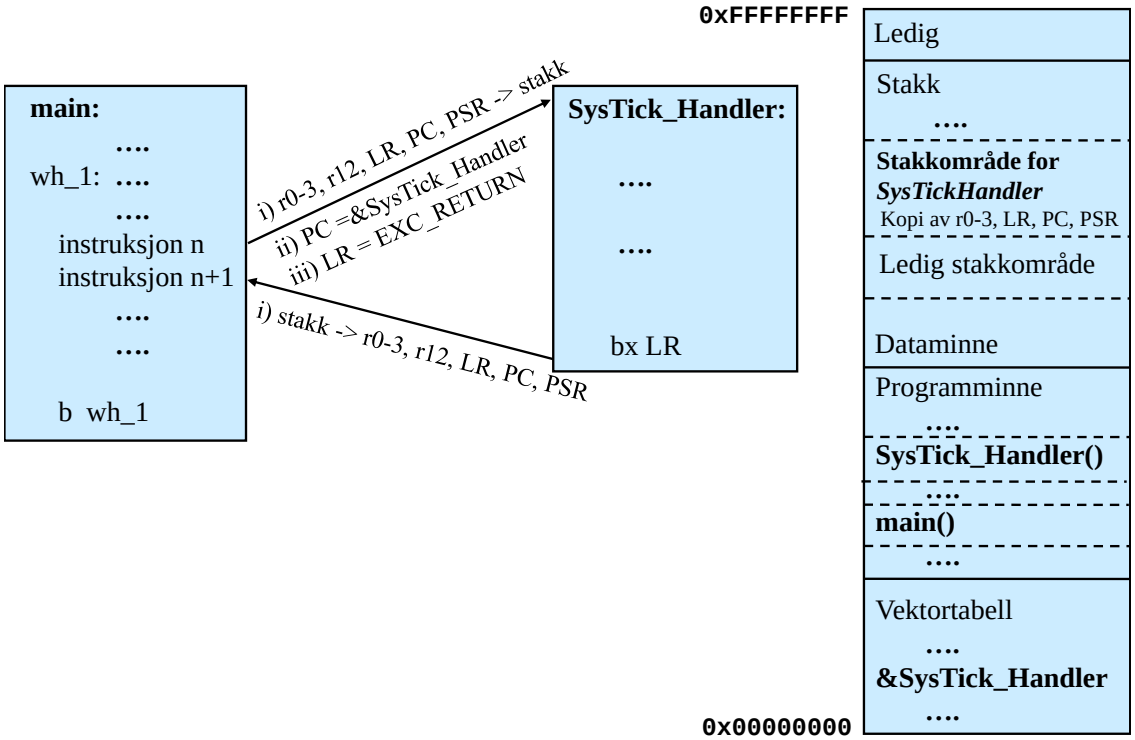
3. Avbrotshandteringa blir gjort av logikk i prosessoren. Instruksjonen som prosessoren held på med, blir først fullført. Etterpå vil logikken så leggja startadressa til avbrotsmetoden inn i programteljaren. Avbrotsmetoden vil då starta opp. Meir om dette kjem under.

Det som skjer ved utføring av ein avbrotsmetode, er vist i figur 2.18. Det er igjen tatt eksempel i taimermodulen SysTick, som her er sett opp til å gi avbrot når han har talt seg ned til null frå ein gitt startverdi.

Mikroprosessoren køyrer som nemnt først ferdig den instruksjonen han held på med, og så vil avbrotsmetoden starta etter eit kort intervall[73].

[72] Sjå figur 2.11 på side 61 med tilhøyrande tekst.

[73] Dette kan i eit vanleg tilfelle vera så kort som 12 klokkesyklar, sjå kapittel 8.3 hjå Yiu, [3]. I tilfelle der ein etter å ha behandla eitt avbrot går rett over til å behandla eit etterfølgjande avbrot, såkalla hale-heng ("tail-chaining"), kan intervallet koma ned i 6 syklar.



Figur 2.18: Køyring av ein avbrotsmetode ("handler").

Som vist i figuren, gjer prosessorlogikken følgjande i løpet av dette intervallet:

1. Registra *r0-3, r12, LR, PC* og *PSR* blir lagra på stakken. Desse registra blir kalla ei **stakkramme** ("stack frame")[74]. Følgjande er då lagra på stakken:
   - Fem av kladderegistra, nemleg *r0-3* og *r12* [75].
   - Returadressa tilbake til der ein nettopp var, dvs. før overgang til avbrotsmetoden. Returadressa ligg som nemnt før, i *LR*.
   - Adressa til den instruksjonen ein skal starta på ved retur[76]. Denne ligg i *PC*.
   - Verdien til prosessorstatusregisteret, dvs. *PSR*, for den metoden ein nettopp var i.

   Innhaldet i denne stakkramma blir ofte kalla **tilstanden** ("state") for metoden som blir avbroten.

2. Programteljaren vil nå bli sett opp med startadressa til avbrotsmetoden ("handler").

3. Lenkeregisteret *LR* får ein spesiell verdi kalla *EXC_RETURN* ("Exception return"). Når prosessoren kjem til siste instruksjonen i avbrotsmetoden, *bx LR*, vil han då gjera det som er spesielt for ein retur frå ein avbrotsmetode. Dette er å henta stakkramma tilbake. Då vil automatisk neste instruksjon som prosessoren hentar, vera neste instruksjon i metoden som blei avbroten. Alt det andre vil også då vera på plass.

[74] Sjå kap. 8.1 hjå Yiu, [3].

[75] Viss avbrotsmetoden brukar fleire register enn desse, må desse lagrast på stakk vhja. ein *push*-instruksjon i sjølve avbrotsmetoden.

[76] Hugs at den førre instruksjonen er nettopp gjort ferdig.

Viss ein etter å ha starta opp avbrotsmetoden får eit **nytt avbrot** med høgare prioritet, vil avbrotsmetoden som køyrer, brytast av. Ei **ny** stakkramme vil då bli lagra på stakken. Dette er altså virkemåten til eit **nesta avbrotssystem**. Viss nestinga er djup, som kan skje viss ein har mange ulike prioritetsnivå i bruk, treng ein stor stakkplass totalt.

Som forklart i punkt 3, så vil prosessorlogikken gå motsett veg ved **retur** frå avbrotsmetoden. Dette er vist som eitt steg, *i)*, i figuren. Stakkramma blir då automatisk henta tilbake som nemnt i punkt 3 over.

Når det gjeld sjølve avbrotssystemet til ARM Cortex-M3, kan ein finna mykje meir om dette i kapittel 7 og 8 hjå Yiu, [3].

2.4.4.12 Andre instruksjonar

Her har ein mellom anna instruksjonane *mrs* og *msr* for flytting mellom *r0 - 15* og spesialregistra i prosessoren. Vhja. desse kan ein f.eks. lesa frå og skriva til statusregisteret PSR.

Andre instruksjonar som kan nemnast her, er *wfe* og *wfi*[77] som set prosessoren i **sovetilstand**. Han vil då i første tilfellet vakna av ei spesifisert hending og i andre tilfellet av eit avbrotssignal ("interrupt"). Ved f.eks. null aktivitet mot ein mobiltelefon over ein viss tidsperiode, vil *wfi* bli køyrt. Eit tastetrykk vil gi eit avbrotssignal som vekkjer han opp att.
Under soving vil effektbruken vera kraftig redusert, noko som forlengar ladeintervallet for eit batteridrive produkt.
Det står meir om utvikling av lågeffektsapplikasjonar i kapittel 9 hjå Yiu, [3].

[77] Tydinga er *w(ait)f(or)e(vent)/i(nterrupt)*.

2.4.5 Oppbygging av programvaren

2.4.5.1 Basis byggjeblokker

Utgangspunktet for utvikling av eit innebygd system er ein spesifikasjon av funksjonane til systemet. Etter å ha vurdert plattform og så funne ut kva funksjonar som skal realiserast i maskinvare[78], kan utviklinga av programvarefunksjonane ta til. Denne kan realiserast på mange måtar. Det er altså **ingen fasitstruktur**, sjølv om nokre strukturar er meir optimale enn andre.

Flytskjema for basisblokker i ein programvarestruktur er vist i figur 2.19.

I alle system vil det vera eit **hovudprogram**, *main()*. Når ein koblar til kraftforsyninga eller trykkjer på ein omstart/resett-knapp, vil prosessoren starta ein **oppstartsmetode**[79]. Denne vil etter å ha initialisert stakkpeikar mm., kalla opp hovudprogrammet.

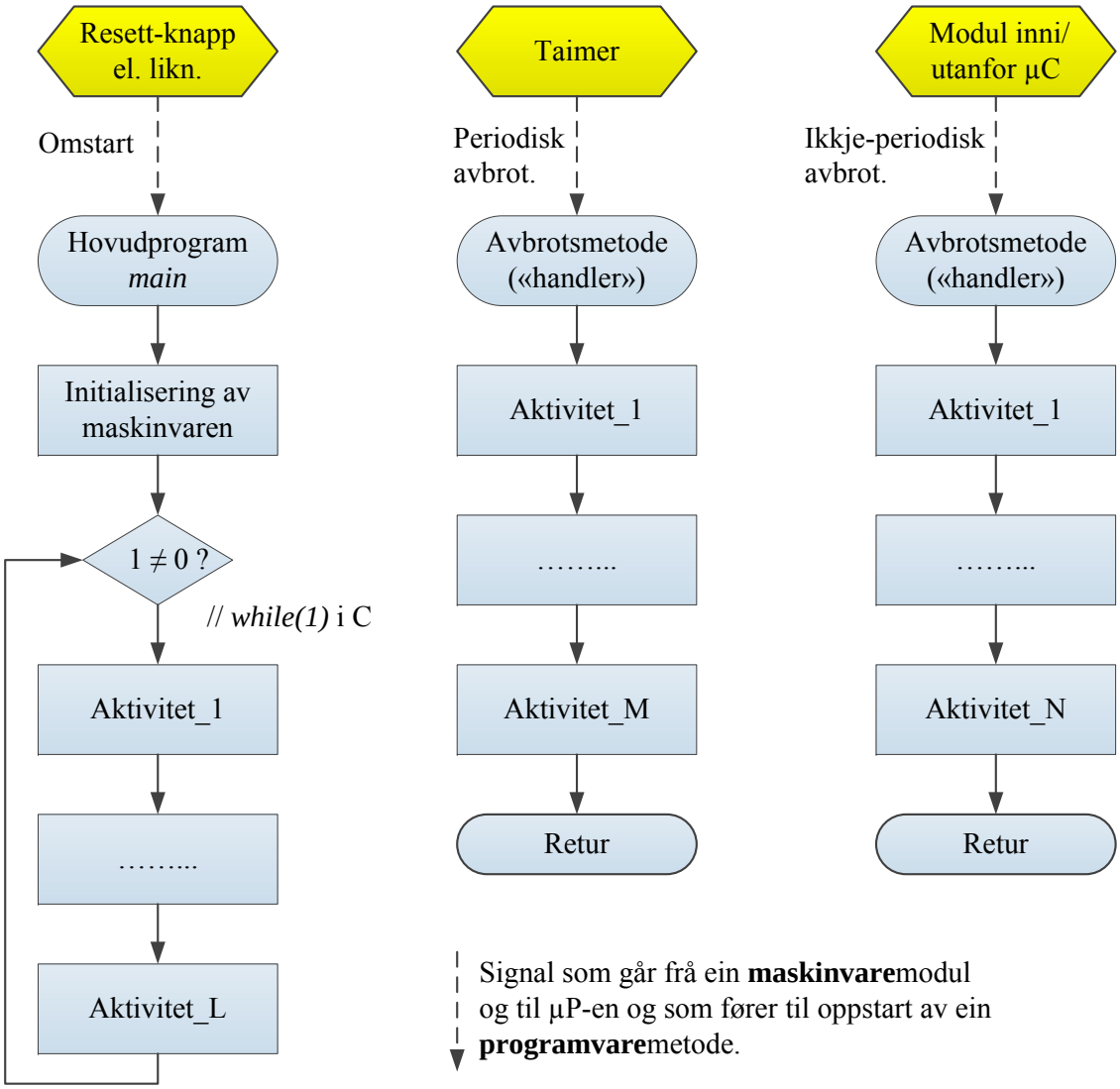
I tillegg kan ein ha programvaremodular som automatisk blir starta opp etter eit **avbrot**. Korleis ein slik oppstart skjer, er kjent frå kapittel 2.4.4.11. I neste kapittel skal ein sjå meir på sjølve oppførselen til eit system basert på avbrot.

Eit **avbrot** er basert på ei **hending** (”event”). Hendingar kan vera **periodiske** eller **ikkje-periodiske**.
Ein taimer som er sett opp til å telja seg ned til null frå ein startverdi og så automatisk starta ny nedteljing etterpå, kan gi opphav til eit periodisk avbrot. Sjølve **hendinga** her er at teljeverdien blir null. Taimeren kan då setjast opp slik at denne hendinga gir eit avbrot.

Ei ikkje-periodisk hending kan vera at ein bit i inngangsregisteret til ein GPIO-modul endrar verdi eller at mottaksregisteret i ein kommunikasjonsmodul har ein ny verdi klar for avlesing.

Me skal nå sjå litt på kva som avgjer kva byggjeblokker ein vel å ta med i ein programstruktur.

[78] Dette står det meir om i kapittel 2.4.5.5.
[79] Om oppstartsmetoden står det litt i slutten av kapittel 2.4.2.4.



Figur 2.19: Ulike byggjeblokker i ein programstruktur.

2.4.5.2 Litt om sanntidssystem

Innebygde system er ofte **sanntidssystem** (”real-time system”), dvs. at dei bl.a. må

- lesa av målesignal og setja ut styresignal på faste tidspunkt og
- reagera **raskt nok** på ytre hendingar.

Kva som er raskt nok, dvs. kor harde sanntidskrava er, varierer frå system til system. Når sensorar i eit styresystem for ei kollisjonspute i ein bil detekterer kollisjon, er krava til reaksjonstida mykje sterkare enn dei f.eks. vil vera til styresystemet inne ei bankID-brikke.

2.4.5.3 Dei to hovudmetodane for oppdaging av hendingar

Når ei hending inntreffer i eit system, så må denne sjølvsagt detekterast før programmet kan reagera. Det er **to hovudmetodar** for dette som vist under.

- **Sjekking** ("polling"): I hovudprogrammet eller metodar kalla opp av dette, les ein av relevante register i perifermodular der hendingar inntreffer. Etter lesing testar programmet dei bitane som indikerer hendingane ein vil detektera.
- **Avbrot** ("interrupt"): Ein set her opp perifermodular slik at dei registerbitane som indikerer hendingar, gir avbrotssignal til prosessoren. Det blir då som kjent automatisk starta opp avbrotsmetodar som reagerer på hendingane.

Ein kan ha kombinasjonar av desse to metodane. Ein kan f.eks. ha eit taimeravbrot som gir ein periodisk oppstart av ein avbrotsmetode for taimeren. I denne metoden kan ein f.eks. starta ein analog/digital-omformar (A/D), men der ein aktivt i metoden står og sjekkar biten *Ferdig* i statusregisteret. Når $Ferdig = 1$ kan ein så lesa den omforma verdien.

Sjekking gir ein relativt enkel struktur, men kan leggja beslag på mykje prosessorressursar. Mange avbrotskjelder gir ein meir omfattande struktur, men som kan vera ryddigare og gi mindre arbeid for prosessoren.

Ein skal nå sjå på eit eksempel som viser desse metodane i praksis.

2.4.5.4 Programstruktur basert på sjekking

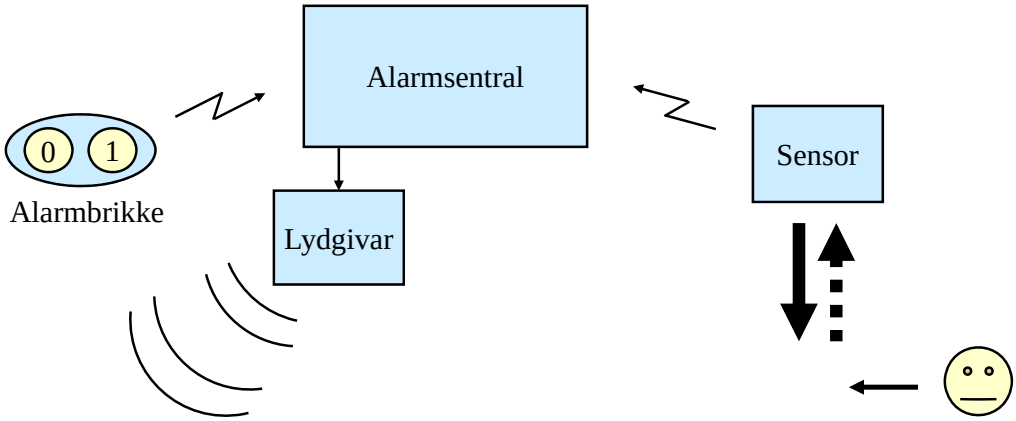
Eksemplet under viser flytskjema for programvare til eit innebygd system der programvaren er basert på sjekking ("polling").

Eksempel 2.23. *Enkelt alarmssystem*

Det skal utviklast eit enkelt alarmsystem. Ønskt oppbygging er vist blokkskjematisk i figur 2.20.

Alarmsentralen inneheld ein mikrokontroller basert på ARM Cortex-M3 og med nødvendige perifermodular samt eksterne komponentar.
Systemet har følgjande funksjonar:

- **Opning og stenging av alarm**
  Vhja. ei **alarmbrikke** kan ein trådlaust opna for alarm ved å trykkja på *1*-knappen og stenga for alarm ved å trykkja på *0*-knappen.
  I eit register i ein kommunikasjonsmodul i mikrokontrolleren, *KM1*, kan programmet lesa den siste knappetrykksverdien som blei mottatt. *KM1* kan også setjast opp til å gi avbrot når ein ny er mottatt.



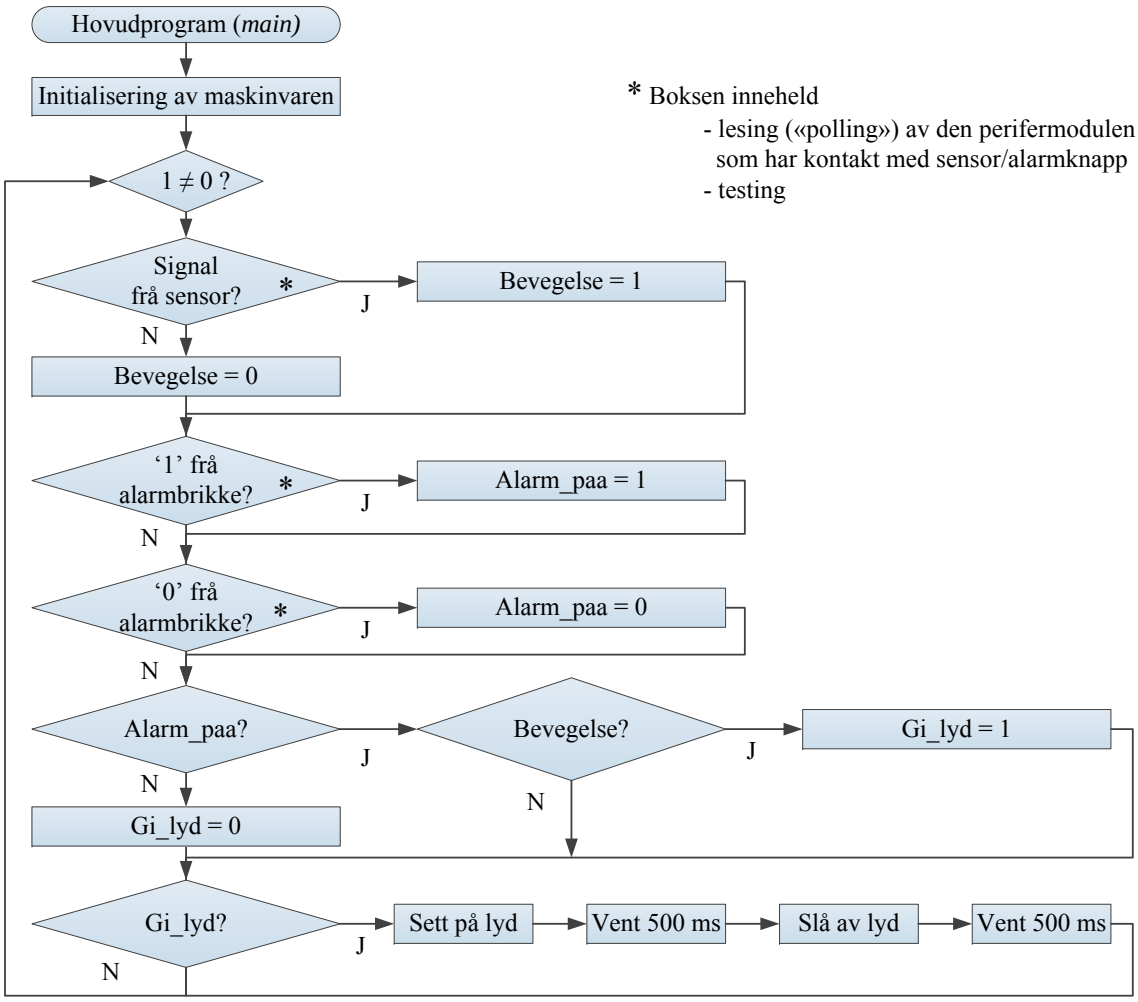
Figur 2.20: Enkelt alarmsystem.

- **Deteksjon av bevegelsar**
  Ein trådlaus sensor sender ei melding når han detekterer bevegelse.
  Eit register i ein annan kommunikasjonsmodul i mikrokontrolleren, *KM2*, vil innehalda verdien *'1'* viss det er mottatt ei slik melding. Etter ei lesing av registeret blir verdien automatisk nullstilt. Modulen kan også setjast opp til å gi avbrot når ei ny melding er mottatt.
- **Signalering av alarm vhja. lyd**
  Ein lydgivar er kobla til ein viss bit b_L i utgangsregisteret til parallellporten (GPIO-modulen) *PP1* i mikrokontrolleren. Når ein set $b_L = 1$, vil lydgivaren gi ein kontinuerleg tone, og $b_L = 0$ slår lyden av.
  Når ein her skal gi alarmsignal, skal det vera ein **pulserande** tone som er vekselvis på og av i 0.5 sekund.

Når det gjeld oppbygging av programvaren for alarmsystemet, skal me først sjå på ei utgåve basert på rein sjekking ("polling"). Eit forslag til mogleg struktur er vist vhja. flytskjema i figur 2.21.

I løpet av eitt omløp av den endelause løkkja i hovudprogrammet får ein utført all sjekking og i tillegg køyrt ein eventuell "lydperiode" som gir lyd i 0.5 sekund og ingen lyd i dei neste 0.5 sekunda.
Akkurat slike langvarige programsekvensar er eitt **hovudproblem** til denne programstrukturen. I dette tilfellet vil brukaren kunne oppleva at responstida på eit knappetrykk blir for lang. Programmet skal altså gjennom to venteløkkjer før det igjen kjem til sjekkesekvensane.
Eit innlysande botemiddel vil vera å få tak i ein lydgivar som gir pulserande lyd. Då ville programmet blitt enklare og responstida kortare. Meir om slike maskinvarevurderingar kjem i neste delkapittel.
I verste fall kan ein i eit system **missa hendingar** når ein køyrer ein slik struktur. I alarmsystemet her vil ein ikkje oppleva dette då kommmunikasjonsmodulane fangar opp meldingane frå omverda.

Eit anna problem med denne strukturen er at **omløpstida** vil **variera** med programvarestørrelsen. Omløpstida vil også vera avhengig av resultatet av dei ulike testane som blir gjort i programvaren. Intervallet mellom to sjekkingar



Figur 2.21: Programstruktur basert på sjekking ("polling").

kan derfor variera mykje. Viss det i eksemplet skal gjevast alarmsignal, blir f.eks. intervallet for sjekking av alarmbrikka lenger enn 1 sekund. Vanlegvis vil slike intervall vera på nokre få mikrosekund!
Det er ikkje så problematisk her, men kan vera veldig kritisk i andre system.

Ei betre løysing her vil vera basert på ein kombinasjon av avbrot og sjekking. Meir om dette kjem i neste kapittel.

2.4.5.5 Eit lite intermesso om å realisera funksjonar i maskinvare kontra programvare

Sidan det nettopp er vist eit eksempel på eit innebygd system, skal me nytta høvet til å sjå litt på korleis ein deler funksjonaliteten til eit system mellom maskinvare og programvare.

Når det gjeld kva funksjonar som skal **realiserast i maskinvare**, har ein i eksemplet over valt løysingar som gir enkel programvare.
Sensor og alarmbrikke er å sjå på som relativt avanserte maskinvarekomponentar då dei realiserer ein kommunikasjonsprotokoll og leverer meldingar.
Eigentleg er sensoren og alarmbrikka også innebygde system då dei vil innehalda kvar sin mikrokontroller mm. Viss desse komponentane var meir primitive, ville programvaren i alarmsentralen bli meir omfattande. Dette systemet ville då blitt meir **programvareintensivt**.

Programvareintensive system gir generelt lenger utviklingstid og dermed **høgare utviklingskostnader**, men **billegare maskinvarekostnadar**. Dette gir totalt sett eit billegare produkt viss **produksjonsvolumet** er stort.

Viss ein bare skal produsera nokre få eksemplar av eit system, svarer det seg å la systemet vera **maskinvareintensivt**. Komponentane bør i størst mogleg grad vera **hyllevare**, og programvareutviklingskostnadane lågast mogleg.

2.4.5.6 Programstruktur basert på eit periodisk avbrot

Ei løysing basert på ein kombinasjon av avbrot og sjekking gir ofte fordelar samalikna med løysinga i kapittel 2.4.5.4. Dette blir igjen vist med eit eksempel.

Eksempel 2.24. *Enkelt alarmssystem med avbrotsbasert programvare.*

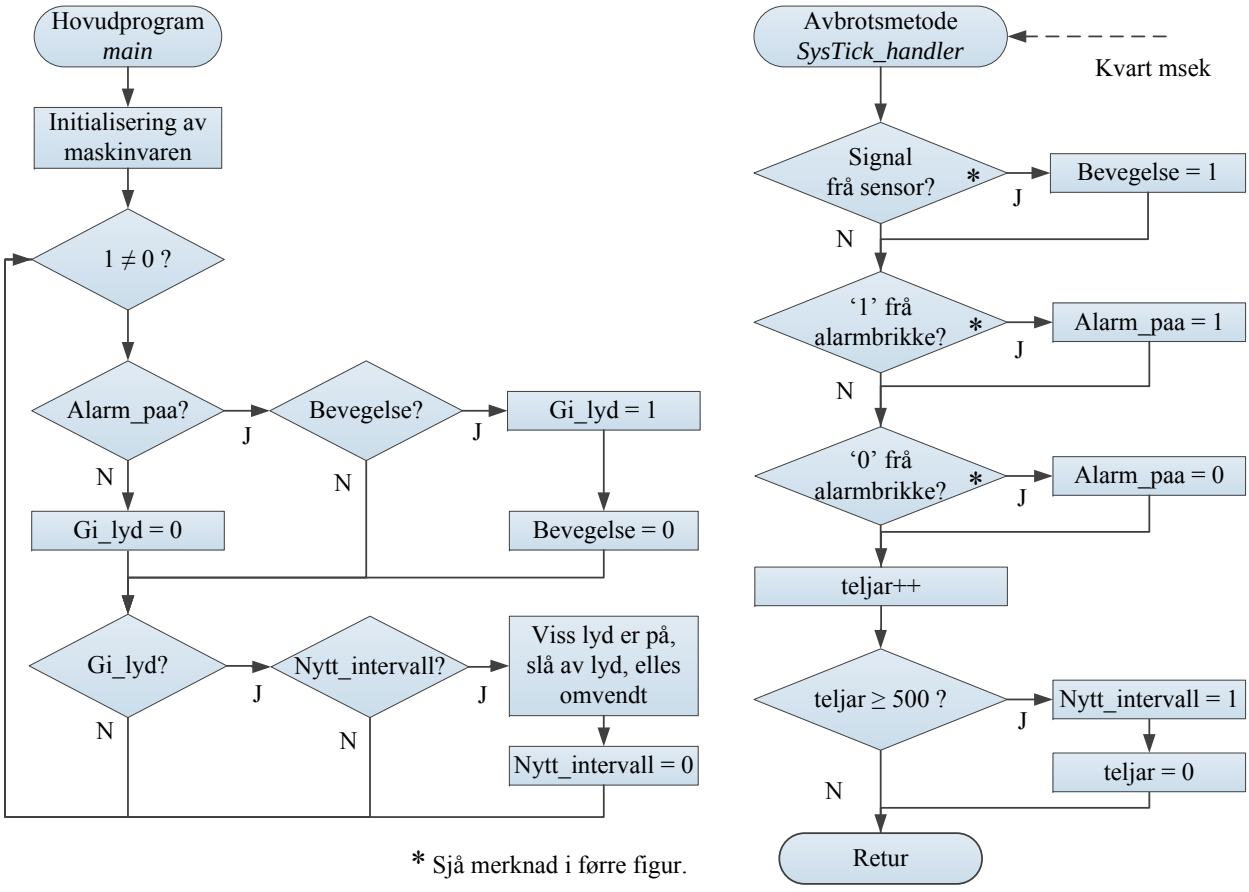
Det enkle alarmsystemet i førre kapittel skal få ein alternativ programstruktur. Målet er at responstida på eit knappetrykk skal bli kortare og at sjekkinga av om det er nye meldingar, skal skje med ein mest mogleg konstant frekvens. For å få til dette, blir det opna for eit periodisk avbrot frå maskinvaren, nærare bestemt taimeren SysTick.
Den resulterande programstrukturen er vist i figur 2.22.

Ein har her sett SysTick-intervallet til 1 millisekund, som er ein vanleg verdi. Sjekkefrekvensen blir då meir enn høg nok her[80]. For å realisera to eller fleire ulike intervall vhja. same avbrotsmetode, er det vanleg å bruka ein teljarvariabel som vist i figuren. Her realiserer ein som vist både intervall på 1 msek og 0.5 sek = 500 msek.

Variablar som feks. *Alarm_paa* som er binære i funksjon, dvs. bare har verdien *0* eller *1*, blir kalla **flagg**.
Dei blir brukt til å **signalisera** hendingar eller tilstandar mellom metodar eller i ein metode i eit programvaresystem.

Her unngår ein altså ventemetodar som gir lang omløpstid og lange sjekkeintervall. All handtering av hendingar i systemet skjer i løpet av kort tid.

[80] Ved sjekking av trykk på tastatur og knappar generelt er vanleg sjekkeintervall 10 msek. Meir om handtering av tastetrykk står i eksempel 3.31 på side 206.



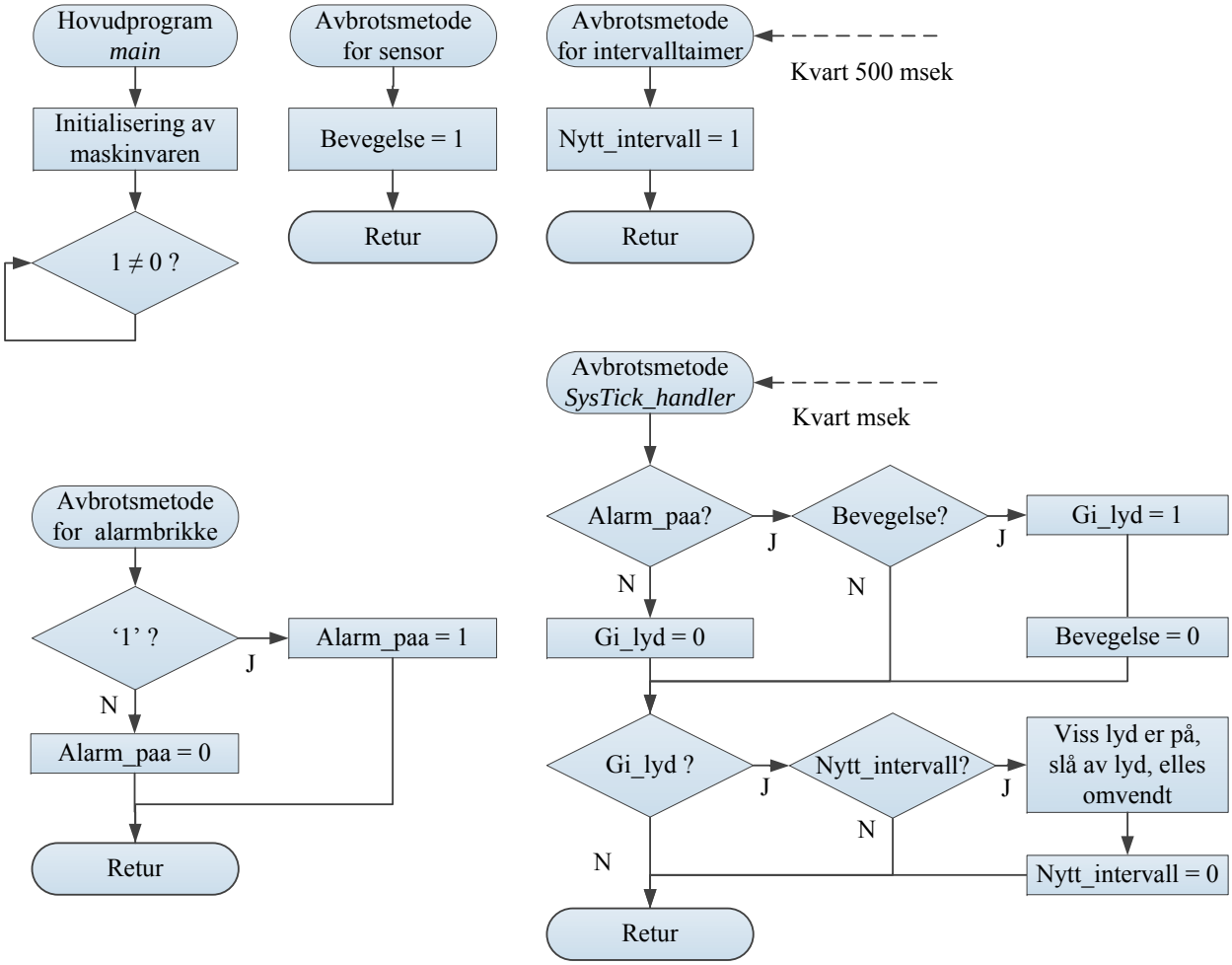
Figur 2.22: Programstruktur basert på periodisk avbrot frå taimeren SysTick.

2.4.5.7 Fullt ut avbrotsbasert programstruktur

I eksemplet frå kapittel 2.4.5.4 skal ein nå erstatta all sjekking av eksterne hendingar med avbrotstyring.

Eksempel 2.25. *Enkelt alarmssystem med full avbrotsstyring.*

Den regelmessige sjekkinga av om det er kome melding frå alarmbrikka eller sensoren i det enkle alarmsystemet skal nå fjernast til fordel for avbrotsstyring. Den resulterande programstrukturen er vist i figur 2.23.



Figur 2.23: Programstruktur med full avbrotsrealisering.

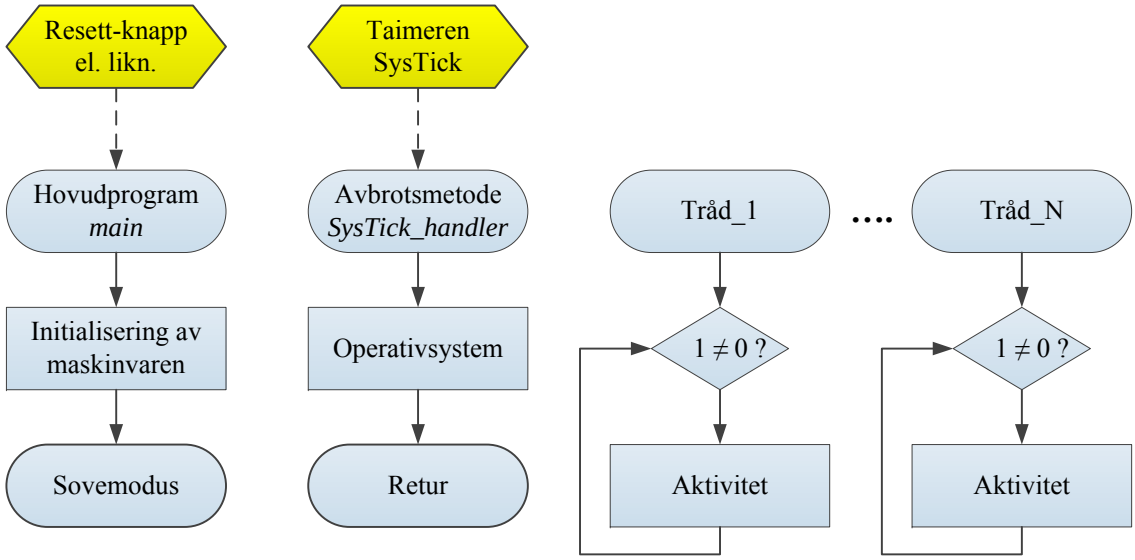
Det er i tillegg som vist, brukt ein eigen taimer for realisering av intervallet på 500 msek.

Programvaren er totalt sett mindre, ryddigare og meir lesbar. Merk at løkkja i hovudprogrammet nå er **tom**, noko som ikkje er uvanleg i slike strukturar.

Ei utfordring er å velja rett prioritet for dei ulike avbrota.
Alle avbrotsmetodane er korte. I tillegg har ein ingen sterke krav til variasjonane i lydintervallet eller reaksjonstidene på meldingar frå alarmbrikka og sensoren. Prioriteten er derfor ukritisk for akkurat dette systemet.

2.4.5.8 Litt om programstruktur for eit sanntids operativsystem

I større innbygde system der ein har mange aktivitetar ("task") og av ulike typar, vil ein ofte bruka eit **sanntids operativsystem** ("real-time operating system", RTOS). Parallelle aktivitetar blir her organiserte som **trådar** ("thread").
Dette er illustrert i figur 2.24.



Figur 2.24: Typisk trådbasert programstruktur.

Ofte blir alt det som føregår i ein tråd, definert som **ein** aktivitet, og ein vil også sjå i delar av litteraturen på dette feltet at heile tråden blir kalla ein aktivitet ("task").
Av dette kjem også uttrykket **"multi-tasking"** som betyr **parallellprosessering**.
Parallellprosesseringa er generelt bare tilsynelatande viss då ikkje mikroprosessoren har fleire kjernar[81]. Prosessortid blir delt mellom trådane, og det er operativsystemet (OS) som gjer dette.

Etter ein omstart vil hovudprogrammet først initialisera systemet. Siste del av initialiseringa vil vera å opna for avbrota. Hovudprogrammet vil så her gå over i sovemodus.

[81] Her kan nokre av trådane køyrast i ekte parallell, men vanlegvis ikkje alle. Å parallellisera programvaren slik at ein verkeleg kan utnytta fleire kjernar, er ei stor utfordring.

Systick-taimeren vil i eit typisk system gi avbrot kvart millisekund. Den tilhøyrande avbrotsmetoden som køyrer OS-et, vil gjera følgjande:

- Avgjera kva tråd som skal køyra.
- Tildela **køyretid** til dei ulike trådane etter ein fastsett **plan** ("schedule").
- Handtera **ressursar** som er felles for trådane.
- Handtera kommunikasjon mellom og synkronisering av trådane. Dei enkle flagga som er omtalt til slutt i kapittel 2.4.6.2, er her erstatta av **semaforar**[82], som er ein avansert synkroniseringsmekanisme.

Det kan elles nemnast at kvar tråd har sitt eige stakkområde slik at han kan halda fram der han slapp når operativsystemet tildeler ny køyretid. Meir om dette og sanntids operativsystem generelt står i kapittel 16 hjå Brown, [2].
Ein kan også finna meir stoff bl.a. hjå "en.wiki Real-time operating system".
Brown bruker FreeRTOS som eksempel. Ei omfattande liste over slike operativsystem er gitt hjå "en.wiki List of real-time operating systems".

[82] Sjå "en.wiki Semaphore (programming)".

2.4.6 Meir om avbrotshandtering

I kapittel 2.4.4.11 såg ein på korleis ARM Cortex-M3 handterer avbrot. I tillegg viste ein kva som må vera ordna for at ein avbrotsmetode skal starta etter at ein maskinvaremodul har gitt eit avbrotssignal.

I det nemnde kapitlet blei taimeren SysTick brukt som eksempel. I mikrokontrolleren vil dei fleste perifermodulane kunne setjast opp til å gi avbrot når eit eller annna hender i modulen. Det er knytta ein eller fleire **avbrotsmetodar** ("handler") til kvar av perifermodulane.
I CMSIS-fila *startup_stm32f10x_md_vl.c* som inneheld sjølve **oppstartsmetoden**[83] for mikrokontrolleren vår, kan ein finna deklarasjonar[84] av alle avbrotsmetodane. Desse er viste under.

```
Reset_Handler();                    NMI_Handler();
HardFault_Handler();                MemManage_Handler();
BusFault_Handler();                 UsageFault_Handler();
SVC_Handler();                      DebugMon_Handler();
PendSV_Handler();                   SysTick_Handler();
WWDG_IRQHandler();                  PVD_IRQHandler();
TAMPER_IRQHandler();                RTC_IRQHandler();
FLASH_IRQHandler();                 RCC_IRQHandler();
EXTI0_IRQHandler();                 EXTI1_IRQHandler();
EXTI2_IRQHandler();                 EXTI3_IRQHandler();
EXTI4_IRQHandler();                 DMA1_Channel1_IRQHandler();
DMA1_Channel2_IRQHandler();         DMA1_Channel3_IRQHandler();
DMA1_Channel4_IRQHandler();         DMA1_Channel5_IRQHandler();
DMA1_Channel6_IRQHandler();         DMA1_Channel7_IRQHandler();
ADC1_IRQHandler();                  EXTI9_5_IRQHandler();
TIM1_BRK_TIM15_IRQHandler();        TIM1_UP_TIM16_IRQHandler();
TIM1_TRG_COM_TIM17_IRQHandler();    TIM1_CC_IRQHandler();
TIM2_IRQHandler();                  TIM3_IRQHandler();
TIM4_IRQHandler();                  I2C1_EV_IRQHandler();
I2C1_ER_IRQHandler();               I2C2_EV_IRQHandler();
I2C2_ER_IRQHandler();               SPI1_IRQHandler();
SPI2_IRQHandler();                  USART1_IRQHandler();
USART2_IRQHandler();                USART3_IRQHandler();
EXTI15_10_IRQHandler();             RTCAlarm_IRQHandler();
CEC_IRQHandler();                   TIM6_DAC_IRQHandler();
TIM7_IRQHandler();
```

Startadressene til desse ligg på kvar sine faste plasser i vektortabellen, som nemnt tidlegare.

[83] Om oppstartsmetoden står det litt i slutten av kapittel 2.4.2.4.

[84] Desse er deklarerte som *WEAK*. Dette vil seia at viss du deklarerer ein avbrotsmetode på nytt i programvaren din, vil dette overstyra den første deklarasjonen.

Nå skal me sjå meir generelt på oppførselen til ein avbrotsbasert programstruktur men vil bruka eit par av avbrotsmetodane over i nokre eksempel.

2.4.6.1 Tidsdiagram for ein programstruktur basert på avbrot

I figur 2.18 på side 88 blir det vist kva operasjonar som blir utførte av mikroprosessoren når eit avbrot blir handtert.
Programstrukturen for dei fleste innebygde system vil vera avbrotsbasert, og ofte vil det i desse systema vera fleire **avbrotskjelder**. Ein har då som nemnt i kapittel 2.4.4.11, eit nesta avbrotssystem. Det er då viktig å velja rett **prioritet** for dei ulike avbrota. **Kritiske** og/eller **raske**[85] avbrotsmetodar bør ha **høg prioritet** og omvendt. Nokre eksempel:

- Viss det f.eks. er kritisk at **intervallet** mellom kvar oppstart av metoden *SysTick_handler()* er mest mogleg **konstant**, må SysTick-avbrotet ha høg prioritet. Prosessoren kan då bryta av ein eventuell annan metode som køyrer når SysTick-avbrotet kjem.
- Viss ein lang metode har høgare prioritet enn f.eks. ein kommunikasjonsmetode, kan ein risikera at kommunikasjonsmetoden ikkje får lese mottatte data raskt nok. Ein **risikerer** då **overskriving** av data i mottaksbufferet.

Programstruktur og prioritetar må vurderast basert på systemspesifikasjonen, og vil variera frå tilfelle til tilfelle.

Når det gjeld handteringa av avbrota i eit **nesta** system, kan dette illustrerast vhja. tidsdiagram som vist i eksemplet under.

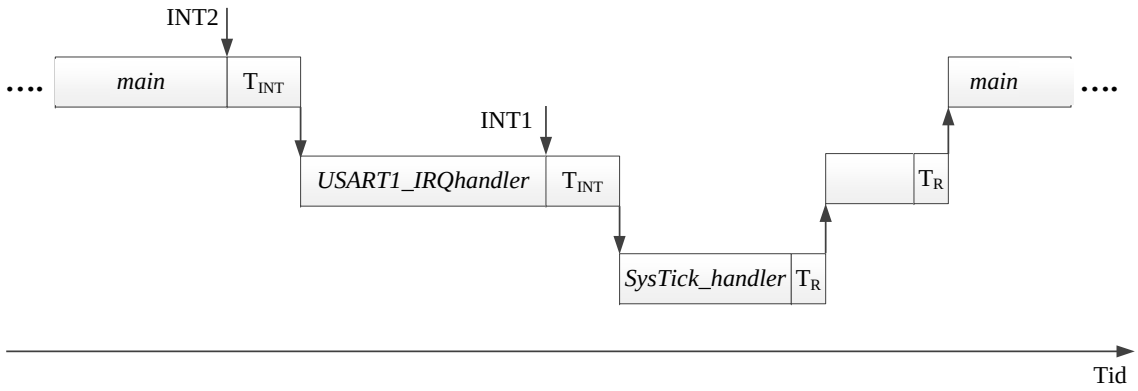
Eksempel 2.26.

I eit visst innebygd system får prosessoren følgjande avbrot:

- *INT1*: Dette avbrotet kjem frå SysTick-taimeren og er her gitt prioritet 1. Avbrotet er periodisk og fører til oppstart av avbrotsmetoden *SysTick_Handler()*. Utføringstida for metoden er i dette systemet maksimalt 200 mikrosek.
- *INT2*: Kjelda for dette er serieporten *USART1* som gir avbrot både når ein databyte er mottatt og når det er klart for sending av ein ny byte. Ingen av delene skjer oftare enn kvart 1 millisek. Prioriteten er ut frå opplysningane i begge punkta sett til 2, altså **lågare** enn for *INT1*.

Ein tenkt, men vanleg situasjon er vist i figur 2.25. Ein avbrotsmetode blir her avbroten av ein annan med **høgare** prioritet.

[85] Det er viktig å laga metodane slik at utføringstida ikkje er for lang.



Figur 2.25: Tidsdiagram som viser handtering av nesta avbrot.

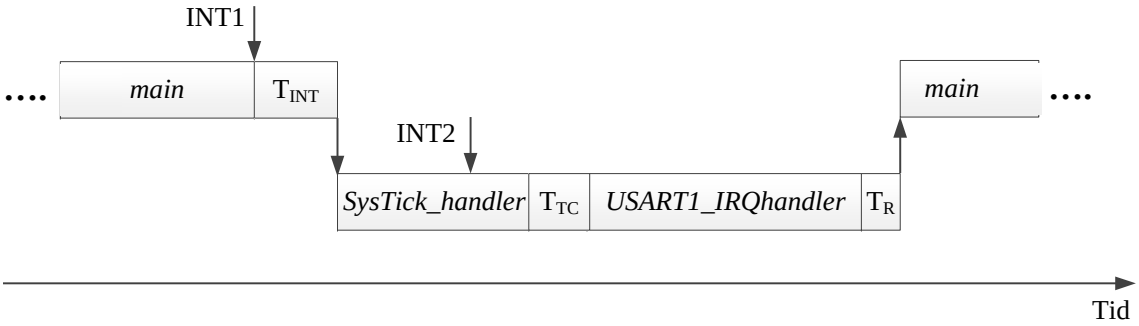
Som vist i kapittel 2.4.4.11, bruker prosessoren litt tid før ein avbrotsmetode blir starta opp. Dette er markert i diagrammet med tida T_{INT}.
I løpet av denne tida blir

- pågåande instruksjon i *main* fullført og
- operasjonane i)-iii) i figur 2.18 på side 88 utførte.

T_{INT} er typisk lik 12 klokkesyklar, men kan også vera lenger[86].

Etter at alle dei ventande avbrotsmetodane er utførte, i dette tilfellet to, vil det koma eit nytt etterslep. Denne returtida, T_R, dekkjer returoperasjonen i) i figur 2.18. T_R vil vera kortare enn T_{INT}.

Ein like vanleg situasjon er vist i figur 2.26. Ein avbrotsmetode er under handtering når det kjem eit avbrot med same eller som her, **lågare** prioritet. Pro-



Figur 2.26: Tidsdiagram som viser handtering av påfølgjande avbrot.

sessoren vil her fullføra den avbrotsmetoden som blir køyrt. Når ein så går rett frå ein avbrotsmetode til neste, treng ein ikkje henta og så lagra stakkramma til den opprinnelege metoden, som her er *main()*. Etterslepet ("latency") blir derfor kortare her. Denne situasjonen blir kalla **haleheng** ("tail-chain", TC)[87].

[86] Sjå kap. 8.3.1 hjå Yiu, [3].
[87] Sjå kap. 8.3.3 hjå Yiu, [3].

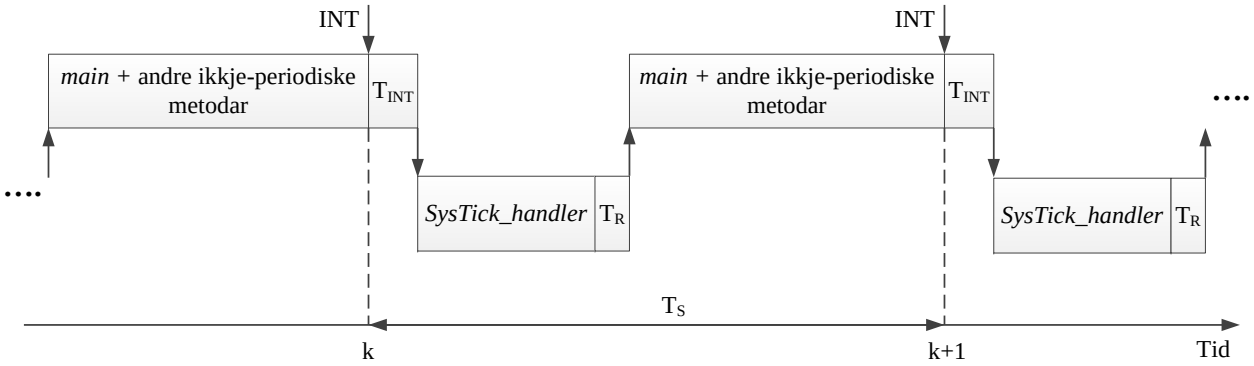
Halehenget er markert i diagrammet med tida T_{TC}.
T_{TC} er typisk lik 6 klokkesyklar, men kan også som Yiu viser, vera lenger.

2.4.6.2 Litt om programstrukturar basert på periodiske avbrot

Det er vanleg å bruka periodiske avbrot i innebygde system. Det kan f.eks. vera krav om at
- måleinstrument skal lesast og styresignal skal genererast regelmessig,
- at knappar i eit brukargrensesnitt må sjekkast med ein viss frekvens eller at
- ein regelmessig skal veksla mellom ulike trådar i eit sanntids operativsystem (RTOS).

I førstnemnde tilfellet blir tida mellom to lesingar kalla **sampleintervall** eller **syklustid**.
I sistnemnde tilfellet blir ofte tida kalla eit **tikk** ("tick" eller "tick time")[88] og er vanlegvis lik tidsintervallet mellom to avbrot frå systemtaimeren i ein datamaskin.

I ARM-baserte system er det SysTick som er **systemtaimer**, og det er vanleg å realisera periodiske aktivitetar vhja. denne.
I figur 2.27 er det vist eit tidsdiagram for eit slikt system.



Figur 2.27: Tidsdiagram for ein programstruktur basert på eit periodisk avbrot.

Utgangspunktet er ofte at ein har eit krav til sampleintervallet T_s.
Ein ser i figuren at mellom tidspunkt $k \cdot T_s$ og $(k+1) \cdot T_s$ må det vera plass til
- utføring av alle dei periodiske aktivitetane, dvs. heile *SysTick_handler()*,
- litt av hovudprogrammet og andre ikkje-periodiske aktivitetar samt
- avbrotshandtering.

Viss dette ikkje går i hop, må ein velja ein plattform med høgare klokkefrekvens eller sjå om ein kan optimalisera programstrukturen og programkoden ytterlegare.
Ein kan også vurdera å distribuera/fordela aktivitetane på fleire mikrokontrollerar eller realisera delaktivitetar som rein maskinvare[89].

[88] Både "en.wiki jiffy (time)" og "en.wiki system time" gir meir om dette.
[89] Meir om dette kjem i kapittel 3 om digitalteknikk.

2.5 Ei lita oppsummering

Det er nå gitt ei innføring i kva ein datamaskin er. Hovudtrekka i utvikling, oppbygging og virkemåte er vist med ein ARM-basert mikrokontroller som eksempel. Ein har nå eit nødvendig grunnlag for å kunne laga enkle program for mikrokontrolleren vår på laboratoriet og kan då sjå litt av potensialet til slike små datamaskinar.

Det er likevel heilt sikkert alt nå detaljar som ikkje er så lette å forstå då ein ikkje har det nødvendige grunnlaget i digitalteknikk. Dette grunnlaget prøver ein å gi i dei neste kapitla.
Etterpå vil ein venda tilbake til mikrokontrolleren og modulane i denne slik at ein kan laga meir avanserte koblingar mot eksterne komponentar og program for desse.

Kapittel 3

Digitalteknikk

3.1 Bakgrunn

3.1.1 Litt om ordet *digital*

Eit **digitalt** system er eit system som opererer med **diskrete** verdiar. Det motsette er eit **analogt** system der verdiane tilhøyrer eit kontinuerleg eller analogt verdiområde.
Ordet *digital* kjem av det latinske *digitus* som betyr *finger*. Fingrar har vore brukte til teljing i uminnelege tider. Koblinga til det som seinare skulle føregå i ein datamaskin er tydeleg. Ordet *digit*, som betyr *siffer* på norsk, er av same opphav.

3.1.2 Kvifor datamaskinen er binær

Som vist i kapittel 1.1.1 om datamaskinen si utvikling, så blei transistoren mellom anna ein erstattar for releet, som var ein dominerande byggjestein i f.eks. automatiserte telefonsentralar og i dei aller første datamaskinane.
Brytarar og rele har som kjent **to tilstandar**, open og lukka. Transistoren brukt som releerstattar har også to tilstandar[1]. Det var difor naturleg å basera både datalagring og databehandling på **totalssystemet**, også kalla det binære talsystemet.

[1] Det er ikkje ennå utvikla teknologi for bruk i digitale kretsar som kan halda ein transistor i fleire enn to tilstandar på ein robust måte. Det er likevel mogleg å realisera tre tilstandar på ei signallinje. Dette gir større informasjonskapaitet over linja og heiter **bipolar enkoding**. Dette blir brukt i høgratekommunikasjon som f.eks. 100 Mbit/sek Ethernett, sjå f.eks. "en.wiki MLT-3".

3.1.3 Kva skal ein med digitalteknikk?

Dei fleste ingeniørar er deltakarar i utviklingsprosjekt. Eit prosjekt dreier seg sjeldan bare om eit smalt fagfelt som alle deltakarane er ekspertar på, f.eks. Java-programmering eller bygging av motorstyringssystem. Dei fleste prosjekta er **tverrfaglege**.
For at prosjektresultatet skal bli bra, er det viktig at deltakarane forstår kvarandre.
Prosjekteksempel kan vera utvikling av eit datamaskinsystem for ein minibank eller for ein bilvaskemaskin. Viss maskinvaren skal halda mål, er det viktig at dei som programmerer, er i stand til å spesifisera behov for datakraft og minnekapasitet og at dei kan gjera seg forstått i samtalar med maskinvarefolk.
Likeeins er det viktig at maskinvarefolka har innsikt i programmering, operativsystem og systemdrift for å kunne utforma maskinvaren på rette måten.
Å bare la andre gjera ting for seg utan at ein sjølv forstår noko av dette, er utrygt og lite optimalt. Det kan i verste fall føra eit prosjekt inn på eit blindspor, noko som kostar tid og pengar.

Det er altså viktig for både data- og elektroingeniørar å ha ein viss felles kompetanse på både program- og maskinvare.
Ofte startar eit utviklingsprosjekt med ei vurdering av kva systemfunksjonar som skal løysast av programvare og kva som skal realiserast av rein maskinvare.
Ved prosessering av data frå f.eks. ein radar, kjem nye data så ofte at ein prosessor ikkje har kapasitet til å ta seg av dette. Ein byggjer då digital elektronikk i ein programmerbar elektronisk krets, FPGA[2], som siler ut det viktige i datastraumen. Dette gir ein redusert datastraum som kan vidarebehandlast av ein mikroprosessor.

Ved behandling av grafisk informasjon, dvs. bilete, gjer ein f.eks. i PC-verda bruk av ein skreddarsydd prosessor for dette. Denne grafikkprosessoren har tilgang på eit eige minne for dei biletdata som står for tur til å bli behandla og køyrt ut på skjermen. For brukaren framstår dette som ein rein maskinvare-akselerator[3]. I andre tilfelle kan det vera snakk om enklare funksjonar som med fordel kan realiserast som maskinvaremodular i programmerbar elektronikk og som då avlastar prosessoren. Dette kan vera filtrering av digitale signal frå brytarar og digitale sensorar, filtrering av data frå analog- til digitalomformarar, køyring av raske sekvensielle og repeterande signalforløp for motorstyring med meir.

Kompetanse på både maskin- og programvare gjer at ein lettare kan ta dei rette val sjølv og saman med andre i eit prosjekt.
Sjølv om ein arbeider med programvare på eit høgt nivå, dvs. langt frå maskinvare, vil ein ofte ha nytte av maskinvarekunnskapane.

[2] Field Programmable Gate Array. Meir om slike kjem i kapittel 3.7.1.3.
[3] Sjå f.eks. "en.wiki Hardware acceleration".

3.2 Hovudgrupper av digital logikk

Eit digitalt system blir realisert vhja. digital elektronikk, og denne er igjen sett saman av ein eller fleire digitale kretsar. Mikrokontrolleren er eit eksempel på ein krets som i hovudsak er digital.
Digital elektronikk blir ofte også kalla **digital logikk**, då byggjesteinane i digitale kretsar er såkalla **logiske portar** ("gates"). Meir om dette kjem ganske snart.

Digital logikk kan delast opp i to hovudgrupper:

- Kombinatorisk logikk.
- Sekvensiell logikk.

Dei neste underkapitla startar med eksempel på kvar av desse hovudgruppene.

3.2.1 Kombinatorisk logikk

Me startar med eit eksempel som illustrerer ein reint kombinatorisk funksjon.

Eksempel 3.1. *Dørlyslogikk*

I instrumentpanelet til ein bil med to dører og bagasjeluke er det eit lys som skal styrast av dørene.
Når minst ei av dørene eller bagasjeluka er open, skal dørlyset vera på.
Ein meir detaljert spesifikasjon av dette logiske styresystemet kan vera som følgjer:

Inngangar:
Signal frå dei to dørene og bagasjeluka som blir kalla *Dør1*, *Dør2* og *Dør3*.

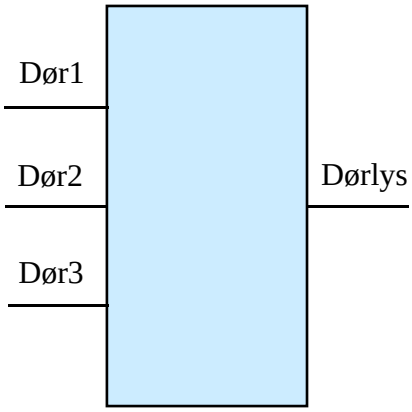
Utgang:
Lyset *Dørlys* i instrumentpanelet.

Funksjon:
Viss minst ei av dørene *Dør1*, *Dør2* eller *Dør3* er open, skal *Dørlys* vera på.

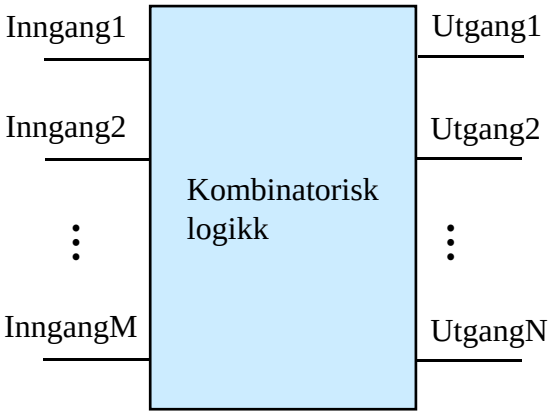
Systemet er illustrert i figur 3.1.

Funksjonen kan realiserast med ei kobling av såkalla **kombinatorisk** logikk. Utgangane er då ein direkte **kombinasjon** av **bitmønsteret** på inngangen, dvs. nivået eller verdien til alle inngangane.

Generelt vil eit kombinatorisk system ha M inngangar og N utgangar som vist i figur 3.2.



Figur 3.1: Dørlyslogikk



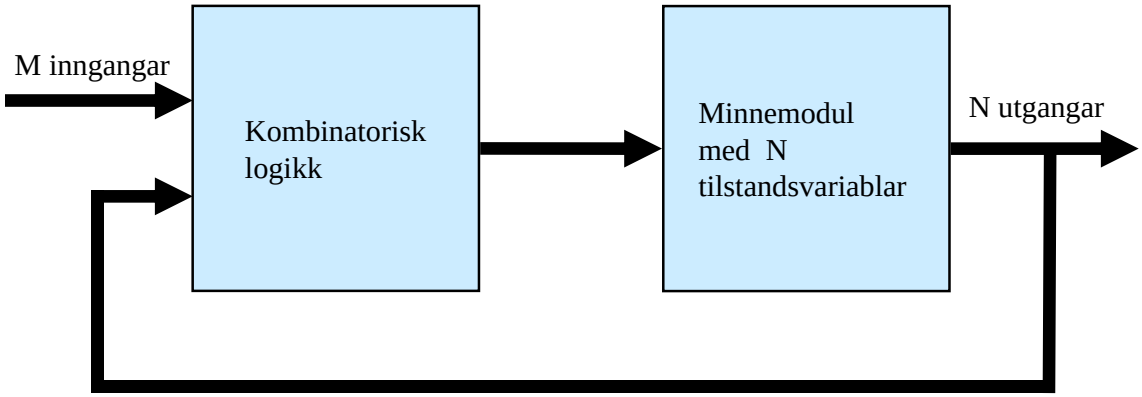
Figur 3.2: Generelt kombinatorisk system.

Me skal i kapittel 3.5 sjå nærare på korleis innsida av eit slikt system kan realiserast.

3.2.2 Sekvensiell logikk

I sekvensielle logiske system vil utgangssignala, som namnet seier, gå i ein **sekvens**. Kva verdiar utgangssignala skal ha i neste steg av sekvensen, er ein kombinasjon av verdiane til inngangssignala og i tillegg verdiane som utgangssignala har nå, dvs. på det nåverande steget av sekvensen.
Eit generelt sekvensielt system er vist i figur 3.3.

Verdiane som utgangssignala har nå, blir ofte kalla **tilstanden** (”state”) til systemet. For å kunne realisera eit slikt system, må altså logikken i tillegg til ein kombinatorisk modul ha ein **minnemodul** som hugsar eller lagrar den nåverande tilstanden.



Figur 3.3: Generelt sekvensielt system.

Det er to hovudtypar sekvensiell logikk:

- **Asynkron**: Tilstandsendringar skjer når inngangssignala endrar seg. Tida mellom slike endringar eller steg er då generelt varierande.
- **Synkron**: Tilstandsendringar går i en fast takt styrt av eit klokkesignal. Tida mellom stega er altså konstant. Dette er den vanlegaste typen sekvensiell logikk.

Eksemplet under kan illustrera dette nærare.

Eksempel 3.2. *Blinkande lys*

Eit system skal gi ut eit lys som blinkar med ein gitt **frekvens**. Lyset skal altså vera **på** i eit gitt **blinkintervall** og **av** i like lang tid[4]. For å kunne få til dette, må systemet kunne generera signal med spesifiserte periodetider. I digitale system kan ein realisera dette med ein **teljar** ("counter"). Ein teljar går i ein takt bestemt av eit klokkesignal med frekvens f_{clk} og er i seg sjølv eit sekvensielt system. Viss teljaren skal telja opp, og utgangsverdien, dvs. tilstanden nå, f.eks. er 5, skal neste tilstand vera lik 6, osv.

Maksimal teljeverdi for ein teljar på n bit er som kjent $2^n - 1$. Når teljaren har nådd maksimalverdi, startar han frå null igjen. Viss ein lar blinkinga bli styrt av nivået på den mest signifikante biten (MSb) til teljaren, kan ein visa at

$$blinkintervall = \frac{1}{2} \cdot blinkperiode = \frac{1}{2}[2^n \cdot (\frac{1}{f_{clk}})] = 2^{(n-1)} \cdot t_{clk}$$

der t_{clk} er periodetida for klokkesignalet.

Viss systemklokkefrekvensen er gitt, er det altså talet på bit i teljaren som bestemmer blinkintervallet.
Meir om korleis teljarar er oppbygde og virkar kjem i kapittel 3.9.

[4]Ein har altså at $blinkintervall = 0.5 \cdot blinkperiode = 0.5/frekvens$.

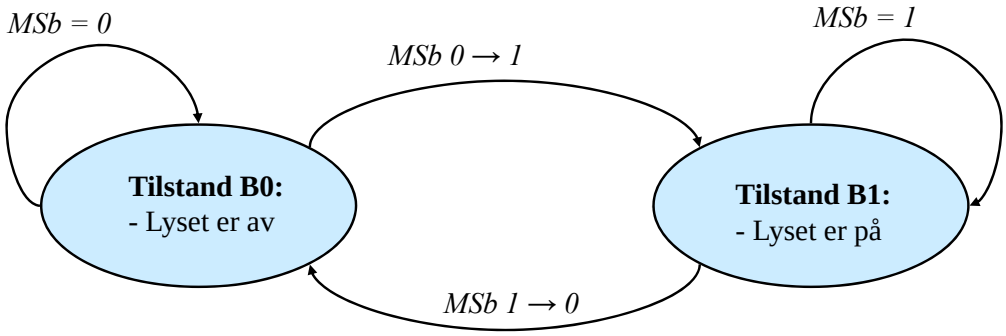
Blinklyssystemet over kan realiserast vhja. sekvensiell logikk[5] som kan spesifiserast slik:

Utgang:
Lys.

Tilstandar:
B0 - Lyset er av.
B1 - Lyset er på.

Funksjon:
Viss tilstand B0 har vart i ei tid lik blinkintervallet, skal ein gå over i tilstand B1.
Viss tilstand B1 har vart i ei tid lik blinkintervallet, skal ein gå over i tilstand B0.

Funksjonen til systemet er vist vhja eit **tilstandsdiagram** i figur 3.4.



Figur 3.4: Tilstandsdiagram for blinklys.

For at ein skal få ein **transisjon** eller overgang mellom to tilstandar, må generelt eitt eller fleire **vilkår** ("condition") vera oppfylte.
I figur 3.4 er vilkåret for transisjon at den mest signifikante biten (MSb) i teljaren endrar verdi.
Vilkår er markerte i *skråskrift*.

Me tar eit eksempel til på eit sekvensielt system. Dette systemet byggjer på blinksystemet i eksemplet over.

[5]Ein kan sjølvsagt også realisera dette systemet vhja. av programvare i ein mikrokontroller. Mikrokontrolleren er eit eksempel på eit avansert sekvensielt logisk system.

Eksempel 3.3. *Lysstyring i ei sykkellykt*

Ei typisk batteridriven sykkellykt, her baklys, er vist i figur 3.5.

Figur 3.5: Baklys for sykkel. (Ref.: "Wiki RearLight1.jpg").

Styringa av lyset i sykkellykta kan realiserast vhja. sekvensiell logikk som igjen kan spesifiserast slik:

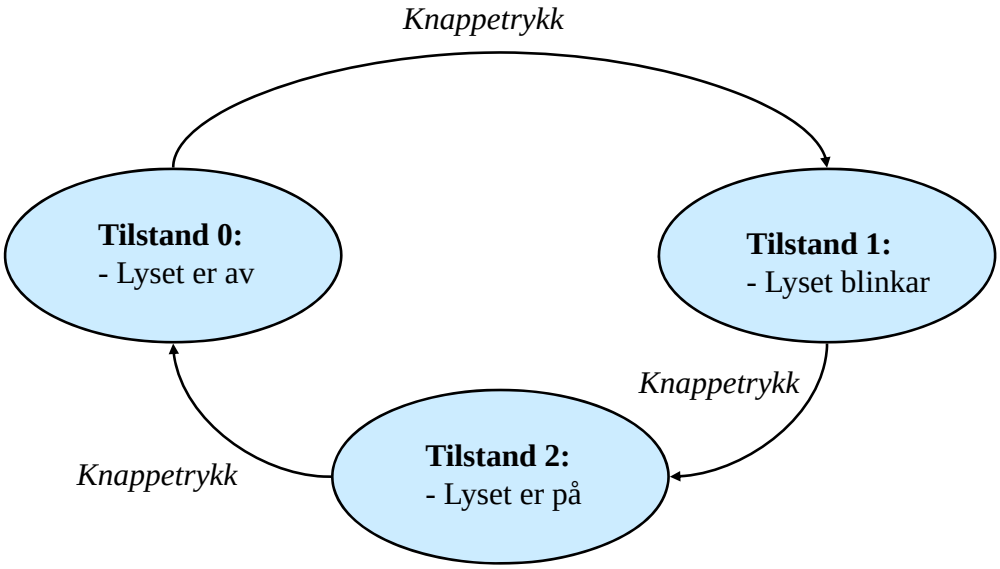
Inngang:
Trykknapp.

Utgang:
Lys.

Tilstandar:
0 - Lyset er av.
1 - Lyset blinkar med ein gitt frekvens.
2 - Lyset er på, dvs. lyser kontinuerleg.

Funksjon:
For kvart nytt knappetrykk går ein over i neste tilstand.

Dette er illustrert i figur 3.6.



Figur 3.6: Tilstandsdiagram for sykkellykt.

> Figuren viser oppsettet av den overordna styringa. Når systemet er i tilstand 1, skal blinkinga styrast slik som vist i figur 3.4. Systemfunksjonane kan altså her spesifiserast vhja. eit **hierarki** av tilstandsdiagram.

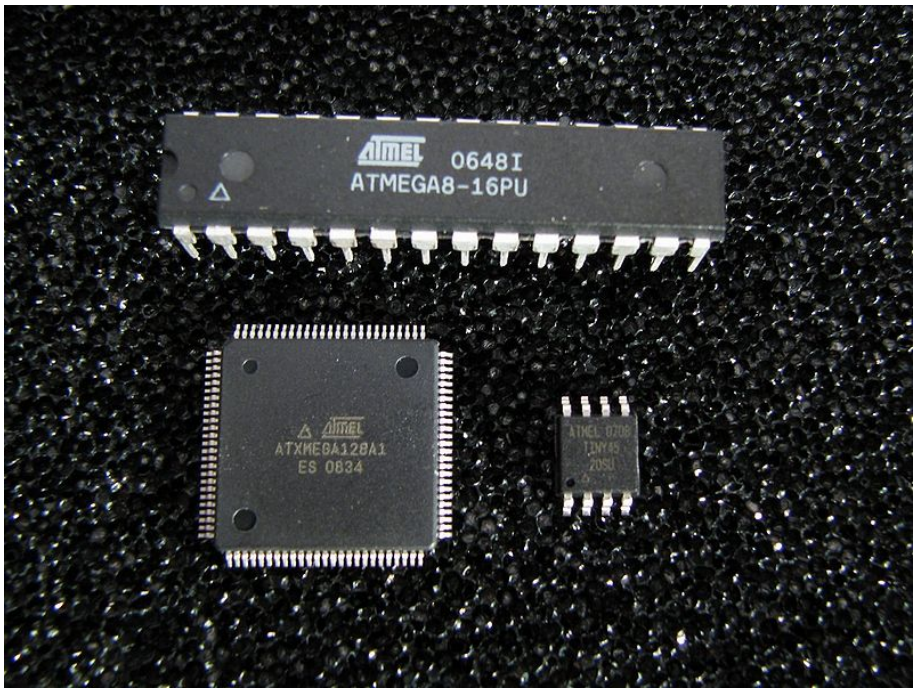
Dei fleste **automatiske** system er sekvensielle[6], og det er veldig vanleg å spesifisera funksjonane til slike vhja. tilstandsdiagram. Slike system blir ofte kalla **tilstandsmaskinar**. Mikroprosessoren er f.eks. ein avansert tilstandsmaskin.
Ved køyring av program går han gjennom følgjande tilstandar:

→ Henting av ny instruksjon → Dekoding av instruksjon → Utføring av instruksjon → Henting.... osv.

Når det gjeld tilstandsmaskinar, kan funksjonane **realiserast** som ei blanding av maskinvare, dvs. digital logikk, og programvare køyrt av ein mikroprosessor. Utfordringa er som sagt tidlegare å finna den optimale blandinga av maskin- og programvare.
Derfor vil ein møta konseptet tilstandsmaskin og tilstandsdiagram både i programmerings- og digitalteknikkundervisning, og i begge tilfelle finst det eksempel på gode utviklingsverktøy der det er mogleg å laga programkode eller maskinvareoppskrifter med basis i tilstandsdiagram.

Styringa av ei vanleg sykkellykt som vist i figur 3.5, er relativt enkel. Viss ein opnar ei slik lykt, vil ein likevel overraskande nok finna at styringa blir realisert vhja. ein mikrokontroller! Ein vanleg mikrokontroller her er den minste medlemmen av ATMEL sin AVR-serie, nemleg Tiny[7]. Han er vist nede til høgre i figur 3.7.

[6] Tenk f.eks. på ein vaskemaskin for bilar.
[7] Sjå "en.wiki Atmel_AVR_ATtiny_comparison_chart".



Figur 3.7: Nokre mikrokontrollerar av typen AVR (Ref.: "Wiki AVR_group.jpg").

Utviklinga har som nemnt tidlegare vore kolossal, og med prisar som nå er så låge[8] at ein kan forsvara å ha ein liten datamaskin inne i kvar ei sykkellykt.

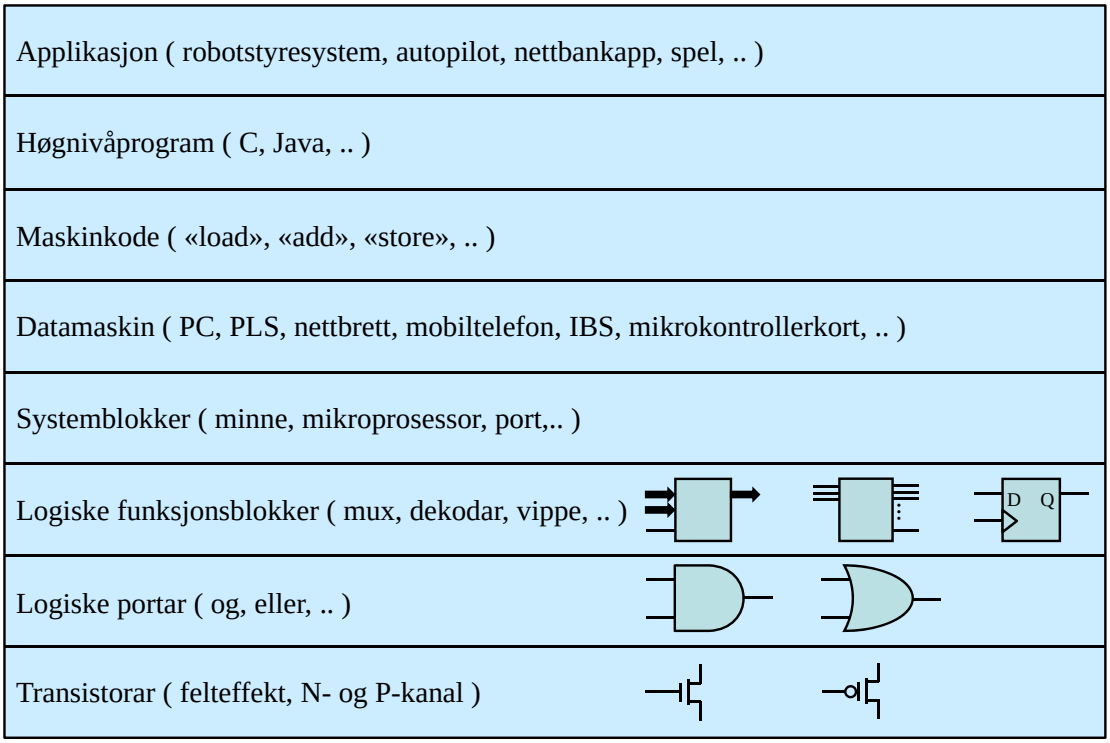
I ei laboratorieøving i emnet Datamaskinarkitektur skal ein realisera sykkellyktstyringa frå eksempel 3.3 på side 111 vhja. plattformkortet vårt, *STM32VLDiscovery*.
Eit alternativ er å realisera denne styringa i programmerbar logikk og utan bruk av ein mikroprosessor. Denne reine maskinvareløysinga blir vist heilt i slutten på dette kapitlet, og vil byggja på mykje av det stoffet som nå skal presenterast.

[8]Ein AVR Tiny mikrokontroller med 8 pinnar, teljar, 6 I/O-linjer, program- og dataminne kan ein få frå rundt 3 kr. i større antal, dvs. 500+, hjå f.eks. ein leverandør som UiS brukar, sjå "Farnell.no".

3.3 Digitalt hierarki

For å realisera digital logikk, trengst ulike byggjesteinar. Den minste er **transistoren**. Sjølvsagt er denne igjen bygt opp av mindre og ulike slags byggjesteinar, men for å forstå korleis datamaskinen virkar, er det nok å starta med transistoren som komponent.

Figur 3.8 viser eit hierarki av byggjeblokker som er nødvendige for å kunne køyra datamaskinbaserte **applikasjonar**, f.eks. eit styresystem for ein lakkeringsrobot.
Under køyring vil millionvis av transistorar inne i datamaskinen slå seg på og av i bestemte mønster og sekvensar for å realisera den spesifiserte systemoppførselen.



Figur 3.8: Digitalt hierarki: Frå transistor til applikasjon.

Transistorane er grupperte i små logiske einingar som ein kallar **portar**. Portane utfører logiske operasjonar på digitale signal, dvs. såkalla OG- og ELLER-operasjonar mfl..
Portane er igjen grupperte i ulike blokker som har ulike **grunnleggjande funksjonar**. Det kan vera minneelement/vippe, kanalveljar/dekodar, sporvekslar/multipleksar (”mux”) for datastraumar og adderar mm.

Systemblokker som mikroprosessor, minne, parallell- og serieportar mm. er bygde opp av desse funksjonsblokkene, og **datamaskinen** er igjen bygd opp av systemblokker.

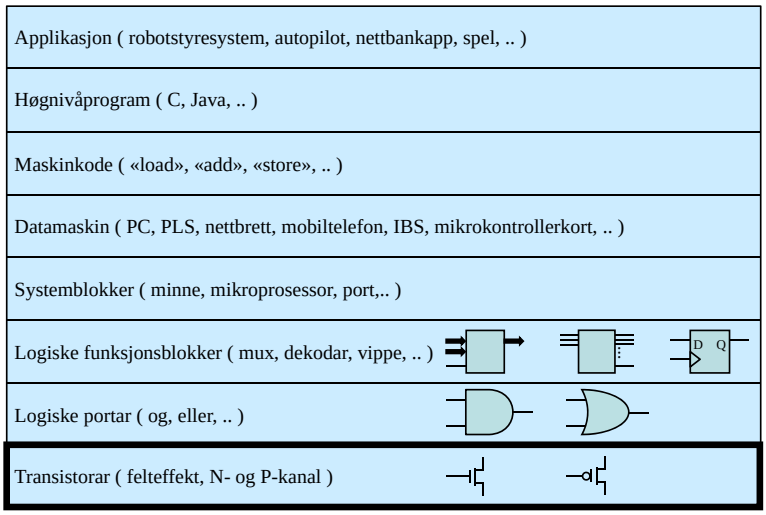
Ein applikasjon vil vera samansett av ulike filer med høgnivåkode. Desse blir under byggjing omgjort til **maskininstruksjonar**, sjå kapittel 2.2.2. Mikroprosessoren er som nemnt før, ein avansert tilstandsmaskin som hentar inn instruksjon etter instruksjon, tol-

kar og utfører desse.
Kvar instruksjon er som vist i kapittel 2.4.4.1, delt opp i ulike delar. Ein del går til ein dekodarmodul som f.eks. aktiverer ei adderarblokk. Andre delar aktiverer ulike minneelement, dvs. register, i mikroprosessoren der data då automatisk blir overført frå, og ein siste del aktiverer minneleementet som resultatet av addisjonen blir overført til. Ein instruksjon blir altså til mange digitale signal som fører til ein **automatisk operasjon** i den digitale elektronikken i datamaskinen.

Meir om dette kjem etter kvart, for først skal dei lågaste nivåa i hierarkiet fram i lyset.

3.4 Transistoren

Dette kapitlet omhandlar altså det lågaste nivået i hierarkiet, sjå figur 3.9.



Figur 3.9: Digitalt hierarki: Transistorlaget.

3.4.1 Bakgrunn

Den første transistoren kom i 1947[9], og med størrelsesorden på centimeter, dvs. 10^{-2} m. I dag innheld dei største minne- og prosessorbrikkene over 1 milliard transistorar, så størrelsen er i ferd med å koma ned til 10 nanometer[10], dvs. 10^{-8} m!

Transistoren blei for det første ein **releerstattar** som vist i kapittel 1.1.1. Han var både mykje raskare og som nemnt over, etter kvart mykje mindre enn relea. Transistoren la

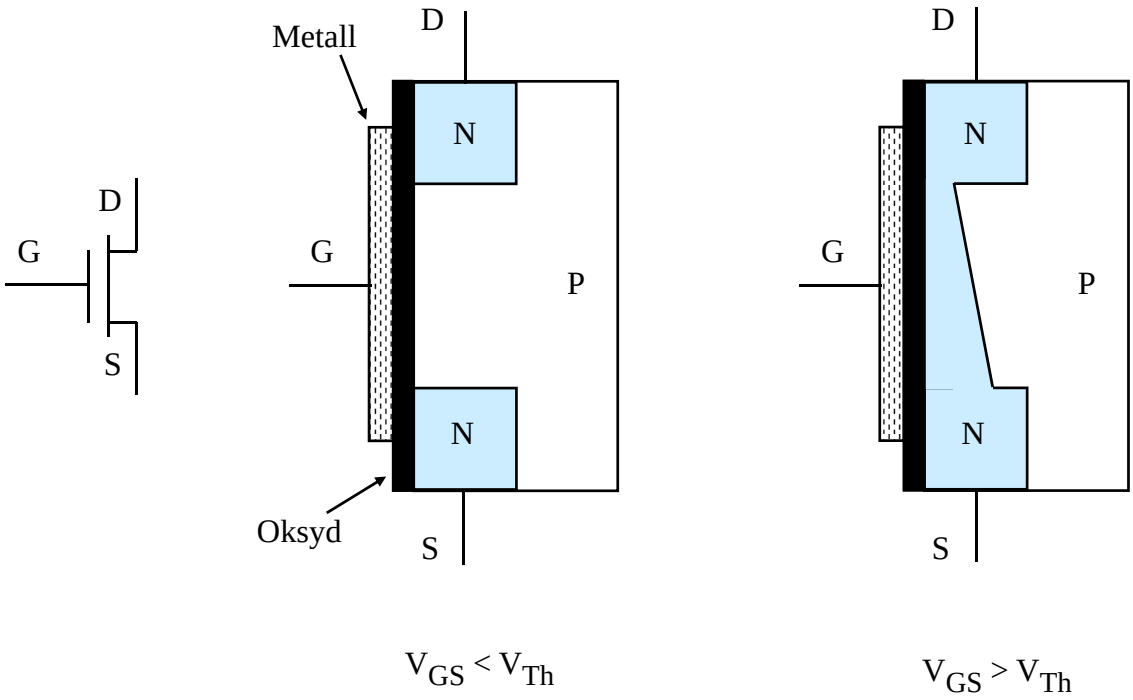
[9] Sjå "Wiki Replica-of-first-transistor.jpg"
[10] Atom har diametrar i området 0.05 - 0.38 nanometer, sjå "nn.wiki Ångstrøm". Ein nærmar seg altså dei nedre grensene for transistorstørrelse.

med dette grunnlaget for den digitale revolusjonen.
I tillegg la transistoren grunnlaget for ei veldig utvikling av **analoge forsterkarar**[11] av ulike slag, for eksempel audioforsterkarar.

Det er etterkvart blitt utvikla mange slags transistorar[12]. Første typen var den **bipolare transistoren** ("Bipolar Junction Transistor").
Seinare kom **felteffekttransistoren** som i dag er den dominerande typen i digital elektronikk.

3.4.2 Litt om felteffekttransistoren

Den mest vanlege utgåva av felteffekttransistoren er **MOSFET**-en ("Metall Oxide Semiconductor Field Effect Transistor")[13]. Symbol samt overordna oppbygging og virkemåte er vist i figur 3.10.



Figur 3.10: MOSFET-en si oppbygging.

Basis for transistoren er **halvleiarmateriale** ("semiconductor"). Som oftast er dette silisium som er **dopa**[14] med visse atom for å oppnå dei eigenskapane ein ønskjer.
Det er to hovudtypar halvleiarmateriale, nemleg **N**- og **P**-type som vist i figuren.

[11] Sjå "en.wiki amplifier".
[12] Sjå "en.wiki transistor".
[13] Sjå "en.wiki MOSFET".
[14] Sjå "en.wiki doping (semiconductor)".

Felteffekttransistoren er ein spenningstyrt ventil for elektrisk straum. Viss ein set på ei elektrisk spenning mellom terminalane *G(ate)* og *S(ource)* på transistoren i figuren, vil det bli danna ein elektrisk leiande **kanal** mellom terminalane *D(rain)* og *S(ource)*. Vilkåret er at spenninga V_{GS} er høgare enn den såkalla **terskelspenninga** for transistoren, V_{Th}. Kanalbreidda er avhengig av V_{GS}, så for å få "opna" transistoren heilt, må denne spenninga liggja eit stykkje over terskelspenninga. Typisk verdi på denne kan finnast i databladet på ein transistor.

Figuren viser ein **N-kanaltransistor**. Like vanleg er transistorar der **kanalen** er av **P-type**. Virkemåten er omvendt då det her er spenninga mellom terminalane *G(ate)* og *D(rain)* som styrer kanalen.
Viss $V_{DG} > V_{Th}$, vil P-kanalen opnast. Dette blir utnytta i såkalla CMOS-kretsar, som me snart skal sjå. I begge typane transistorar vil **straumretning** vera
frå D til S.

G-terminalen i ein MOSFET er knytt til eit såkalla metallsjikt[15] som igjen er elektrisk isolert frå dei to andre terminalane vhja. eit oksyd-sjikt som vist i figuren. Det går altså bare ein fosvinnande liten straum frå G til S. Dette er ein av faktorane som gjer at MOSFET-basert elektronikk har **lågt effektforbruk**, som er viktig for batteridrivne system. Lågt effektforbruk er også viktig i kraftige mikroprosessorbrikker, der høg effekt gir mykje varme som må leiast vekk for å unngå oppheiting. Meir om effektforbruk kjem i kapittel 3.4.4.4.

3.4.3 RTL, den første transistorkoblinga

Dei første integrerte kretsane var baserte på ei basiskobling som vist i figur 3.11. Denne typen oppkoblingar fekk namn etter komponentane og blei kalla "Resistor-TransistorLogikk", RTL[16]. Oppkoblinga i figuren er vist med felteffekttransistor, mens dei opprinnelege kretsane var basert på bipolare transistorar.

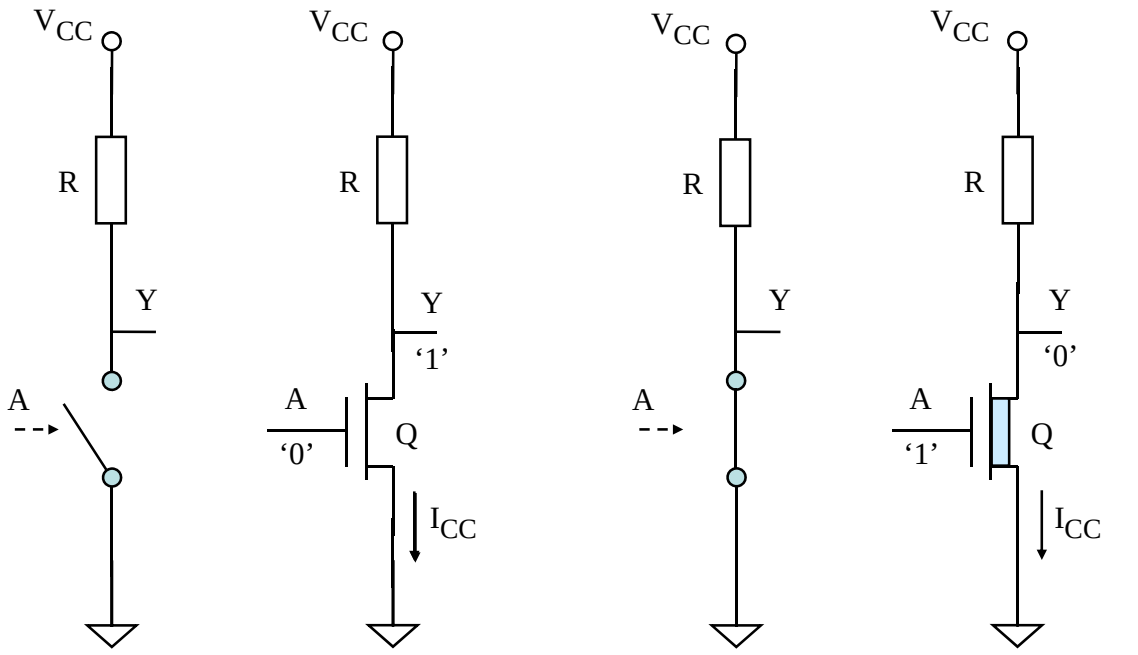
Transistoren blir i digitale kretsar brukt som ein styrt brytar, dvs. eit rele. Anten er transistoren av, dvs. ingen kanal, eller så er han på, dvs. med full kanal mellom D og S.
Digitale kretsar er laga slik at alle elektriske signal inntar eitt av to nivå, nemleg **høg/'1'** eller **låg/'0'**, som sagt før.
Høgt nivå er gitt av at signalspenninga ligg nær **forsyningsspenninga** til elektronikken[17] og lågt nivå er gitt av at signalspenninga ligg nær 0 Volt, dvs. "jord".

Ein ser av oppkoblinga og tabellen i figuren over at **lågt nivå** på inngangssignalet **A**, vil slå av transistoren. Straumen $\mathbf{I_{CC}}$ **frå kraftforsyninga**, dvs. mellom $\mathbf{V_{CC}}$ og jord, vil

[15] Opprinneleg var dette av metall, men ein har nå i lengre tid brukt andre material, sjå "en.wiki MOSFET".

[16] Sjå "en.wiki resistor-transistor logic".

[17] Vanlege kraftforsyningsspenningar er 5, 3.3, 2.5, 1.8, 1.5, 1.2 og 1.0 Volt der dei høgaste blir brukt ved forsyning av brikker på eit kretskort, mens dei lågaste blir brukte inni større brikker, f.eks. mikroprosessorar. Fordelen med dette opplegget er vist i kapittel 3.4.4.4.



A	V_A	Q	Icc	V_Y	Y
'0'	< V_{Th}	Av	0	V_{CC}	'1'

A	V_A	Q	Icc	V_Y	Y
'1'	> V_{Th}	På	V_{CC}/R	0	'0'

Figur 3.11: Ei basisoppkobling av en transistor vist i to situasjonar. Tilsvarande relekobling er til venstre for transistorkoblinga. Ein transistor som er på, er markert med ein blå kanal.

då bli null. Det vil derfor vera **null** spenningsfall over motstanden, og utgangssignalet **Y** vil bli lik forsyningsspenninga. I praksis er utgangen kobla til ein eller fleire nye inngangar som trekkjer litt straum. Då vil spenninga Y bli litt lågare, men fortsatt vera definert som logisk høg.

Eit **høgt nivå** på inngangsspenninga A vil slå transistoren på. Nivået på utgangen vil då som vist i tabellen i figuren, bli lågt. I praksis vil det vera litt elektrisk motstand i ein transistorkanal som fører til eit lite spenningsfall. Spenninga Y vil altså bli litt høgare enn 0, men fortsatt definert som logisk låg.
Straumen I_{CC} vil pga. av kanalresistansen $R_{DS,on}$[18] i praksis bli litt mindre enn verdien V_{CC}/R.

[18]Vanleg symbol for kanalresistansen mellom D og S der "on" siktar til at transistoren er på.

Den logiske funksjonen til kretsen i figur 3.11 er altså **invertering**.
Funksjonstabellen ("truth table") for denne digitale logiske kretsen blir då som vist i tabell 3.1.

A	$Y = \overline{A} = A' = xA$
0	1
1	0

Tabell 3.1: Funksjonstabell for invertarkrets.

Her ser ein at det er fleire måtar å visa invertering på. Notasjonen som vil bli brukt i dette skrivet, er det venstre alternativet i tabellen.
Funksjonen til ein invertar er altså $Y = \overline{A}$.

3.4.4 CMOS

RTL-oppkoblingane blei etterkvart erstatta av betre koblingar. Hovudulemper med RTL var at svitsjinga mellom dei to tilstandane var treg og at effektforbruket var høgt. Ein ser for eksempel av figur 3.11 at når inngangssignalet A er høgt, vil det gå ein **kontinuerleg** straum I_{CC} frå forsyninga og til jord.

Oppfølgjaren var TTL, "Transistor-Transistor Logic"[19]. TTL blei veldig populær, og hadde både raske utgåver samt relativt effekt-gjerrige, men tregare utgåver.

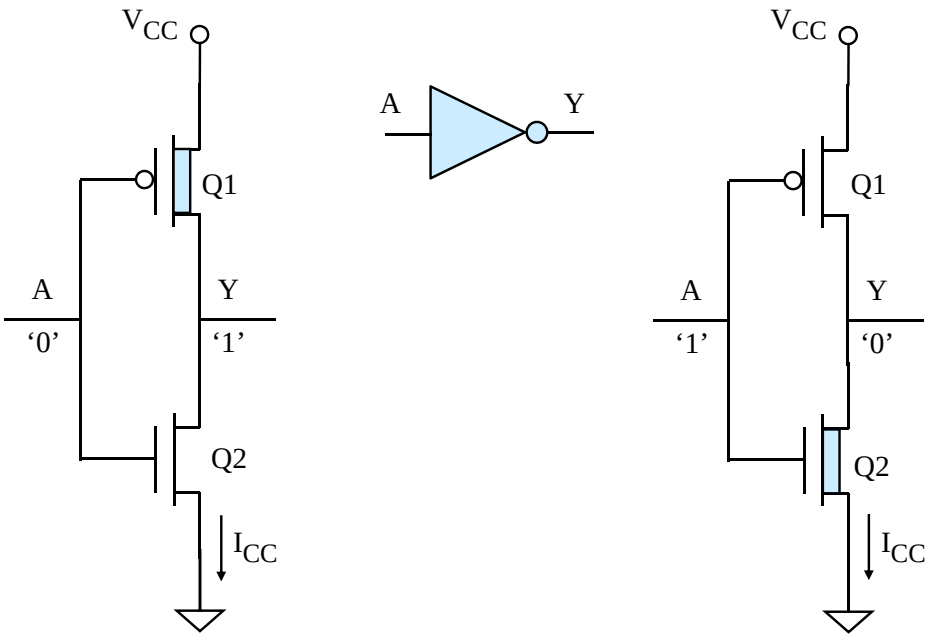
Etterkvart som logikkompleksiteten på brikkene auka, var det behov for endå gjerrigare koblingar. Svaret blei logikkfamilien "Complementary MOS", CMOS, som har vore dominerande teknologi i digital elektronikk dei siste 20 åra[20]. MOSFET-ar trengde altså vekk bipolare transistorar, som var basis for dei føregåande logikkfamiliane.

3.4.4.1 Basisoppsett for eit CMOS-trinn

Oppbygging og virkemåte av eit **"Complementary"** MOS-trinn, dvs. den grunnleggjande byggjeblokka i CMOS-teknologi, er vist i figur 3.12.

Funksjonen er den same som for RTL-utgåva i figur 3.11, men motstanden er erstatta av ein **P-kanaltransistor**. Denne typen transistor er symbolisert med eit **inverteringsteikn** på G-terminalen til Q1, då ei høg spenning **A** på inngangen vil slå av denne transistoren.
Tabellen til venstre i figuren viser verdiane til dei sentrale parametrane når inngangsspenninga **A** er låg. Tabellen til høgre viser situasjonen når **A** er høg.

[19] Sjå "en.wiki transistor-transistor logic".
[20] Sjå "en.wiki CMOS".



A	V_A	Q1	Q2	Icc	V_Y	**Y**
'0'	$< V_{Th}$	På	Av	0	V_{CC}	**'1'**

A	V_A	Q1	Q2	Icc	V_Y	**Y**
'1'	$> V_{Th}$	Av	På	0	0	**'0'**

Figur 3.12: CMOS-trinn. Ein transistor som er på, er markert med ein blå kanal.

Komplementær betyr altså her ei kobling som består av to transistorar med motsett/komplementær virkemåte, dvs. ein N- og ein P-kanaltransistor[21].

Den store fordelen med dette er at ein av transistorane alltid vil vera av som vist i tabellen i figuren. Det vil derfor aldri gå ein kontinuerleg straum I_{CC} frå kraftforsyninga og gjennom trinnet, dvs. mellom $\mathbf{V_{CC}}$ og jord. Effektforbruket er altså i utgangspunktet lågt i CMOS. Meir om dette kjem i kapittel 3.4.4.4.

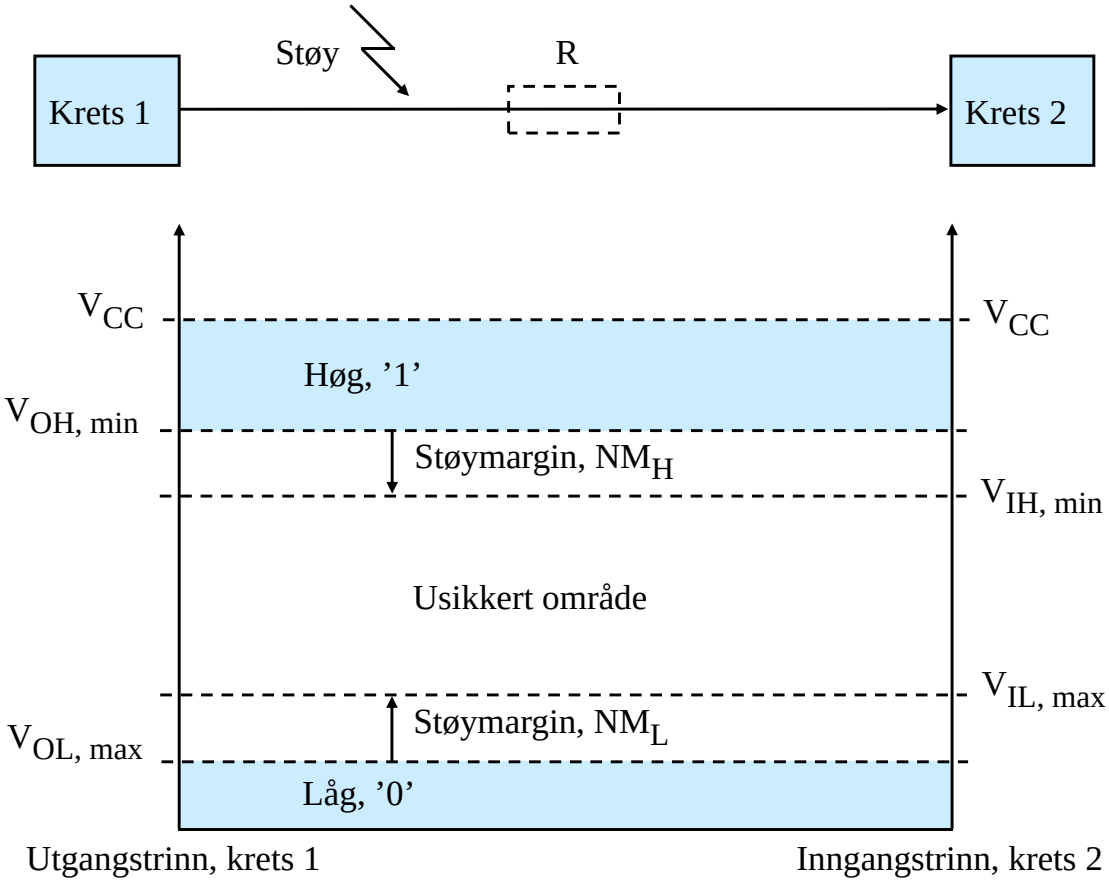
Invertaren er den enklaste **logiske porten** ("logic gate"), då det bare trengst to transistorar for å realisera denne. **Symbolet**[22] for denne porten er vist i figuren.

[21] Eit slikt utgangstrinn der transistorar og eventuelle andre komponentar står oppå kvarandre, blir kalla ein **totempåle** ("totem pole"). Dette namnet kom med TTL-teknologien.

[22] Som ein ser hjå "en.wiki logic gate", så er det fleire symbolstandardar ute og går. ANSI/IEEE-standarden er nok den mest vanlege og vil bli brukt her.

3.4.4.2 Litt om spenningsforhold for digitale kretsar

Figur 3.13 viser ein digital krets som gir ut eit signal, og dette går inn på ein annan krets. For at Krets 2 skal forstå at signalet har eit bestemt logisk nivå, må Krets 1 gi ut ei spenning som ligg innanfor det rette **spenningsområdet** for det logiske nivået.
Det blir altså brukt eit visst spenningsområde[23] for logisk høgt nivå, '1', og eit anna spenningsområde for logisk lågt nivå, '0'.



Figur 3.13: Spenningsområde for dei ulike logiske nivåa.

Spenninga for eit signal kan endra seg under overføring langs ein bane på eit kretskort eller ein leiar i ein kabel. Årsaker kan vera **spenningsfall** pga. elektrisk motstand i banen/kabelen eller at det blir indusert **støyspenningar** som legg seg oppå signalet. Dette er forsøkt illustrert i figuren.

Derfor må det vera visse marginar, her kalla **støymarginar** ("noise margin") mellom det som utgangen gir ut av spenningsnivå for eit logisk nivå og det spenningsnivået ein mottakar må ha på inngangen for å **tolka** det logiske nivået rett.

[23] Kva spenningsområde ein bruker, er avhengig av kva forsyningsspenninga V_{CC} er og kva teknologi dei integrerte kretsane er basert på. Vanlegast i dag er som nemnt CMOS-teknologi.

Ein går nå som vist i figuren, ut frå følgjande:

- Ei spenning innan eit visst område opp mot forsyningsspenninga V_{CC} representerer ein logisk '1'.
- Ei spenning innan eit visst område ned mot jordnivå, dvs. 0 Volt, representerer ein logisk '0'.

Først vil det bli gitt ei forklaring av dei viktige parametrane i figur 3.13. Så vil det bli vist eit eksempel på ein støymarginanalyse for ei konkret kobling.

3.4.4.3 Støymarginar og andre viktige parametrar

Ved analyse av om ei overføring vil gå greitt, dvs. bli tolka rett av mottakaren, må ein sjå på overføring av både av logisk høge og logisk låge verdiar.

a) Høg tilstand
Ved overføring av ein logisk høg verdi melder det seg nokre sentrale spørsmål:
1. Kva er den lågaste spenninga som utgangen i verste fall kan gi for ein '1'? Denne parameteren blir kalla $V_{OH,min}$ ("Output High").
2. Kva er den lågaste spenninga som blir godtatt som ein '1' på inngangen? Denne parameteren blir kalla $V_{IH,min}$ ("Input High").
Desse parameterverdiane kan me finna i databladet for ein komponent. **Støymarginen** ("noise margin") i høg tilstand, NM_H, er som vist i figur 3.13, gitt av:

$$NM_H = V_{OH,min} - V_{IH,min}$$

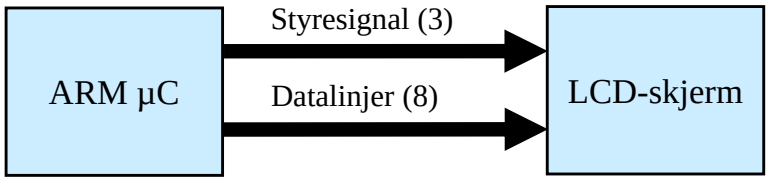
b) Låg tilstand
Ved overføring av ein logisk låg verdi melder det seg tilsvarande sentrale spørsmål:
1. Kva er den høgaste spenninga som utgangen i verste fall kan gi for ein '0'? Denne parameteren blir kalla $V_{OL,max}$ ("Output Low").
2. Kva er den høgaste spenninga som blir godtatt som ein '0' på inngangen? Denne parameteren heiter $V_{IL,max}$ ("Input Low").
Igjen kan me gå til databladet for ein komponent for å finna desse parameterverdiane.

Støymarginen ("noise margin") i låg tilstand, NM_L, er som vist i figuren, gitt av:

$$NM_L = V_{IL,max} - V_{OL,max}$$

Eksempel 3.4. *Grensesnitt mellom ein ARM-kontroller og ein LCD-skjermmodul*

Ein tenkjer seg her at ein LCD-skjermmodul er kobla til parallellportane på mikrokontrolleren vår, STM32F100RB, sjå figur 2.7 på side 36. Ein vil her finna ut kva støymarginar ein har ved overføring av data til skjermen.
Systemet er illustrert i figur 3.14.



Figur 3.14: Grensesnitt mellom ARM-kontroller og LCD-skjermmodul.

a) Parameterverdiar
Parameterverdiar for ARM-kontrolleren kan finnast i tabell 35 på s. 59 i databladet på STM32F100-familien, [16].
Her ser me bare på utgangsparametrar sidan alle dei tre styresignala går frå mikrokontrolleren og til skjermen[24] og at data bare skal overførast til skjermen:

$V_{OL,max} = 0.4$ Volt, $V_{OH,min} = V_{CC,min} - 0.4 = 2.6$ Volt.

Ein har her tatt utgangspunkt i plattformkortet *STM32VLDiscovery*, der forsyningsspenninga til mikrokontrolleren er 3.0 Volt.

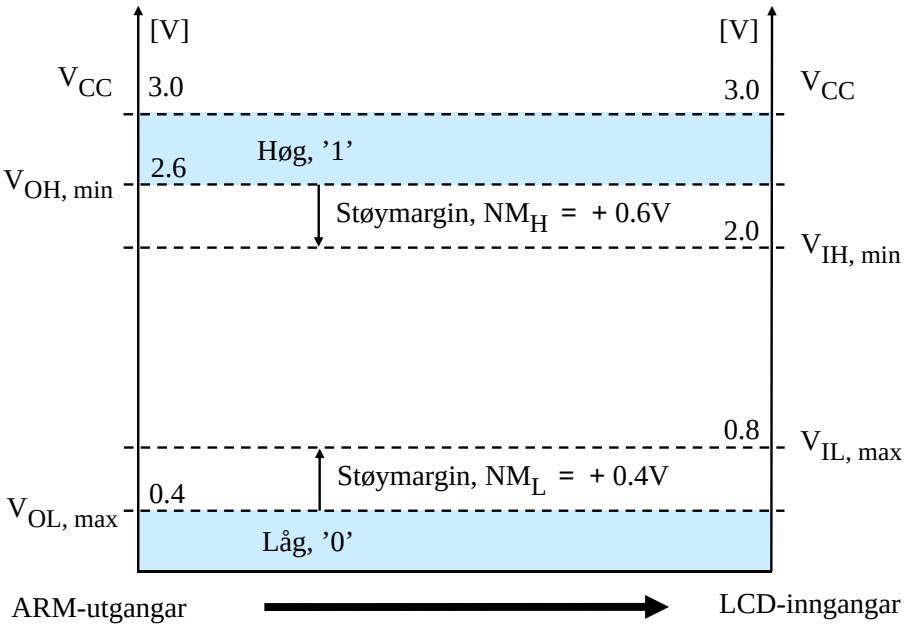
Parameterverdiar for inngangane på LCD-modulen kan finnast i databladet for denne. Me tenkjer oss at forsyningsspenninga er som for mikrokontrolleren, dvs. 3.0 Volt, og at dei aktuelle parameterverdiane er

$V_{IL,max} = 0.8$ og $V_{IH,min} = 2.0$ V.

Ved å setja alle verdiane opp i diagram for høgt og lågt nivå får ein fram støymarginane for overføringa som vist i figur 3.15.

Ein kan konkludera med at marginane her er gode. Det skal relativt store endringar til for at mottakaren skal misforstå dei logiske nivåa til inngangssignala.

[24] Mange grensesnitt vil også signal som går andre vegen. Dette kan f.eks. vera eit signal med namn *Busy* som fortel mikrokontrolleren om den tilkobla kretsen er klar til å ta imot data. Slike signal blir altså ikkje styrt av mikrokontrolleren, men vil likevel og naturleg nok høyra til styrebussen.



Figur 3.15: Støymarginanalyse for grensesnittet mellom ARM og LCD.

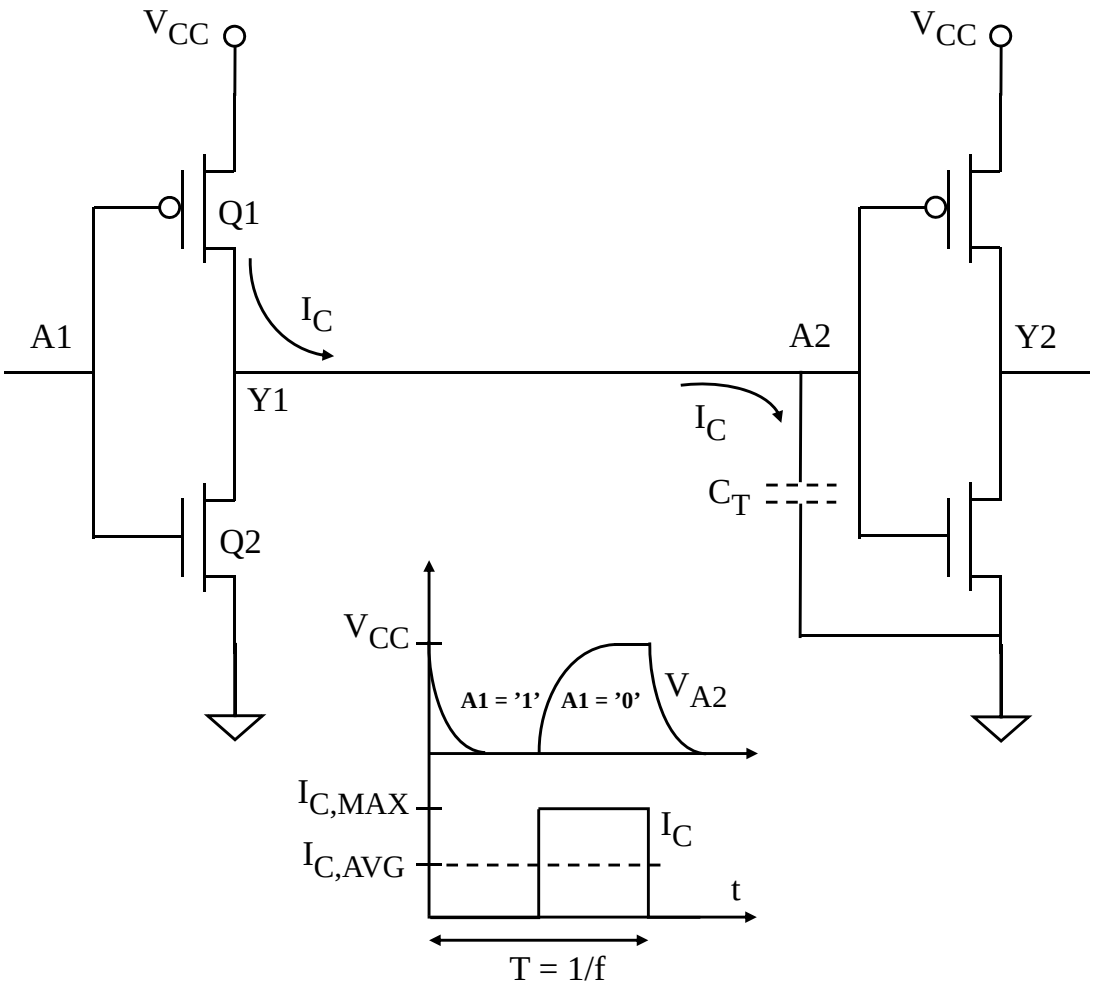
3.4.4.4 Litt om effektforbruk

Som vist i kapittel 3.4.4.1, vil ein av transistorane i eit CMOS utgangstrinn alltid vera av. Det vil derfor i prinsippet aldri gå ein kontinuerleg straum, også kalla DC-straum, frå V_{CC} til jord gjennom trinnet. Det **statiske effektforbruket**[25] eller DC-effektforbruket er derfor ideelt sett lik **null**.
I praksis vil det gå litt straum då transistorane vil ha litt lekkasje. Det vil også vera litt lekkasje gjennom inngangen til ein krets, dvs. gjennom G(ate)-terminalane på transistorane kretsen. Slike lekkasjestraumar vil ikkje vera større enn typisk $1\mu A$, så det statiske effektforbruket blir likevel svært lite.

Det som er hovudårsaken til at effektforbruket i CMOS-kretsar likevel kan vera høgt, er **svitsjinga** mellom tilstandane høg og låg. Ein mikroprosessor køyrer for eksempel med ein gitt klokkefrekvens, og dette gir millionar av svitsjingar i sekundet rundt i dei ulike delane av prosessoren. Effektforbruket pga. av denne dynamikken, eller meir konkret svitsjinga, i ein digital krets blir kalla **dynamisk effektforbruk**.

Inngangen på ein MOSFET-transistor er som vist i figur 3.10, isolert frå sjølve kanalen. Ein kan sjå på overgangen mellom "Gate"-inngangen og kanalen som ein **kondensator**. Det vil også vera kapasitans mellom G-inngangen og S-terminalen.
For at transistoren skal slå seg på, dvs. danna ein kanal, må ein derfor **lada opp** inngangen slik at spenninga kjem over terskelspenninga for transistoren, som vist i kapittel 3.4.2. Dette er vist med ein forenkla illustrasjon i figur 3.16.

[25] Effekt er som kjent produktet av forsyningsspenning V_{CC} og straumtrekk og blir målt i Watt.



Figur 3.16: Opplading av inngangen til eit CMOS-trinn.

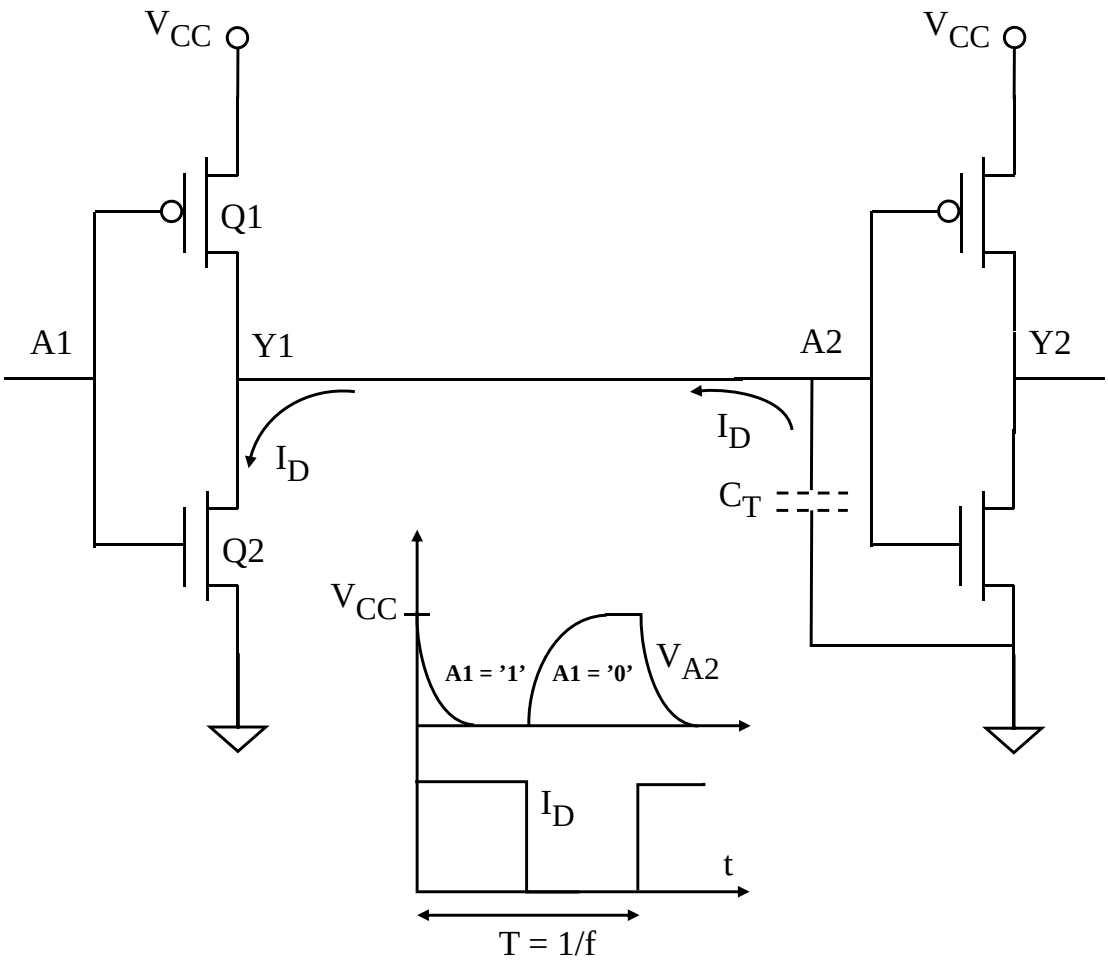
Eit CMOS-trinn driv her eit etterfølgjande CMOS-trinn. Ved å driva ein straum I_C[26] i ei viss tid, vil det tilførast nok ladningar til inngangskondensatoren for trinn 2.
I figuren er denne kondensatoren illustrert med eit symbol[27] på inngangen. Kapasitansen er C_T som er den **totale** kapasitansen ein ser gjennom inngangen.

Den viste sekvensen i figuren startar med at inngangen på det første trinnet er høg, dvs. *A1 = '1'*. Dette gir ei låg spenning V_{A2} inn på trinn 2.
Når A1 så går låg/'0', vil transistoren Q1 bli slått på. Inngangen til trinn 2 vil då pga. den tilførte straumen I_C frå kraftforsyninga ladast opp, og spenninga V_{A2} på vil nå høgt nivå etter ei viss tid.

[26] C for "Charge", (opp)lading.
[27] Symbolet er stipla då det ikkje er ein eigen kondensator som heng på linja inn til trinnet, men ein eigenskap ved trinnet sjølv.

Utgangen på trinn 1 i figur 3.16 er der i såkalla **"source"**-situasjon, då den **leverer** straum til inngangen på trinn 2.

Kva som skjer under utlading av inngangen, er vist i figur 3.17.



Figur 3.17: Utlading av inngangen til eit CMOS-trinn.

I første delen av sekvensen er A1 høg, og transistoren Q2 vil då vera på. Ein straum I_D[28] vil då gå til jord. Etter i ei viss tid, vil denne ha fjerna nok ladningar på kondensatoren slik at inngangen på trinn 2 når lågt nivå.

Utgangen på trinn 1 er nå i såkalla **"sink"**[29]-situasjon, då den **tar imot** straum frå inngangen på trinn 2.

Straumen er framstilt som konstant under opplading og under utlading. I praksis vil straumen variera ein del mens dette pågår, men det er ikkje viktig for effektutrekninga som nå

[28] D for "Discharge", (ut)lading.

[29] Ein "sink" betyr eigentleg ein utslagsvask. Omgrepa "source" og "sink" er velbrukte i elektronikkverda. Dette blir også kalla **"push-pull"** der "push" viser til straumleveranse og "pull" det motsette.

skal gjerast.

Viss svitsjinga går med ein viss **frekvens f**, vil sekvensen i figur 3.16 ha ei periodetid lik

$$T = 1/f$$

Kraftforsyninga leverer straum bare i halve svitsjeperioden. Gjennomsnittleg straumtrekk frå kraftforsyninga blir derfor

$$I_{C,AVG} = I_{C,MAX}/2$$

Frå fysikken har ein fylgjande samanheng mellom ladning, kapasitans og spenning[30]:

$$Q = C \cdot V$$

For sekvensen i figur 3.16 får ein då følgjande uttrykk for tilført ladning til inngangen på trinn 2:

$$Q = I_{C,MAX}\frac{T}{2} = I_{C,AVG}T = I_{C,AVG}\frac{1}{f} = C_T V_{CC}$$

Dette gir følgjande gjennomsnittlege straumtrekk:

$$I_{C,AVG} = C_T V_{CC} f \tag{3.1}$$

Effekten P_D levert av kraftforsyninga og avgitt[31] i CMOS-trinn 2 blir

$$P_D = V_{CC} I_{C,AVG} = C_T V_{CC}^2 f \tag{3.2}$$

Denne viktige formelen viser at for å oppnå at batteridriven digital elektronikk kan køyra i lang tid mellom ladingar eller at store mikroprosessorbrikker ikkje blir for varme, må ein elektronikkutviklar prøva å

- bruka så låg forsyningsspenning som mogleg, sjå fotnote i kapittel 3.4.3 på side 117.
- ikkje bruka høgare klokkefrekvens enn ein må i ulike delar av eit digitalt system[32].
- redusera samla kapasitans i elektronikken.

Reduksjon av samla kapasitans kan ein oppnå ved å redusera størrelsen på transistorane og pakka desse tettare saman. I tillegg kan ein bruka meir effektive kretsar, dvs. kretsar som krev færre transistorar men med same funksjonalitet som konkurrentane[33].

[30] Sjå "nn.wiki kapasitans"

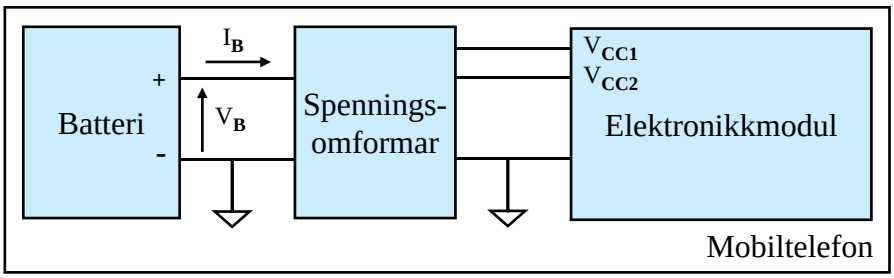
[31] D i parameteren P_D står her for "Dissipated", dvs. avgitt.

[32] Ei løysing er også å stansa klokka heilt til delar som ikkje er i bruk. Mikrokontrolleren STM32F100RB har ein eigen **RCC**-modul ("Reset and Clock Control") der ein kan aktivera klokkesignalet til dei modulane som ein skal bruka i ein applikasjon. Dei andre modulane vil vera inaktive og vil derfor bruka minimalt med effekt. STM32F100RB er som alle andre moderne mikrokontrollerar, basert på CMOS-teknologi.

[33] Det er mellom anna her ARM-prosessoren er så konkurransedyktig. Intel-prosessorane er mindre effektive, men her satsar ein på å optimalisera sjølve transistorane, sjå f.eks. "en.wiki multigate device".

Eksempel 3.5. *Ladeintervall for mobiltelefon*

Eit grovt blokkskjema av ein mobiltelefon er vist i figur 3.18.



Figur 3.18: Blokkskjema av kraftforsyninga i ein mobiltelefon

Mobiltelefonen her har eit batteri med kapasitet $K_B = I_B \cdot T_B = 2000$ mAh og spenning $V_B = 3.6$ Volt.

Det er ofte behov for fleire ulike forsyningsspenningar til ulike delar av ein elektronikkmodul. Dette er eksemplifisert med spenningane V_{CC1} og V_{CC2} i figuren. I tillegg er som regel ingen av forsyningsspenningane i eit system lik kjeldespenninga, som her er batterispenninga.
Det er derfor eit generelt behov for kombinert spenningsomforming og -regulering i eit elektronisk system. **Omforming** går ut på å generera forsyningsspenningar som altså anten er lågare eller høgare enn kjeldespenninga. **Spenningsregulering** går ut på å halda dei genererte forsyningsspenningane stabile sjølv om straumtrekket i elektronikkmodulen varierer.

Ein kombinert spenningsomformar og -regulator kan lagast på ulike måtar[34]. Her skal me gå ut frå at denne modulen er ideell, dvs. har tilnærma null tap.

Mobiltelefonen i figuren har samla kapasitans $C_T = 1.0$ μF (10^{-6} Farad) og gjennomsnittleg frekvens under bruk på 100 kHz. Gjennomsnittleg spenning på dei ulike kretsane i elektronikkmodulen under bruk er 2.5 V.

Me vil nå finna ut kor mange timar ein kan bruka telefonen mellom ladingar. Ein skal også rekna ut denne brukstida for ein ny mobiltelefonvariant som har gjennomsnittleg forsyningsspenning på bare 1.8 V men ellers same data.

[34] I eit enkelt system som får kraft utanifrå, vil den greiaste løysinga vera å bruka ein såkalla lineær spenningsomformar og -regulator, sjå f.eks. "en.wiki Linear regulator". Med denne løysinga vil ein "fyra litt for kråkene", men med ekstern kraftforsyning er ofte dette ukritisk.
I eit batteriforsynt system eller eit system med med stort straumforbruk, vil ein svitsja spenningsomformar og -regulator eller såkalla DC/DC-omformar vera beste løysinga, sjå f.eks. "en.wiki DC-to-DC converter". Slike har høg virkningsgrad, dvs. lite tap.

Basert på spesifikasjonen over, kan svara finnast slik:

Energien lagra i batteriet er gitt av *spenning x straumkapasitet*, altså:
$E_B = V_B \cdot I_B \cdot T_B = 3.6\ V \cdot 2000\ mA \cdot 1\ h = 7.2\ Watt \cdot 3600\ sekund \approx 26.0$ kJoule.

Når spenningsomforminga skjer tilnærma utan tap, vil heile denne energien vera tilgjengeleg for elektronikkmodulen. Med eit effektforbruk P_D (D for "Dissipation", tap) vil heile batterienergien vera oppbrukt i løpet av tida mellom ladingar, T_{CI} (CI for "Charge Interval", ladeintervall). Samanhengen blir:

$$E_B = T_{CI} P_D = T_{CI} \cdot (\overline{f} C_T \overline{V_{CC}}^2)$$

der

$\overline{f}$ - gjennomsnittleg frekvens, og

$\overline{V_{CC}}$ - gjennomsnittleg forsyningsspenning.

Dette gir følgjande ladeintervall:

$$T_{CI} = \frac{E_B}{P_D} \approx \frac{26.0\ kWattsek}{100kHz \cdot 1.0 \cdot 10^{-6} Farad \cdot (2.5Volt)^2} = 41600sek = 12h$$

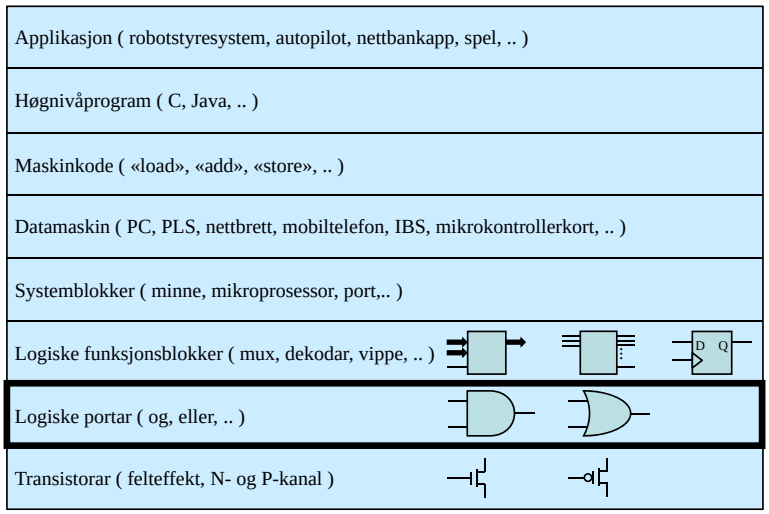
Viss den gjennomsnittlege forsyningsspenninga blir redusert frå 2.5 til 1.8 Volt, blir det nye ladeintervallet:

$$T_{CI,1.8} = T_{CI,2.5} \frac{2.5^2}{1.8^2} \approx T_{CI,2.5} \cdot 1,93 \approx 22h$$

Det å redusera forsyningsspenninga gir altså stor innverknad på effektforbruket i elektronikk.

3.5 Logiske portar

Ein skal her sjå på neste lag i det digitale hierarkiet, nemleg dei logiske portane (”gate”), sjå figur 3.19.



Figur 3.19: Digitalt hierarki: Portlaget.

Dei logiske portane er sentrale byggjesteinar i digitale kretsar. Kapitlet startar med sjølve funksjonane til dei ulike logiske portane, og går så over på korleis ein kan beskriva oppførselen til eit logisk system. Til slutt ser ein på oppbygging og symbol for portane.

3.5.1 Digitale logiske funksjonar

3.5.1.1 Formulering av ein logisk funksjon

Korleis ein kan formulera ein logisk funksjon i rein tekst, er illustrert vhja. eit eksempel, nemleg bildørlogikken frå eksempel 3.1 på side 107.

Eksempel 3.6. *Formulering av dørlyslogikken*

Funksjonen til bildørlogikken kan formulerast slik:

”Viss *dør1* **eller** *dør 2* **eller** *dør 3* er open så skal *dørlyset* vera på.”

Det kan og seiast slik:

”Viss *dør1* **og** *dør 2* **og** *dør 3* er stengde, så skal *dørlyset* vera av.”

Stengt blir her det same som **ikkje** open, og av det same som **ikkje** på.

Orda **og**, **eller** og **ikkje** kallar ein **logiske operatorar**.
Ein har av og til også behov for operatoren **eksklusiv eller**.
Viss ein **inverterer** resultatet av operasjonane **og**, **eller** og **eksklusiv eller**, får ein såkalla **negative** operatorar.

3.5.1.2 Logiske operatorar

Dei logiske operatorane som blei nemnte i førre kapittel, er summert opp i tabellar under.
I første tabellen er det vist namn på norsk og engelsk samt logisk symbol.

Norsk namn	Norsk forkorting	Engelsk fork.	Logisk symbol m/eksempel
Ikkje	N(egativ)	N	$\overline{A}$
Og	OG	AND	$A \cdot B$
Negativ og	NOG	NAND	$\overline{A \cdot B}$
Eller	ELLER	OR	$A + B$
Negativ eller	NELLER	NOR	$\overline{A + B}$
Eksklusiv eller	XELLER	XOR	$A \oplus B$
Negativ eksklusiv eller	XNELLER	XNOR	$\overline{A \oplus B}$

Tabell 3.2: Namn og symbol på logiske operatorar.

Funksjonstabellen ("truth table") for alle operatorane er vist under.

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

Tabell 3.3: Funksjonstabell for logiske operatorar.

3.5.1.3 Digresjon til logiske og bitvise operatorar i høgnivåspråk

Ein kan sjølsagt også som kjent bruka logiske operatorar i høgnivå programkode for mikroprosessorar.
Ein skil då mellom ein **logisk operator** og ein **bitvis operator**. Operatorane som språket C tilbyr, er viste i tabellen under.

Det er her brukt andre bokstavar på talverdiane for å markera at desse er n-bitsstørrelsar der n vanlegvis er 8, 16 eller 32. Talverdiane A er B i tabellane 3.2 - 3.3 er 1-bitsstørrelsar.

Norsk fork.	Engelsk fork.	Logisk operator i C m/eks.	Bitvis operator i C m/eks.
N	N	$!M$	$\tilde{}N$
OG	AND	$M\&\&N$	$M\&N$
NOG	NAND	$!(M\&\&N)$	$\tilde{}(M\&N)$
ELLER	OR	$M\|\|N$	$M\|N$
NELLER	NOR	$!(M\|\|N)$	$\tilde{}(M\|N)$
XELLER	XOR		$M\hat{}N$
XNELLER	XNOR		$\tilde{}(M\hat{}N)$

Tabell 3.4: Namn og symbol på logiske og bitvise operatorar i høgnivåspråket C.

Eksempel 3.7. *Logiske kontra bitvise operatorar*

Gitt tala $M = 0$, $N = 5$, $R = 141 = 10001101_2$ og $S = 201 = 11001001_2$.

Då har me at

$!M = 1$, $!N = 0$, $!R = 0$ mens $\sim R = 01110010_2$,

$M\&\&N = 0$, $N\&\&R = 1$ då begge er ulik 0, mens $R\&S = 10001001_2$,

$M||N = 1$, mens $R|S = 11001101_2$ og $R \hat{} S = 01000100_2$.

3.5.1.4 Spesifikasjon av logisk funksjon vhja. funksjonstabell

Me bruker igjen dørlyslogikken som vist under.

Eksempel 3.8. *Funksjonstabell for dørlyslogikken*

Me definerer nå følgjande:

- open tilsvarer "Høg", dvs. "1".
- på tilsvarer "Høg", dvs. "1".

Av dette følgjer at tilstandane stengt, dvs. ikkje open, og av tilsvarer "Låg"/"0".

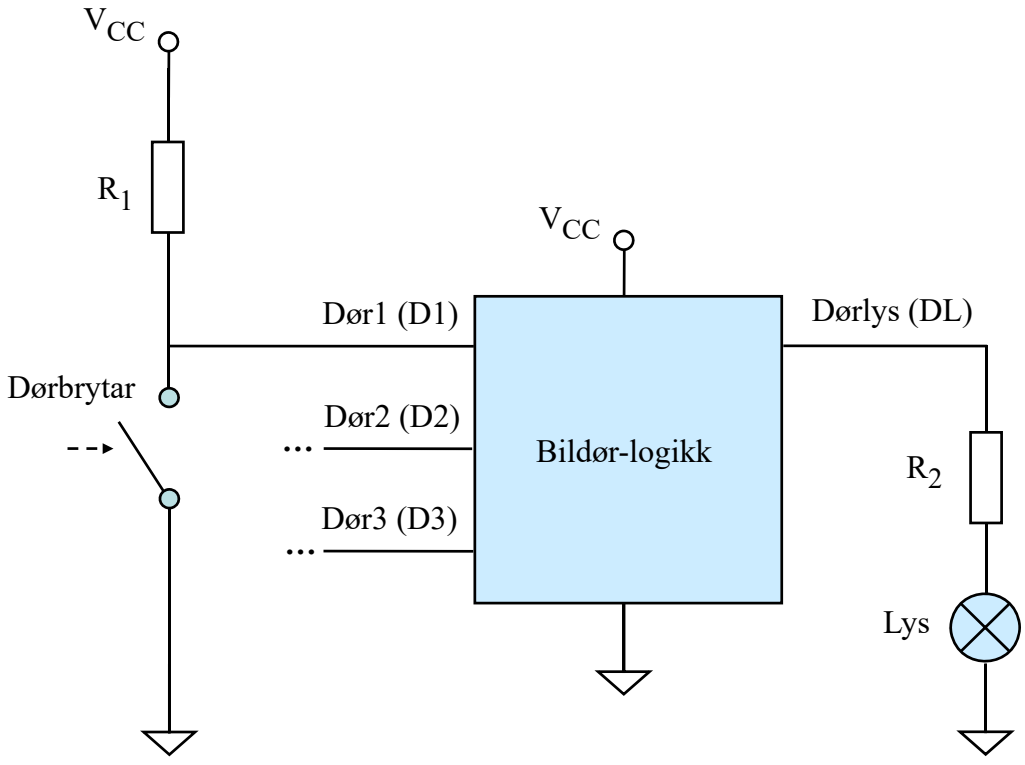
Oppførselen til bildørlogikken kan då spesifiserast med funksjonstabell som vist i tab. 3.5.

I figur 3.20 er det vist ei tilkobling av dører og lys som er i samsvar med definisjonane gjort over. Alle dørene er kobla på same måte, dvs. som vist for Dør 1. Når ei dør blir lukka, vil også dørbrytaren bli lukka.

For å kunne realisera bildørlogikken i figur 3.20, treng ein **digitale elektroniske byggjesteinar** som har nokre av dei logiske funksjonane vist i tabellane over. Slike byggjesteinane kallar ein som nemnt før, logiske **portar** ("logic gates").
Desse skal presenterast i tur og orden i det følgjande.

Dør1	Dør2	Dør3	Dørlys
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

Tabell 3.5: Funksjonstabell for bildørlogikken.



Figur 3.20: Tilkobling av dører og lys til bildørlogikken.

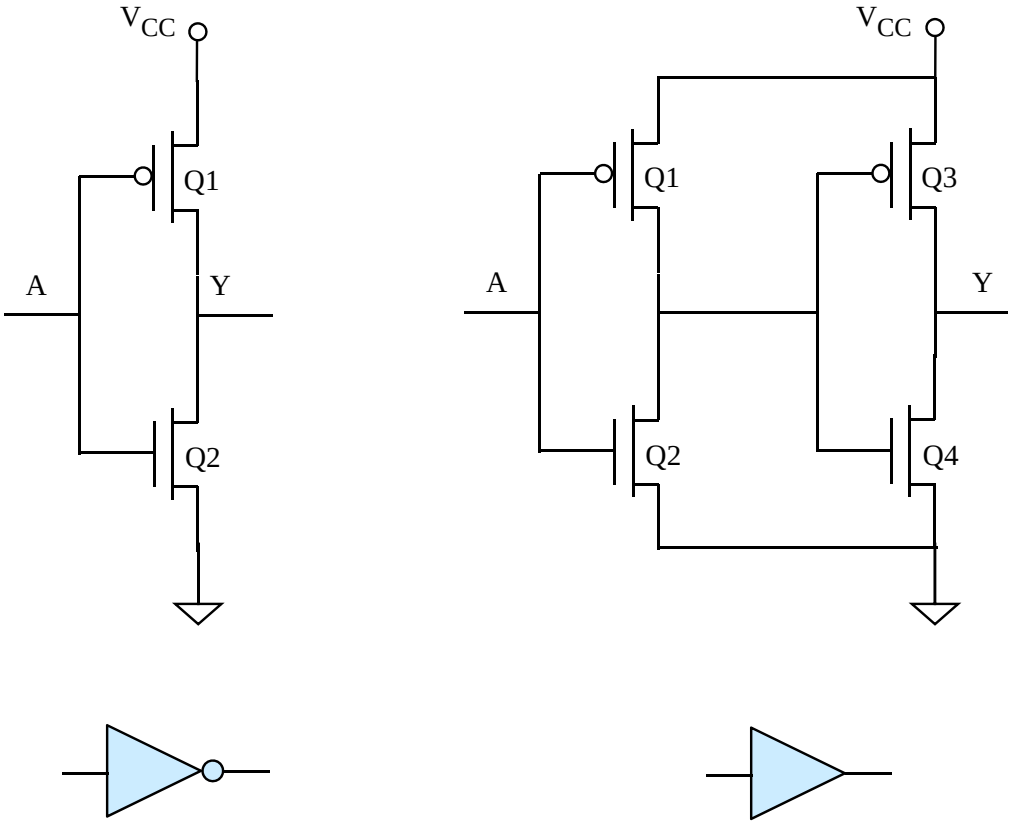
3.5.2 Dei enklaste portane: Invertaren og bufferet

3.5.2.1 Invertar

Den enklaste logiske porten er ikkje noko anna enn CMOS-trinnet i kapittel 3.4.4.1. Med inngang A og utgang Y såg ein der at likninga for denne porten er

$$Y = \overline{A}$$

Vanleg symbol og oppbygging av invertaren er vist til venstre i figur 3.21.



Figur 3.21: Invertar og buffer.

3.5.2.2 Vanleg buffer

Den nest enklaste porten er ein buffer, dvs. ein **ikkje-inverterande** port. Symbol og oppbygging av både invertar og buffer er vist til høgre i figur 3.21. CMOS-utgåva av ein buffer er som vist gitt av to påfølgjande invertartrinn. Med inngang A og utgang Y blir

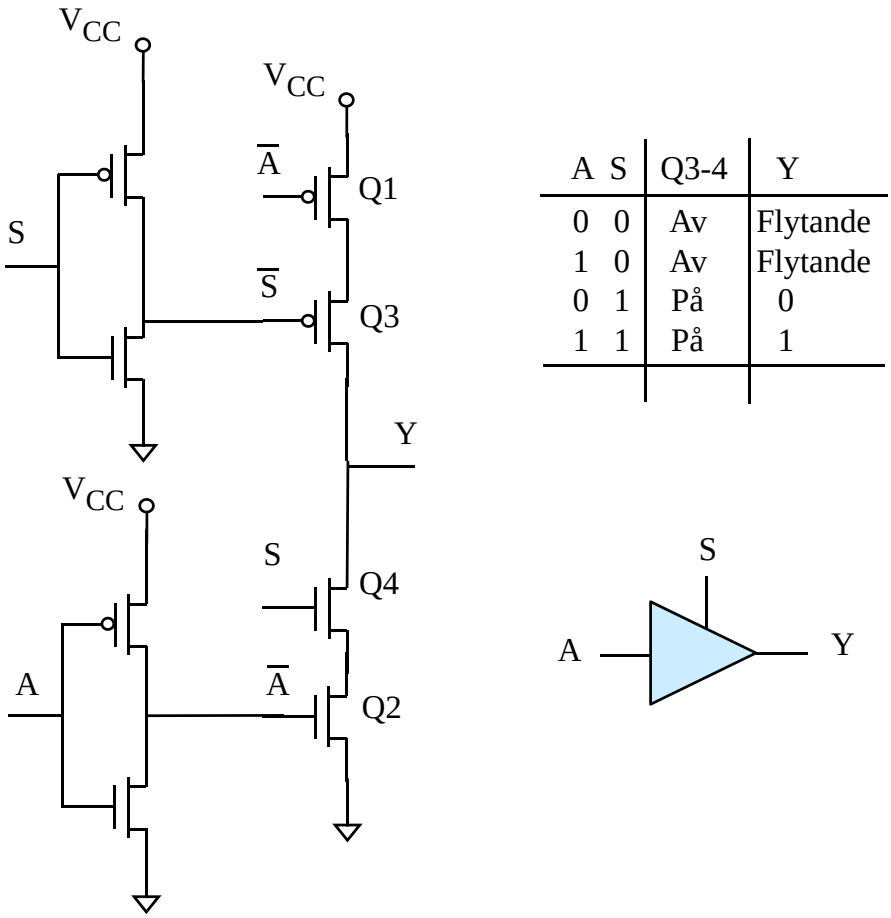
då likninga for bufferen rett og slett

$$Y = A$$

Kva er vitsen med ein port som ikkje gjer noko med signalet?
Kapasiteten når det gjeld å levera straum kan vera for liten på inngangssida i forhold til alt som er tilknytta på utgangssida. Ein kan då bruka ein buffer med stor drivekapasitet eller parallellkobla fleire buffer og fordela dei tilknytte kretsane på desse.

3.5.2.3 *Tristate*-buffer

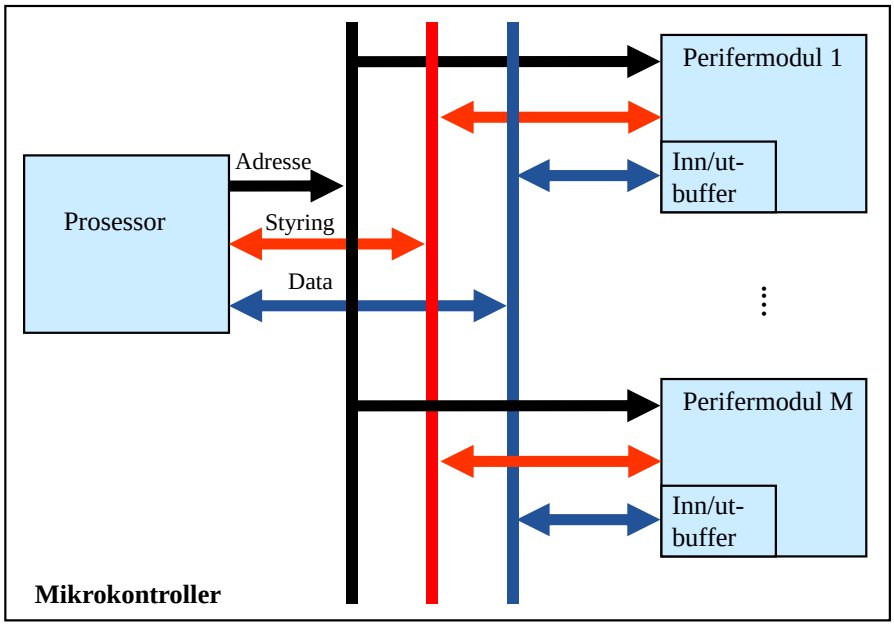
I tillegg kan ein ha buffer der ein kan kobla inngangen vekk frå utgangssida av bufferet vhja. eit ekstra styresignal.
Oppbygging og funksjon til ein slik port er vist i figur 3.22.



Figur 3.22: *Tristate*-buffer.

Vhja. styresignalet S kan ein stenga begge transistorane $Q3$ og $Q4$ samtidig. Utgangen Y vil då **flyta**, dvs. vera upåverka av denne porten. Då er det dei andre komponentane som er kobla til denne Y-linja som bestemmer det logiske nivået. Slik ein buffer blir kalla ein *tristate*-buffer der den flytande eller fråkobla tilstanden gjer at ein har **tre** utgangstilstandar ("tristate")[35] Denne tredje tilstanden blir også kalla ein **høg-impedanstilstand** ("high-Z") då det her er svært høg impedans både mellom Y og V_{CC} og mellom Y og jord (GND).

Tristate-bufferar er nødvendige når mange ulike portar er kobla til dei same signallinjene, f.eks. i eit bussystem. Dette er illustrert i figur 3.23. Mikrokontrolleren her har ein systembuss der det er tilkobla M perifermodular.



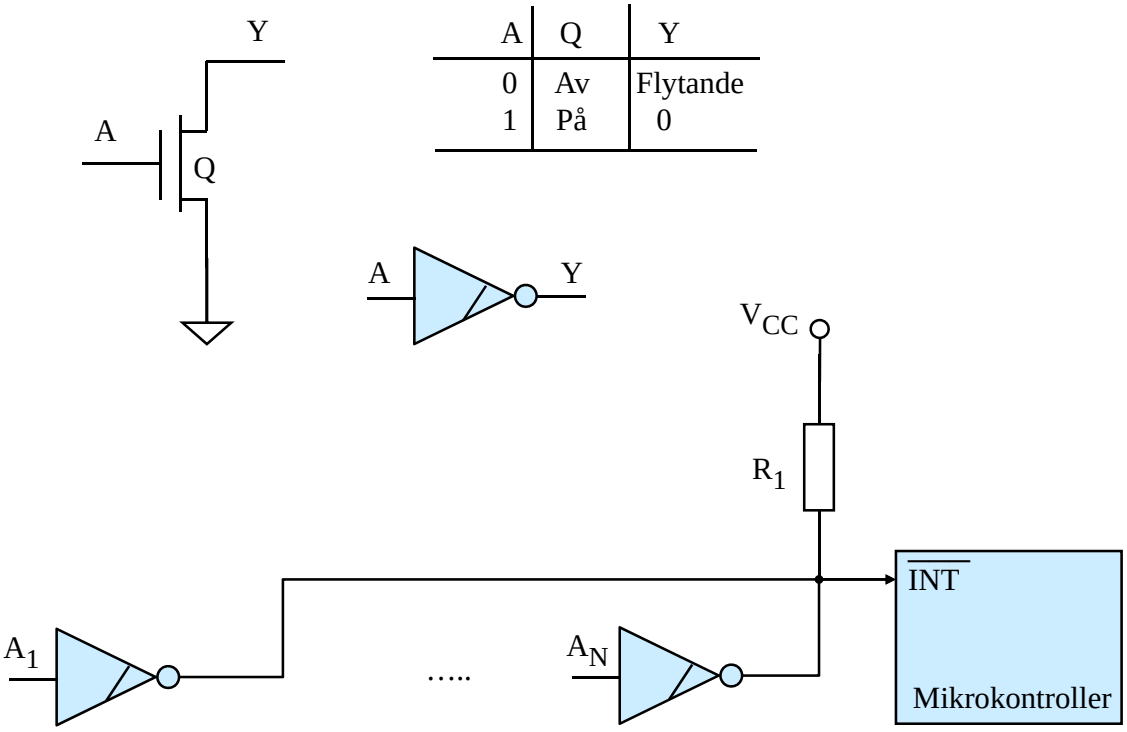
Figur 3.23: Bussystem i enkel mikrokontroller.

Når ein av perifermodulane skal levera data til mikroprosessoren, er det viktig at dei andre perifermodulane ikkje påverkar nivået på dei datalinjene som er felles. Alle modulane inneheld derfor kvar sin *tristate*-buffer på alle utgangane slik at utgangane kan fråkoblast bussen når perifermodulen ikkje er aktivert. Utbufferane i figuren er altså *tristate*-buffer. Innbufferane i figuren er av den vanlege typen vist i figur 3.21. Utan desse ville det blitt for tungt for mikroprosessoren å driva busslinjene[36].

[35] Sjå "en.wiki three-state logic".
[36] Det vil vera bufferar på adresse- og styrebussen også, men dette er ikkje vist i figuren.

3.5.2.4 Buffer med open *drain*

I nokre spesielle tilfelle kan ein bruka eit enklare utgangstrinn. Viss fleire utgangar skal koblast til same linja der dei bare skal kunne trekkja linja ned til lågt logisk nivå og ellers vera fråkobla, kan ein bruka ein buffer med såkalla **open "drain"**[37] eller logiske kretsvariantar med eit slikt utgangstrinn, sjå figur 3.24.



Figur 3.24: Buffer med open "drain".

Eit eksempel er som vist nedst i figuren, ei avbrotslinje inn til ein mikrokontroller der N ytre kjelder skal kunne signalisera eit avbrot ("interrupt"). Ein eller fleire kan "dra i snora" [38]. Dei andre er fråkobla/flytande og påverkar ikkje nivået til linja.
Merk at avbrotssignalet $\overline{INT}$ i figuren vil vera **aktivt lågt** då eit lågt nivå signaliserer avbrot.

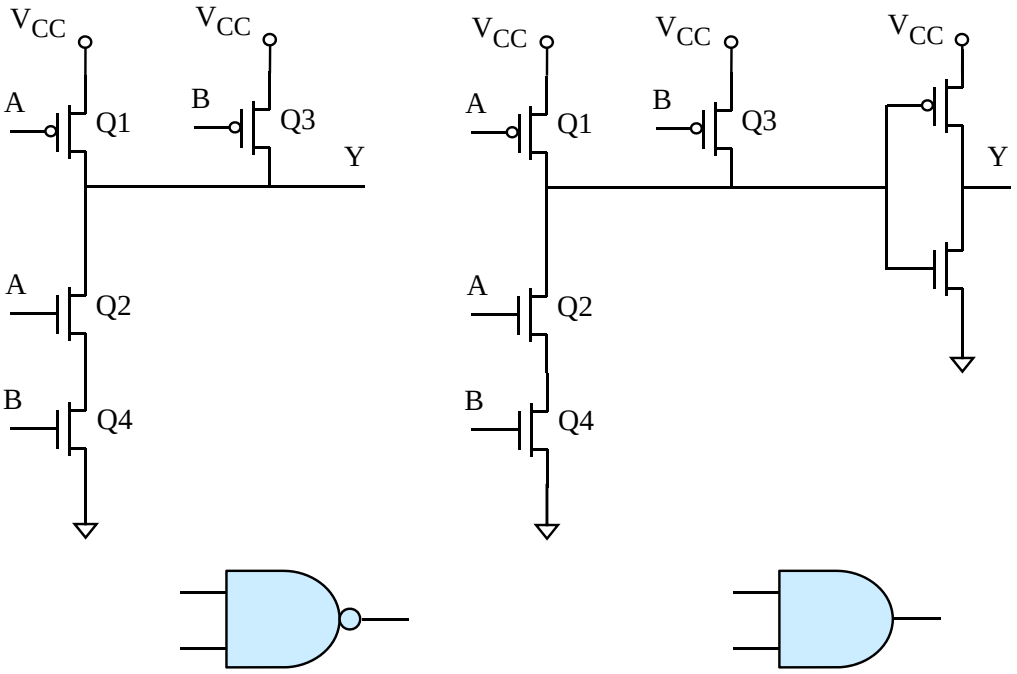
[37] Sjå "en.wiki open collector".
[38] Som i ein buss med snor for å gi stoppsignal.

3.5.3 OG-portar

Ein har to hovudtypar:

- OG-portar som inverterer resultatet, såkalla NOG-port ("NAND gate").
- Rein OG-port ("AND gate").

Oppbygging i CMOS-teknologi og symbol for begge typane er viste i figur 3.25.



Figur 3.25: NOG- og OG-port.

Av desse krev NOG-porten færrast transistorar, faktisk bare fire.
Merk: Det er vanleg å måla størrelsen til ein digital krets vhja. talet på portar ("gates"). Ein pleier då å rekna at **ein gjennomsnittleg port har fire transistorar**.

OG-porten krev som vist eit ekstra invertartrinn på utgangen.

Eksempel 3.9. *Verifisering av portfunksjon*

Ein skal her verifisera funksjonen til NOG-porten i figur 3.25. Dette skal gjerast vhja. ein utvida tabell av funksjonen $Y = f(A, B)$ for porten, der ein også viser tilstanden til dei enkelte transistorane Q1-4.

Ut frå figuren ser ein at dei ulike kombinasjonane av inngangssignala A og B vil gi transistortilstandar og utgangsverdiar som vist i tabell 3.6.

A	B	$Q1$	$Q2$	$Q3$	$Q4$	Y
0	0	På	Av	På	Av	1
0	1	På	Av	Av	På	1
1	0	Av	På	På	Av	1
1	1	Av	På	Av	På	0

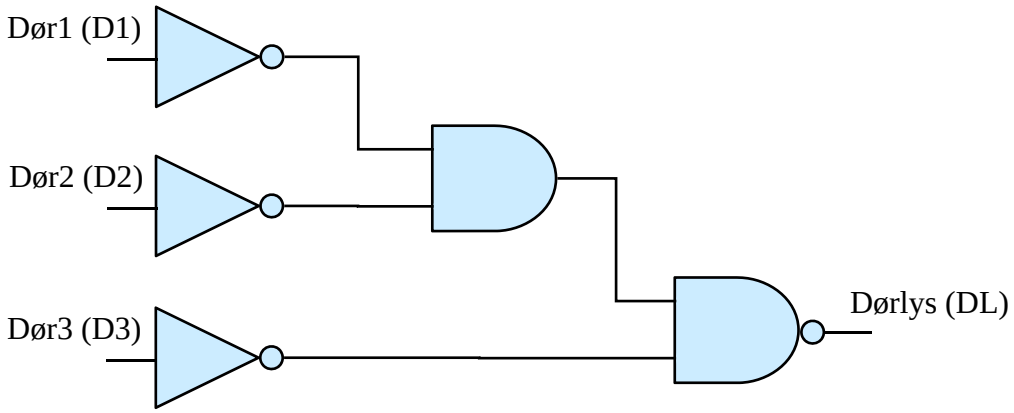
Tabell 3.6: Utvida funksjonstabell for NOG-porten.

Med referanse til tabell 3.3 på side 131, ser me at transistorkoblinga gir rett funksjon, dvs. NOG.

Eksempel 3.10. *Realisering av dørlyslogikken, alternativ 1*

Funksjonen til bildørlogikken kan altså formulerast slik: "Viss *dør1* **og** *dør 2* **og** *dør 3* er stengde, så skal *dørlyset* vera av."

Basert på funksjonstabellen i eksempel 3.8 på side 132 kan ei realisering av logikken sjå ut som i figur 3.26.
Det **logiske skjemaet** i figuren inneheld her begge dei to typane OG-portar samt tre invertarar.



Figur 3.26: Eksempel på realisering av bildørlogikken vist vhja. eit logisk skjema.

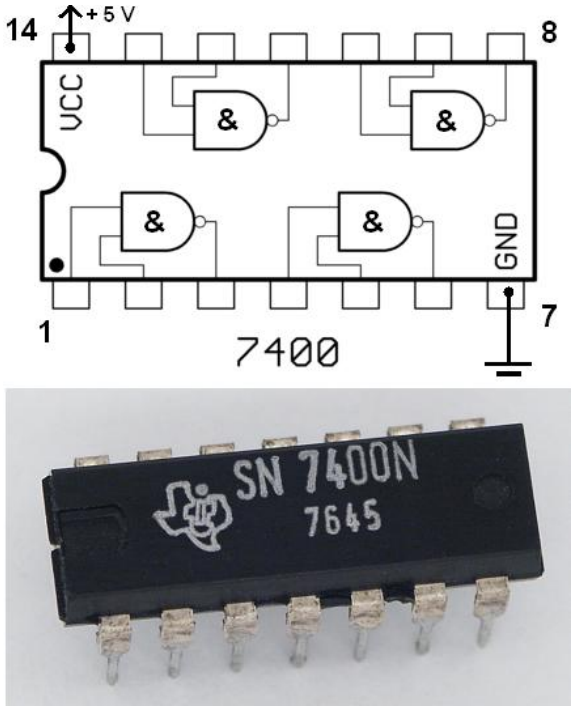
Denne realiseringa vil oppfylla funksjonstabellen i eksempel 3.8[39], men bruken av kretsar er ikkje optimal.
Samla sett går det med 16 transistorar, dvs. 4 standard-portar, også kalla **portekvivalentar**.

Ei meir optimal realisering blir vist i neste kapittel.

Før dei såkalla programmerbare elektroniske kretsane slo i gjennom for fullt, realiserte ein logikk som i eksemplet over ved å kobla saman integrerte kretsar på eit kretskort.

[39] Du kan jo gjera ei verifisering av dette.

Dei første IC-ane kom som nemnt før, for fullt på 60-talet. Dette kunne f.eks. vera kretsar av typen vist i figur 3.27 som inneheld 4 NOG-portar.



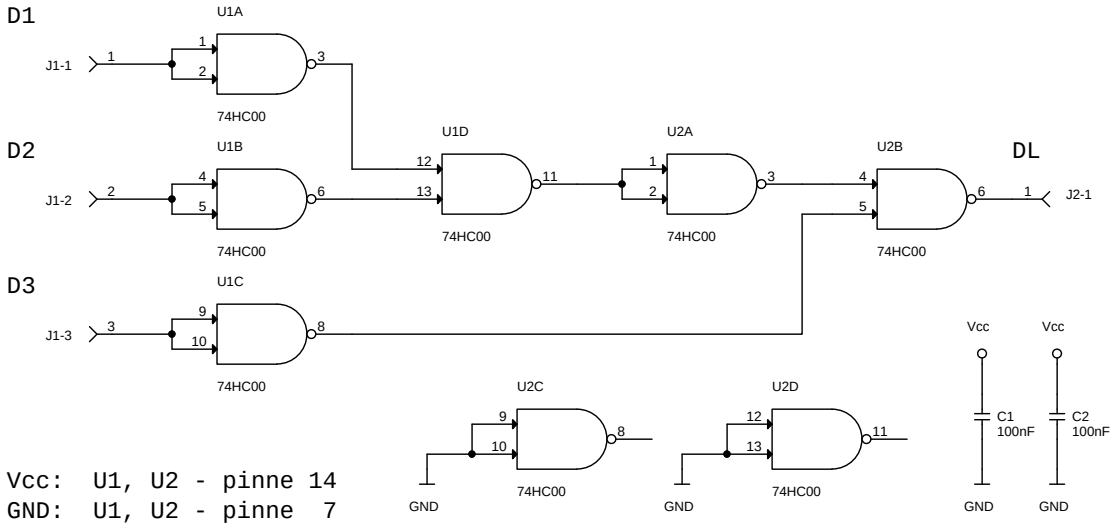
Figur 3.27: IC-eksempel. (Ref.: ”en.wiki 7400_adriusa_wiki_logic_gate”.

Etter kvart blei det utvikla ei heil rekkje ulike funksjonar realisert som integrerte kretsar av ein bestemt teknologi. Dei første IC-ane blei realisert vhja. såkalla *resistor-transistorlogikk* (RTL), jfr. kapittel 3.4.3.
Så kom den lenge populære *transistor-transistorlogikken* (TTL), som var raskare. Begge desse var basert på bipolare transistorar, som har eit mykje høgare statisk effektforbruk enn felteffekttransistoren.
Felteffekttransistoren er som vist i kapittel 3.4.4, basis for CMOS-teknologien som dominerer i dag.
Med inntoget av CMOS-teknologien kom det CMOS-alternativ for alle funksjonane som før bare var realiserte som TTL-kretsar. Digitale logiske funksjonar blei då i aukande grad realiserte vhja. CMOS-kretsar som byggjesteinar[40].

[40]Ei liste over logiske funksjonar realisert som IC-kretsar kan finnast med søkjeorda ”en.wiki List of 7400 series integrated circuits”.

3.5.4 Litt om å teika koblingsskjema for elektronikk

Figur 3.28 viser eit **koblingsskjema**[41] for bildørlogikken[42] basert på to IC-kretsar med same innhald som i figur 3.27[43].



Figur 3.28: Koblingsskjema for bildørlogikken realisert vhja. NOG-portar.

Når ein skal kobla opp elektronikk på laboratoriet, er det viktig å teikna[44] **eit detaljert koblingsskjema først**. Figuren gir eit godt eksempel på dette. I lista under er det både nyttige samt heilt nødvendige tips ved teikning.

- Pinnenr. og evt. namn på alle **tilkoblingspunkt** må visa i skjemaet. Unntak er motstandar, R, og upolariserte kondensatorar, f.eks. C-ane i figuren, der ein kan kobla seg til i kva ende ein vil.
- Kor forsyningsspenning V_{CC} og jord (GND) skal koblast til, må også visa i skjemaet. I standardserien *74*00* av IC-ar er pinnane for V_{CC} og jord (GND) plassert som vist i figur 3.27.
- Alle komponentar må **nummererast** og ha forklarande **typenamn** og eventuell **komponentverdi**. Vanleg nummerering er
  - *Un* for IC-ar der *n* er komponentnummer,
  - *Jn* for kontaktar og *Cn* for kondensatorar. Ein har også
  - *Sn* for brytarar, *Rn* for motstandar mfl[45].

[41] Skjemaet er teikna i OrCad, som er eit kraftig verktøy for konstruksjon av elektronikk. Ein **konstruksjonsprosess** inneheld fasane spesifikasjon, skjemateikning, evt. simulering, så utlegg og produksjon av kretskort.

[42] Sjå figur 3.26.

[43] Som ein ser, kan ein NOG lett brukast som ein inverter.

[44] For eit enkelt system kan dette greitt teiknast for hand på eit papir.

[45] Ei liste over slike merkelappar (”designator”) er vist hjå ”en.wiki Electronic symbol”.

Eit eksempel på typenamn er *74HC00* som vist i figuren. **Merk**: Viss komponentar inneheld fleire modular av same type slik som dei to IC-ane i figuren, blir modulane/portane nummererte som A, B, osv.

- Signalflyten bør vera frå venstre mot høgre, dvs. at inngangar kjem inn frå venstre og utgangar ut mot høgre.
- Komponentar som f.eks. portane i figuren, bør også ha ein hovudretning. Ein kan likevel ved behov også ha komponentar som er orienterte 90^o på hovudretninga.
- Alle ubrukte inngangar må **terminerast**, dvs. koblast til eit punkt som har eit sikkert logisk nivå, f.eks. jord (GND). Elles kan inngangen sveva og gi opphav til oscillasjonar og støy i kretsen. Dette vil også gi høgare effektforbruk.
- Viss ein har som mål å laga kretskort av systemet, må ein som i figuren ta med **avkoblingskondensatorar**. Desse skal stå nær kvar sin IC-krets og stabiliserer[46] forsyningsspenninga til kretsen.

3.5.5 ELLER-portar

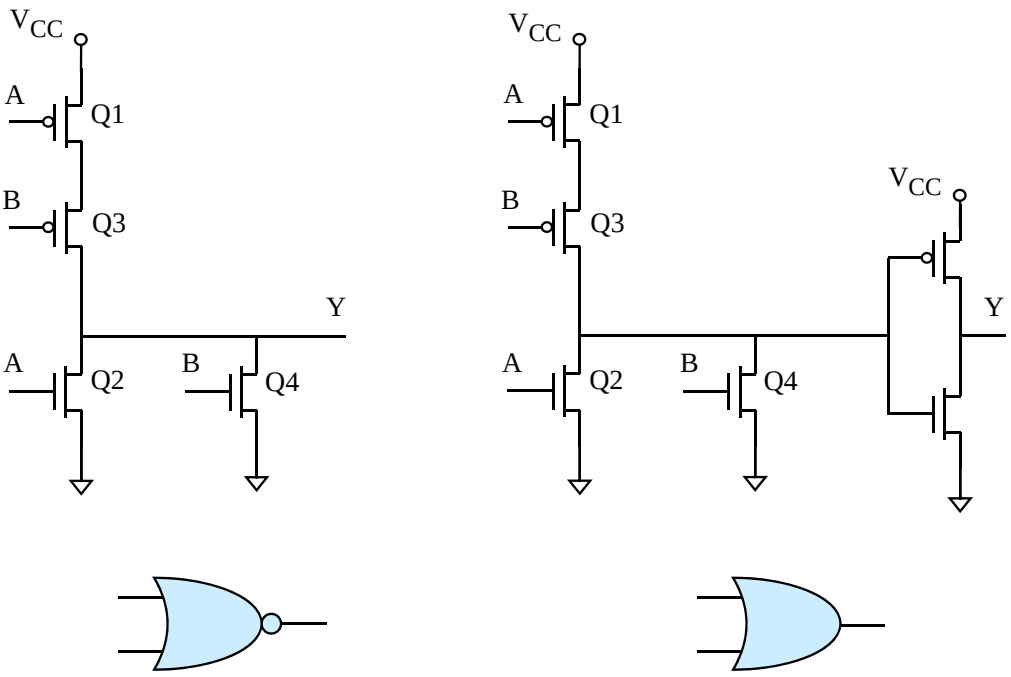
Ein har her også to hovudtypar:

- ELLER-portar som inverterer resultatet, såkalla NELLER-port ("NOR gate").
- Rein ELLER-port ("OR gate").

Oppbygging CMOS-utgåver av portane og symbol for begge typane er viste i figur 3.29.

Som for Og-portane, krev den inverterande ELLER-porten færrast transistorar, nemleg bare fire.
ELLER-porten krev eit ekstra invertartrinn på utgangen som vist i figuren.

[46] Sjå f.eks. "wiki Decoupling capacitor".



Figur 3.29: NELLER- og ELLER-port.

Eksempel 3.11. *Realisering av dørlyslogikken, alternativ 2*

Funksjonen til bildørlogikken kan som vist før, også formulerast slik:

"Viss *dør1* **eller** *dør 2* **eller** *dør 3* er open så skal *dørlyset* vera på."

Denne logikken skal realiserast ved å gjera følgjande:
- Setja opp logisk uttrykk.
- Teikna logisk skjema basert på bruk av ELLER-porten i figur 3.20.

Realiseringa skal baserast på definisjonane i eksempel 3.8 på side 132 og oppkoblinga i figur 3.20 på side 133.

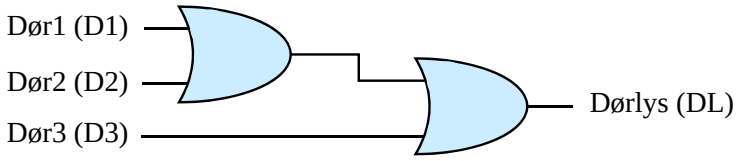
Ut frå det logiske skjemaet skal ein finna ut kor mange transistorar går med her. Dette skal også gjerast også dette om til portekvivalentar.

Ut frå den "prosaiske" formuleringa av logikken som er gitt over, blir det logiske uttrykket enkelt og greitt slik:

$$DL = D1 + D2 + D3$$

Det logiske skjemaet blir då som vist i figur 3.30.

Ut frå denne figuren samt figur 3.29 ser ein at det vil gå med 12 transistorar, dvs. 3 portekvivalentar her.
Dette er 4 færre transistorar enn løysingsalternativ 1, som blei vist i eksempel 3.10.
Alternativ 2 er altså ei meir effektiv løysing.



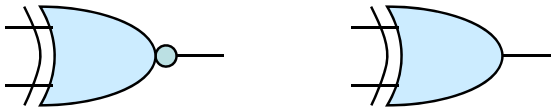
Figur 3.30: Logisk skjema for realisering av bildørlogikken vhja. ELLER-portar.

3.5.6 Eksklusive ELLER-portar

Ein har to hovudtypar:

- Ekslusiv ELLER-port som inverterer resultatet, såkalla XNELLER-port ("XNOR gate").
- Rein XELLER-port ("XOR gate").

Symbol for begge typane er viste i figur 3.31.



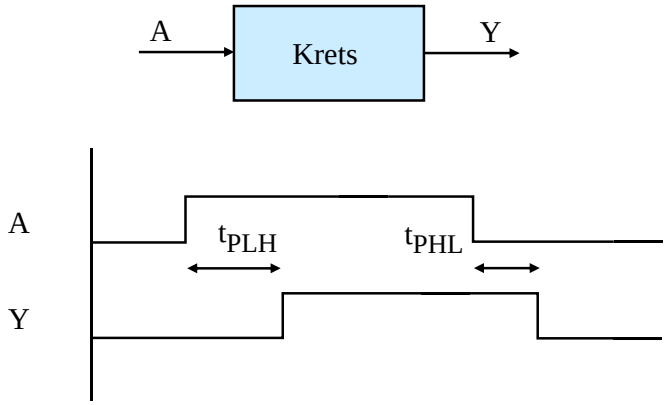
Figur 3.31: Dei to variantane av portar med eksklusiv ELLER-funksjon

Portar med eksklusiv ELLER-funksjon kan realiserast vhja. ein passande kombinasjon av dei andre portane me har sett på. Meir om dette kjem i kapittel 3.6.5.

3.5.7 Litt om tidsforseinking gjennom portar

Som me såg i kapittel 3.4.4.4 om effektforbruk i CMOS-kretsar, så må eit inngangstrinn ladast opp før transistorar kan slå seg på. Spenninga på inngangen må som kjent ladast opp til over terskelspenninga V_{th} før det kan begynna å danna seg ein kanal i ein transistor, jfr. kapittel 3.4.2.
Spenninga må så auka ytterlegare før kanalen er fullt utvikla og transistoren er slått heilt på.

På same vis må inngangar ladast ut før transistorar slår seg av. Viss ein port er bygt opp av fleire etterfølgjande trinn, blir dette ein slags **domino-effekt** gjennom alle trinna i porten som endar med at utgangen endrar tilstand.
Poenget er altså at dei digitale signala bruker tid på å koma seg gjennom ein port, og denne tida blir vanlegvis kalla **etterslep**, **tidsforseinking** eller **transportforseinking** ("propagation delay"), t_{PD}[47].
Denne tida kan vera avhengig av kva tilstand kretsen skiftar frå som illustrert i figur 3.32.



Figur 3.32: Etterslep gjennom ein digital krets.

Tidene t_{PLH}[48] og t_{PHL} eller alternativt ei felles tid t_{PD} er oppgitt i databladet for ein krets.
I figuren er som ein ser, kretsen eit vanleg, dvs. ikkje-inverterande buffer.

Eksempel 3.12. *Etterslep gjennom ein CMOS-krets*

Ein moderne CMOS-variant er den såkalla AHC-teknologien ("Advanced High speed CMOS").
Etterslepa for invertaren 74AHC04 finn ein saman med med mykje anna i databladet[49] for kretsen.

[47] Sjå også "en.wiki Propagation delay".
[48] Dvs. tida frå ei inngangsendring og til utgangen går frå lågt til høgt nivå.
[49] Søk på "Farnell.no SN74AHC04D" og opna databladet "Technical data sheet" frå produsenten Texas Instruments. Det finst også fleire andre produsentar av slike kretsar.
Farnell er ein stor leverandør av elektronikk og blir brukt av mellom anna UiS.

Med ei vanleg forsyningsspenning på 3.3 Volt og vanleg kapasitiv last på utgangen, vil ein ha
- minimalt etterslep, $t_{PLH,min} = t_{PHL,min} = 1$ nanosekund og
- maksimalt etterslep, $t_{PLH,max} = t_{PHL,max} = 10.5$ ns.

I databladet kan ein også mellom anna finna spenningsområda for høgt og lågt logisk nivå[50], sjå side 3.

Det er viktig å ta omsyn til etterslepa ved konstruksjon av digital elektronikk, og spesielt i sekvensiell logikk. Meir om dette kjem i kapittel 3.9.

3.6 Boolsk algebra

3.6.1 Innleiing

Som ein såg ved realisering av dørlyslogikken i kapitla 3.5.3 og 3.5.5, **så kan fleire ulike portkombinasjonar realisera same digitale funksjon**.
Slik dørbrytarane og dørlyset var tilkobla, viser det seg at bruk av ELLER-portar vil gi meir optimal logikk[51] enn første varianten, som blei vist i figur 3.26.

Det er ikkje lett å sjå kva portkombinasjon som gir den beste løysinga, sjølv for ein relativ enkel funksjonsspesifikasjon. Det har derfor vist seg nyttig å kunne bruka matematikk her.
Logiske operasjonar kan beskrivast vhja. såkalla Boolsk algebra[52]. I digitalteknikk blir Boolsk algebra brukt på variablar som har bare to ulike verdiar, nemleg **sann/høg/1** og **usann/låg/0**.

Som ein såg i kapittel 1.1.1, så var det **Claude Shannon** som viste at digitale system kunne beskrivast eller **modellerast** slik. Han viste også med basis i denne algebraen måtar å forenkla eller **optimalisera** slike system på. I tillegg såg han på korleis ein vhja. brytar- og relekoblingar kunne **løysa algebraiske problem**, dvs. gå frå ein funksjonsspesifikasjon til ei realisering av eit system.
Han la med dette grunnlaget for **syntese** eller **konstruksjon av datamaskinar**.

[50] Jfr. kapittel 3.4.4.2.
[51] Sjå eksempel 3.11 på side 143.
[52] Boolsk algebra blei utvikla av George Boole og publisert i 1854 i boka hans "An Investigation of the Laws of Thought", sjå "en.wiki Boolean algebra".

3.6.2 Grunnleggjande reglar

Grunnleggjande reglar for Boolsk algebra er viste under.

Ombytting/kommutering:

$$A \cdot B = B \cdot A \tag{3.3}$$
$$A + B = B + A \tag{3.4}$$

Kobling/assosiering:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C) \tag{3.5}$$
$$(A + B) + C = A + (B + C) \tag{3.6}$$

Fordeling/distribuering:

$$(A + B) \cdot C = (A \cdot C) + (B \cdot C) \tag{3.7}$$
$$(A \cdot B) + C = (A + C) \cdot (B + C) \tag{3.8}$$

Eigenskapar ved OG-operasjonen:

$$A \cdot 0 = 0 \tag{3.9}$$
$$A \cdot 1 = A \tag{3.10}$$
$$A \cdot A = A \tag{3.11}$$
$$A \cdot \overline{A} = 0 \tag{3.12}$$

Eigenskapar ved ELLER-operasjonen:

$$A + 0 = A \tag{3.13}$$
$$A + 1 = 1 \tag{3.14}$$
$$A + A = A \tag{3.15}$$
$$A + \overline{A} = 1 \tag{3.16}$$

Invertering:

$$\overline{\overline{A}} = A \tag{3.17}$$

De Morgan[53] sine reglar:

$$\overline{A \cdot B} = \overline{A} + \overline{B} \tag{3.18}$$
$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{3.19}$$

I eksemplet under blir den første regelen til De Morgan vist vhja. funksjonstabell.

[53] Sjå "nn.wiki Augustus De Morgan".

Eksempel 3.13. *Bevis for ein av De Morgan sine reglar*

Beviset for regelen $\overline{A \cdot B} = \overline{A} + \overline{B}$ kjem fram av tabellen under.

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

Viss den logiske funksjonen til eit system er spesifisert vhja. eit sett med Boolske likningar, kan me bruka dei viste reglane til å omarbeida likningane. Det vil alltid vera behov for å optimalisera likningssettet slik at ein oppnår følgjande:

- Færrast mogleg portar som igjen gir minimalt logikkareal og effektforbruk.
- Minimalt etterslep t_{PD} gjennom logikken, dvs. raskast mogleg logikk.

3.6.3 Spesifikasjon av logisk funksjon vhja. Boolsk algebra

Vhja. Boolsk algebra kan ein setja opp **logiske likningar** som beskriv oppførselen til eit logisk system. Me bruker igjen her dørlyslogikken som eksempel.

Eksempel 3.14. *Logisk likning for dørlyslogikken*

Med formuleringa
"Viss *dør1* **og** *dør 2* **og** *dør 3* er stengde, så skal *dørlyset* vera av",

og definisjonane som vist tidlegare, der
- stengt tilsvarer "Låg", dvs. "0" og
- av tilsvarer "Låg", dvs. "0",

blir den logiske likninga for dørlyslogikken slik:

$$\overline{DL} = \overline{D1} \cdot \overline{D2} \cdot \overline{D3}$$

Ein har her brukt dei namneforkortingane som er viste i figur 3.26 på side 139.

Altså:
Når f.eks. dør 1 er stengt, er variabelen $D1 = 0$. Då følgjer det at $\overline{D1} = 1$.
Likeeins har me at når dørlyset er av, er variabelen $DL = 0$, og dermed $\overline{DL} = 1$.

Av dette ser me at likninga er i samsvar med formuleringa i starten av eksemplet.

3.6.4 Forenkling av logisk funksjon vhja. Boolsk algebra

Vhja. dei grunnleggjande reglane i kapittel 3.6.2 kan ein optimalisera, dvs. forenkla likningssettet for eit logisk system. Spesielt De Morgan sine reglar er sentrale her.

Eksempel 3.15. *Bruk av dei grunnleggjande reglane*

Gjer følgjande uttrykk enklare:

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

Det er fleire måtar å gjera dette på. Her er ein av dei:

$$\begin{aligned} A \cdot \overline{B} + \overline{A} &= \overline{\overline{A\overline{B}} \cdot \overline{\overline{A}}} = \overline{(\overline{A} + \overline{\overline{B}}) \cdot A} \\ &= \overline{\overline{A}A + BA} = \overline{BA} = \overline{B} + \overline{A} \end{aligned} \tag{3.20}$$

Viss ein støyter på eit tilsvarande utgangspunkt ved oppgåveløysing, er det bare å visa til resultatet i dette eksemplet og bruka det i utrekningane.

Eksempel 3.16. *Forenkling av logisk likning for dørlyslogikken*

Likninga for dørlyslogikken frå eksempel 3.14 er som følgjer:

$$\overline{DL} = \overline{D1} \cdot \overline{D2} \cdot \overline{D3}$$

Denne kan forenklast slik ved bruk av De Morgan sine reglar:

$$\begin{aligned} \overline{DL} &= \overline{D1} \cdot \overline{D2} \cdot \overline{D3} = (\overline{D1} \cdot \overline{D2}) \cdot \overline{D3} \\ &= \overline{\overline{(\overline{D1} \cdot \overline{D2})}} \cdot \overline{D3} = \overline{(D1 + D2)} \cdot \overline{D3} \\ &= \overline{(D1 + D2) + D3} \end{aligned} \tag{3.21}$$

Ved å invertera begge sidene av likninga får ein så at

$$DL = D1 + D2 + D3$$

Realisering av denne likninga gir enklare logikk, dvs. færre logiske portar, og samsvarer med resultatet i eksempel 3.11 på side 143.

3.6.5 Generering av logiske likningar ut frå funksjonstabell vhja. SoP-metoden

Basert på funksjonstabellar kan ein også koma fram til logiske likningar for eit system. Likningane kan då vera på ei form som beskriv funksjonane til systemet som **produktsummar** (”Sum of Products”, SoP).
Framgangsmåten kan illustrerast vhja. eit enkelt eksempel.

Eksempel 3.17. *Generering av ein produktsum frå funksjonstabell*

Me tar utgangspunkt i funksjonstabellen til eit system med to inngangsvariablar A og B og ein utgangsvariabel Y:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Av tabellen ser me greitt at systemet har ein rein ELLER-funksjon.

SoP-metoden går ut på å setja opp ei logisk likning for kvar utgang som ein **sum** av dei inngangskombinasjonane som gir utgangsverdi lik 1. Altså:

$$Y = \overline{A} \cdot B + A \cdot \overline{B} + A \cdot B$$

Likninga kan lesast slik:
"Y er 1 viss A invertert er 1 og B er 1 **eller** *A er 1 og B invertert er 1* **eller** *A er 1 og B er 1".*

Me skal ikkje her føra noko generelt bevis for at likningane blir rette[54] men skal nøya oss med å visa ved bruk av Boolsk algebra at dette er ein ELLER-funksjon:

$$\begin{aligned} Y &= \overline{A} \cdot B + A \cdot \overline{B} + A \cdot B = \overline{A} \cdot B + A \cdot (\overline{B} + B) = \overline{A} \cdot B + A \cdot 1 \\ &= (\overline{A} \cdot B) + A = \overline{[(\overline{A} \cdot B) \cdot \overline{A}]} = \overline{[(\overline{\overline{A}} + \overline{B}) \cdot \overline{A}]} \\ &= \overline{[A \cdot \overline{A} + \overline{B} \cdot \overline{A}]} = \overline{[0 + \overline{A} \cdot \overline{B}]} = A + B \end{aligned} \tag{3.22}$$

q.e.d.

[54] Ein kan lesa meir om dette under "en.wiki Canonical form (Boolean algebra)".

Eksempel 3.18. *Generering av ein produktsum for bildørlogikken*

Me tar utgangspunkt i funksjonstabellen for bildørlogikken som blei sett opp i eksempel 3.8 på side 132, men med namneforkortingane frå eksempel 3.14:

$D1$	$D2$	$D3$	DL
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

Ei logisk likning på SoP-form for utgangsvariabelen DL blir då:

$$\begin{aligned} DL = &\overline{D1} \cdot \overline{D2} \cdot D3 + \overline{D1} \cdot D2 \cdot \overline{D3} \\ &+ \overline{D1} \cdot D2 \cdot D3 + D1 \cdot \overline{D2} \cdot \overline{D3} + D1 \cdot \overline{D2} \cdot D3 \\ &+ D1 \cdot D2 \cdot \overline{D3} + D1 \cdot D2 \cdot D3 \end{aligned} \tag{3.23}$$

Dette er ei fullt ut gyldig likning for systemfunksjonen. Likninga er likevel **lite optimal** samanlikna med den minimale varianten frå eksempel 3.16 på side 149, nemleg:

$$DL = D1 + D2 + D3$$

Det er altså eit generelt behov for optimalisering av dei logiske likningane ein kjem fram til basert på ein funksjonsspesifikasjon. Elles kan logikken bli unødvendig stor og tilsvarande treg[55].
Det står meir om dette rett rundt hjørnet i kapittel 3.6.7.

Eksempel 3.19. *Realisering av logikk for funksjonen eksklusiv eller.*

Som nemnt i kapittel 3.5.6, gjenstår det å visa realisering av ein XELLER-funksjon. Dette skal gjerast i dette eksemplet gjennom følgjande steg:

1. Oppsett av logisk likning basert på funksjonstabellen i kapittel 3.5.1.2.
2. Teikning av logisk skjema.

Steg 1:
Eit utdrag av funksjonstabellen, dvs. tabell 3.3 på side 131, viser sjølve XELLER-funksjonen.

[55] Tilsvarande motivasjon har ein for optimalisering av programvare.

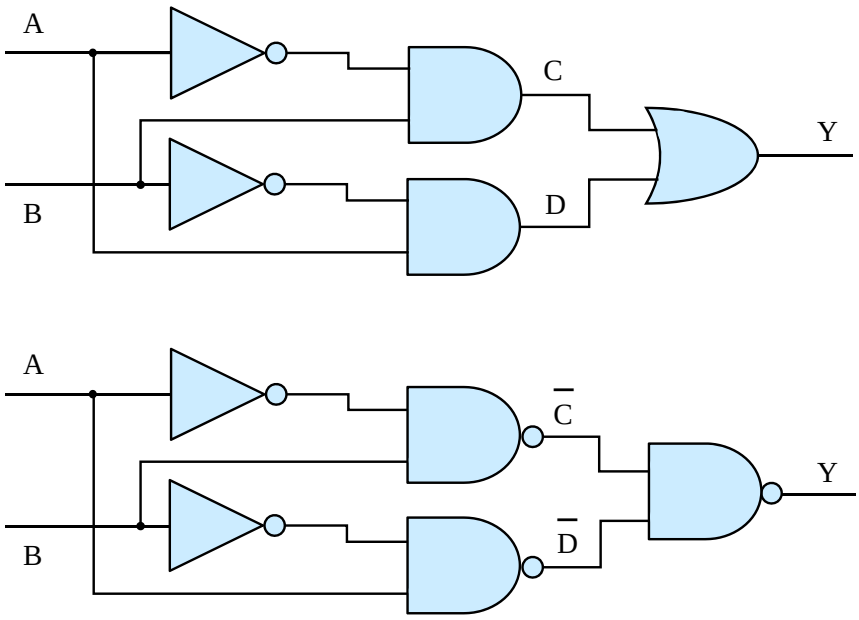
A	B	$Y = f(A,B) = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Basert på denne tabellen blir den logiske likninga på SOP-form for utgangen Y som følgjer:

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

der C og D er hjelpevariablar som blir brukte i neste steg.

Steg 2:
Eit logisk skjema for den logiske likninga over er vist øvst i figur 3.33.



Figur 3.33: To realiseringar av funksjonen eksklusiv eller.

Ei meir effektiv realisering kan ein få ved å utnytta følgjande:

$$Y = C + D = \overline{\overline{C} \cdot \overline{D}}$$

Denne realiseringa er vist nedst i figuren. Ein vil greitt kunne telja seg fram til at denne samla sett vil krevja seks færre transistorar enn den øvre realiseringa.

3.6.6 Ei lita oppsummering: Korleis spesifisera funksjonane til logiske system

Me har nå sett litt på dei tre hovudmetodane ein bruker for å spesifisera funksjonane til logiske system, nemleg

- Funksjonstabell
- Logiske likningar
- Logisk skjema

Som vist i førre kapittel, kan ein generera dei logiske likningane ut frå ein funksjonstabell viss det er ønskjeleg. Det er også ei smal sak å generera eit logisk skjema eller ein funksjonstabell basert på dei logiske likningane for eit system.

3.6.7 Litt om realisering av digital logikk

Realisering av logikk blir også ofte kalla **logisk syntese** ("logic synthesis") der siste ordet er gresk og betyr "å setja saman". Når det gjeld realisering, må ein skilja mellom følgjande logiske system:

1. Moderat støttelogikk for ein mikrokontroller i eit innbygd system.
2. Avansert støttelogikk. Dette kan f.eks. vera forprosesseringslogikk for digital video frå eit kamera i eit maskinsynssystem eller rask styrelogikk for ein robotarm.
3. Avansert logikk integrert på ei brikke og produsert i store volum.
   Slike kretsar kallar ein ofte kundespesifiserte integrerte kretsar ("Application Specific Integrated Circuit", ASIC)[56]. Dette er kretsar med funksjonar som er skreddarsydde for ein kunde, men omgrepet vil i vid forstand også inkludera mikroprosessor- og mikrokontrollerbrikker.
   Eit nyare omgrep er applikasjonsspesifiserte standardkretsar ("Application Specific Standard Product", ASSP) som er skreddarsydde kretsar som ein produsent tilbyr alle som vil ha.
4. Avansert logikk integrert på ei fleksibel brikke[57] i eit lite antal. Utviklingstida er her mykje kortare og modifikasjonar og reprogrammering kan gjerast veldig raskt, noko som er veldig nyttig i prototypefasen av eit utviklingsprosjekt. Mange kommersielle produkt i små antal blir også realiserte slik, men det er produkt der ein kan leva med at mellom anna effektforbruket i brikkene er høgare enn ved ASIC- eller ASSP-realisering.

[56] Sjå "nn.wiki ASIC" samt "en.wiki Application-specific integrated circuit"
[57] Brikketypen heiter FPGA ("Field Programmable Gate Array"), sjå neste kapittel.

I dette skrivet har ein hovudfokus på punkt 1, nemleg vanleg støttelogikk i eit elektronisk system, men punkt 4 blir også omtalt nærare i neste kapittel om programmerbar logikk.

Før dei programmerbare logiske kretsane blei dominerande, blei slik støttelogikk realisert ved å **teikna koblingsskjema** for det logiske systemet med utgangspunkt i **optimaliserte** logiske likningar og tilgjengelege IC-ar. Eit IC-eksempel blei vist i figur 3.27.
Optimalisering blei gjort for hand vhja. Boolsk algebra eller vhja. ein veldig populær grafisk metode basert på såkalla **Karnaugh-diagram**[58]. Ei SoP-likning med typisk opptil 6 variablar kunne forenklast til eit minimalt uttrykk.

Ved inntoget av **programmerbar elektronikk** skjedde følgjande:

- Det dukka opp eigne språk for spesifisering av maskinvare, såkalla "Hardware Description Language", **HDL**. Her kunne ein spesifisera både likningar og funksjonstabellar for kombinatorisk logikk og tilstandslikningar for sekvensiell logikk.
- Det kom verktøy for programmering av slik elektronikk. Her legg ein då inn HDL-kode, og verktøyet køyrer automatisk optimalisering[59] av dei logiske likningane spesifisert i HDL-språket. Koden blir så kompilert og lasta ned i ein elektronisk krets.
  Alt dette tilsvarer byggjeprosessen som eit utviklingsverktøy for Java- eller C-program køyrer basert på programkoden som ein lagar. Køyring av automatiske algoritmar for optimalisering er også her ein del av byggjinga.
- Etterkvart kom det verktøy av **høgare generasjon** der ein set saman logikkmodular. Første fasen av kompileringsprosessen genererer då HDL-kode for systemet. Denne blir så kompilert vidare som nemnt i førre punkt.

  Eit eksempel er verktøyet **System generator** frå Xilinx[60]. System Generator blir køyrt med verktøyet **Simulink** frå MathWorks som overbygg.
  Ein teiknar logikken i Simulink og har her tilgang på eit bibliotek frå Xilinx som inneheld ei rekkje ulike logikkblokker. Nokre kan også fyllast med din eigen HDL-kode. System Generator er sjølve kompilatoren.
  Dette verktøyet er eigentleg laga for realisering av **signalbehandlingsmetodar**[61] i FPGA. Dette er tunge metodar, og å leggja desse i maskinvare sparer mykje mikroprosessortid i ein datamaskin.
  System Generator er tidlegare brukt i eit signalbehandlingsemne ved UiS, og har også bibliotek med eit utval vanlege logikkblokker. Det blir derfor brukt i emnet Datamaskinarkitektur også for å illustrera realisering av logikkfunksjonar i FPGA.
  Meir om dette verktøyet kjem i kapittel 3.7.2.4.

  Eit anna eksempel på verktøy av høgare generasjon er **Embedded Development Kit** (EDK). Vhja. dette verktøyet kan ein byggja mikrokontrollerar i FPGA. Modular i mikrokontrolleren kan då i tillegg til sjølve mikroprosessoren vera serielle

[58] Sjå "en.wiki Karnaugh map".
[59] Dei manuelle metodane blei altså vidareutvikla til metodar som kunne køyrast i datamaskin. Det står ein del om denne utviklinga i "en.wiki Logic synthesis".
[60] Som nemnt før, er Xilix ein dei aller største produsenten av programmerbar elektronikk i verda.
[61] Eksempel er filtrering, frekvensomforming (FFT) og desimering mm.

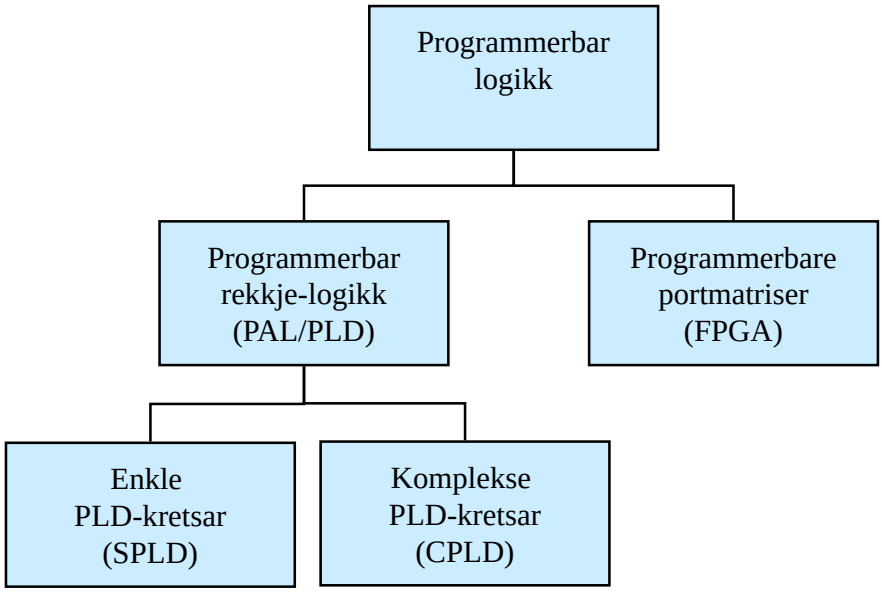
og parallelle portar, taimerar og minne mm. Mikrokontrolleren kallar me då **mjuk** ("soft"). Meir om mjukC-ar og EDK kjem i kapittel 3.7.4.

I neste kapittel vil ein sjå nærare på programmerbar elektronikk.

3.7 Programmerbar logikk

3.7.1 Hovudtypar av programmerbar logikk

Programmerbar logikk kan delast inn som vist i figur 3.34.

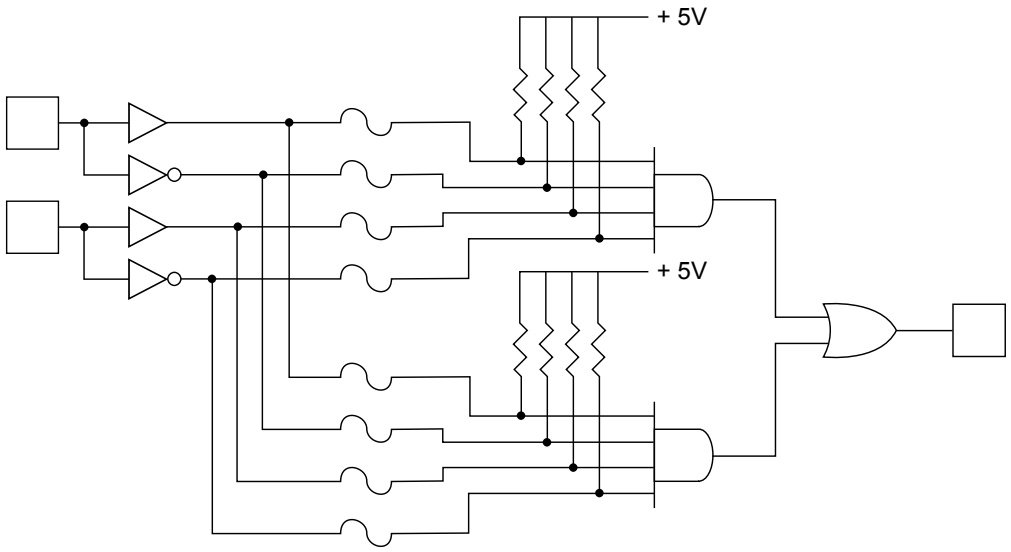


Figur 3.34: Ulike typar programmerbar logikk.

3.7.1.1 SPLD

Dei første programmerbare kretsane kan karakteriserast som **enkle programmerbare kretsar**, "Simple Programmable Logic Devices", som vist i figuren.
Slike kretsar kom på 1970-talet og var av typen MSIC, sjå tabell 1.2 på side 9. Kretsane blei i starten kalla "Programmable Array Logic", PAL[62]. Oppbygginga av dei første variantane er vist i figur 3.35.

[62] Sjå "en.wiki Programmable Array Logic".



Figur 3.35: Oppbygging av den første typen programmerbar logikk.
(Ref.: "Wiki Programmable_Logic_Device.svg".)

Kretsane inneheldt sikringar ("fuse") markert som liggjande *S*-ar i figuren. Programmeringa gjekk ut på å brenna av desse eller ikkje. Der ei sikring blir brent, vil opptrekksmotstanden til høgre for sikringa gjera at ein får eit høgt signal inn på den tilhøyrande OG-porten. Utgangen på OG-porten vil slik ikkje bli påverka av ei linje der sikringen er brent.
Som ein ser, har basismodulen i figuren to inngangar og ein utgang og kan realisera ei SOP[63]-likning med to produkt. I produkta kan ein ha inngangsvariablane som dei er eller som inverterte utgåver.

Ein PAL-krets var bygt opp av fleire slike basismodular for å kunne ha ein viss kapasitet når det gjaldt inngangar, utgangar og SOP-produkt. Som vist i figuren, er ELLER-delen fast, dvs. ikkje programmerbar. To produkt vil altså summerast gjennom ELLER-porten i basismodulen.
I ein vanleg PAL-krets kunne ein typisk summera saman 8 produktledd for kvar utgang.

[63] Sjå kapittel 3.6.5.

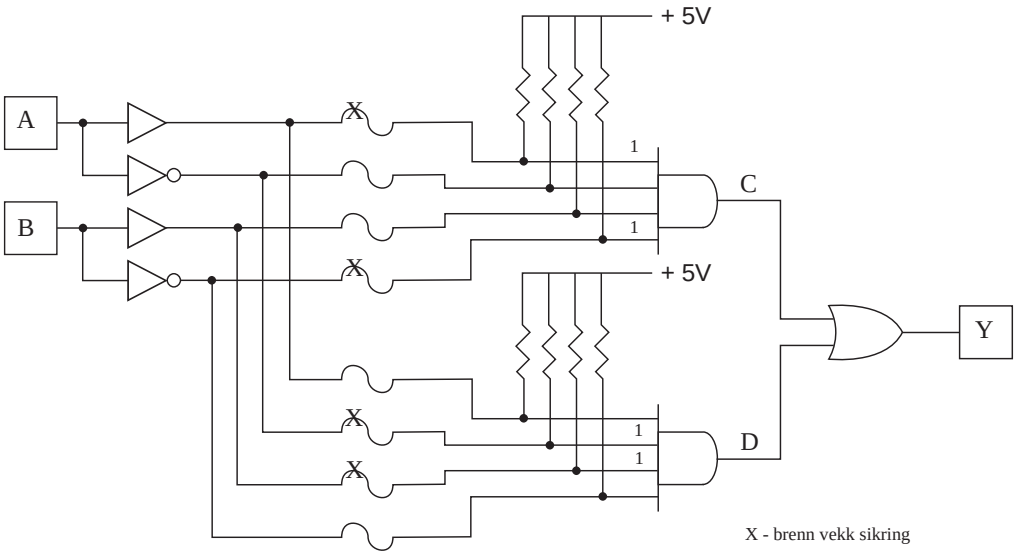
Eksempel 3.20. *Realisering av logikk for funksjonen eksklusiv eller i ein PAL-krets.*

I eksempel 3.19 på side 151 såg ein på to ulike realiseringar av ein XELLER-funksjon basert på vanlege logiske portar. Logisk likning for den eine av desse realiseringane var som følgjer:

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

der C og D er hjelpevariablar.

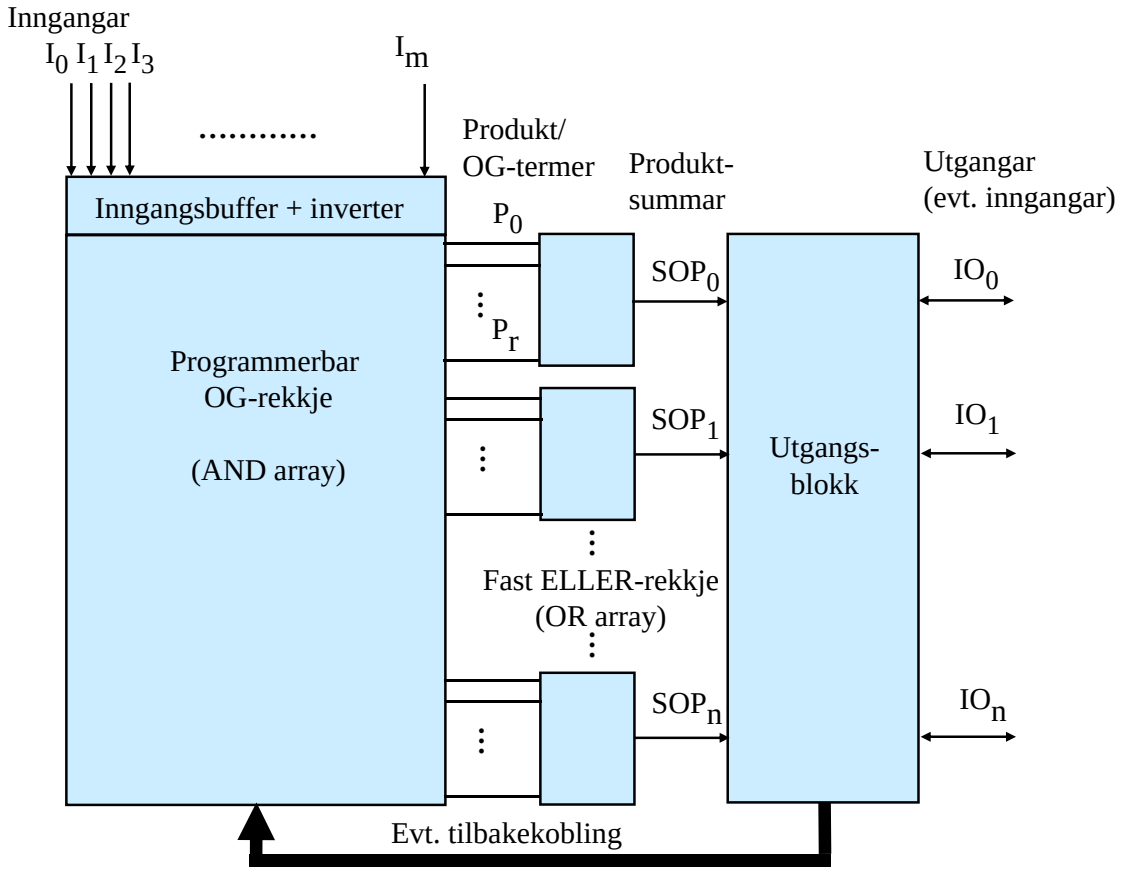
Ved å brenna av bestemte sikringar kan ein realisera denne funksjonen i ein PAL-krets. Dette er vist i figur 3.36.



Figur 3.36: Realisering av XELLER i ein PAL-krets.
(Fritt etter: "Wiki Programmable_Logic_Device.svg".)

Neste generasjon av programmerbare kretsar blei ofte kalla ”Generic Array Logic”, GAL, som er ein fleirgongsprogrammerbar PAL. Sikringane er då erstatta av spesielle transistorkoblingar, som kan reprogrammerast[64].

Som i PAL-kretsen i figur 3.35, er ELLER-rekkja i SOP-strukturen til GAL-kretsar fast. Dette er også illustrert i figur 3.37.



Figur 3.37: Oppbygginga av enkle programmerbare kretsar, SPLD.

Kretsar av typen **SPLD** av i dag, har ein struktur som vist i figuren.

[64]Dette er tilsvarande teknologi som i eit Flash-minne, der kvar minneselle også består av ein spesialtransistor. Meir om dette i kapittel 3.10.9.7.

Eksempel 3.21. *Generering av ein produktsum, SOP, for eit system med fire inngangar og ein utgang*

Ein vil bruka kretsen i figur 3.37 til å realisera følgjande logiske funksjon:

$$IO_0 = I_1 \cdot I_2 + \overline{I_3} \cdot I_4$$

Dette kan gjerast ved å la dei såkalla **produkttermene** eller OG-termene P_0 og P_1 bli slik:

$$P_0 = I_1 \cdot I_2$$
$$P_1 = \overline{I_3} \cdot I_4$$

Dette blir som vist før, realisert ved å programmera samband mellom inngangar og tilhøyrande OG- eller produktrekkjer. Ein har også som vist i figur 3.37, dei inverterte inngangsnivåa tilgjengelege for bruk i OG-rekkja.
Dei to produkta P_0 og P_1 blir kobla til den faste ELLER-modulen med utgangen SOP_0. Denne er igjen er kobla til pinnen IO_0 via ei utgangsblokk.

Vanlege logiske funksjonar som den i eksemplet over, kan altså realiserast som ein **sum av produkt, SOP**.
I tillegg kan ein vhja. **utgangsblokka** realisera meir komplekse logiske funksjonar. Denne blokka kan mellom anna innehalda:

- Logikk som gjer at ein kan bruka IO-linjer som inngangar. Desse blir kobla inn på OG-rekkja via den breie pila i figur 3.37.
  Kretsen GAL22R10 er f.eks. ein krets med til saman 22 inn- og utgangar der 12 av desse er I-linjer og 10 er IO-linjer.
- Ei såkalla D-vippe[65], dvs. eit minnelement, for kvar utgang.
  Utgangen på sjølve vippa kan koblast tilbake til OG-rekkja slik at ein kan realisera ein tilstandsmaskin som f.eks ein teljar.
  Bokstaven "R" i namnet GAL22R10 vil seia at ein har eit register i utgangsblokka, dvs. ei D-vippe for kvar av utgangane.

Adressedekodaren i mindre datamaskiner, jfr. figur 2.2, er eit eksempel på ein logisk modul som ofte blei realisert i SPLD-kretsar.

[65] Meir om dette kjem i kapittel 3.9.

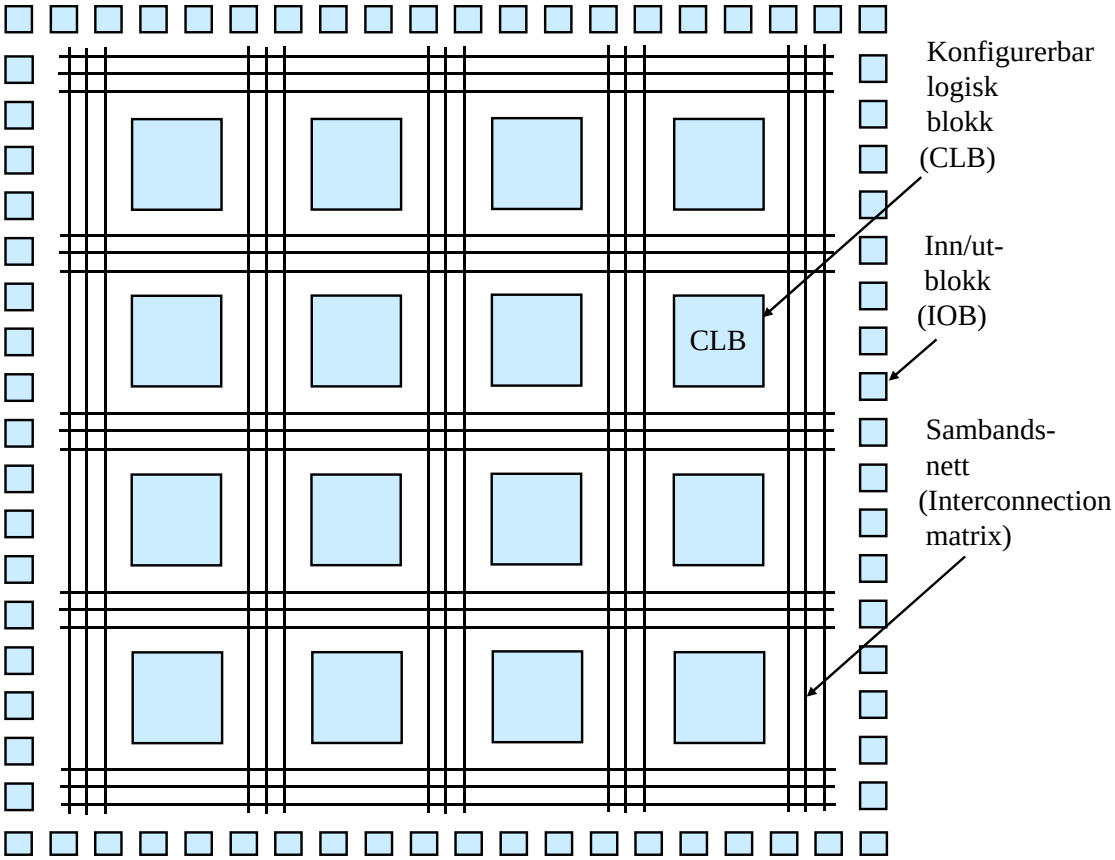
3.7.1.2 CPLD

Komplekse PLD-kretsar, CPLD, har ein meir omfattande og fleksibel innmat. Dei er grovt sett bygde opp av mange SPLD-modular som er knytte saman vhja. eit programmerbart grensesnitt ("programmable interconnection array", PIA).
CPLD-kretsane er av typen LSIC eller VLSIC[66].

3.7.1.3 FPGA

Dei aller største programmerbare kretsane er såkalla felt-programmerbare portmatriser[67], FPGA, der alle er av typen VLSIC eller ULSIC.
Ordet **"felt"** siktar til at kretsane kan programmerast "i felten", dvs. ute hjå brukaren. Det motsette vil vera å programmera kretsen i fabrikken, dvs. hjå kretsprodusenten.

Strukturen til FPGA-kretsar er vist i figur 3.38.



Figur 3.38: Oppbygging av tradisjonelle FPGA-kretsar.

[66] Sjå klassifiseringa i tabell 1.2 på side 9.
[67] Frå det engelske "Field Programmable Gate Array".

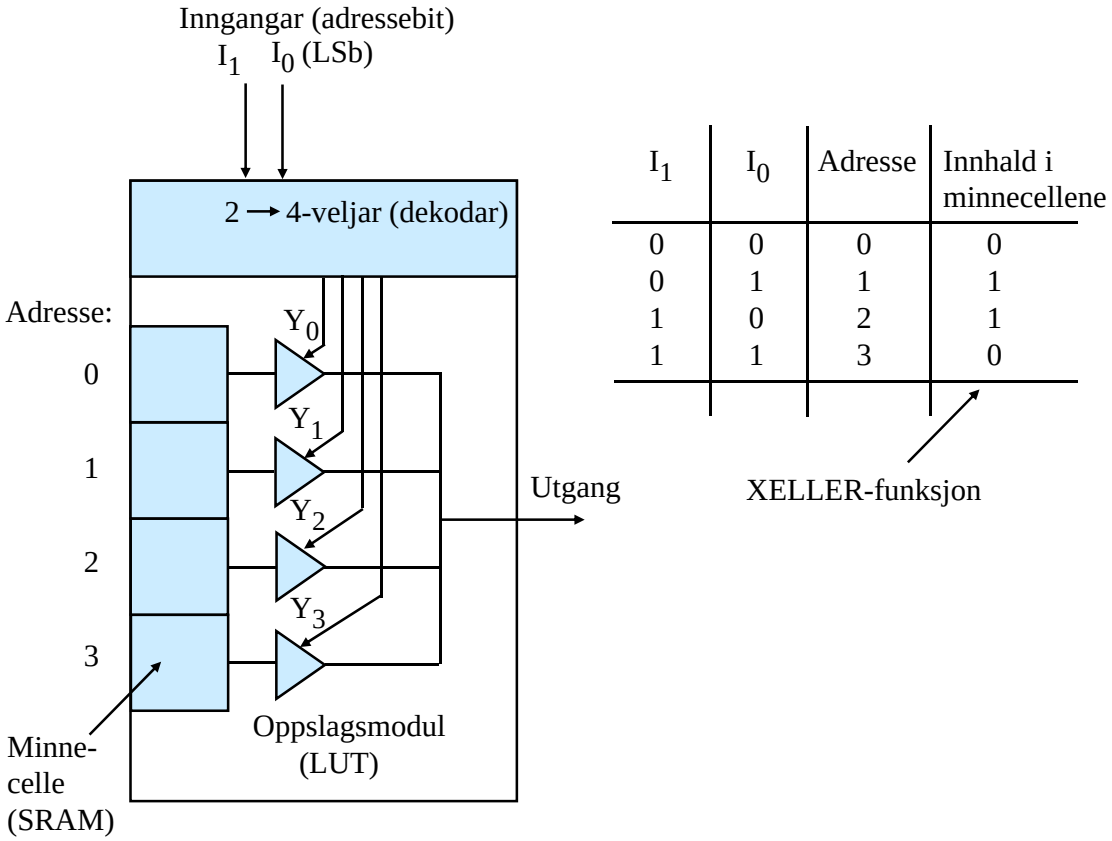
Dei konfigurerbare, dvs. programmerbare logiske blokkene, CLB, er kvar for seg mindre enn blokkene i ein CPLD, men det er mange fleire av desse. Det er også eit meir omfattande sambandsnettverk i ein FPGA.
Slik blir FPGA-kretsane generelt sett mykje kraftigare og fleksible enn CPLD-kretsar.

Kvar CLB i ein FPGA er bygd opp av fleire mindre logiske blokker. I kvar av desse blir ofte dei logiske funksjonane genererte vhja. **oppslagstabellar** ("look-up table", LUT). **LUT** er eit alternativ til SOP-prinsippet som altså blir brukt i SPLD-kretsar, jfr. figur 3.37.

Virkemåten til ein LUT er illustert vhja. eit eksempel.

Eksempel 3.22. *Enkel LUT for realisering av XELLER-funksjon i FPGA*

Figur 3.39 viser ein enkel logisk oppslagstabell eller -modul med to inngangar og ein utgang.



Figur 3.39: Modul som realiserer ein enkel logisk funksjon basert på oppslagstabell.

Inngangane $I0-1$ utgjer her samla sett ei 2-bits **adresse**, og vhja. veljar-/dekodarlogikk[68] vil ein på utgangen få ut innhaldet i den minnecella[69]

[68] I kapittel 3.8.2 skal ein sjå på korleis ein dekodar er bygt opp.
[69] Minnecellene er realisert som små SRAM-blokker i FPGA-en.

som har den valde adressa. Dekodaren aktiverer 1 av dei 4 styresignala Y_0 - Y_3 som er kobla til kvart sitt tri-statebuffer[70]. Vhja. styresignala og desse bufrane stenger eller opnar ein linjene frå minnecelle til utgang. Dei fire minnecellene er altså ein tabell, og ein gjer **oppslag** i denne vhja. inngangssignala.
Viss ein vil at modulen skal realisera ein viss logisk funksjon, i dette tilfellet "eksklusiv eller", må ein fylla cellene med rett innhald som vist i funksjonstabellen i figuren.

FPGA dominerer i dag marknaden for programmerbare kretsar.
Bruk av oppslagstabellar, LUT, er dermed den dominerande realiseringsforma for digital logikk nå. Ein kan som me har sett, spesifisera logiske funksjonar på fleire ulike måtar, men funksjonane blir altså vanlegvis realiserte vhja. LUT-ar i FPGA.
Meir om prosessen frå spesifikasjon til realisering kjem i kapittel 3.7.2.

Vanlegvis er talet på adressebit for ein LUT større enn vist i figur 3.39.
Her skal me bl.a. illustrera dette med eit FPGA-eksempel frå Spartan-familien til produsenten Xilinx. Den nest minste av desse, XC3S200, har i ei årrekkje vore i bruk undervisninga ved UiS, og litt frå databladet [30] er vist under.

- 200.000 portekvivalentar, dvs. typisk 800.000 transistorar.
- 480 CLB-ar der kvar CLB inneheld mellom anna 8 LUT-ar.
- Kvar av dei nær 4000 LUT-ane har 4 inngangar og 1 utgang.
- Typisk klokkefrekvens er 50 MHz, men det finst også raskare kretsutgåver.
- 32 kByte SRAM.
- 12 multiplikasjonsmodular.

Skal ein realisera bare ein enkel XELLER-port som vist i figur 3.39, treng ein bare 1 LUT.
Skal ein derimot realisera ein liten mjuk[71] mikrokontroller i denne kretsen, treng ein over 60% av alle CLB-ane i kretsen.

LUT-modulane gir ei fleksibel og effektiv realisering av logiske funksjonar.
Merk at ein **LUT-modul** ikkje er noko anna enn ein rein **SRAM**-modul, dvs. ein minnemodul. Meir om ulike minnetypar kjem i kapittel 3.10.9.

Figur 3.38 viser grovstrukturen til ein tradisjonell FPGA-krets. Mange FPGA-kretsar av i dag kan i tillegg til CLB-ane og inn/ut-blokkene innehalda følgjande:

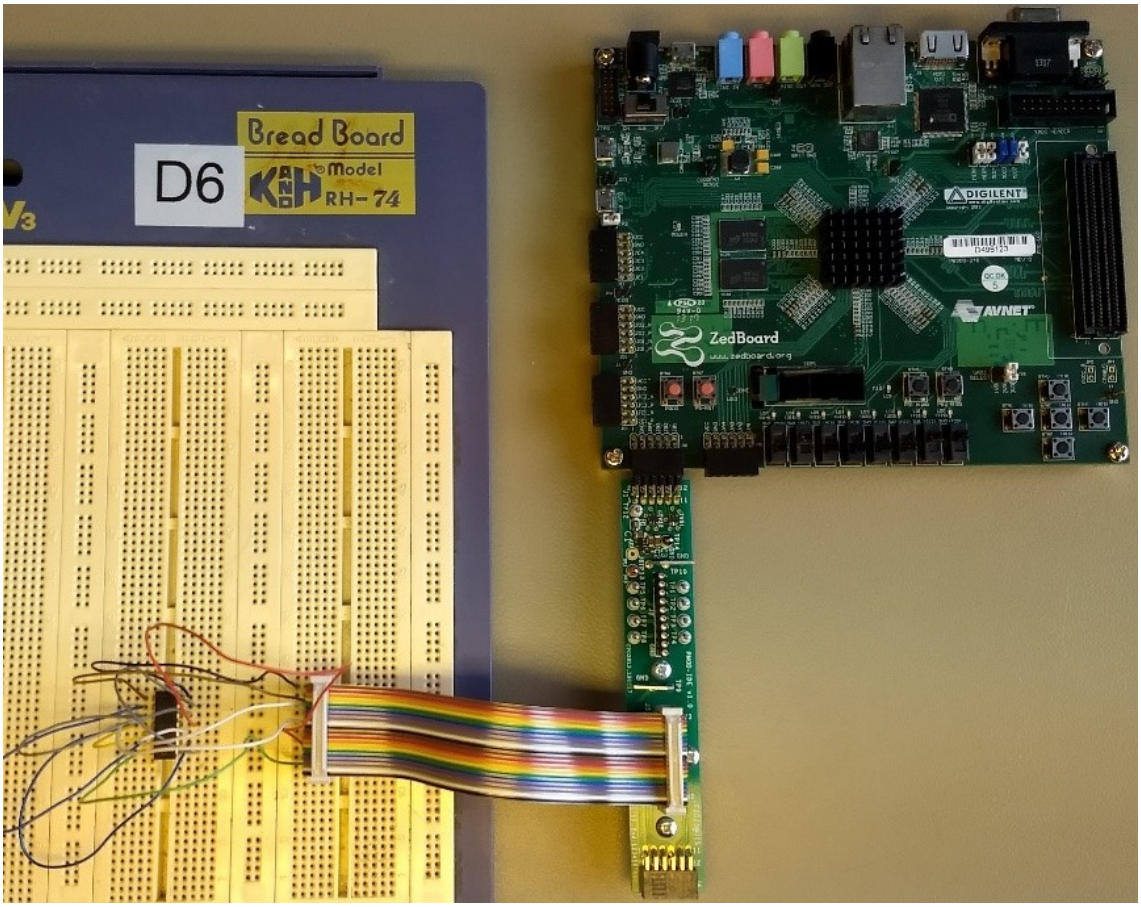
- *B(lock)*RAM-modular, som kan brukast til å realisera data- og programminne for ein mjuk μP.
- Multiplikasjonsmodular.
- Raske kommunikasjonsmodular, dvs. med bitratar $>$ 1 Gbit/sek.

[70] I kapittel 3.5.2 såg ein mellom anna på oppbygginga av ulike typar buffer.
[71] Meir om mikrokontrollerar i FPGA kjem i slutten av dette delkapitlet.

- Faste mikroprosessorar.

Xilinx sin nyaste familie av programmerbare kretsar, Zynq-familien, inneheld ARM-kjernar som eigne modular i kretsane. Eit plattformkort basert på nettopp Zynq-familien blir brukt i undervisninga i Datamaskinarkitektur i tillegg til plattformkortet *STM32VLDiscovery*[72].

Det Zynq-baserte plattformkortet frå produsenten Digilent er vist i figur 3.40.



Figur 3.40: FPGA-kortet *ZedBoard Zynq-7000 Development Board* i bruk på laboratoriet. (Foto: J. Fidjeland, UiS.)

Figuren viser eit typisk laboratorieoppsett i Datamaskinarkitektur der *ZedBoard* er kobla mot eit koblebrett via eit robust grensesnittkort, *UiS_PMOD_Ide*[73].

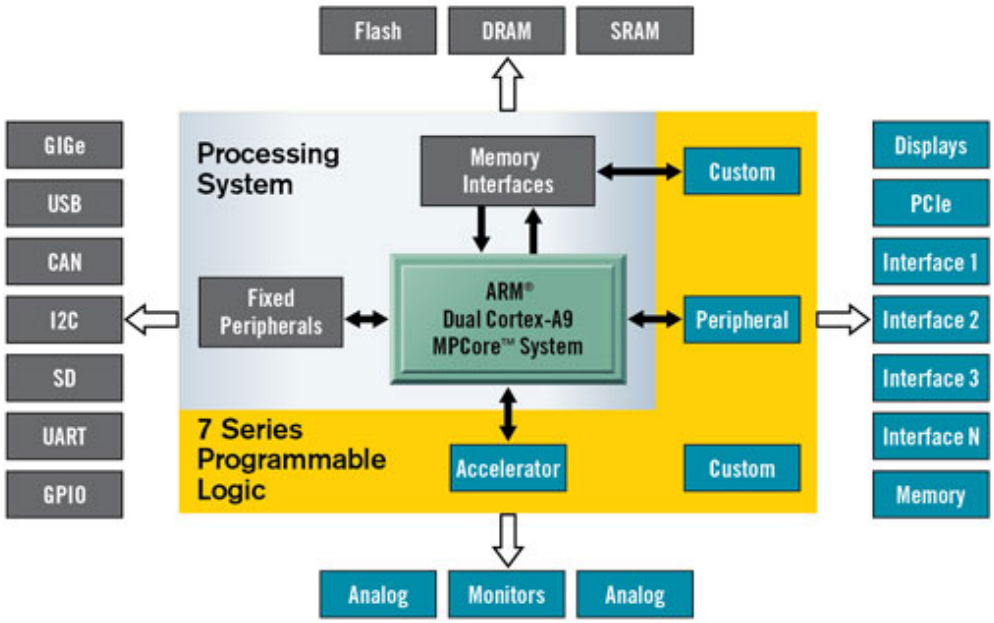
Litt meir om plattformkortet *ZedBoard Zynq-7000 Development Board*:

- FPGA med rundt 1.3 millionar portekvivalentar.
- FPGA-en inneheld om lag 53000 LUT-ar med typisk 6 inngangar og 1 utgang.
- Dobbelkjerne (mikroprosessor) av typen ARM Cortex-A9.

[72] Sjå figur 1.9 på side 12.
[73] Konstruert av J. Fidjeland, UiS.

- Opptil 667 MHz klokkefrekvens.
- 512 MByte *D(ouble)D(ata)R(ate)3* DRAM-minne.
- 256 Mbit Flash-minne med *Quad S(erial)P(eripheral)I(nterface)*-grensesnitt.
- Grensesnitt for 4 GB *S(ecure)D(igital)*-kort (Flash-minnekort).
- 10/100/1000 Mbit/sekund Ethernet-samband.
- USB.
- HDMI og VGA.
- P-modulkontaktar.
- Lysdiodar og brytarar.
- Mikrofon- og øyretelefonkontaktar.

Den programmerbare kretsen blir her kalla eit **SoC** ("System-on-Chip"), då han i tillegg til det reine FPGA-arealet har prosessorar og perifermodular. Dette er vist blokkskjematisk i figur 3.41.



Figur 3.41: Blokkskjema av ein FPGA-krets frå Xilinx-familien Zynq-7000.
(Ref.: "Wiki Xilinx_Zynq-7000_AP_SoC.jpg".)

Blokkene i det gule området er sjølve portmatrisa. Viss ein lagar generell logikk og lastar dette ned, vil det hamna i den nedre "Custom"-blokka. Den øvre av desse blokkene er tiltenkt logikk som skal kommunisera med prosessorkjernane via eit minnegrensesnitt. Både prosessorane og det gule området inneheld SRAM-modular, men med det nemnde minnegrensesnittet mot omverda kan ein utvida minnekapasiteten kraftig.

Med som i dette tilfellet, ein eller fleire faste prosessorar, kan ein realisera ein datamaskin inne i FPGA-kretsen. Denne datamaskinen vil vera ein mikrokontroller sidan han er på ei brikke.

Den einaste modulen som ein vil mangla samanlikna med ein "vanleg" mikrokontroller, er **permanentminne**, som f.eks. Flash. Slike minne held som kjent på innhaldet sjølv om ein slår av kraftforsyninga.
Slike minne har ein vanlegvis ikkje til disposisjon i FPGA. Dei aller fleste FPGA-familiane er som nemnt baserte på SRAM-teknologi og er difor tome etter straumpåslag. Konfigurasjonsdata og μP-program må då lastast inn på nytt. FPGA-en er difor kobla til ein ekstern Flash-krets som innheld dette. ZedBoard har som vist eit slikt Flash-minne på 256 Mbit (= 32 MB).
Det er nokre unntak til dette, mellom anna følgjande:

- Ein SoC-krets med ei blokk frå Xilinx-familien Spartan-3AN og eit Flash-minne for konfigurasjon.
- Små, men Flash-baserte FPGA-brikker i IGLOO-serien frå produsenten Microsemi.

Som eit alternativ til fast prosessorkjerne i FPGA eller ein mikroprosessorkrets i tillegg til ein FPGA, kan ein bruka ein **mjukprosessor**, mjukP, i FPGA-en. MjukP-en med minne og andre nødvendige modular vil då utgjera ein **mjuk mikrokontroller**, mjukC, i FPGA-en. Heile mjukC-en er då spesifisert i eit eige språk. Ein kallar ofte desse mjuke modulane for **IP-modular**[74].

Xilinx har som nemnt før, eit eige verktøy for utvikling av mjukC-baserte system i FPGA, Embedded Development Kit (EDK). I EDK spesifiserer ein først modulane ein vil ha med i den mjuke mikrokontrolleren. Så kompilerer ein dette og lastar den resulterande maskinvarekonfigurasjonen ned til FPGA-en saman med programkoden for mjukC-en.

Meir om programmering av kretsar kjem i neste kapittel.
Når det gjeld mjukprosessorar og programmering av denne, kjem det meir om dette i kapittel 3.7.4.

3.7.2 Programmering av logikk

3.7.2.1 Spesifikasjon vhja. programmeringsspråket VHDL

Som oppsummert i kapittel 3.6.6, kan ein logisk funksjon **spesifiserast** vhja. **funksjonstabell**, **logiske likningar** eller **logisk skjema**.

[74] *I(ntellectual)P(roperty)* tilkjennegir at dette ikkje er eigne harde modular i form av IC-brikker, men spesifikasjonar eller oppskrifter på korleis modulane skal realiserast i programmerbar logikk. Ein IP-modul er altså ein ferdiglaga digital modul for gjenbruk i for eksempel ein FPGA. "Intellectual Property" betyr at nokon har eigedomsretten til modulen. Ein IP-modul kan vera ein prosessormodul, ein serie- eller parallellport, ein taimer eller ein grensesnittmodul mot eit eksternt minne.

Når ein skal programmera logikk i ein programmerbar krets, dvs. **realisera** ein logisk funksjon, bruker ein vanlegvis eit passande **logikkprogrammeringsspråk** ("Hardware Description Language", HDL) for å spesifisera denne.
Det vanlegaste er å bruka dei logiske likningane ved programmering, men ein kan også leggja inn funksjonstabellar.

Logikk av lite omfang kan alternativt spesifiserast vhja. eit logisk skjema. Fleire programmeringsverktøy tilbyr dette som eit alternativ i tillegg til HDL.

Eit vanleg språk for maskinvareprogrammering er **VHDL**[75]. Andre aktuelle språk er mellom anna Verilog og ABEL.

Eit eksempel på oppsett av ein enkel funksjon i VHDL er gitt under.

Eksempel 3.23. *NOG-port spesifisert i VHDL*

```
-- Først deklarasjon av bibliotek
library IEEE;
use     IEEE.STD_LOGIC_1164.ALL;

-- Portdeklarasjon
entity NOG is

   -- Oppsett av inngangar og utgang
   port (A, B: in  std_logic;
         Y   : out std_logic);

end NOG;

-- Deklarasjon av funksjon/oppførsel til porten
architecture behavioral of NOG is
begin

   Y <= not (A and B);

end behavioral;
```

Som ein ser i figuren, set ein først opp **inn- og utgangar** og spesifiserer så **arkitekturen**, dvs. dei logiske funksjonane. Dei logiske operatorane heiter følgjande i VHDL:

not, and, nand, or, nor, xor, xnor

[75] "VHSIC Hardware Description Language", VHDL, der VHSIC står for "Very High Speed Integrated Circuit". Ein kan lesa meir om dette språket bl.a. hjå "en.wiki VHDL".

Viss ein treng interne hjelpevariablar, her kalla **signal**, kan ein gjera dette som vist i neste eksempel.

Eksempel 3.24. *VHDL-kode for NOG-port med hjelpevariablar*

```
-- Først deklarasjon av bibliotek
library IEEE;
use     IEEE.STD_LOGIC_1164.ALL;

-- Portdeklarasjon
entity NOG is

   -- Oppsett av inngangar og utgang
   port (A, B: in  std_logic;
         Y   : out std_logic);

end NOG;

-- Deklarasjon av funksjon/oppførsel til porten
architecture behavioral of NOG is

   -- Deklarasjon av hjelpevariablar
   signal X: std_logic;

begin
   X <= (A and B);
   Y <= not X;

end behavioral;
```

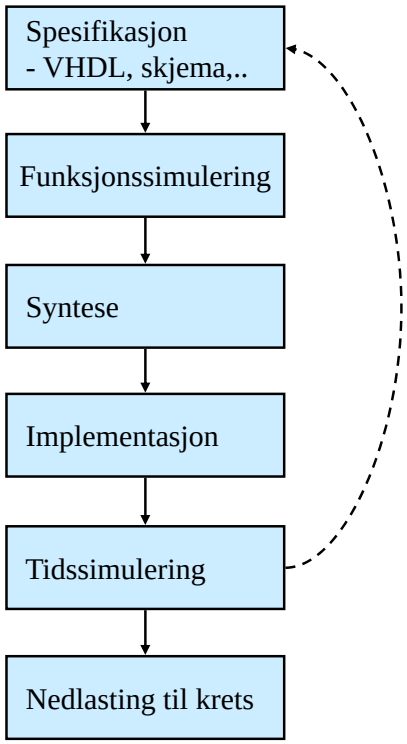
3.7.2.2 Litt om spesifikasjon vhja. verktøy av høgare generasjon

Ved programmering av logikk av stort omfang, bruker ein ofte verktøy av såkalla **høgare generasjon**.
Xilinx har som nemnt i slutten av kapittel 3.7.1.3, eit eige verktøy for utvikling av mjukC-baserte system i FPGA, Embedded Development Kit (EDK). Når ein i EDK set opp eit slikt system, vil verktøyet sjølv generera VHDL-kode som i sin tur kan kompilerast og lastast ned til FPGA-en saman med programkoden for mjukC-en.
Meir om dette verktøyet kjem i kapittel 3.7.4.

Viss ein skal realisera eit logisk system **utan** ein mikroprosessor, vil det vera verktøy av høgare generasjon for dette også. Eit slikt verktøy frå Xilinx er som nemnt i kapittel 3.6.7, **System Generator**.
Meir om dette verktøyet kjem i kapittel 3.7.2.4.

3.7.2.3 Om generering og nedlasting

Vanlege trinn i ein byggje- og nedlastingsprosess er viste i figur 3.42.



Figur 3.42: Omsetting av programkode for logikk.

I dei ulike trinna skjer følgjande:

- **Spesifikasjon**
  Her set ein altså opp korleis ein ønskjer at logikken skal virka, sjå VHDL-eksempla over.
  Viss ein bruker ein verktøy av høgare generasjon, vil ein spesifisera kva logikkmodular som skal vera med og koblinga mellom desse. Verktøyet vil då omforma dette til VHDL-kode før det går vidare i byggjeprosessen.

- **Funksjonssimulering**
  Her kan ein få simulert den logikken ein har spesifisert for å sjå om den virkar som tenkt.

- **Syntese**
  Her blir logikken først optimalisert[76] der ein fjernar overflødig logikk og bruker dei mest effektive portkombinasjonane. Til slutt blir det produsert ei nettliste, der ein

[76] Dette står det litt meir om i kapittel 3.6.7.

nummererer alle portar, inn- og utgangar og sambandslinjer, dvs. nett, og spesifiserer tilknytningspunkta for dei ulike netta.

- **Implementasjon**
  Her blir dei ulike delene av nettlista tilordna fysiske inn-og utgangar, logiske blokker og nett i den programmerbare kretsen som skal brukast.

- **Tidssimulering**
  Denne simuleringa blir køyrt for å sjå om signala går som venta gjennom logikken slik ein vil implementera denne. Viss det her blir avdekka tidsproblem pga. etterslepa ("propagation delay") gjennom portane, må ein endra spesifikasjonen.

- **Nedlasting**
  Implementasjonstrinnet gir som resultat ei såkalla **bitfil**. Denne blir overført som ein **bitstraum** til kretsen via eit standard grensesnitt.
  Bitar i denne straumen vil bli ruta til oppslagstabellar, sjå figur 3.39, slik at dei logiske modulane i FPGA-en samla sett realiserer den maskinvaren som er spesifisert. Viss det er ein mjuk mikrokontroller som skal realiserast, vil ein del av bitfila vera programkode. Denne vil bli lagt i ei minneblokk, vanlegvis eit SRAM-minne.

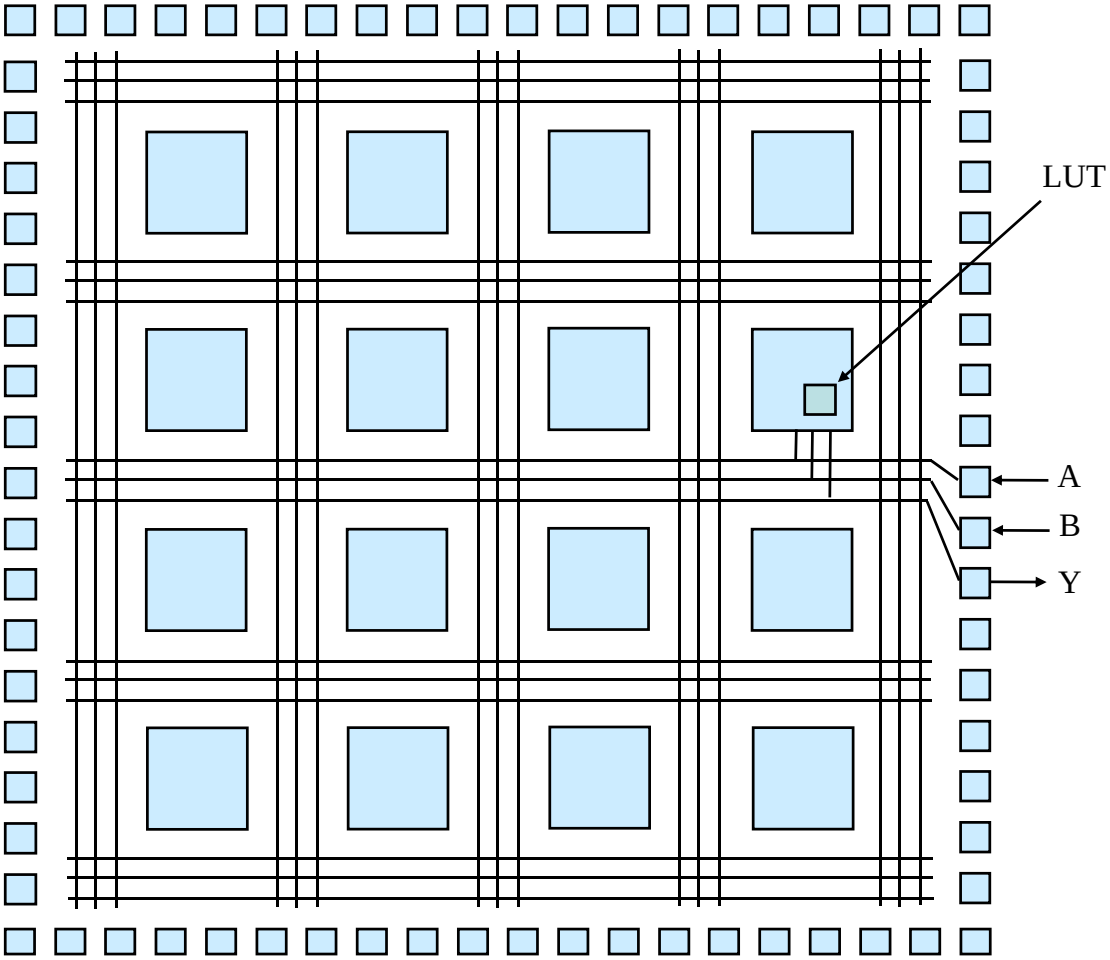
Eksempel 3.25. *Bygging og nedlasting av XELLER-logikk.*

I eksempel 3.20 på side 157 blei ein XELLER-funksjon for to inngangssignal A og B realisert i programmerbar logikk av typen PAL.
Nå skal denne funksjonen realiserast i FPGA. Ein kan då ta utgangspunkt i VHDL-kode. Denne kan lagast på tilsvarande måte som i eksempel 3.23.
Kompilering av VHDL-koden vil då gi ein bitstraum som i tillegg til annan nødvendig informasjon innheld bitmønsteret

..0110[77]..

Dette bitmønsteret vil ved nedlasting bli plassert i minnecellene til den LUT-en som skal brukast her. LUT-en blei vist i eksempel 3.22 på side 161.
FPGA-kretsen vil etter nedlasting oppføra seg som ein XELLER-port med pinnane viste i figur 3.43.

[77] Sjå figur 3.39 på side 161.



Figur 3.43: FPGA brukt til realisering av ein XELLER-port.

3.7.2.4 Litt om utviklingsverktøyet System Generator frå Xilinx

System Generator er som nemnt før, eit utviklingsverktøy for digital elektronikk som er basert på *Matlab* og *Simulink*.
Verktøyet er som nemnt i kapittel 3.6.7 eigentleg laga for realisering av **signalbehandlingsmetodar** i FPGA, men er brukt i emnet Datamaskinarkitektur også for å illustrera realisering av logikkfunksjonar i FPGA.

Ein teiknar logikkskjema vhja. biblioteksmodular i *Simulink* eller skriv kode (VHDL) som ein så byggjer vhja. *System Generator* og lastar ned i ei programmerbar brikke (FPGA). Ein vil her kalla verktøyet *Simulink med System Generator* og bruka forkortinga *SSG*.

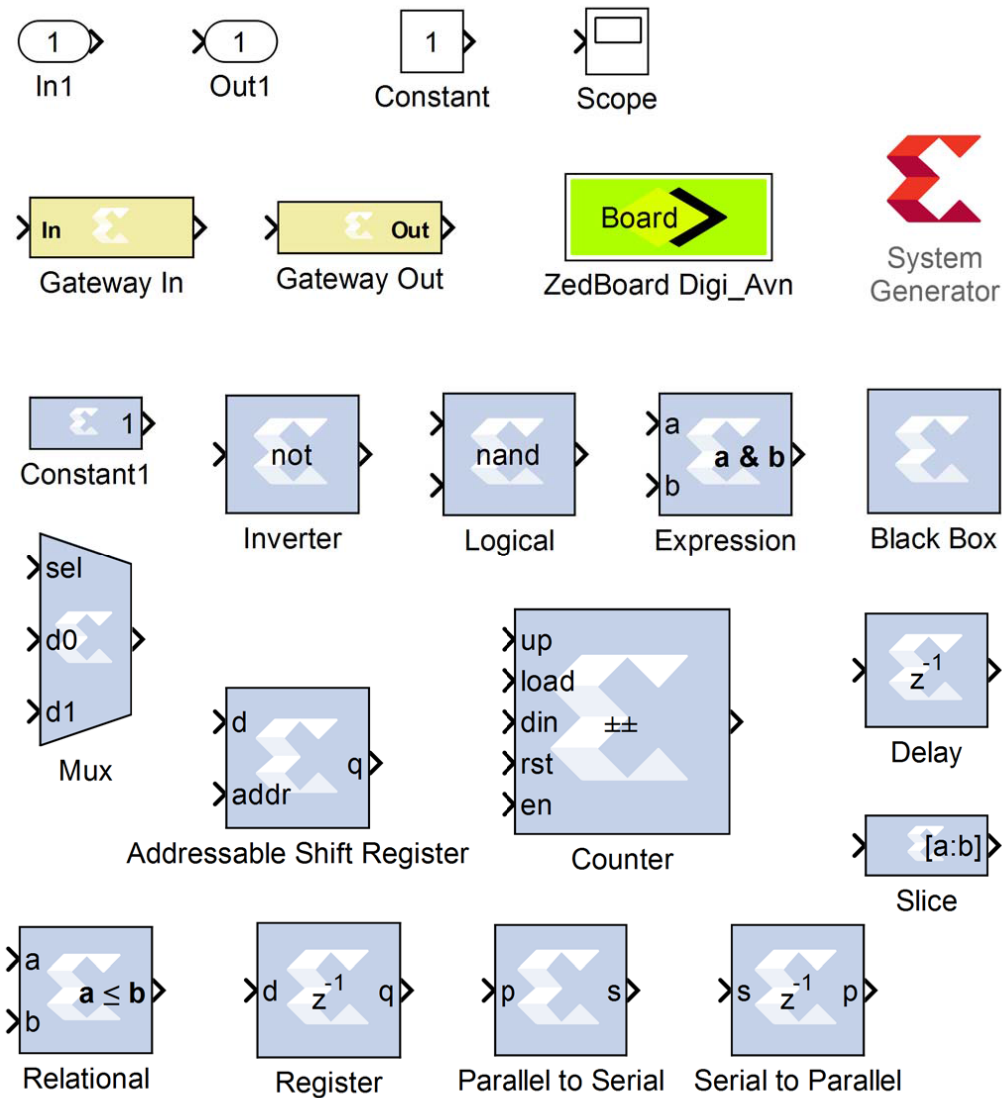
Plattformkortet som blir brukt i digitalteknikkdelen av emnet Datamaskinarkitektur, er altså *ZedBoard*, sjå figur 3.40 på side 163.
Ein skal seinare i kapitlet visa korleis ein kan realisera og testa bildørlogikken vhja. av dette kortet, men først skal ein sjå litt på kva **biblioteksmodular** som er tilgjengelege.
Eit lite utdrag frå biblioteka til SSG er vist i figur 3.44.
I figuren er det vist nokre vanlege blokker frå tre ulike bibliotek:

- *Simulink Library*: Dei kvite blokkene gir det ytre rammeverket for Simulink-modellen. Her markerer ein kva som er inngangar og utgangar. Innanfor inn- og utblokkene spesifiserer ein sjølve logikken, plattformen og verktøyet vhja. blokker frå dei to andre biblioteka, sjå dei to neste kulepunkta.
  Viss ein ønskjer å **simulera** modellen før testing på plattformkortet, kan ein i staden for innblokkene leggja inn **stimuliblokker** samt ei **skopblokk**.
  Ei stimuliblokk kan som vist gi ut ein **konstant** verdi, men kan også setjast opp til å gi ut ein tidsserie frå ei Matlab-fil.
  Vhja. skopblokka kan ein sjå korleis utgangane reagerer på stimulia.
- *UiSBlockset*: I dette biblioteket finn ein **plattformblokka** *ZedBoard Digi_Avn*. I denne ligg informasjon om plattformkortet inkludert kva portar og andre modular dei ulike FPGA-pinnane er kobla til.
- *Xilinx Blockset*: **Verktøyblokka** *System Generator* høyrer til dette biblioteket. Vhja. denne blokka spesifiserer ein mellom anna kva FPGA-krets plattformen inneheld og kva filer som skal lagast.
  Vhja. denne blokka startar ein også **bygging** av logikken ein har teikna.

  I dette biblioteket finn ein også **portblokkene** *Gateway In* og *Gateway Out* samt ei rekkje **logikkblokker** av ulike slag.
  ZedBoard inneheld som ein ser i figur 3.40, mellom anna brytarar, trykknappar og I/O-portar. Vhja. portblokkene definerer ein kva ein vil kobla seg til.
  Vhja. logikkblokkene realiserer ein den funksjonaliteten systemet skal ha.

  Ein kan her merka seg blokka *Black Box*. Denne kan spesifiserast vhja. VHDL-kode.

  Når ein køyrer bygging og så nedlasting, er det port- og logikkblokkene og koblingane mellom desse som blir realiserte i FPGA-en.



Figur 3.44: Litt frå biblioteka til SSG.

Skrivet "Xilinx System Generator for ZedBoard" av Jon Fidjeland, [11], viser detaljert framgangsmåte for å setja opp Simulink og System Generator.

Til slutt i dette kapitlet skal ein visa korleis dørlyslogikken frå figur 3.20 på side 133 kan realiserast i FPGA vhja. verktøyet SSG.

Eksempel 3.26. *Realisering av dørlyslogikken i FPGA.*

I eksempel 3.18 på side 151 kom ein fram til følgjande logiske likning for dørlyslogikken:

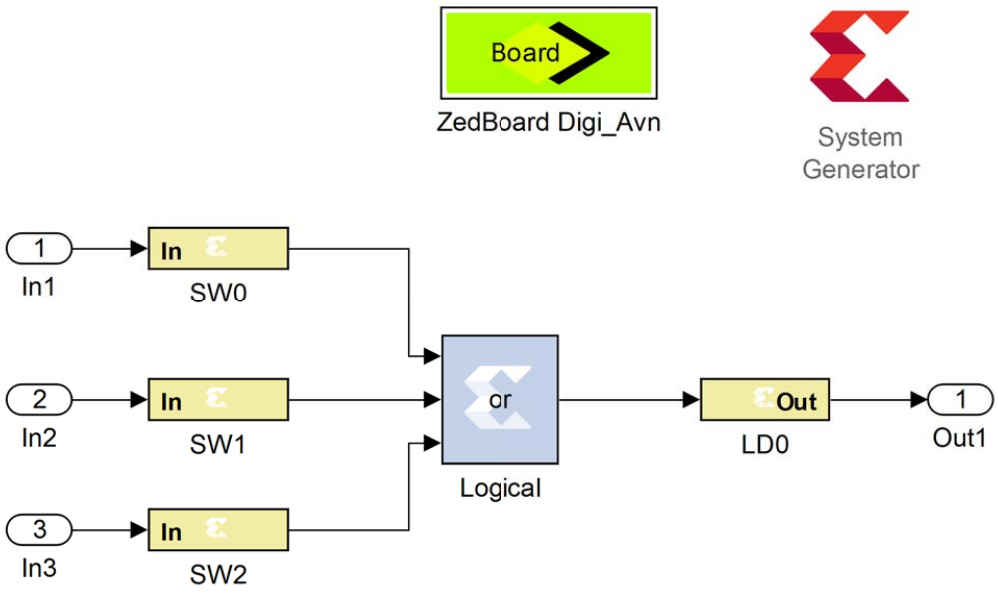
$$DL = D1 + D2 + D3$$

Ein ønskjer å realisera denne i FPGA-en på plattformkortet ZedBoard.
I første omgang vil ein bare testa logikken vhja. tre brytarar og ein lysdiode på kortet.

Spesifikasjon av tilkoblingane på plattformkortet er vist i tabellen under.

Signal	Skal koblast til
$D1$	Brytaren $SW0$
$D2$	$SW1$
$D3$	$SW2$
DL	Lysdioden $LD0$

Ein SSG-modell for dørlyslogikken med desse tilkoblingane er vist i figur 3.45.



Figur 3.45: SSG-modell for testing av dørlyslogikken.

Legg merke til i dette eksemplet at logikkblokkene er fleksible. I blokka *Logical* kan ein spesifisera både logisk funksjon og talet på inngangar mm.

3.7.3 Fordelar og ulemper med programmerbar logikk

Bruk av programmerbare kretsar gir mange fordelar, så som:

- Fleksibel maskinvare, dvs. same plattform for mange produktvariantar og dermed lågare systemkostnad. Dette gir kort produktutviklingstid, som blir stadig viktigare. Fleire firma har difor gått over til bruk av FPGA i kombinasjon med faste eller mjuke mikroprosessorar.
- Kan realisera tung/tidskritisk programvare som eigen logikkmodul (IP-modul) og integrera med mjuk eller hard mikroprosessor i FPGA.
- Ingen store sprang i utviklingsverktøy og maskinvare, slik som ved hopp frå ein fast prosessortype til ein annan.
  Prosessorar og perifermodular frå ein og same produsent endrast lite over tid. Det er i hovudsak bare sjølve kretskapasiteten og
  klokkefrekvensen som varierer frå system til system.
- Oppdateringar av både program- og maskinvare kan overførast til sluttbrukar og utførast der.

Pga. fleksibiliteten har kretsane relativt høgt effektforbruk. Dette er ein hovudulempe ved FPGA. Bruk av FPGA i mobilt utstyr er derfor utfordrande, men det finst lågeffekts-utgåver av nokre mindre FPGA-kretsar.
Konstruksjonen kan alternativt overførast til ASIC, men dette gir lenger utviklingstid og krev omtanke.

3.7.4 Mjukprosessor

Den første mjuke prosessoren ("soft processor") kom rundt år 2000.
I dag fins det ei rekkje slike, der nokre av dei første og mest kjende[78] på marknaden er følgjande:

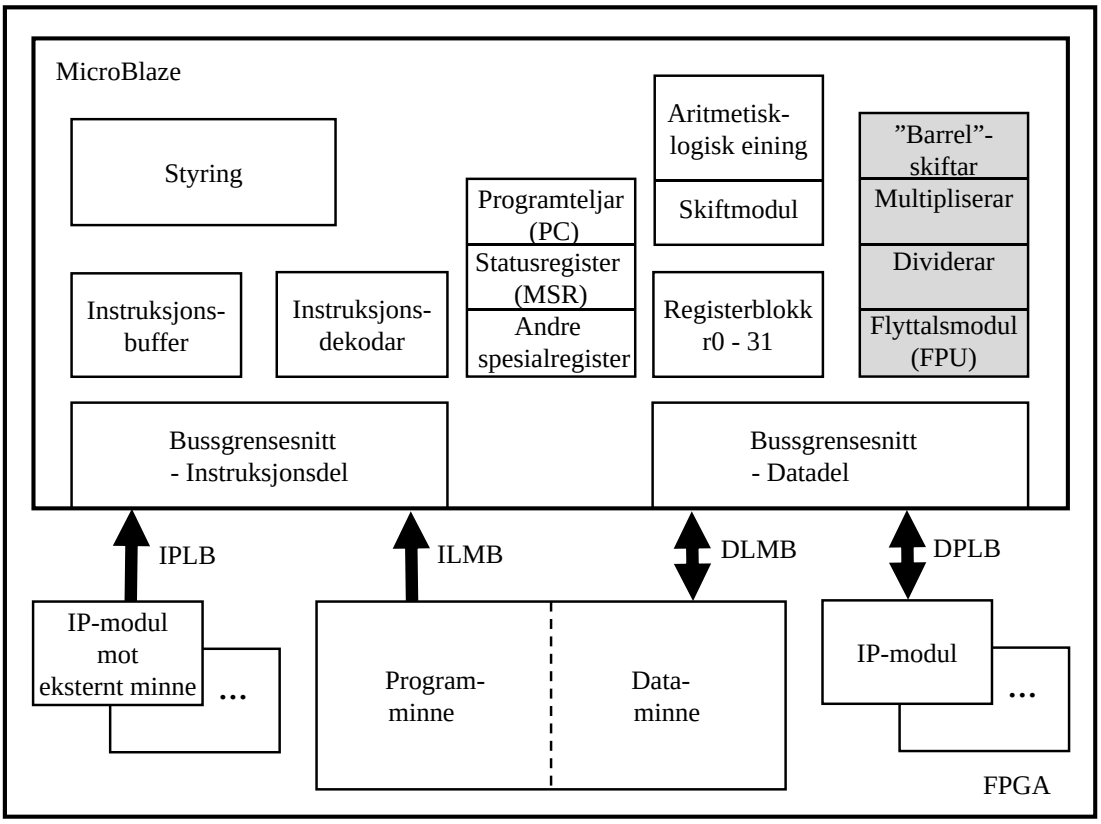
- Altera Corp.: Nios II (32bit).
- Lattice Semiconductor Corp.: Mico32 (32bit), Mico8 (8bit).
- Xilinx Inc.: MicroBlaze (32bit), PicoBlaze (8bit).

Ein vil her sjå vidare på MicroBlaze, som er blitt brukt ved UiS i ei årrekkje.

[78] Sjå og "en.wiki Soft microprocessor".

3.7.4.1 Litt om MicroBlaze sin struktur

MjukP-en MicroBlaze har som nemnt i kapittel 2.1, såkalla Harvard-arkitektur. Eit forenkla blokkskjema av prosessorstrukturen er vist i figur 3.46.



Figur 3.46: Forenkla MicroBlaze-struktur.

MicroBlaze har tre ulike bussystem:

- **Local Memory Bus, LMB**
  Denne knyttar prosessoren saman med interne program- og dataminne, dvs. minne som også ligg i FPGA-en, og bussen er altså delt i ein instruksjons- og ein datadel, ILMB og DLMB. Dette gjer som nemnt før, at ein unngår den såkalla Von Neumann-flaskehalsen, dvs. at eit felles bussystem blir ein flaskehals i prosesseringa.
  Instruksjonsdelen og datadelen av bussystemet er igjen delt opp i ein adresse-, data- og styrebuss som vist i kapittel 2.

- **Processor Local Bus, PLB**
  Denne knyttar prosessoren saman med modular som igjen kan koblast til eksterne komponentar, dvs. på utsida av FPGA-en.
  Slike modular er serie- og parallellportar, taimermodular, grensesnittmodular mot eksternt minne mm.

Denne bussen er også delt i ein instruksjon- og ein datadel. Kvar av desse er igjen delt opp i ein adresse-, data- og styrebuss.
PLB overtok som periferbuss f.o.m. med EDK-versjon 10.1 og erstatta då periferbussen OPB (On-Chip Peripheral Bus). Det blei då samstundes tilgjengeleg PLB-utgåver av dei IP-modulane ein hadde til OPB tidlegare.

- **Fast Simplex Link, FSL**
  Denne er for spesiell bruk. Viss ein f.eks. vil realisera tyngre algoritmer som eigne maskinvaremodular for å avlasta prosessoren, vil det vera gunstig å knyta desse til prosessoren via FSL. Dette blir ikkje omtala nærare her, men det visest til referansemanualen, [29].

MicroBlaze har ei registerblokk med 32 register, *r0-31*. Kvart register kan lagra ein verdi på 32 bit. I tabell 3.2 i referansemanualen, [29], er det vist ein såkalla konvensjon for registerbruk. Dei som utviklar kompilatorar for ulike prosessorar, følgjer slike konvensjonar. Nokre av registra er altså tenkt brukt til bestemte ting frå μP-produsenten si side, mens dei fleste andre er til generell bruk.
Litt frå konvensjonen for MicroBlaze-registra:
- *r0* er alltid null.
- *r1* er stakkpeikar.
- *r5-10* blir brukt til variabelverdiar overført i eit metodekall[79].
- *r15* er lenkeregister, dvs. inneheld returadressa frå ein metode.

I tillegg har MicroBlaze mange spesialregister. Me skal her nøya oss med to sentrale eksempel:
- *PC* er som kjent programteljaren, Program Counter, og inneheld adressa til den instruksjonen som blir køyrt.
- *MSR* er statusregisteret, Machine Status Register.

Merk at nokre av blokkene i MicroBlaze er valfrie, sjå figur 3.46. Dei kvite delane utgjer minimumsutgåva av mikroprosessoren, mens dei grå må spesifiserast ut frå behov. Desse, som f.eks. ein multiplikator i maskinvare, vil gi meir effektiv prosessering, men vil krevja ekstra plass i FPGA-en. Bare ein del av alle dei valfrie modulane i prosessoren er viste i figuren. Mellom anna kan prosessoren setjast opp med hurtigminne ("cache").
Når det gjeld blokkene ellers i figur 3.46, vil funksjonane til desse vera rimeleg klare ut frå det som er skrive om mikroprosessorar generelt i kapittel 2.

3.7.4.2 Litt om MicroBlaze sitt instruksjonssett

Instruksjonssettet til MicroBlaze er vist i tabell 3.7. Ved å spesifisera bruk av tilleggsmodular som f.eks. dei som er gråfarga i figur 3.46, kan ein få tilgang til delar av dei skråstilte instruksjonane i tabellen.

[79] Metode er som nemnt før, ofte brukt som namn i objektorienterte språk, mens funksjon og subrutine er vanlege ord i språket C.

Eit eksempel er såkalla ”barrel shift”-instruksjonar. Spesifisering av dette kan gi meir effektiv kode, men vil som nemnt før, gi ein meir plasskrevjande mikroprosessor i FPGA-en.

Aritmetikk		**Logikk**		**Program-styring**		**Flytting**	
Addisj./ subtraksj.	**Multipl./ divisj. mm**	**Logikk**	**Saman-likning**	**Hopp u/ vilkår**	**Hopp m/ vilkår**	**Last/ lagre**	**Spesial-instr.**
add	*mul*	and	cmp	br	beq	lbu	imm
addc	*muli*	andi	cmpu	bra	beqd	lbui	mfs
addk	*mulh*	andn	*fcmp*	brd	beqi	lhu	mts
addkc	*mulhu*	andni	*pcmpbf*	brad	beqid	lhui	*msrclr*
addi	*mulhsu*	or	*pcmpeq*	brld	bge	lw	*msrset*
addic	*idiv*	ori	*pcmpne*	brald	bged	lwi	*get*
addik	*idivu*	xor		bri	bgei	sb	*getd*
addikc	*fmul*	xori		brai	bgeid	sbi	*put*
rsub	*fdiv*	sra		brid	bgt	sh	*putd*
rsubc	*fsqrt*	src		braid	bgtd	shi	*wdc*
rsubk		srl		brlid	bgti	sw	*wic*
rsubkc		sext8		bralid	bgtid	swi	
rsubi		sext16		brk	ble		
rsubic		*bsll*		brki	bled		
rsubik		*bslli*		rtbd	blei		
rsubikc		*bsra*		rted	bleid		
fadd		*bsrai*		rtid	blt		
frsub		*bsrl*		rtsd	bltd		
		bsrli			blti		
		fint			bltid		
		flt			bne		
					bned		
					bnei		
					bneid		

Tabell 3.7: Nokre sentrale MicroBlaze-instruksjonar.

To spesielle detaljar ved instruksjonssettet skal framhevast her, nemleg **indeksen ”d”** og **indeksen ”l”** som blir brukt i nokre av **hoppinstruksjonane**.

Indeksen ”d” står for ”delay”, og ein hoppinstruksjon med d-indeks gir ikkje hopp før instruksjonen etterpå også er utført. Ein får altså eit forseinka hopp.
Dette er innført for å kompensera for litt av den daudtida som oppstår i samband med hopp pga. parallelkøyringa (”pipelining”) til prosessoren.
Når hoppinstruksjonen er kome til utføringsfasen, vil dei to neste instruksjonane også liggja i røyret, og dette blir utnytta av kompilatoren som vist i det enkle eksemplet under.

Eksempel 3.27.

C-koden

```
.....
a++; //lokal variabel f.eks. liggjande i register r3.
rutine_A();
.....
```

kan bli til følgjande assemblykode:

```
.....
brlid   r15,rutine_A
addi    r3,r3,1
.....
```

I koden over vil altså "d"-en i hoppinstruksjonen *brlid* fortelja MicroBlaze at *addi*-instruksjonen skal utførast før han hoppar vidare i koden.

Indeksen "l" står for "link". Ein hoppinstruksjon med l-indeks som f.eks. *brlid* i eksempel 3.27, vil leggja ein kopi av programteljaren sin verdi, dvs. adressa til hoppinstruksjonen, i prosessorregister *r15*. Slik kan prosessoren returnera og halda fram i hovedprogrammet etter å ha køyrt ei subrutine. Registeret *r15* er då eit lenkeregister.

3.7.4.3 Utvikling av MicroBlaze-basert mikrokontroller i FPGA

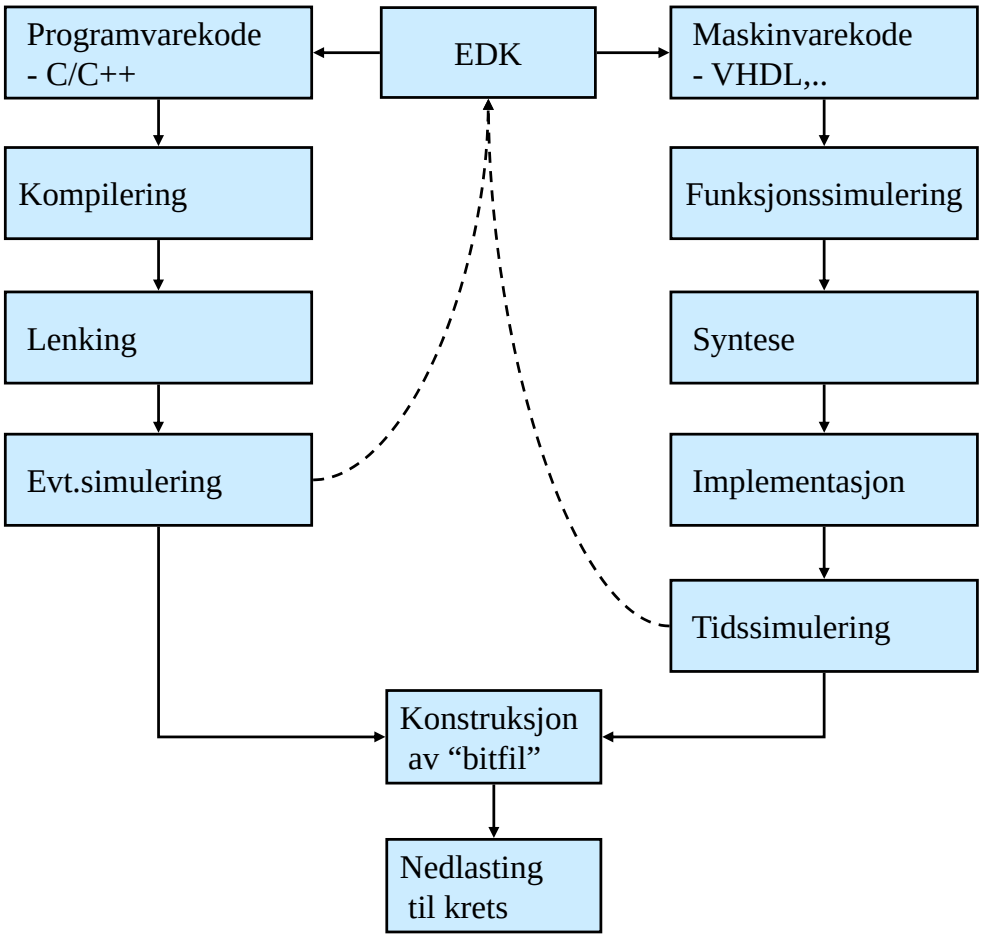
Ved å byggja MicroBlaze saman med eit passande utval av andre modular som minne, inn/ut-portar med meir, kan ein altså realisera ein mjuk mikrokontroller i FPGA.
Når det gjeld krav til FPGA-en her, tilseier erfaring at kretsen bør ha **minimum 200.000 portar** for at ein skal ha nok plass til mikrokontrolleren.
Ein mikrokontroller med to einvegs parallellportar på 8bit, ein serieport, til saman 8kByte program- og dataminne samt eit minimum av modular for bitfiloverføring og programavlusing, vil ta i bruk litt under 150.000 portar.

I utviklingsverktøyet **Embedded Development Kit** (EDK) frå Xilinx kan ein både setja opp maskinvaren i ein MicroBlaze-basert mikrokontroller og utvikla programkode for denne.

Ein av pakkane i EDK er **Base System Builder** (BSB), der ein gjer sjølve maskinvarespesifikasjonen.
Alle maskinvareendringar **etter** bygging av mikrokontrolleren i BSB kan gjerast i pakken **Xilinx Platform Studio** (XPS), som også er ein del av EDK.
Eit eksempel på oppsetting av ein enkel mikrokontroller vhja. BSB er vist i

vedlegg A i [1]. I vedlegg B er det vist eit eksempel på utviding av ein mikrokontroller vhja. XPS.

Sjølve programutviklinga for mikrokontrolleren kan ein gjera vhja. den ”eclipse”-baserte EDK-pakken **Software Development Kit (SDK)**.
Maskinvare- og programvarekode blir kompilert kvar for seg i fleire trinn før nedlasting som vist i figur 3.47.
Maskinvarekoden eller -spesifikasjonen blir her først omsett til VHDL-kode. Dei ulike trinna i kompileringa av denne igjen er som omtalt i kapittel 3.7.2.



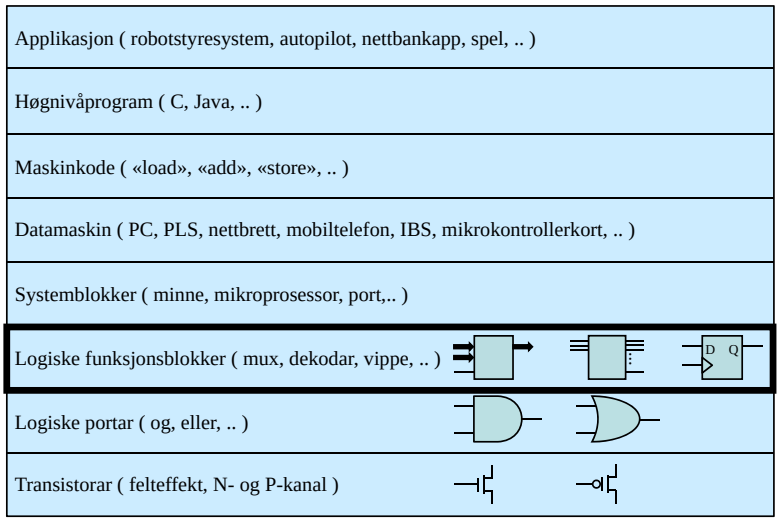
Figur 3.47: Omsetting av kode generert i verktøyet Embedded Development Kit

Omsett maskinvare- og programkode blir kombinert i ei bitfil som kan overførast til FPGA-en.
Som regel vil det på FPGA-kort som nemnt i kapittel 3.7.1.3 vera eksterne Flash-minne som kan programmerast permanent med den resulterande bitfila. Ein kan då få til automatisk overføring til FPGA-en etter at kraftforsyninga til kortet blir slått på.

3.8 Kombinatoriske funksjonsblokker

3.8.1 Litt om funksjonsblokklaget i det digitale hierarkiet

Me vender nå tilbake til det digitale hierarkiet. Neste lag i hierarkiet er logiske funksjonsblokker, sjå figur 3.48.



Figur 3.48: Digitalt hierarki: Funksjonsblokklaget.

Funksjonsblokkene kan byggjast opp vhja. dei logiske portane me såg på i kapittel 3.5. Funksjonsblokkene kan delast inn i følgjande to hovudgrupper:

- Kombinatoriske byggjeblokker (KB).
- Sekvensielle byggjeblokker (SB).

Forskjellane på kombinatorisk og sekvensiell logikk er forklart i kapittel 3.2. Ein hovudforskjell er som vist der at sekvensiell logikk har eit minne. Ei slik blokk kan altså lagra eit bitmønster, og dette mønsteret kallar me ofte for tilstanden til blokka.

Eitt av måla med kapittel 3, dvs. digitalteknikkdelen, at ein skal forstå grunnleggjande oppbygging og virkemåte for ein mikroprosessor. Basert på figur 2.15 på side 73 av ein enkel prosessor, så kan me identifisera nokre viktige funksjonsblokker som vil bli omtalt nærare i dette og neste kapittel. Det er også vist med forkortingane KB eller SB kva type blokk dette er.

Dekodar (KB): Med ein verdi inn på inngangane, dvs. eit bitmønster, kan ein f.eks. velja kva for register som skal motta eit resultat frå ein operasjon i ALU-en, dvs. den aritmetisk-logiske eininga i mikroprosessoren.

Multipleksar, MUX (KB): Dette er ein kanalveljar. På inngangssida er det kobla til to eller fleire digitale kanalar. Ein kanal kan vera ei digital signallinje eller ein buss, dvs. ein datastraum med breidde på eit vist antal bit. Eit eksempel er datalinjene frå eit register i mikroprosessoren.

I tillegg har multipleksaren slik som dekodaren nokre inngangssignal der ein kan setja ein verdi eller bitmønster. Dette mønsteret vil bestemma kva kanal som skal sleppa gjennom multipleksaren. F.eks. kan ein då styra kva register som skal levera verdien sin til ALU-en.

Adderar (KB): Blokk i ALU-en. Når du f.eks. skriv $C = A + B$ i programkoden din, går verdiane til A og B, dvs. bitmønstra, gjennom logikken i denne blokka. Resultatet hamnar i eit prosessorregister.

OG-, ELLER-, XELLER- og Inverteringsblokk (KB): Blokker i ALU-en. Desse blir realiserte som vist i kapittel 3.5, og utvida til å handtera verdiar med ei viss breidde, f.eks. 32 bit.

Komparator (KB): Blokk i ALU-en. Når du f.eks. skriv *if* $(A > B)$ i programkoden din, går verdiane til A og B, dvs. bitmønstra, gjennom logikken i ei komparatorblokk.

Skift-blokk (SB): Dette er også ei blokk i ALU-en. Skiftoperasjonar er vanlege i program. Ei skiftblokk er også f.eks. ein viktig del av ein omformarar frå parallelle til serielle datastraumar og omvendt.

Register (SB): Dette er eit minne som kan lagra ein verdi av ei viss breidde. Registra i ARM-kjernen vår er på 32 bit.

Eit slikt register er bygd opp av 32 minneelement på 1 bit, der kvart minnelement er basert på ei såkalla **data-vippe** ("D flip-flop").

Teljar (SB): Denne blokka er også bygd opp av vipper og på ein slik måte at verdien som er lagra, kan aukast eller minkast. Ei slik blokk er bl.a. ein sentral del av ein taimer, sjå f.eks. SysTick-taimeren i figur 2.11 på side 61.

Dei viste kombinatoriske blokkene kjem i det følgjande. Dei sekvensielle blokkene blir omhandla i kapittel 3.9.

3.8.2 Dekodar

Ein dekodar er ein krets der ein med **n** inngangar kan setja ein av 2^n utgangar høg mens resten er låge. Det er også ein såkalla **aktivt låg** variant der den valde utgangen går låg mens dei andre er høge.

Ein mikroprosessor vil bruka ei slik funksjonsblokk for å velja kva for modul som f.eks. skal motta eller levera data. Inngangssignala er då ofte del av ein **adresseverdi** som mikroprosessoren bruker for å nå den komponenten han vil "snakka med".

Me tar nå utgangspunkt i **funksjonstabellen** til ein dekodar med
- $n = 2$ inngangsvariablar S_0[80] og S_1 og
- $2^n = 4$ **aktivt høge** utgangsvariablar Y_0, Y_1, Y_2 og Y_3.

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Tabell 3.8: Funksjonstabell for ein $2 \rightarrow 4$-dekodar.

Av tabellen ser me at dei **logiske likningane** for alle utgangane på SOP-form[81] blir enkelt og greitt som følgjer:

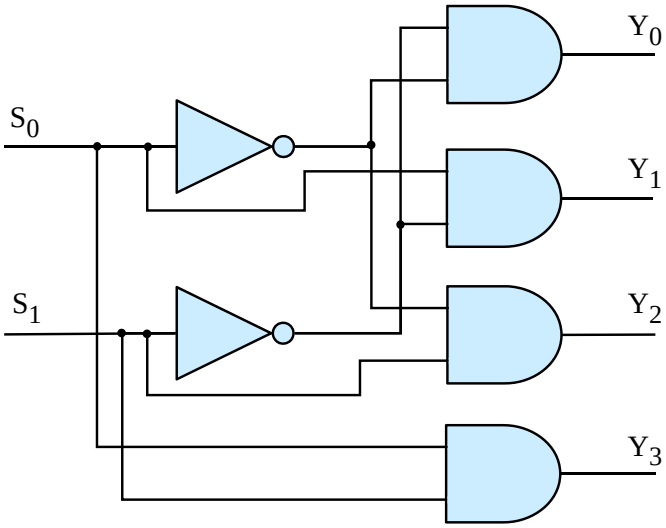
$Y_0 = \overline{S_1} \cdot \overline{S_0}$
$Y_1 = \overline{S_1} \cdot S_0$
$Y_2 = S_1 \cdot \overline{S_0}$
$Y_3 = S_1 \cdot S_0$

Eit logisk skjema for dekodaren er vist i figur 3.49.

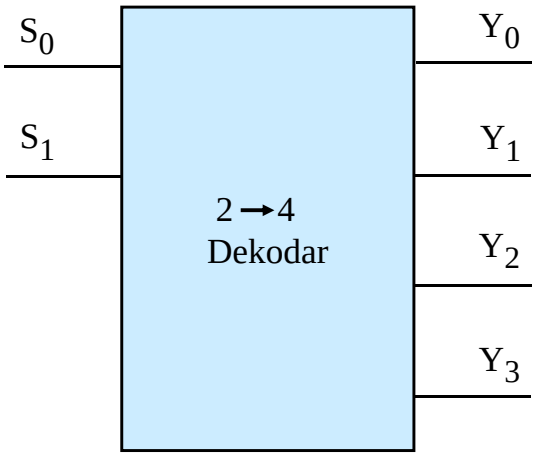
Eit funksjonsblokksymbol er vist i figur 3.50.

[80] Av "Select".

[81] SoP-metoden går altså ut på å setja opp ei logisk likning for kvar utgang som ein sum av dei inngangskombinasjonane som gir utgangsverdi lik 1.



Figur 3.49: Dekodarlogikk.



Figur 3.50: Dekodarsymbol.

Eksempel 3.28. *Dekodar som kan opnast og stengast.*

Det skal lagast ein $2 \rightarrow 4$-dekodar med følgjande ut- og inngangar:

- Fire **aktivt låge** utgangar[82] $\overline{Z}_0, \overline{Z}_1, \overline{Z}_2$ og $\overline{Z}_3$.
- To styresignal S_0 og S_1.
- Eit **aktivt høgt** signal kalla E for å opna ("enable") eller stenga dekodaren. Når $E = 0$ skal altså ingen utgangar kunne gå lågt, dvs. bli 0.

Kjernen i denne dekodaren, dvs. sjølve $2 \rightarrow 4$-logikken, vil bli som i figur 3.49. Logikken for å opna/stenga og generera aktivt låge signal må leggjast til på utgangssida av denne kjernelogikken.
Me skal som vanleg ved konstruksjon gå gjennom følgjande steg:

Spesifikasjon > Funksjonstabell > Logiske likningar > Logisk skjema

Spesifikasjonen er gitt øvst i eksempelteksten. Når det gjeld funksjonstabell og logiske likningar, kan ein sjå på kvart utgangssignal for seg.
Funksjonstabell for eit aktivt lågt utgangssignal $\overline{Z}_i, i = 0, .., 3$, er vist i tabell 3.9. Hugs at utgangssignala Y_i frå kjernelogikken er aktivt høge. I tabellen er det også tatt med den inverse utgåva av $\overline{Z}_i$, nemleg Z_i.

E	Y_i	$\overline{Z}_i$	Z_i
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabell 3.9: Funksjonstabell for eit utgangssignal frå dekodaren.

Logisk likning på SOP-form for utgangssignalet $\overline{Z}_i$ kan me lettast finna ved å ta utgangspunkt i SOP-likninga for det inverse utgangssignalet:

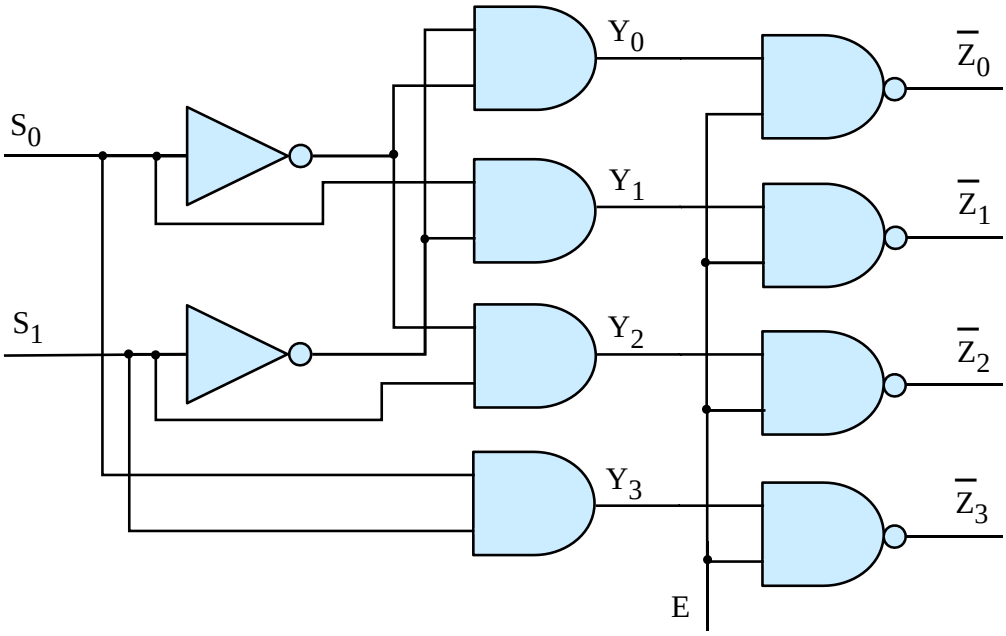
$$Z_i = E \cdot Y_i$$

Dette gir at

$$\overline{Z}_i = \overline{E \cdot Y_i}$$

Tilleggslogikken kan då realiserast med fire NOG-portar som vist i det logiske skjemaet i figur 3.51.

[82] Merk: Det er vanleg å gi aktivt låge signal namn med ein inverteringsstrek over. Dette skal då visa at desse er *aktive* når verdien er '0'. I dette eksemplet vil den linja som skal aktiverast, få verdien '0' mens dei *inaktive* linjene vil ha verdien '1'.



Figur 3.51: Dekodar med opne/stengesignal.

3.8.3 Multipleksar

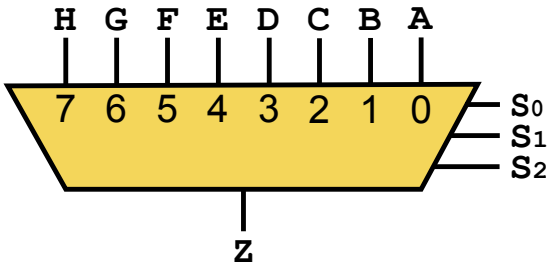
3.8.3.1 Funksjon

Ein multipleksar[83], MUX, er ein krets der ein med **n** inngangar kan velja kva for eit av 2^n digitale inngangssignal[84] eller **kanalar** som skal få passera gjennom funksjonsblokka. Det er altså bare **ein** utgang.
Dei **n** val-linjene (”select lines”) blir som for dekodaren ofte kalla **adresselinjer**.

Ein mikroprosessor vil bruka ei slik funksjonsblokk til f.eks. å kanalisera data frå rett prosessorregister til ALU-en. I prosessoren vår vil det då vera 16 kanalar inn på multipleksaren, og kvar kanal vil ha ei breidde på 32 bit. Ein treng då 4 adressebit for å velja kva datakanal som skal sleppast gjennom multipleksaren.

Me har alt sett eit eksempel på ein $4 \rightarrow 1$-multipleksar i kapittel 3.7.1.3 på side 160 om programmerbar logikk. MUX-en i figur 3.39 er som ein ser, bygt opp av ein dekodar og fire tri-statebuffer[85] og er ein sentral del av LUT-modulen.

Neste trinn i multipleksar-rekkja er ein $8 \rightarrow 1$-multilpleksar. Eit vanleg symbol for denne er vist i figur 3.52.



Figur 3.52: 8→1-multilpleksar. (Ref.: ”Wiki Multiplexer_8_to_1.svg”.)

3.8.3.2 Eksempel på kommersiell multipleksar

Multipleksaren i figur 3.52 er vanleg som kommersiell IC-krets. Eit eksempel er kretsen 74HC151 frå produsenten NXP. Denne er som vist i databladet[86], basert på ”vanlege” portar, dvs. utan tri-stateoppførsel på utgangen. Dette er greitt i mange system, mens i andre tilfelle har ein som nemnt i kapittel 3.5.2 behov for å kobla kretsen vekk i frå det som er kobla til utgangane. Då er slike buffertrinn nødvendige på utgangen av kretsen.

[83] Sjå også ”en.wiki Multiplexer”.
[84] Eit signal kan ha ei breidde på m bit. For eit enkelt digitalt signal er som kjent $m = 1$.
[85] Sjå kapittel 3.5.2.
[86] Søk på ”Farnell.no 74HC151N” frå produsenten NXP og opna databladet ”Technical data sheet”.

3.8.3.3 Realisering og analyse av ein enkel multipleksar

Me skal nå sjå på den minste multipleksaren og vil realisera denne på same måte som den kommersielle multipleksaren 74HC151, dvs. med vanlege portar.

Funksjonstabell og logisk likning
Me tar utgangspunkt i ein $2 \rightarrow 1$-multipleksar der ein vhja. signalet S kan sleppa eitt av inngangssignala A og B gjennom til utgangen Y .

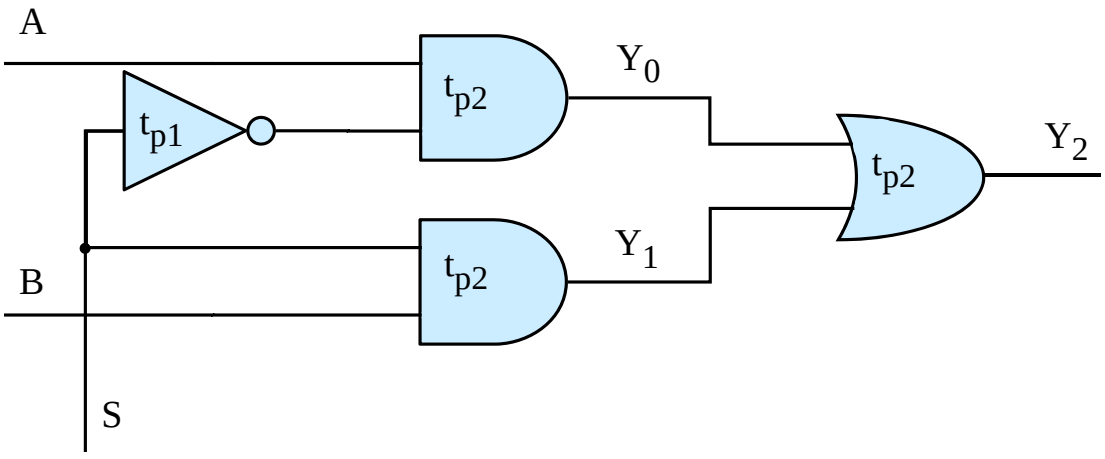
Funksjonstabellen blir som vist under:

S	Y
0	A
1	B

Logisk likning for utgangen blir då:

$Y = \overline{S} \cdot A + S \cdot B$

Logisk skjema
Ei realisering basert på "vanlege" portar, dvs. utan tri-stateoppførsel på utgangen, er vist i figur 3.53.
I figuren er det Y_2 som er sjølve utgangen på multipleksaren. Y_0 og Y_1 er bare signal på vegen gjennom mux-en.



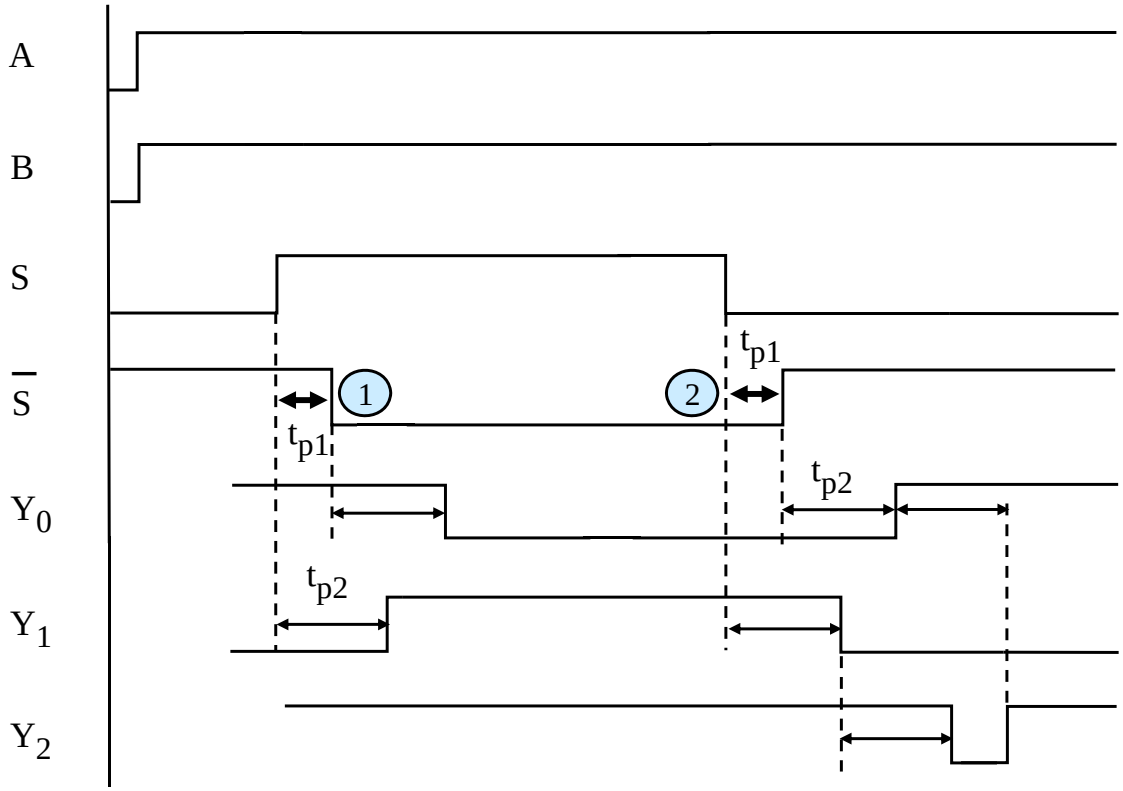
Figur 3.53: Realisering av $2 \rightarrow 1$-multilpleksar.

Tidsdiagram
Me skal bruka denne enkle multipleksaren til å visa nokre viktige sider ved oppførselen til

digital elektronikk. Me tenkjer oss følgjande som **utgangspunkt**:

- Begge inngangssignala A og B går **høge**.
- Etter ei stund går styresignalet S **høgt**.
- Ei stund etter dette går styresignalet S **lågt** igjen.

Eit tidsdiagram som viser oppførselen til inn- og utgangane er vist i figur 3.54.



Figur 3.54: Tidsdiagram for $2 \rightarrow 1$-multilpleksaren.

Som ein såg i kapittel 3.5.7, er det **tidsetterslep** ("propagation delay") gjennom portane. For å gjera det enkelt, har ein her gått utfrå følgjande:

- Etterslepet frå høgt til lågt nivå er det same som for motsett **overgang/transisjon**, dvs. $t_{PLH} = t_{PHL} = t_P$.
- Etterslepet for invertaren er t_{P1} og for dei andre portane t_{P2}.

Analyse
Ut frå figuren kan ein gjera følgjande observasjonar:

- Når styresignalet S går høgt, vil det ta ei tid t_{P1} før det inverterte signalet $\overline{S}$ går lågt. I dette tidsintervallet som er markert med tjukk pil og talet **1** i figuren, **vil *begge* signala bli slept gjennom multipleksaren**. Viss signala har ulikt nivå, vil det signalet som er høgt, vinna[87].
- Når styresignalet S går lågt, vil det ta ei tid t_{P1} før det inverterte signalet $\overline{S}$ går høgt. I dette tidsintervallet som er markert med tjukk pil og talet **2** i figuren, **vil *ingen* signal bli slept gjennom multipleksaren**. **Merk** at virkninga av dette vil koma på utgangen Y_2 først etter to portetterslep, dvs. $2 \cdot t_{P2}$, som vist i figuren.

Etter ein **transisjon** i styresignalet S, går det altså ei viss tid før ein kan **stola på** det som kjem på utgangen. Dette er ein ganske vanleg situasjon i digital elektronikk. Etter signaltransisjonar må det generelt gå ei viss tid før utgangssignal er **stabiliserte**.

Pulsen som kjem på utgangen Y_2, blir kalla ei **glipe** ("glitch")[88]. Den skal ideelt sett ikkje koma då både A og B er høge, men kan altså dukka opp i praktiske koblingar som ein ser her.
Gliper kan vera veldig korte. Dei dukkar i tillegg ofte bare opp ved spesielle signalnivåkombinasjonar og er dermed vanskelege å detektera. Viss ein ikkje tar tilstrekkjeleg omsyn til dei reelle eigenskapane til komponentar ved konstruksjon av maskinvare, vil oppførselen til maskinvaren bli upåliteleg. I fotnota over er det nemnt nokre eksempel. Eit nyare eksempel er ei tidleg utgåve av Xbox360 der forsyningsspenninga falt i eit kort intervall ved heilt spesielle signalnivåkombinasjonar i bussystemet. Desse var ekstra krevjande for kraftforsyninga i maskinen, og den resulterande spenningsglipa gjorde at prosessoren hengde seg opp.
Slikt kan skje pga. ei spenningsglipe inn til prosessoren, men gliper kan også gi signalnivåendringar på bussane. Viss bare eitt bit endrar verdi på f.eks. ein instruksjonsbuss, vil prosessoren gjera noko heilt anna enn det han skulle og kan hamna i uføret.

Ei vanleg løysing er å laga logikken slik at utgangssignalet ikkje blir endra før etter ei viss tid $t > t_P$, der t_P er etterslepet i logikkmodulen. Ein hindrar då utgangssignalet å påvirka etterfølgjande logikkmodular før signalet er stabilt. Den mest vanlege måten å realisera dette på, er ved å innføra eit klokkesignal. Utgangen får då lov til å endra seg bare når klokkesignalet f.eks. går frå lågt til høgt nivå. Meir om dette kjem i kapittel 3.9.5 om **vipper**. Slik logikk kallar me **klokkestyrt** eller **synkron** logikk.

I andre tilfelle ordnar ein dette vhja. spesielle styresignal. Viss ein f.eks. skal overføra data til eit minne, må både datasignal samt adressesignala få stå ei stund på inngangane før minnekretsen låser dataverdien inn på rett adresse i minnet. Dette ordnar ein ofte ved at den komponenten som overfører dataverdien, f.eks. ein mikroprosessor, køyrer ein

[87] Jfr. funksjonen til ELLER-porten på utgangen.
[88] "en.wiki Glitch".

transisjon på eit styresignal[89] ei stund etter at data- og adresseverdiar er stabiliserte.

Mange logikkmodular fungerer greitt utan klokkestyring. Bildørlogikken i figur 3.26 på side 139 er eit eksempel på dette. Slik logikk kallar ein **asynkron** logikk.

Oppsummering av tidsanalysen
Multipleksaren er ei funksjonsblokk som slepp gjennom eitt av fleire inngangssignal. Ved skifte mellom ulike inngangssignal kan det oppstå tidsintervall der utgangen ikkje er til å stola på. Dette er ein vanleg situasjon i digital logikk og er vist her for å illustrera behovet for **synkronisering**. Dette vil altså seia at etterfølgjande logikkmodular reagerer på inngangssignala på faste og klokkestyrte tidspunkt, og etter at endringar er stabiliserte. Det er likevel mykje logikk som ikkje treng vera klokkestyrt, og denne blir kalla **asynkron** logikk.

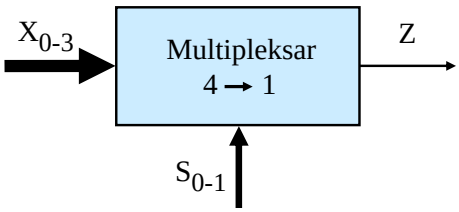
3.8.3.4 Konstruksjonseksempel

Ein skal her visa med eit eksempel koss ein kan laga ein multipleksar med eitt hakk større kompleksitet enn i førre kapittel.

Eksempel 3.29. $4 \rightarrow 1$-*multipleksar.*

Det skal lagast ein $4 \rightarrow 1$-multipleksar der ein vhja. styresignala S_0 og S_1 kan sleppa eitt av inngangssignala
X_0 - X_3 ut på utgangen Z.

Eit blokkskjema av multipleksaren er vist i figur 3.55.



Figur 3.55: $4 \rightarrow 1$-multipleksar.

Me skal igjen gå gjennom følgjande steg:

Spesifikasjon > Funksjonstabell > Logiske likningar > Logisk skjema

Spesifikasjonen er gitt øvst i eksempelteksten.
Funksjonstabell er vist i tabell 3.10.

[89] Dette signalet heiter ofte $WR(ite)$.

S_1	S_0	Z
0	0	X_0
0	1	X_1
1	0	X_2
1	1	X_3

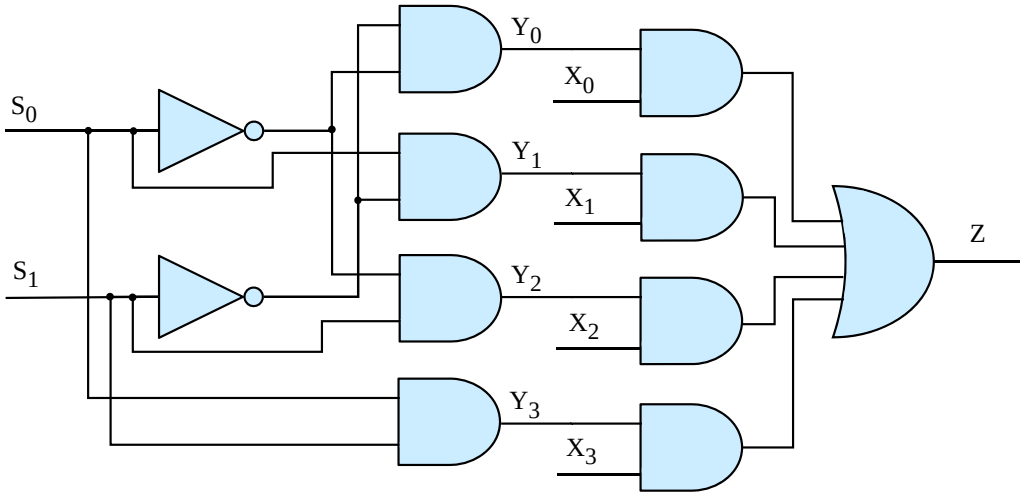
Tabell 3.10: Funksjonstabell for ein $4 \rightarrow 1$-multipleksar.

Av tabellen ser me at den **logiske likninga** for utgangen Z på SOP-form blir som følgjer:

$$\begin{aligned} Z &= \overline{S_1} \cdot \overline{S_0} \cdot X_0 + \overline{S_1} \cdot S_0 \cdot X_1 + S_1 \cdot \overline{S_0} \cdot X_2 + S_1 \cdot S_0 \cdot X_3 \\ &= Y_0 \cdot X_0 + Y_1 \cdot X_1 + Y_2 \cdot X_2 + Y_3 \cdot X_3 \end{aligned} \tag{3.24}$$

Sjølve styringslogikken, dvs. logikken mellom styresignala S_0 og S_1 og hjelpesignala Y_0 - Y_3, er altså som for $2 \rightarrow 4$-dekodaren i figur 3.49 på side 183.

Dette går også greitt fram av det logiske skjemaet for multipleksaren som er vist i figur 3.56.



Figur 3.56: Logisk skjema for $4 \rightarrow 1$-multipleksar.

3.8.4 Digital komparator

3.8.4.1 Innleiing

Det finst to hovudtypar komparatorar, nemleg analoge og digitale komparatorar.

Ein **analog** komparator samanliknar nivået til to analoge signal A og B og gir ut resultatet på ein digital utgang. A og B er vanlegvis tidsvarierande spenningar. Høgt nivå tilseier at spenninga A er høgast. Omvendt situasjon gir lågt signal ut[90].

I dette kapitlet skal me sjå på **digitale** komparatorar. Ein slik komparator samanliknar talverdiane A og B og har vanlegvis følgjande tre digitale utgangar:

- $Q_{A>B}$: Denne er høg når A > B.
- $Q_{A=B}$: Denne er høg når A = B.
- $Q_{A<B}$: Denne er høg når A < B.

Verdiane A og B er på binær form og har ei oppløysing på n bit.
Altså:

$A: \ A_{n-1}A_{n-2}\ldots A_i\ldots A_2A_1A_0$
$B: \ B_{n-1}B_{n-2}\ldots B_i\ldots B_2B_1B_0$
der
i - bitnummer der *0* viser kva som er LSbit.
A_i, B_i - siffer i tala A og B.

Samanlikningsoperasjonen kan realiserast vhja. følgjande vanlege metodar:

- Dedikert komparatorlogikk.
- Subtraksjonslogikk med tilhøyrande statusflagg.

Dei neste underkapitla tar for seg desse metodane.

3.8.4.2 Dedikert komparatorlogikk

Utgangspunktet her er å utvikla logikk som utfører samanlikninga av to n-bitstal A og B. Det aller enklaste tilfellet er at dette er **1-bitstal**, dvs. at $A: A_0$ og $B: B_0$.

Ein ser då greitt at funksjontabellen for logikken blir som vist i tabell 3.11.

[90] Då analoge signal i prinsippet har uendeleg oppløysing, vil ikkje signala bli eksakt like nokon gong.

A_0	B_0	$Q_{0,A>B}$	$Q_{0,A=B}$	$Q_{0,A<B}$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

Tabell 3.11: Funksjonstabell for ein 1-bitskomparator.

Ut frå denne kan me setja opp følgjande logiske likningar for dei tre utgangane basert på SoP-metoden:

$$\begin{aligned} Q_{0,A>B} &= A_0\overline{B_0} \\ Q_{0,A=B} &= \overline{A_0} \cdot \overline{B_0} + A_0 B_0 = \overline{A_0 \oplus B_0} \\ Q_{0,A<B} &= \overline{A_0} B_0 \end{aligned} \tag{3.25}$$

Når ein nå skal gå eitt steg vidare og sjå på samanlikning av **2-bitstal**, kan det vera lurt å setja opp ein meir kompakt funksjonstabell.
I tabell 3.12 er dette gjort på same måten som i databladet på 4-bitskomparatoren 74HC85, [26].

A_1,B_1	A_0,B_0	$Q_{1,A>B}$	$Q_{1,A=B}$	$Q_{1,A<B}$
$A_1 > B_1$	x (dvs. vilkårleg.)	1	0	0
$A_1 < B_1$	x	0	0	1
$A_1 = B_1$	$A_0 > B_0$ (Dvs. $Q_{0,A>B} = 1$)	1	0	0
$A_1 = B_1$	$A_0 = B_0$ (Dvs. $Q_{0,A=B} = 1$)	0	1	0
$A_1 = B_1$	$A_0 < B_0$ (Dvs. $Q_{0,A<B} = 1$)	0	0	1

Tabell 3.12: Funksjonstabell for ein 2-bitskomparator.

Som ein ser av tabellen, vil f.eks. følgjande gjelda:

- Viss $A > B$, så skuldast det at $A_1 > B_1$ **eller** at $A_1 = B_1$ **og** $A_0 > B_0$.
Her er då vilkåret for at $A_0 > B_0$ gitt av $Q_{0,A>B}$ i likning 3.25.

Basert på dette kan me på tilsvarande måte som sist, setja opp følgjande logiske likningar for dei tre utgangane:

$$\begin{aligned} Q_{1,A>B} &= A_1\overline{B_1} + \overline{A_1 \oplus B_1} \cdot Q_{0,A>B} \\ Q_{1,A=B} &= \overline{A_1 \oplus B_1} \cdot Q_{0,A=B} \\ Q_{1,A<B} &= \overline{A_1} B_1 + \overline{A_1 \oplus B_1} \cdot Q_{0,A<B} \end{aligned} \tag{3.26}$$

Utgangane er kalla $Q_{1,\ldots}$ for å skilja desse frå $Q_{0,\ldots}$, og det er nå f.eks. utgangen $Q_{1,A>B}$, som fortel om **heile** 2-bitstalet A større enn talet B eller ikkje.

Som ein ser av likningane over, er det nå greitt å utvida logikken til kor mange bit ein ønskjer.
Logikk som samanliknar f.eks. to 4- eller 8-bits talverdiar, finst som nemnt realisert som eigne integrerte kretsar.
I det nemnde databladet på 4-bitskomparatoren 74HC85[91], [26], kan ein også finna logisk skjema for kretsen.

Denne komparatorlogikken kan sjølvsagt også realiserast i FPGA. I verktøyet *Simulink med System Generator* har ein som vist i figur 3.44 på side 172, biblioteksmodulen *Relational* som innheld slik logikk. Dette er ei fleksibel blokk der ein kan spesifisera kva slags samanlikningsoperasjon ein vil gjera.

3.8.4.3 Komparatoroperasjonar basert på subtraksjon

Ein samanlikningsoperasjon kan også utførast av ein mikroprosessor. Som ein ser av tabell 2.10 på side 76 har ARM Cortex-M3 to slike *cmp*-instruksjonar. Viss ein slår opp på ein av desse i kapittel 7 i arkitekturreferansemanualen, [21], vil ein sjå at instruksjonen blir realisert vhja. **subtraksjon**. Instruksjonsdekodaren som er vist i figur 2.15 på side 73, vil altså her aktivera subtraksjonslogikken[92] i ALU-en.

Som resultat av subtraksjonen vil det automatisk setjast flagg i statusregisteret til prosessoren. Ein etterfølgjande hoppinstruksjon vil då testa desse flagga som vist i kapittel 2.4.4.9 på side 81.
Assemblysekvensen under gir eit eksempel på dette:

```
        ....
merke1  ....            ;
        ....
        cmp   r1, #10   ; Subtraksjonen r1 - 10 blir utført. Flagg blir sett
                        ; eller nullstilt.
        bne   merke1    ; Viss Z-flagget = 0, hopp. Ved Z = 1 går ein vidare.
       ....             ; (Z = 1 betyr at resultatet av operasjonen blei 0.)
```

Andre hoppinstruksjonar vil som kjent testa på andre enkeltflagg eller kombinasjonar av flagg.

Samanlikningsoperasjonar i ein prosessor vil altså ikkje krevja eigen logikk i ALU-en. Det er ikkje behov for å leggja til den logikken som finst i dedikerte samanlikningskretsar/blokker.

Ein kan også finna komparatorstoff f.eks. på Wikipedia, sjå "en.wiki Digital comparator".

[91] Databladet kan lett finnast på internettet ved å bruka søkjeorda "nxp 74hc85".
[92] Som vist i kapittel A.2.2.2, blir subtraksjon realisert vhja. toar-komplement og addisjon. Det er altså eigentleg addisjonslogikken som blir aktivert.

3.8.5 Adderar

Ein skal her sjå på korleis ein kan realisera digital logikk som adderer tal. Det er viktig då å ha på plass eit grunnlag i talsystem og aritmetikk. Dette grunnlaget blir gitt i vedlegg A.1 - 2.

3.8.5.1 Logikk som realiserer addisjon

Første variant: Halv-adderar
Me tar utgangspunkt i korleis addisjon av to tal A og B blir utført, sjå addisjonseksemplet i figur A.4 på side 295. Tala har ei gitt breidde på n bit, og kan då framstillast slik:

$A: \ A_{n-1}A_{n-2}...A_i...A_2A_1A_0$
$B: \ B_{n-1}B_{n-2}...B_i...B_2B_1B_0$
der
i - bitnummer der *0* viser kva som er LSbit.
A_i , B_i - siffer i tala A og B.

Me innser greitt at funksjonstabellen for addisjon av dei to binære siffera A_0 og B_0 i tala, blir som vist i tabell 3.13.

A_0	B_0	S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabell 3.13: Funksjonstabell for ein halvadderar.

S_0 er **summen** og C_1[93] er **mente** ("carry").

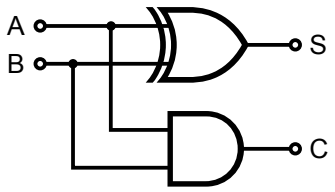
Av tabellen ser me at summen S_0 er ein rein XELLER-funksjon av inngangane A_0 og B_0, og mentet C_1 er ein OG-funksjon.

Dette gir følgjande logiske likningar:

$$S_0 = A_0 \oplus B_0 \tag{3.27}$$
$$C_1 = A_0 \cdot B_0 \tag{3.28}$$

Desse kan realiserast[94] vhja. logiske portar som vist i figur 3.57.

[93] Dette er vanleg notasjon då mentet er ein inngang til neste addisjonssteg, her steg 1.
[94] Sjå også f.eks. "en.wiki Adder (electronics)".



Figur 3.57: Halv-adderar. (Ref.: "Wiki Half_Adder.svg")

Dette blir som vist i figuren, kalla ein **halv-adderar**. Når ein skal utføra addisjonen eit hakk til venstre i addisjonsprosessen, må ein som kjent ta omsyn til at addisjonen til høgre kan gi mente. Dette mentet blir ein **ekstra inngangsvariabel** i addisjonsprosessen. Logikk for dette blir vist i det følgjande.

Full-adderar

Med mentet[95] frå addisjonen til høgre som ekstra inngangsvariabel, innser me her også greitt at funksjonstabellen for addisjon av dei to binære siffera A_i og B_i i tala, blir som vist i tabell 3.14.
S_i og C_{i+1} blir då utgangsvariablar.

C_i	A_i	B_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabell 3.14: Funksjonstabell for ein fulladderar.

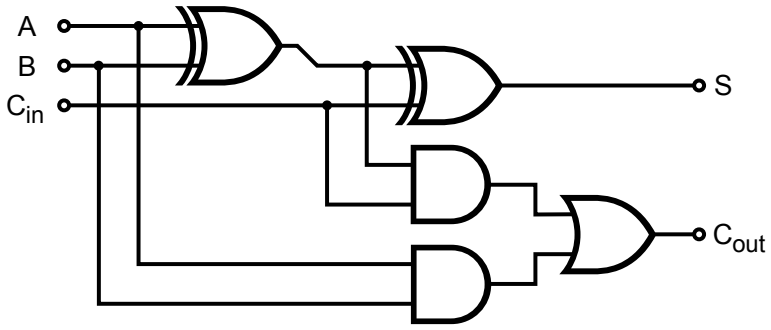
Viss ein set opp dei logiske likningane på SOP-form og så forenklar desse, kan ein koma fram til følgjande:

$$S_i = B_i \oplus (A_i \oplus C_i) \tag{3.29}$$

$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i \tag{3.30}$$

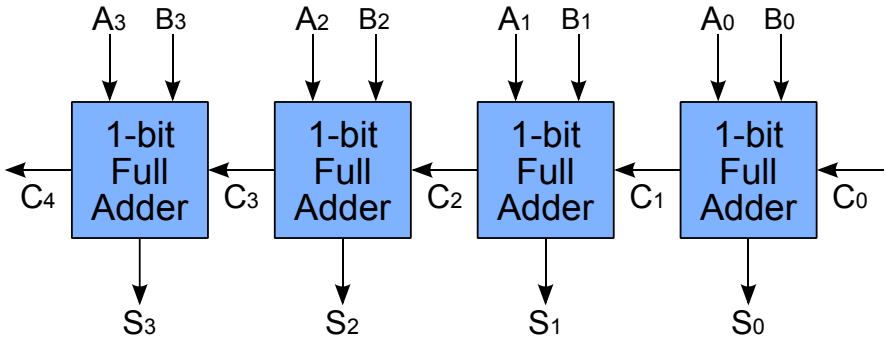
Desse kan realiserast vhja. logiske portar som vist i figur 3.58. I figuren er **utgåande** mente C_{out} det same som C_{i+1} og **inngåande** mente C_{in} det same som C_i i likningane over.

[95] Hugs at mentet også er ein variabel som er 0 eller 1.



Figur 3.58: Full-adderar. (Ref.: "Wiki Full_Adder.svg")

Adderar av n-bitstal
Viss ein nå definerer logikken i figur 3.58 som ei adderarblokk, kan ein vhja. n slike realisera ei maskinvareløysing for addisjon av to n-bitstal.
Ei realisering av ein **full-adderar** for 4-bitstal er vist i figur 3.59.



Figur 3.59: Full-adderar for 4-bitstal. (Ref.: "Wiki 4_bit_ripple_carry_adder.svg")

Merk at ein bruker ein full-adderar for A_0 og B_0 også. I nokre tilfelle har ein bruk for C_0. Eit eksempel er der ein skal realisera 64-bitsaddisjon vhja. ein ALU på 32-bit. Dette vil då køyrast som to etterfølgjande addisjonar der mentet C_{32} frå den første blir inngangsvariabel C_0 for den neste addisjonen.

Ei stor ulempe ved oppsettet i figur 3.59 er at addisjonane må skje **sekvensielt** då ein addisjon vil vera avhengig av resultatet, dvs. mentet, frå addisjonen til høgre for seg.
Ein får då eit **tidsetterslep** proporsjonalt med bitbreidda, noko som gir ein treg operasjon.
Utgåva i figuren blir ofte kalla ein **rippeladderar**, då operasjonen gir ein **mente-rippel** eller **-bølgje** gjennom logikken.
Ei raskare adderarutgåve blir omtala i det følgjande.

Adderar med menteframsyn

Det kan visast at likningane for dei ulike menta C_i, $i = 1, \ldots n$, kan omformast til funksjonar av alle føregåande A_i og B_i, dvs. for $i-1, i-2, \ldots, 1, 0$, samt C_0[96]. Funksjonane er viste i referansen i fotnota og kan skrivast som følgjer:

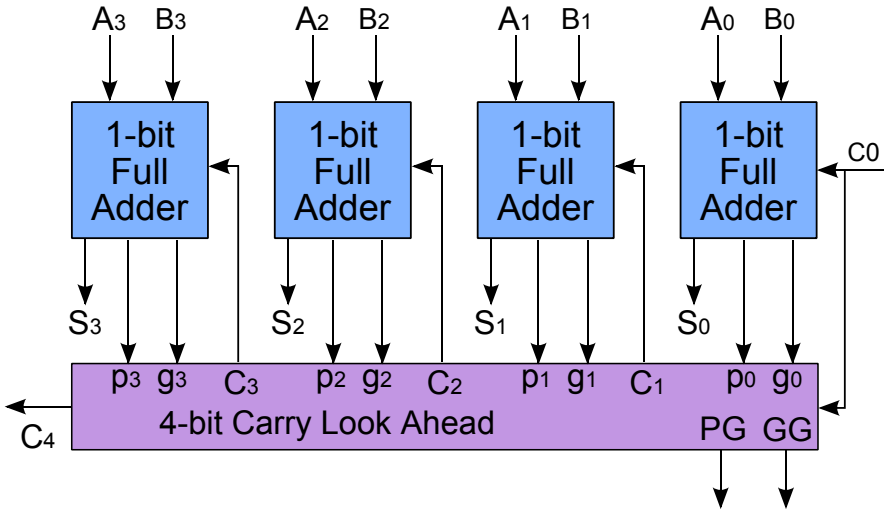
$$g_i = A_i \cdot B_i \tag{3.31}$$

$$p_i = A_i \oplus B_i \tag{3.32}$$

$$C_{i+1} = f(g_i, p_i, g_{i-1}, p_{i-1}, \ldots, g_0, p_0, C_0) \tag{3.33}$$

Fullstendige mentelikningar kan finnast i referansen.
Med likningane på denne forma er ein ikkje avhengig av å vite C_{i-1} **før** ein reknar ut C_i. Ein kan altså **sjå forbi** addisjonsresultata når ein reknar ut menta, noko som blir kalla **menteframsyn** ("carry-lookahead") eller **mentepredikSjon**, sjå fotnota over.

For å realisera menteframsyn må ein laga ei eiga blokk som genererer dei ulike menta basert på likningane over.
Eit oppsett for addisjon av 4-bitstal er vist i figur 3.60.



Figur 3.60: Full-adderar for 4-bitstal med menteframsyn. (Ref.: "Wiki 4_bit_carry_lookahead_adder.svg")

Dette vil gi ein mykje raskare addisjon. Artikkelen i fotnota over, tar eksempel i ein 16-bitsaddisjon der menteframsyn aukar reknefarten med typisk 4 gonger.

[96] Sja f.eks. "en.wiki Carry-lookahead adder".

3.8.5.2 Subtraksjon

Som vist i vedlegg A.2.2, kan ein bruka ei adderar-blokk til å utføra både addisjon og subtraksjon. Vilkåret er at tala er representerte på toar-komplementform.
I datamaskinar blir derfor denne forma brukt, og den aritmetiske logikken i ein prosessor blir då mest mogleg kompakt og effektiv.

3.8.5.3 Litt om multiplikasjon og divisjon

Som nemnt under avsnittet om skiftoperasjonar i kapittel 2.4.1.2, så er den tradisjonelle algoritmen for multiplikasjon bygd opp av addisjons- og skiftoperasjonar.
Divisjon er på tilsvarande måte basert på subtraksjons- og skiftoperasjonar. Addisjonsmodulen er altså sentral i begge desse aritmetiske operasjonane, men i dei mest effektive algoritmene unngår ein dei tradisjonelle og samla sett tidkrevjande sekvensane av addisjon og skifting[97].
Prosessorar som ARM Cortex-M3 har ein innebygd maskinvaremodul som utfører multiplikasjon av to heiltal på 32bit i løpet av **ein** klokkesyklus.

Algoritmen for divisjon er også effektivisert. Ein populær variant for maskinvarerealisering, SRT-divisjon, er basert på oppslagstabellar ("look-up table")[98]. Divisjon er likevel ein mykje tyngre operasjon og kan i ARM Cortex-M3 ta opptil 12 klokkesyklar.

[97] Litt om effektive multiplikasjonsalgoritmar for realisering i maskinvare står hjå "en.wiki Binary multiplier".
[98] Sjå "en.wiki Division algorithm".

3.9 Sekvensielle funksjonsblokker

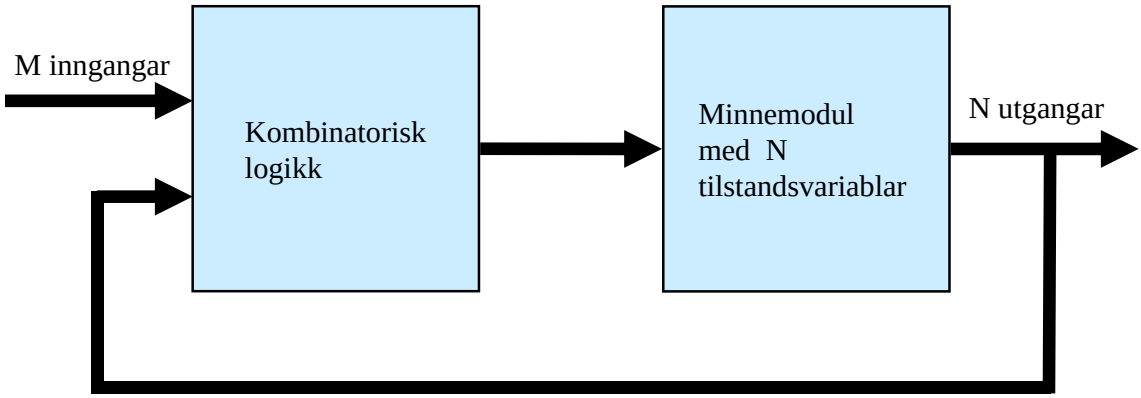
3.9.1 Struktur

I **sekvensiell logikk** er neste tilstand som vist i kapittel 3.2.2, avhengig av verdiane på inngangane i det gjeldande tidspunkt, men også av nåverande tilstand.
Eit enkelt eksempel er som vist før, blinklyslogikk[99]. Viss tilstanden nå er **av**, skal neste tilstand vera **på**.

Karakteristiske kjenneteikn ved sekvensielle logiske system er:

- Systemet kan vera i ulike **tilstandar**.
- Systemet inneheld eit **minne** som "hugsar" nåverande tilstand.

Eit generelt sekvensielt system blei vist i kapittel 3.2.2, og er gjentatt i figur 3.61.



Figur 3.61: Generelt sekvensielt system.

Det er som nemnt før, to hovudtypar sekvensiell logikk:

- **Asynkron**: Tilstandsendringar skjer når inngangssignala endrar seg. Tida mellom slike endringar eller steg er då generelt varierande.
- **Synkron**: Tilstandsendringar går i en fast takt styrt av eit klokkesignal. Tida mellom stega er altså konstant. Dette er den vanlegaste typen sekvensiell logikk.

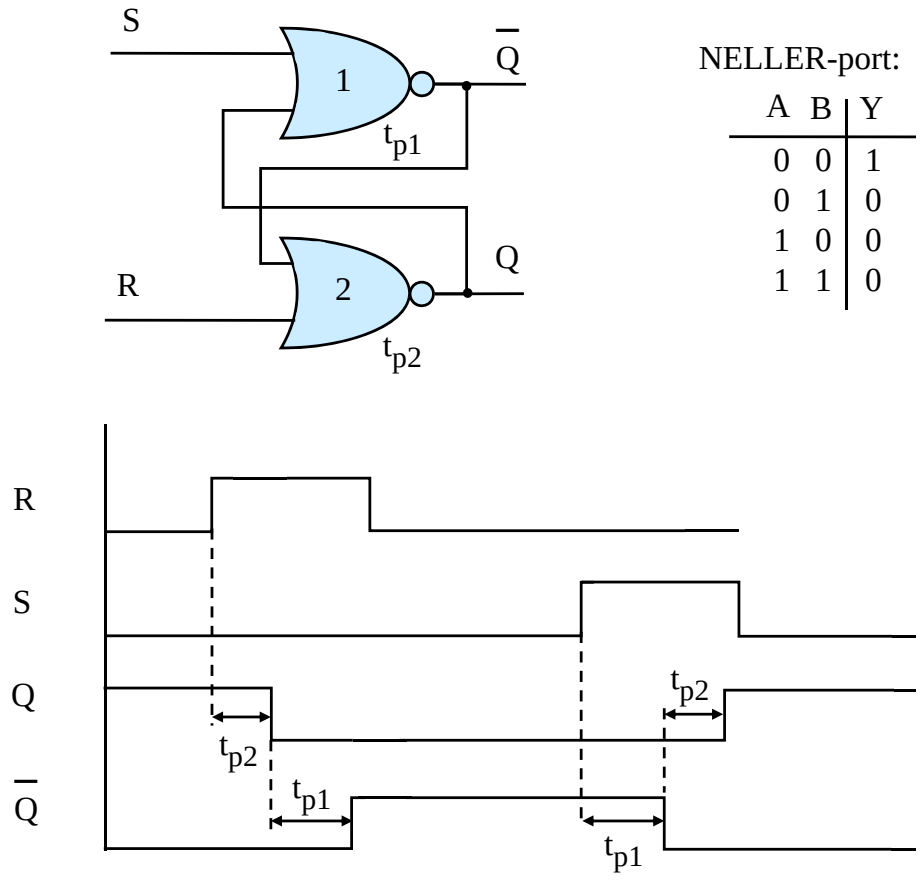
[99] Sjå eksempel 3.2 på side 109.

3.9.2 Minneelement i sekvensiell logikk

Eit minneelement har den eigenskapen at det har to stabile tilstandar, *høg/1* eller *låg/0*.
Det er to typar minneelement, nemleg **lås** ("latch") og **vippe** ("flip-flop")[100].
Hovudforskjellen mellom desse er at i ei vippe kan tilstandsendringane bare skje på faste tidspunkt styrt av ei klokke, mens i ein lås kan dette skje kva tid som helst når visse ting skjer på inngangen. Ein lås er altså asynkron mens ei vippe er synkron.
Dette vil truleg koma klart fram i løpet av kapitlet.

3.9.3 Enklaste minnelementet, ein S/R-lås

S/R-låsen er ein krets som er bygd opp av to krysskobla portar.
Ei utgåve basert på NELLER-portar er vist i figur 3.62. Funksjonstabellen for NELLER er i tillegg repetert i figuren.



Figur 3.62: SR-lås med NELLER-portar.

[100]Begge typane blir også presenterte under "en.wiki Flip-flop (electronics)".

Me ser først litt på inngangar, utgangar og tilstandar:

To inngangar: $S =$ Sett og $R =$ Resett.

To utgangar: Q og $\overline{Q}$

To tilstandar:
- $Q = 1$ ($\overline{Q} = 0$): *Sett*-tilstand
- $Q = 0$ ($\overline{Q} = 1$): *Resett*-tilstand

Figuren illustrerer også oppførselen til låsen vhja. eit tidsdiagram[101].
Basert på dette diagrammet kan me tenkja oss fylgjande **sekvens** av hendingar:

1. Låsen er *sett*, dvs. at han er i tilstanden gitt av $Q = 1$.

2. Ein **puls** blir påtrykt R-inngangen. Dette vil *resetja* låsen, dvs. få han over i tilstanden gitt av $Q = 0$.
   Merk at endringane ikkje skjer momentant, då det som me hugsar frå kapittel 3.5.7, vil vera tidsetterslep gjennom dei to portane. Desse er her kalla t_{p1} og t_{p2}[102].

3. Pulsen forsvinn, dvs. *Resett*-signalet går lågt. Tilstanden er likevel **stabil** pga. **tilbakekoblingane** ("feedback") frå utgangane.
   Merk at det er først etter tida $t_{p,SR} = t_{p1} + t_{p2}$ at tilbakekoblinga inn på port 2 har skifta nivå.
   Resett-pulsen må derfor vara lenger enn tida $t_{p,SR}$ for at SR-låsen garantert skal skifta tilstand.

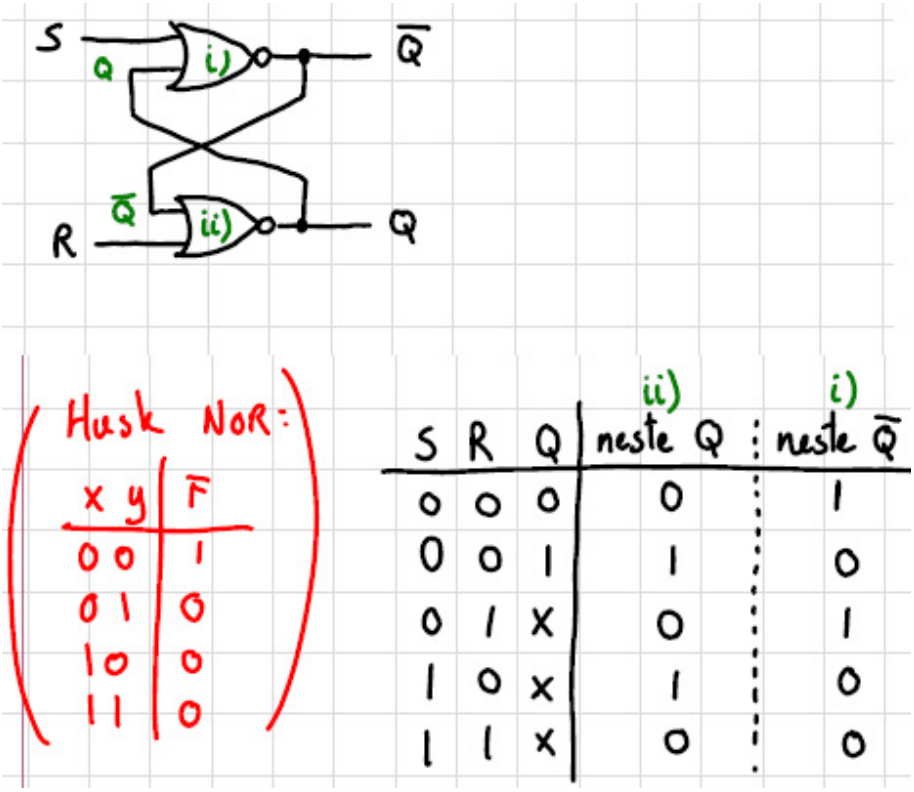
4. Når låsen nå er *resett*, vil derfor ein ny *Resett*-puls **ikkje** føra til noko endring. Det må ein Sett-puls til for å endra tilstanden i dette tilfellet.

5. Ein **puls** blir så påtrykt S-inngangen. Dette vil *setja* låsen, dvs. få han over i tilstanden gitt av $Q = 1$ etter ei viss tid. Og når låsen nå er *sett*, vil ein ny *Sett*-puls ikkje føra til noko endring. Låsen vil nå stå i denne stabile tilstanden til det kjem ein eventuell *Resett*-puls.
   Kravet til *Sett*-pulsbreidde blir det same som for *Resett*-pulsen.

Sekvensen over viser at ein med dei to inngangssignala kan **styra/programmera** kva tilstand låsen skal vera i. Mellom programmeringspulsane har ein stabil lagring av informasjonen, dvs. tilstanden. Låsen fungerer altså som eit **minne**. Den stabiliserande mekanismen er altså dei to tilbakekoblingane[103].

[101] SR-låsen blir animert på "en.wiki Flip-flop (electronics)".
[102] Etterslepa kan vera ulike sjølv om portane er av same type.
[103] Tilbakekobling er ein nødvendig del av mange system. Meir om dette står f.eks. hjå "en.wiki Feedback".

Sidan sekvensiell logikk er avhengig av både nåverande tilstand og inngangssignala, kan det vera nyttig å operera med ein **utvida funksjonstabell** for slike system. Figur 3.63 viser dette for den NELLER-baserte SR-låsen.



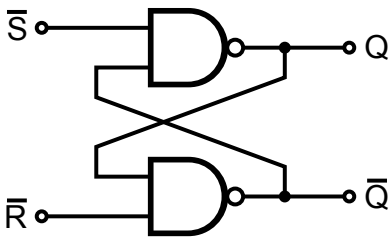
Figur 3.63: SR-lås og funksjonstabell. (Ref.: Tom Ryen, [12].)

Ein har altså utvida inngangssida av tabellen. Neste tilstand, dvs. nivået på utgangane er altså generelt ein funksjon av inngangane samt nåverande tilstand. Dette kan formulerast slik:

$$Q_{k+1} = f(R, S, Q_k) \tag{3.34}$$

der $f(.)$ er ein **logisk funksjon** og k er nåverande **steg** i ein sekvens.

I figur 3.64 er det vist ein SR-lås basert på NOG-portar og **aktivt låge** styresignal.



Figur 3.64: SR-lås basert på NOG-portar.
(Ref.: "Wiki SR_Flip_flop_Diagram.svg".)

Styresignala $\overline{S}$ og $\overline{R}$ er her høge når dei er **inaktive**. Viss f.eks. låsen er resett, dvs. i *Resett*-tilstand, vil ein låg puls på $\overline{S}$-inngangen setja låsen, dvs. gi $Q = 1$.
I det følgjande eksemplet skal me sjå i detalj på oppførselen til denne låsen.

Eksempel 3.30. *Illustrasjon av oppførselen til ein NOG-basert SR-lås*

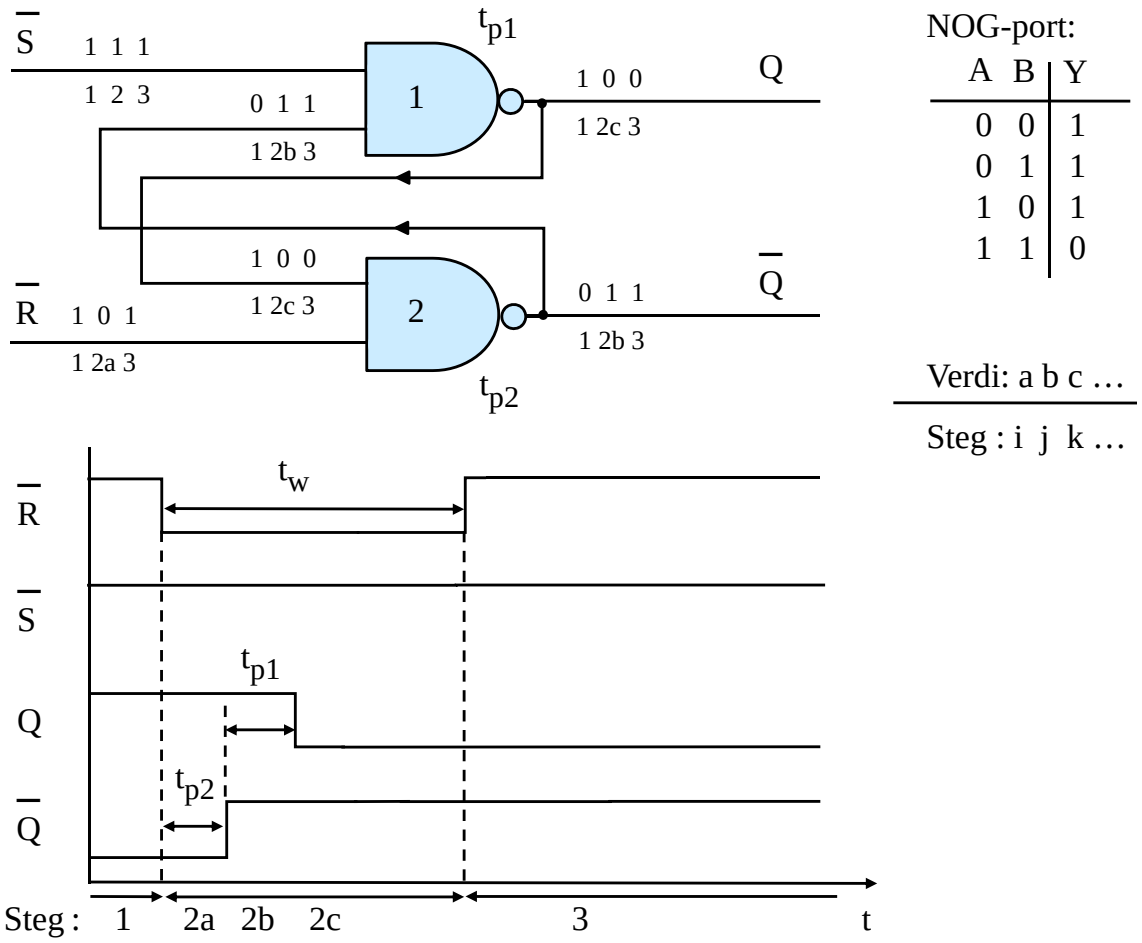
Ein skal her sjå på NOG-utgåva av SR-låsen som vist i figur 3.64 og lar utgangspunktet vera følgjande:

- Etterslepet gjennom den øvre porten er t_{p1} og t_{p2} gjennom den nedre porten.
  Ein ser vekk frå etterslepet gjennom tilkoblingslinjene.
- Låsen er *sett*, dvs. at han er i tilstanden gitt av $Q = 1$.
- *Sett*-signalet er deaktivert, dvs. høgt.
- Ein **puls** blir så påtrykt R-inngangen.
  Pulsbreidda ("width") $t_w > t_{p1} + t_{p2}$.

I figur 3.65 er det vist i detalj kva som skjer i denne situasjonen.

I tillegg til det vanlege tidsdiagrammet, er det eit logisk skjema der ein prøver å visa heile sekvensen av hendingar inne i SR-låsen. Sekvensen er delt opp i fleire tidssteg. I steg 1 er SR-låsen i *Sett*-tilstand og i steg 3 er han over i ein ny og stabil tilstand, nemleg *Resett*-tilstanden.
Sjølve transisjonsfasen, dvs. steg 2, er delt opp i 3 mindre steg for å visa korleis signala går gjennom SR-låsen. I alle stega er også signalverdiane viste.

Ein resettpuls som er breiare enn det samla portetterslepet vil altså garantert *resetja* låsen som vist i figuren, dvs. få han over i tilstanden gitt av $Q = 0$.

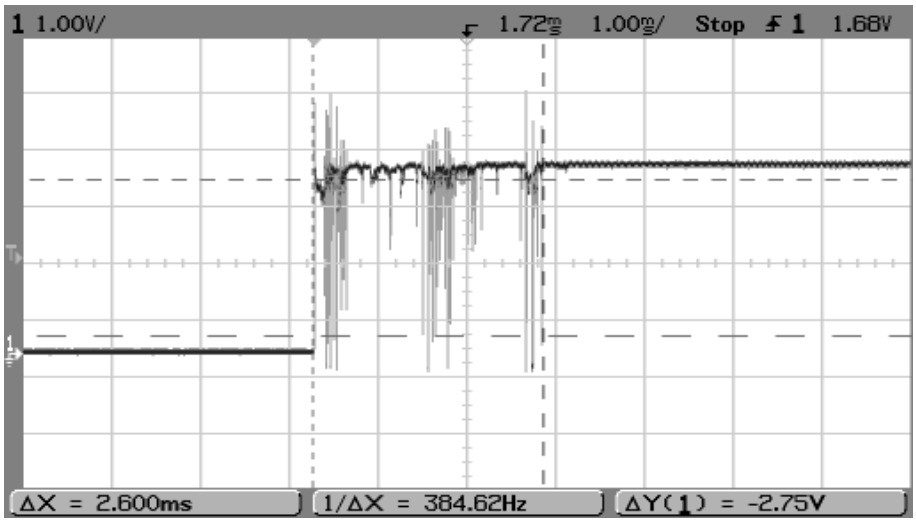


Figur 3.65: Transisjonsfasen til ein SR-lås.

Eit eksempel på bruksområde for SR-låsar er vist under.

Eksempel 3.31. *Avprelling av brytarar vhja. SR-lås*

Alle brytarar med unntak av heilt spesielle typar som kvikksølvbaserte brytarar, har kontaktprell[104]. Ved trykk på brytaren kan det i verste fall gå opptil 10 - 40 millisekund før kontaktflatane ligg stabilt mot kvarandre og ein har ei stabil elektrisk spenning ut frå brytarkoblinga. I mellomtida kan denne spenninga svinga fleire gonger mellom høgt og lågt nivå som vist i eksemplet i figur 3.66.



Figur 3.66: Kontaktprell. (Ref.: "Wiki Bouncy_Switch.png".)

Eitt og same brytartrykk vil derfor generera mange spenningspulsar. Viss ein bare les spenningssignalet inn i datamaskinen utan å behandla dette, vil dette kunne **tolkast som mange trykk** av datamaskinen.
Det er derfor nødvendig med avprelling ("debouncing") av enkle brytarar og tastatur som er kobla til ein datamaskin. Avprelling blir gjort vhja. ulike metodar som vist i det følgjande.

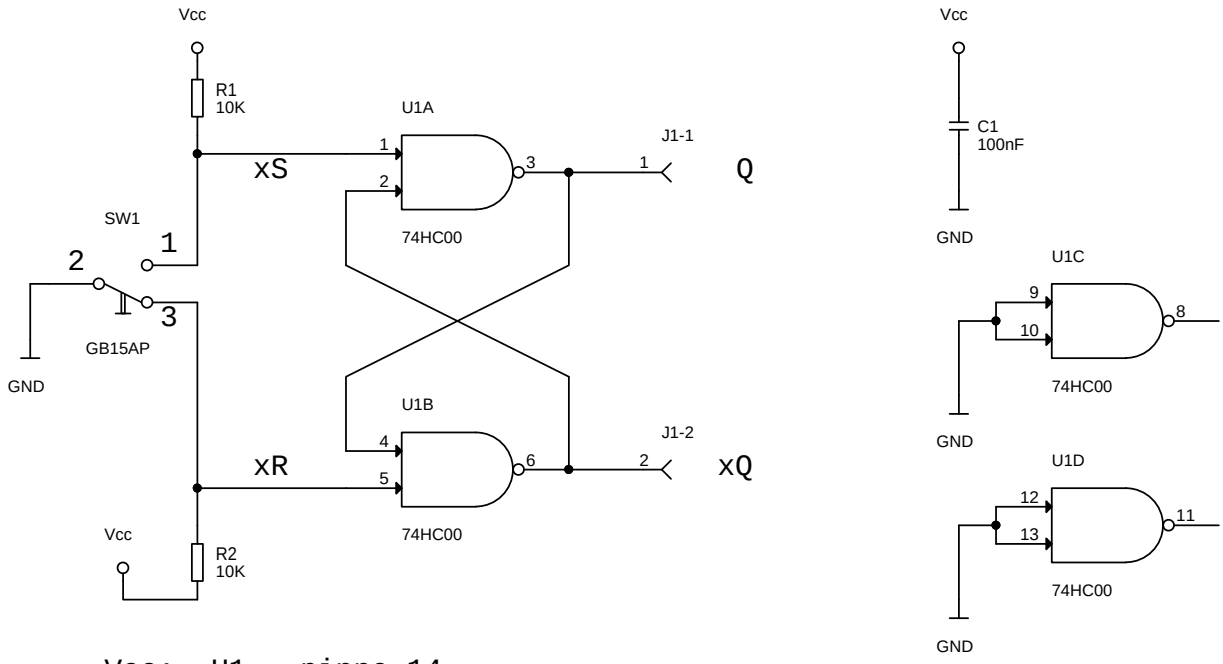
- **Analog filtrering:** Vhja. ein motstand R og kondensator C kan ein laga eit RC-filter for spenningssignalet. Det er ei enkel kobling, men ikkje like enkelt å dimensjonera filteret slik at oppførselen blir robust. Eksempel her er brytarkoblingane på mikrokontrollerkortet *STM32VLDiscovery* der ein kan setja inn ein kondensator[105] og dermed realisera eit RC-filter, sjå kretsskjema bakerst i brukarmanualen, [15].
- **Digital filtrering:** Ein kan la spenningssignalet passera gjennom ein logikkmodul som ikkje godkjenner eit **trykk** før signalet har vore stabilt

[104] Sjå og "en.wiki contact bounce".
[105] Det er altså lagt til rettes for dette på kretskortet, men kondensatoren er ikkje montert.

høgt i ein viss tidsperiode. Tilsvarande godkjenner logikken ikkje eit brytarslepp før signalet har vore lågt like lenge.
Ei spesiell, men vanleg løysing er bruk av SR-vippe som digital filterkrets. Dette er ei enkel og robust løysing, men er bare aktuell for **vekselbrytarar**. Dette er brytarar med tre kontaktflater 1 - 3 der trykk gir kontakt mellom 1 og 2, og slepp gir kontakt mellom 1 og 3[106].

- **Avprelling i programvare:** Dette er også ei svært vanleg løysing og liknar i virkemåte på det digitale filteret. Ein vil her lesa brytarsignalet på faste tidspunkt, f.eks. kvart 10. millisekund, og kan då krevja at ein f.eks. skal ha tre etterfølgjande høge avlesingar for å godkjenna eit trykk. Tilsvarande krav vil gjelda for eit slepp.

Ei vanleg kobling for ein vekselbrytar[107] med SR-basert avprelling er vist i figur 3.67.



Figur 3.67: Vanleg oppsett for avprelling av vekselbrytar.

Her kan ein f.eks. kobla utgangsignalet Q til datamaskinen.

Når ein trykkjer brytaren over i øvre stilling, dvs. lagar kontakt mellom kontaktflatane 1 og 2 i figuren, vil signalet xS, dvs. $\overline{S}$, vera aktivert. Utgangsignalet Q vil då gå høgt. Sjølv om brytaren i starten av trykket vil dansa eller prella litt mot den øvre kontaktflata (1) slik at signalet xS svingar litt mellom høgt

[106] Ein vanleg brytar er anten lukka eller open, dvs. at det er kontakt eller ikkje kontakt.

[107] Denne kan også heita endevendar ("Single Pole Double Trow", SPDT). Sjå "no.wiki Strømbryter" om ulike brytartypar.

og lågt nivå, vil utgangsignalet Q som me har sett tidlegare, halda seg **stabilt** høgt.
Brytaren må sleppast slik at han går heilt over på den nedre kontaktflata (3) før utgangen går låg. Og igjen vil litt prell mot den nedre flata ikkje gi noko endring når utgangen først er gått låg.
Dette er altså ein velbrukt avprellingsmetode for slike brytarar.

Funksjonstabell for den NOG-baserte SR-låsen i figur 3.65, som altså har aktivt låge inngangar, er vist i tabell 3.15.

$\overline{S}$	$\overline{R}$	Q_k	Q_{k+1}	$\overline{Q}_{k+1}$	Merknad
1	1	0	0	1	Stabil lagring av tilstand.
1	1	1	1	0	
0	1	0	1	0	Sett tilstand til '1'.
0	1	1	1	0	
1	0	0	0	1	Resett tilstand til '0'.
1	0	1	0	1	
0	0	0	1	1	Ustabil tilstand der både
0	0	1	1	1	Q og $\overline{Q}$ er '1'.

Tabell 3.15: Funksjonstabell for den NOG-baserte SR-låsen i figur 3.64.

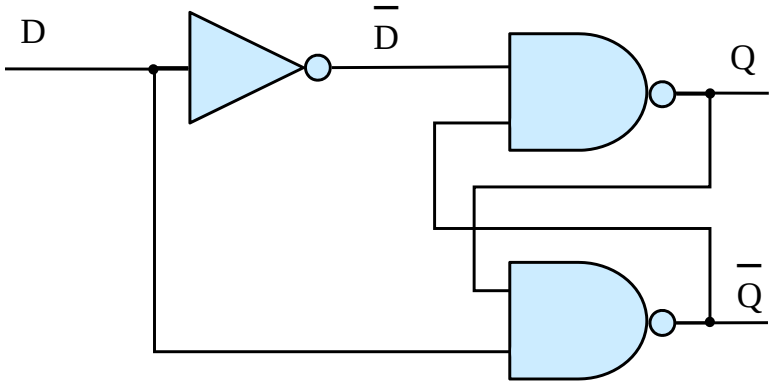
SR-låsen er laga for å vera i ein av to stabile tilstandar, og han blir anten *sett* eller *resett* vhja. inngangane. Viss ein derimot aktiverer begge inngangssignala **samtidig**, så vil begge utgangane som vist i funksjonstabellen, vera høge så lenge begge signala er aktive. Dette innser ein greitt ved f.eks. å studera den NOG-baserte SR-låsen i figur 3.64.
Dette kan me kalla ein **ustabil tilstand** då SR-låsen vil gå over i ein av dei to stabile tilstandane med ein gong[108] me deaktiverer ein eller begge inngangane. Den som blir deaktivert **sist**, bestemmer då kva tilstand dette blir.
Dette gjeld for situasjonar der det er stor nok forskjell på deaktiveringstidspunkta.

Viss ein **deaktiverer** begge inngangssignala samtidig eller veldig nært i tid, får me ein interessant situasjon. Det blir då eit **kappløp/res** ("race")[109][110] der den tregaste **signalvegen**[111]bestemmer kva tilstand låsen hamnar i.
Då ein vanlegvis ikkje veit kva signalveg som er den tregaste i elektronikk, vil samtidig deaktivering gi eit **usikkert resultat**.

[108]Rett nok etter eit visst etterslep gjennom portane.
[109]Ordet er opphaveleg nordisk, sjå "http://www.nob-ordbok.uio.no/" der du legg inn ordet *res*. Ordet kom truleg til Storbrittania med vikingane på 900-talet. Det kom då saman med andre ord som f.eks. norrønt *baggi* som blei til "bag". Desse orda har me altså nå henta tilbake igjen.
[110]Du kan finna meir om res i digitale kretsar hjå "en.wiki race condition".
[111]Signalvegen inkluderer både portar og banar på og utanfor ei brikke som signala må gjennom.

3.9.4 Datalås

For å unngå at begge inngangane på ein lås er aktive samtidig, kan ein tenkja seg ei kobling som vist i figur 3.68.

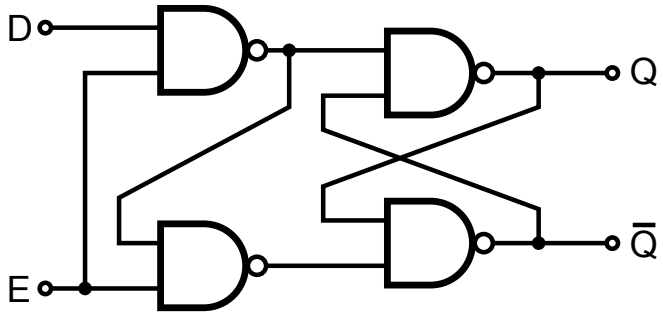


Figur 3.68: Enkel D-lås.

Inngangane har her ideelt sett alltid ulikt nivå. Pga. inverteren vil det likevel vera korte tidsintervall der begge inngangane er aktive. Dette er tilsvarande oppførselen til multiplekssaren som me såg på i kapittel 3.8.3.3 på side 187. Ein unngår derfor ikkje heilt at det kan førekoma korte **gliper** (”glitch”) her.

Koblinga er likevel viktig på ein annan måte, då ho er første enkle steg fram mot eit robust lagringselement for databitar. Inngangssignalet er i figuren kalla D for **data**, då dette altså kan sjåast på som ein enkel datalås, dvs. eit minneelement for ein databit. Etter ei endring til f.eks. $D = 1$ vil låsen etter ei lita stund gå over i den stabile tilstanden gitt av at $Q = 1$.

I figur 3.69 har ein innført eit tilleggssignal, E, som er eit signal for å **opna** (”Enable”) låsen.

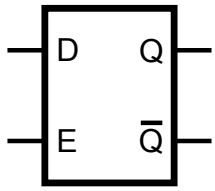


Figur 3.69: Transparent D-lås. (Ref.: ”Wiki D-Type_Transparent_Latch.svg”.)

Ein kallar ofte dette for ein **transparent** D-lås då utgangen Q vil **gjenspegla** inngangen D når låsen er open, dvs. $E = 1$. Når låsen er stengt, dvs. når $E = 0$, vil han lagra den verdien som D hadde rett før stenging.

Låsen er altså open for dataendringar så lenge $E = 1$, dvs. at dataendringar vil **utløysa/trigga** tilstandsendringar så lenge **nivået** til E ligg høgt. Låsen er altså **nivå-trigga** ("level-triggered").

Eit vanleg symbol for ein transparent D-lås er vist i figur 3.70.



Figur 3.70: Symbol for transparent D-lås.
(Ref.: "Wiki Transparent_Latch_Symbol.svg".)

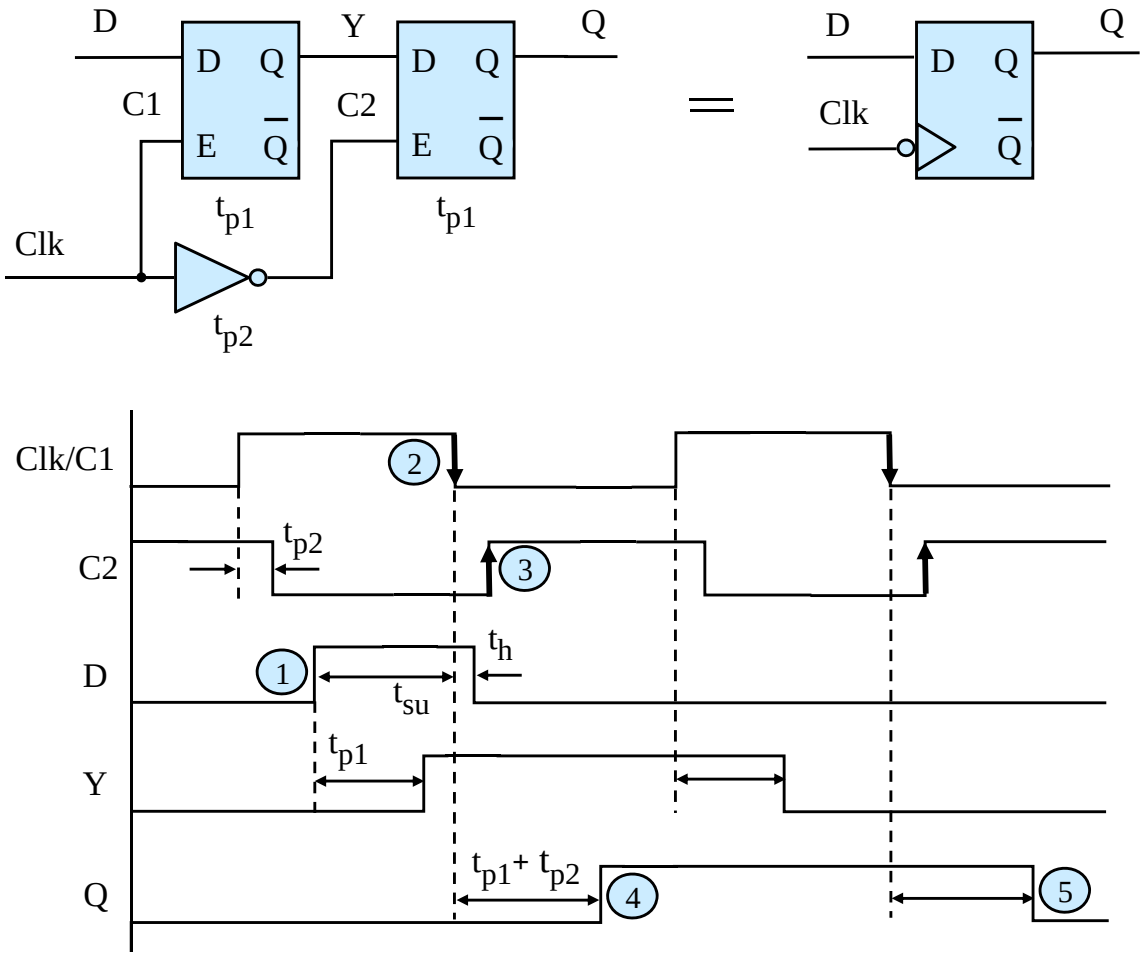
Det er viktig å ha opne/stenge-signalet E av fleire grunnar:

- D-inngangen kan vera **gyldig** ("valid") bare i bestemte tidsintervall. Viss minneelementet f.eks. tar imot data via eit bussystem, vil det bare vera i heilt bestemte intervall at databitane på bussen er meint for akkurat dette minneelementet.
  Signalet E vil då typisk vera kobla til utgangen på ein dekodar. Dekodaren vil på inngangssida ta inn adresselinjer og styresignal frå bussen. Ved ein bestemt kombinasjon av desse signala er det dette minneelementet som skal ha data, og då blir E automatisk aktivert av dekodaren. Meir om dette kjem bl.a. i kapittel 3.10.9 om minne.
- Som me har sett eksempel på, så kan det som ein konsekvens av signalendringar koma **gliper**, dvs. intervall der eit signal har feil nivå. Viss f.eks. D nettopp er under endring, er det vanleg å forseinka aktiveringa av E til D er stabil.

3.9.5 Datavippe

Det å ha eit opne/stenge-signal E er altså viktig. I mange system ønskjer ein i tillegg at endring av tilstand bare skal skje på bestemte tidspunkt. Ved nivåtrigging får ein ikkje til dette då endringar kan skje i **heile** tidsintervallet der E er høg.
Ein vil også ha den situasjonen at det etter ei endring av E vil vera korte intervall der både S- og R- linja internt i låsen er aktive pga. forskjellar i etterslepa gjennom dei ulike portane. Det er derfor også viktig i mange system at utgangen Q ikkje får påvirka etterfølgjande system før han er stabil.

Dette kan ein oppnå vhja. såkalla **flanketrigging** ("edge triggering"). Eit flanketrigga minne for ein databit kan ein laga ved å setja saman to datalåsar som vist øvst i figur 3.71.



Figur 3.71: Oppsett og oppførsel til ei negativt trigga D-vippe.

Styresignalet er nå kalla *CLK*. Viss nå ein ny dataverdi kjem inn på datainngangen *D*, vil verdien i dette tilfellet bli lagra når CLK går frå høgt til lågt nivå, dvs. på **negativ flanke** av klokkesignalet. Eit **flanketrigga** minneelement blir kalla **datavippe** eller D-vippe ("D flip-flop"[112]).

Symbolet for ei negativt trigga D-vippe er vist til høgre i figuren.

Oppsettet i figur 3.71 blir også kalla ei **Master/Slave**-kobling der første låsen er master-låsen. Den siste låsen blir kalla slave då han blindt følgjer etter det som skjer av endringar i masterlåsen.

Me skal nå sjå meir inngåande på korleis D-vippa virkar.
Eit tidsdiagram for den negativt trigga D-vippa er vist nedst i figuren.
Her legg ein merke til at slaven blir driven av eit klokkesignal som er ei invertert utgåve av masterklokka. Dette vil då ha eit lite etterslep i forhold til masterklokka.

[112]Sjå igjen "en.wiki Flip-flop (electronics)".

Det er **slaven** som **lagrar** sjølve **tilstanden**. Ei tilstandsendring kan delast opp i følgjande steg med stegnummer viste i tidsdiagrammet:

1. **Ny dataverdi kjem inn til vippa:** Når *Clk* går låg, er masterlåsen ikkje mottakeleg for dataendringar. Den nye dataverdien må derfor **setjast opp** på inngangen ei viss tid før negativ klokkeflanke, slik at dataendringa får tid til å koma seg gjennom inngangsportane i masterlåsen og setja eller resetja denne[113]. Denne tida kallar ein ofte **oppsettingstid** ("set-up time") t_{su}. Det vil vera eit minstekrav for denne tida som ein kan finna av databladet[114].
   Etter ei viss tid, dvs. etterslepet t_{p1} gjennom masterlåsen, vil utgangen Y få ny verdi.

2. **Masterklokka Clk går låg:** Det er viktig at den nye dataverdien blir halden heilt til Clk går lågt. Viss ikkje, vil ein kunne få ei utilsikta endring i aller siste liten. Det er altså i tillegg til **oppsettingstid** også eit krav til **haldetid** ("hold time") t_h for data. I utgangspunktet er det nok at data blir halde på inngangen fram til klokkeflanken, men pga. etterslepa i ulike greiner i masterlåsen vil ofte minstekravet vera at $t_h > 0$[115].
   Når *Clk* har gått låg, er **masterlåsen stengt**. Han vil då ikkje påverkast av nye endringar på datainngangen. Det er nå klart for innlåsing i slaven.

3. **Slaveklokka $\overline{Clk}$ går høg:** Den nye dataverdien Y på utgangen av masteren blir nå låst inn i slaven.

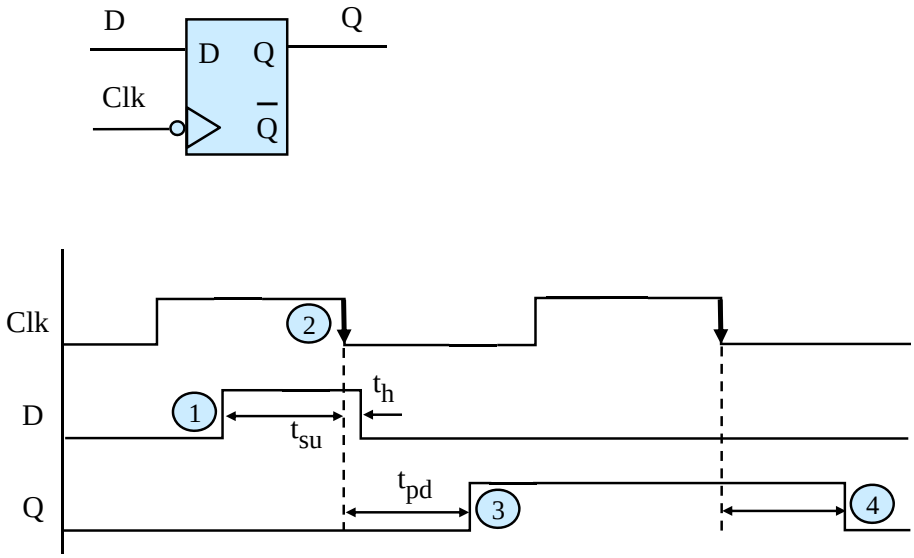
4. **Utgangen Q på D-vippa får ny verdi:** Etter ei viss tid, dvs. etterslepet t_{p1} gjennom slavelåsen, vil ein ha fått ny Q-verdi på utgangen til D-vippa. Det har då gått tida $t_{p1} + t_{p2}$ sidan negativ *Clk*-flanke.

5. **Ennå ein ny D-verdi har kome seg gjennom D-vippa:** Som ein ser av dette eksemplet på signalforløp, så går datainngangen D låg rett etter den negative *Clk*-flanken i punkt 2. Denne endringa blir låst inn i masteren når denne opnar igjen og går vidare når så slaven opnar på den neste negative klokkeflanken.

Sjølv om både master- og slavedelen av D-vippa er **nivåtrigga**, så kan D-vippa samla sett sjåast på som **flanketrigga**. På negativ klokkeflanke **lukkar ein** bokstaveleg talt data inn i vippa, og etter ei viss tid vil den nye dataverdien opptre på utgangen av vippa. Vippa vil då **lagra** denne verdien, også kalla tilstanden, til neste gong vippa opnar for ei eventuell ny endring.
Ein seier ofte at endringa her er **synkronisert** med negativ klokkeflanke.

Viss ein ser på D-vippa under eitt, kan oppførselen og dei viktige tidene for vippa oppsummerast som vist i figur 3.72.

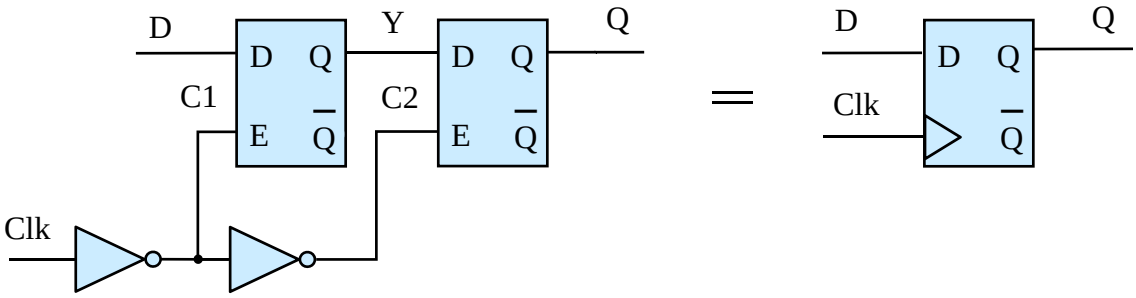
[113] Her blir masterlåsen sett då $D = 1$.
[114] Sjå f.eks. databladet til D-vippa 74HC74 frå NXP.
[115] Sjå igjen f.eks. databladet til D-vippa 74HC74 frå NXP.



Figur 3.72: Oppførsel og viktige tider for ei D-vippe.

Etterslepet ("propagation delay") frå negativ *Clk*-flanke og til Q endrar seg, er her summert saman til tida t_{pd}. Denne tida kan ein også finna av databladet for ei vippe. Tidene viste i figuren er sentrale for alle komponentar som lagrar data, dvs. register og minnekretsar mm. Det er derfor viktig å forstå opphavet til desse tidene slik det er vist i det føregåande. Kor raskt ein kan presentera nye dataendringar, dvs. maksimal klokkefrekvens for slike logiske blokker, er også gitt av desse tidene. I kapittel 3.9.7 skal me sjå meir på dette.

Til nå har ein bare vist negativt trigga D-vipper. Minst like vanleg er positivt trigga vipper. Endringar er då synkroniserte med positiv klokkeflanke. Klokkesignalet går her gjennom ein ekstra inverter før det går vidare til låsane i vippa. Oppsett og symbol for ei positivt trigga D-vippe er vist i figur 3.73.



Figur 3.73: Logisk skjema og symbol for ei positivt trigga D-vippe.

Til slutt her er det vist eit eksempel på korleis ein kan realisera ei D-vippe vhja. VHDL-kode. Oppsettet er vist under:

```
-- Først deklarasjon av bibliotek
library IEEE;
use     IEEE.STD_LOGIC_1164.ALL;

-- Portdeklarasjon
entity D_vippe is

   -- Oppsett av inngangar og utgang
   port (clk_in: in  std_logic;
         D_in  : in  std_logic;
         Q_out : out std_logic);

end D_vippe;

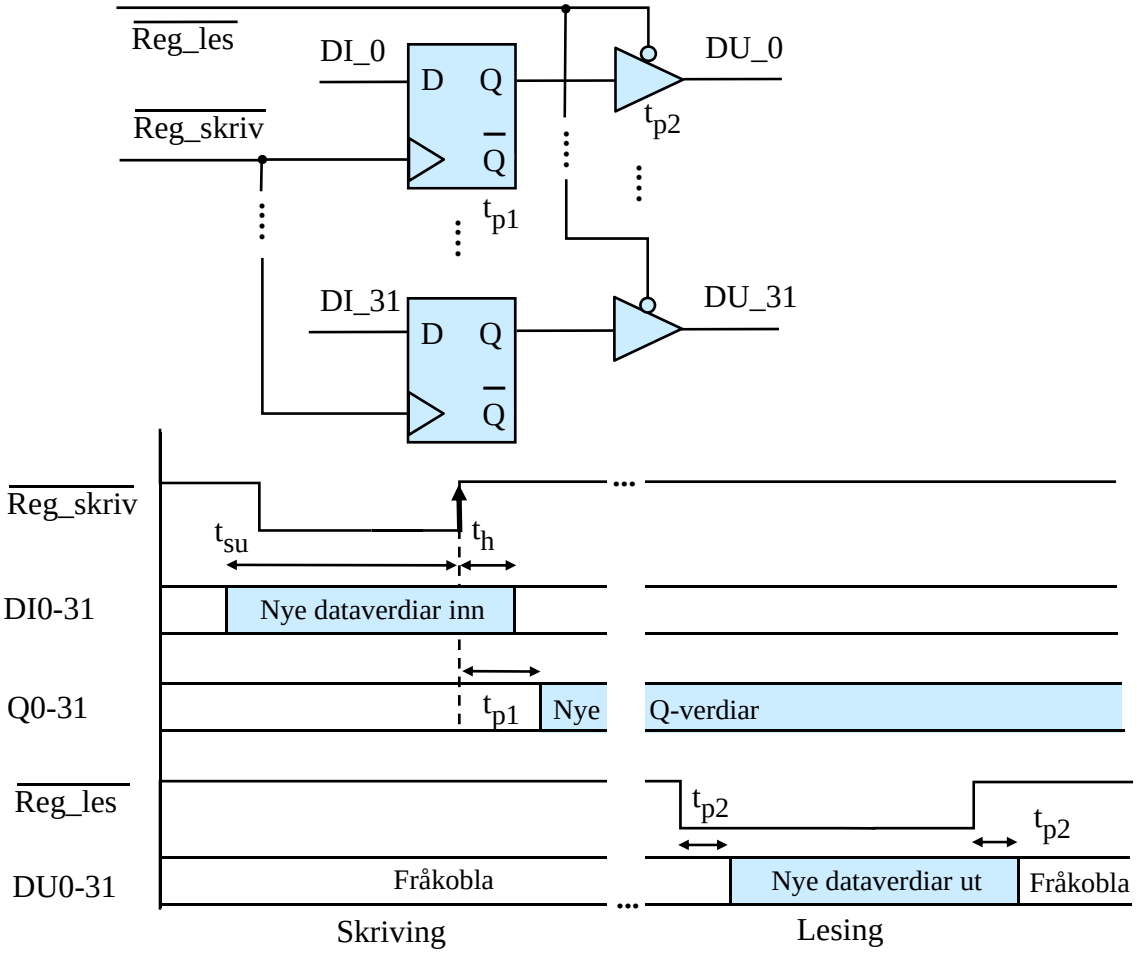
-- Deklarasjon av funksjon/oppførsel til D-vippa
architecture behavioral of D_vippe is
begin
process (clk_in, D_in)
begin
   -- Positiv flanke?
   if clk_in'event and clk_in = '1' then
      Q_out <= D_in;
   end if
end process

end behavioral;
```

3.9.6 Dataregister

Register har me sett mange eksempel på i kapittel 1 - 2. Både prosessorar og perifermodular har ei rekkje register, og desse blir brukt til mange ulike ting.

Eit register med breidde på n bit er bygd opp av n D-vipper kobla i parallell. I tillegg kjem logikk for å kunne lesa data parallelt og å kunne kobla utgangane vekk frå ein databuss mm.
I figur 3.74 er det vist eit eksempel på eit 32-bitsregister der ein **skriv** ein ny dataverdi og ei stund etterpå **les** denne ut av registeret igjen. Figuren viser altså begge typane **aksess**, dvs. skriving og lesing.



Figur 3.74: Skriving til og lesing frå eit register.

Styresignala for skriving og lesing, $\overline{Reg\_Skriv}$ og $\overline{Reg\_Les}$, kjem frå ein **dekodar**.
Legg merke til at det generelle klokkesignalet *Clk* er erstatta av styresignalet $\overline{Reg\_Skriv}$.
Å endra innhaldet i eit register er ikkje noko som skjer med ein fast frekvens, men når det er **behov** for dette. Dette blir ofte bestemt av instruksjonane i eit program, men registerinnhald kan også endrast av maskinvare.
I eit program kan ein f.eks. endra bitane i eit utgangsregister til ein GPIO-modul, mens

nivåendringar på inngangane vil endra innhaldet i inngangsregisteret automatisk.

Av figuren ser ein at registeret er positivt trigga, dvs. at ei endring av innhaldet er **synkronisert** med positiv flanke av styresignalet, $\overline{Reg\_Skriv}$.

Ein må også leggja merke til korleis endringar av data med breidde $n > 1$ bit, dvs. databussverdiar, blir markerte i tidsdiagrammet. Her kan ein ikkje visa nivåa for dei ulike linjene kvar for seg, men markerer verdiane som tjukke "pølser" og med endringar viste som loddrette streker[116].

Ved skriving er det altså viktig å halda minstekrava til **oppsettings**- og **haldetid**. Når det gjeld lesing, er den såkalla **aksesstida** t_{access} viktig. For registeret i figuren er dette er tida frå styresignalet $\overline{Reg\_Les}$ går lågt og til data kjem ut på databussen, dvs. tida t_{p2}. Når ein f.eks. skal kjøpa tilleggsminne til ein datamaskin, er det viktig at aksesstida for minnet er kort nok for den prosessoren og den klokkefrekvensen som blir brukt i datamaskinen.

3.9.7 Skiftregister

Eit **skiftregister** er eit register der ein kan **klokka** bitane **på langs** av registeret, dvs. frå den eine D-vippa og over i neste. Denne flyttinga blir også kalla for ein skiftoperasjon. Ein prosessor har eigne instruksjonar for å skifta bitane i eit register til høgre eller venstre, og desse blir utført vhja. logikk for bitskifting. Skiftoperasjonar er f.eks. sentralt i alle perifermodular for seriell kommunikasjon.

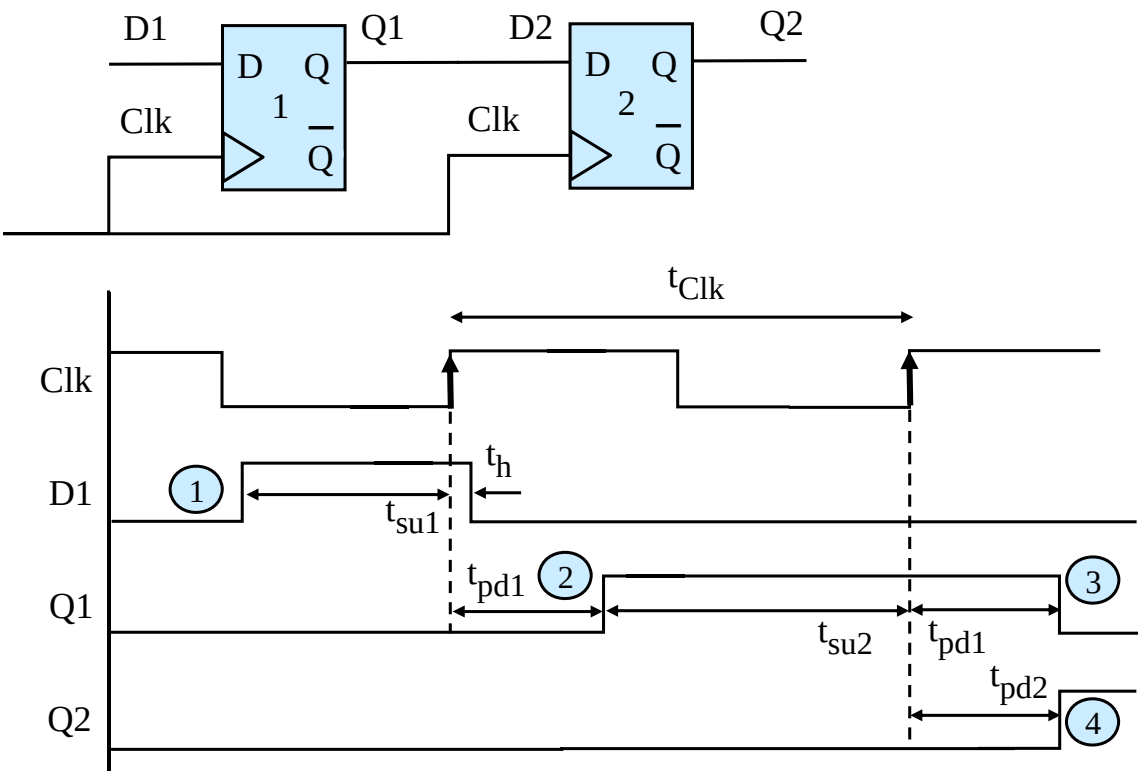
Eit skiftregister med **lengde** på to bit er vist i figur 3.75.

D-vippene har etterslepa t_{pd1} og t_{pd2} som er teikna like lange.

Eit 2-bitsskift kan delast opp i følgjande steg med stegnummer viste i tidsdiagrammet:

1. **Ny dataverdi kjem inn til første vippa:** Data må som me alt veit, **setjast opp** på inngangen ei viss tid før positiv klokkeflanke. Når *Clk* går høg, vil *D* bli klokka inn i første vippa.
2. **Ny tilstand blir lagra i første vippa:** Etter ei tid gitt av etterslepet t_{pd1} vil første vippa få ny tilstand *Q1*.
3. **Neste dataverdi gir nytt skift i tilstand for første vippa:** Som ein ser, endrar *D* verdi igjen. Denne blir klokka inn på andre positive klokkeflanke og kjem på utgangen av første vippa etter tida t_{pd1}.
4. **Ny tilstand blir lagra i andre vippa:** På andre positive flanke vil den første dataverdien bli klokka inn i den andre vippa.

[116] I datablad er ofte endringane meir forseggjorte med fine kryss.



Figur 3.75: Logisk skjema og tidsdiagram for skiftregister med lengde på to bit.

Etter ei tid gitt av etterslepet t_{pd2} vil den andre vippa få ny tilstand *Q2*.
Dataverdien er då skifta gjennom begge vippene.

Skiftregisteret kan utvidast til den breidda ein ønskjer.

Kor **raskt** ein kan køyra skiftinga, går fram av tidsdiagrammet i figuren. Ser ein på periodetida t_{Clk} i figuren, må denne denne ha rom for både etterslepet t_{pd1} gjennom første vippa og nødvendig oppsettingstid t_{su2} mot andre vippa.
Mellom to positive klokkeflankar må altså data få tid til å koma seg ut av ei vippe og få stå lenge nok mot neste vippe slik at data blir oppfatta rett.
Minimal klokkeperiode er då gitt av:

$$t_{Clk,min} = t_{pd,max} + t_{su,min}$$

der
- $t_{pd,max}$ er maksimalt etterslep gjennom ei vippe.
- $t_{su,min}$ er minimumskrav til oppsettingstid mot ei vippe.

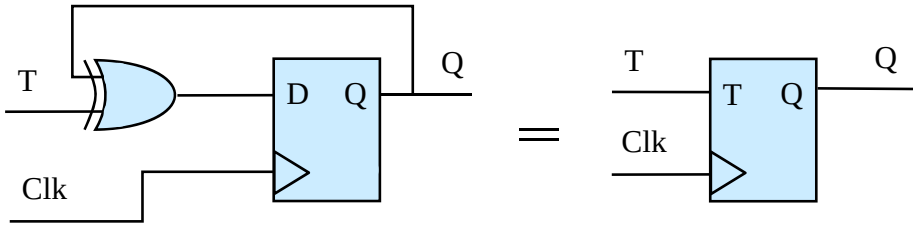
Basert på dette vil då øvre grense for klokkefrekvensen bli $f_{Clk,max} = 1/t_{Clk,min}$.

3.9.8 Teljar

Ein teljar er ei sekvensiell blokk då sjølve teljarverdien blir lagra. Teljarverdien er å sjå på som **tilstanden** til teljaren.
Viss teljaren er positivt trigga, vil han auka teljeverdien med 1 når han mottar ein positiv klokkeflanke.

Maksimal teljeverdi er gitt av kor brei teljaren er, dvs. kor mange bit teljaren har. Når teljaren har telt seg opp til maksimalverdien vil neste verdi vera null.

Ein teljar kan realiserast på fleire måtar, f.eks. vhja. **T-vipper**. Ei positivt trigga T-vippe eller ein **vendar** er vist i figur 3.76.

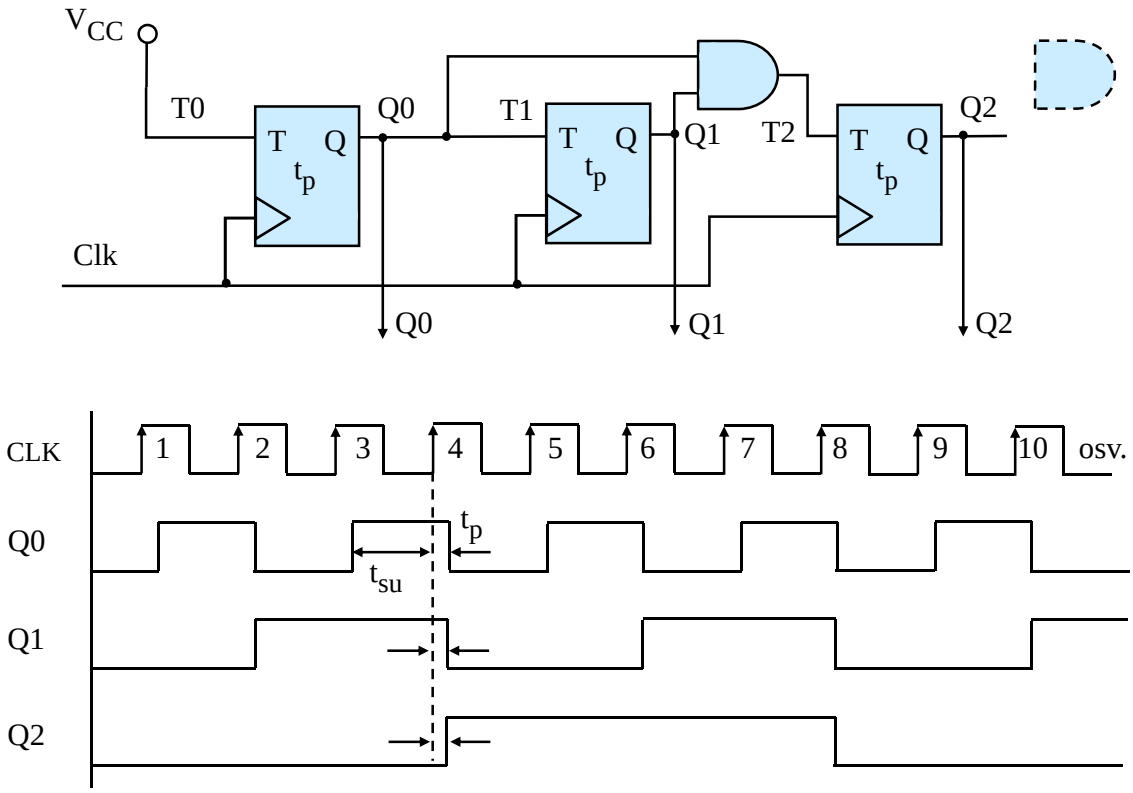


T	Q_k	D_k	Q_{k+1}
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Figur 3.76: Logisk skjema, symbol og funksjonstabell for T-vippe.

Ein ser av det logiske skjemaet og funksjonstabellen at når styresignalet T ("toggle") er null, vil neste tilstand blir lik den førre.
Når $T = 1$ vil derimot tilstanden bli motsett av den førre. På same vis som ein bitvis XELLER-operator kan brukast til å snu eller invertera bit i eit register, er ein XELLER-port sentral i denne vendaren, dvs. T-vippa.
Dette kan brukast til å laga ein teljar.

For at logikken ikkje skal bli for kompleks, er det i figur 3.75 vist oppbygging og oppførsel til ein tre-bitsteljar.



Figur 3.77: Logisk skjema og tidsdiagram for ein 3-bitsteljar.

I tillegg til dei tre vendarane eller T-vippene er det ein OG-port som bidrar til at den tredje biten ikkje reagerer på positiv klokkeflanke før både bit 0 og bit 1 er blitt 1. Skal ein utvida teljaren, blir det ein OG-port[117] og ei T-vippe for kvar ny bit som skal leggjast til.

Alle vippene reagerer på same klokkesignal og vil difor telja utan etterslep seg i mellom. Det einaste etterslepet er frå klokkeflanken kjem og til vippeutgangane endrar seg, men det er ukritisk[118] her.

Me konstaterer som nemnt tidlegare at når teljaren kjem til maksimalverdien gitt av $Q0 = Q1 = Q2 = 1$, vil neste verdi bli null. Slik held då dette fram så lenge klokkesignalet går.

Etter å ha sett på sentrale kombinatoriske og så sekvensielle funksjonsblokker skal me nå heva oss opp eit nivå i hierarkiet og sjå på systemblokker, dvs. logiske modular som er oppbygde av funksjonsblokker av ulike slag.

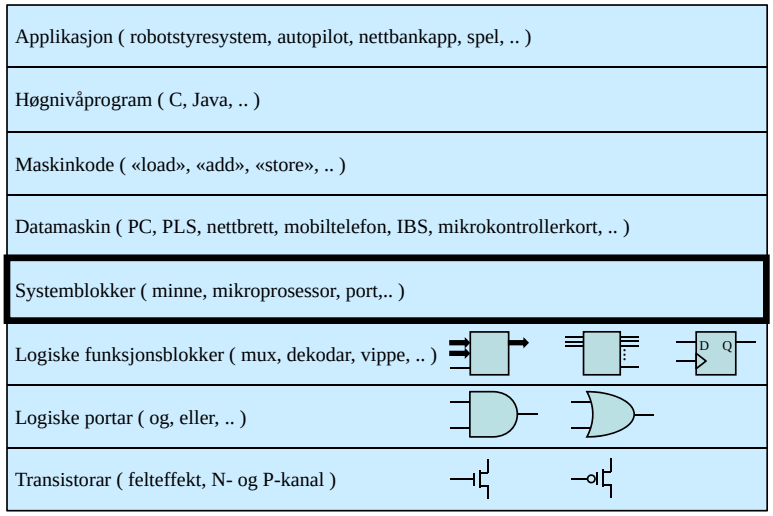
[117] OG-porten for eit evt. nytt trinn er stipla i figuren.

[118] Etterslepet må sjølvsagt ikkje vera lenger enn periodetida for klokkesignalet minus etterslepet gjennom OG-porten og nødvendig oppsettingstid for T-signalet.

3.10 Systemblokker

3.10.1 Litt om systemblokklaget i det digitale hierarkiet

Neste laget i hierarkiet er systemblokklaget, sjå figur 3.78.



Figur 3.78: Digitalt hierarki: Systemblokklaget.

Systemblokkene er bygde opp av funksjonsblokker av ulike slag.

Me skal her visa grovstrukturen til viktige systemblokker i ein datamaskin og med spesiell vekt på dei systemblokkene ein kjenner frå mikrokontrolleren *STM32F100RB*. Den mest komplekse systemblokka i ein datamaskin er mikroprosessoren.

Sjølv om detaljeringsgraden i presentasjonen her ikkje er stor, vil ein sjå klart kva funksjonsblokker dei ulike systemblokkene er bygde opp av. Dette kan vera til hjelp når ein skal sjå på grensesnittkonstruksjon i kapittel 4, samt når ein skal setja seg inn i parallell og seriell dataoverføring generelt, avbrotssystem og operativsystem med meir.
Dei systemblokkene som ein vil sjå på her, er:

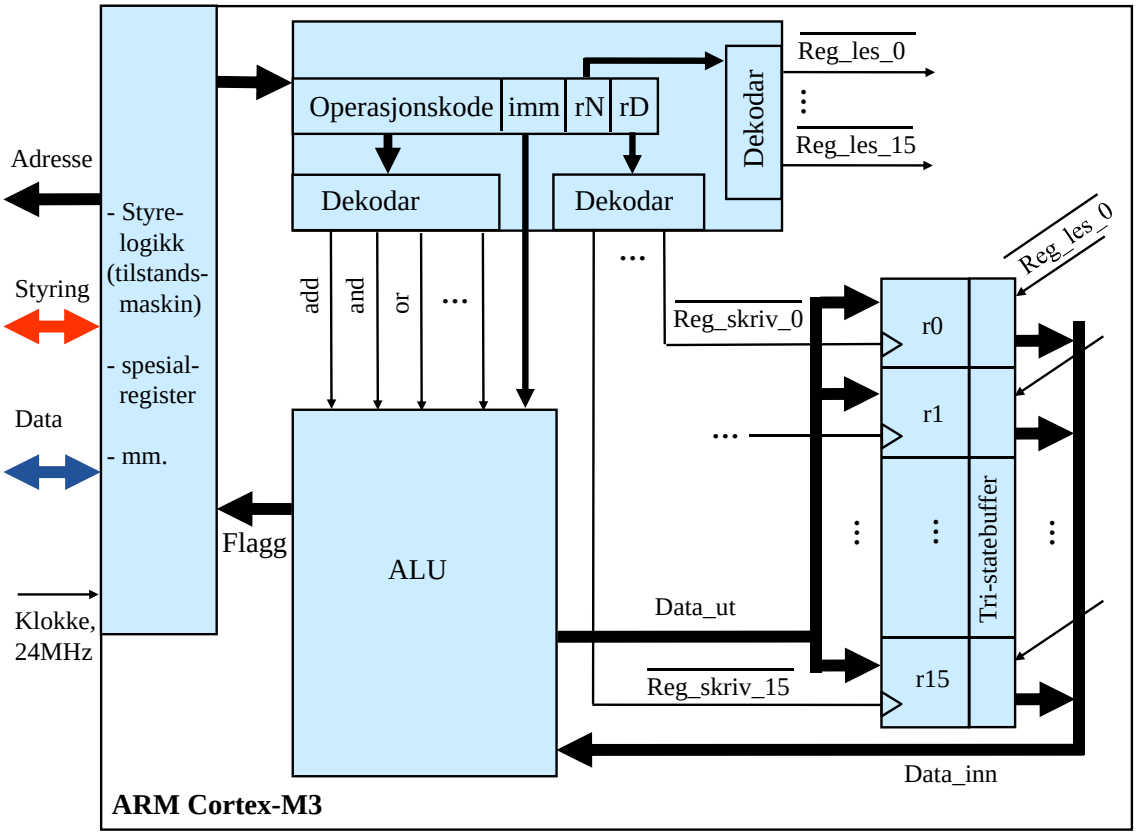
- **Prosessor:** Grovstruktur av ARM Cortex-M3 med vekt på instruksjonsdekoding.
- **Parallellport:** GPIO-modulane i mikrokontrolleren.
- **Taimer:** Systemtaimeren SysTick i ARM Cortex-M3 samt TIM-modulane, som er generelle taimerar i mikrokontrolleren.
- **Serieport:** USART- og SPI-modulane i mikrokontrolleren.
- **Minne:** Minnemodular av ulike slag.

Når det gjeld minne, vil det også bli vist minnetypar som ein ikkje finn i mikrokontrolleren vår.

3.10.2 Mikroprosessoren

3.10.2.1 Overordna struktur med vekt på dekodingslogikken for instruksjonar

Me skal først gå laus på den mest komplekse systemblokka i datamaskinen, nemleg mikroprosessoren.
Ei grov skisse av ARM Cortex-M3 med vekt på instruksjonsdekoding er vist i figur 3.79.



Figur 3.79: Grov skisse av ARM Cortex-M3 med vekt på instruksjonsdekoding.

Ein mikroprosessor er som me har sett i kapittel 2, eit komplekst sekvensielt system. Systemet har svært mange tilstandar og er klokkestyrt.

Den overordna funksjonen til mikroprosessoren er å henta instruksjonar frå programminnet, dekoda og så utføra desse. Me skal nå sjå nærare på logikken for instruksjonsdekoding.

3.10.2.2 Meir om instruksjonsdekoding

I figur 3.79 er ein ny instruksjon overført til instruksjonsdekodaren. Som vist i kapittel 2.4.4.1, vil kvar type maskinkodeinstruksjon vera bygt opp av ulike delar som blir dekoda kvar for seg.
Me skal her igjen ta eksempel i følgjande instruksjon for å illustrera virkemåten til prosessoren:

```
adds r2, r0, #1 //Altså: Adder talet 1 til innhaldet i prosessorregister r0
                //        og legg resultatet i prosessorregister r2.
```

Som vist før, kan me i kapittel A7.7.3 i arkitekturreferansemanualen for ARM Cortex-M3, [21], finna ut korleis dei 16 instruksjonsbitane er sett opp. Oppsettet er altså slik:

Bit nr.	15 9	8 6	5 3	2 0
Felt	Opkode	imm	rN	rD
Verdi	0001110	001	000	010

Figur 3.79 viser at **operasjonskoden** går inn på ein **dekodar** som aktiverer den rette logiske modulen i ALU-en. I eksemplet her vil operasjonskoden `0001110` velja ut ut **addisjonsmodulen** i ALU-en som mottakar for inndata.
Styresignalet *add* i figuren blir då aktivert mens dei andre er deaktiverte.

Talverdien *imm* går rett til ALU-en. Denne talverdien skal altså adderast direkte ("immediate") til innhaldet i det spesifiserte **kjelderegisteret** *rN*.

Dekodaren av kjelderegisteret *rN* vil aktivera lesesignalet for det spesifiserte registeret. I eksempelinstruksjonen er det sett av tre bit til dette, så i dette tilfellet skal då eitt av registra *r0*-*7*[119] levera data til ALU-en. Her viser koden at *r0* blir kjelderegister.
Korleis ein leseoperasjon mot eit register føregår, har me sett på i kapittel 3.9.6.

Summen skal plasserast i **destinasjonsregisteret** *rD*.
I eksemplet vil dekoding av 3-bitskoden for dette resultera i at skrivesignalet for register *r2* blir aktivert. Ein skriveoperasjon mot eit register blei forklart i kapittel 3.9.6.

Figuren viser at **køyring av eit program** i ein datamaskin ikkje er noko anna enn ein **klokkestyrt sekvens av logiske operasjonar**. Eksempel på slike operasjonar er dataoverføring, aritmetikk, bitvis logikk mm., og alle desse er realiserte vhja. kombinatoriske og sekvensielle logikkmodular i datamaskinen, altså rein maskinvare.

[119] Ein 32-bitsvariant av instruksjonen har eit fire-bitsfelt her. Ein kan då spesifisera eitt av registra *r0-15*.

3.10.2.3 Nokre tilleggsmerknader

Figur 3.79 har som sagt eit fokus på instruksjonsdekoding, der det viktige er å visa koblinga mellom instruksjonsoppsett, registeraksessar og ALU. Sentralt er bruken av dekodarar.
Nokre tilleggsmerknader er gitt i det følgjande:

- Instruksjonssettet til ARM Cortex-M3 har både 16-bits- og 32-bitsinstruksjonar og gjer bruk av mange ulike instruksjonsformat. For å kunne handtera dette, må instruksjonsdekodaren vera mykje meir fleksibel enn det som er vist i figuren.
- Prosessoren parallellkøyrer som kjent henting[120], dekoding og utføring ("pipelining"). Dette er ikkje vist her.
- Aksessane mot registra under f.eks. addisjonsoperasjonen over går i sekvens. Ein kan som kjent f.eks. spesifisera same register som både kjelde- og destinasjonsregister. Då må det vera slik at første steg i addisjonssekvensen er ein leseaksess mot registeret og siste steg er ein skriveaksess mot same registeret.

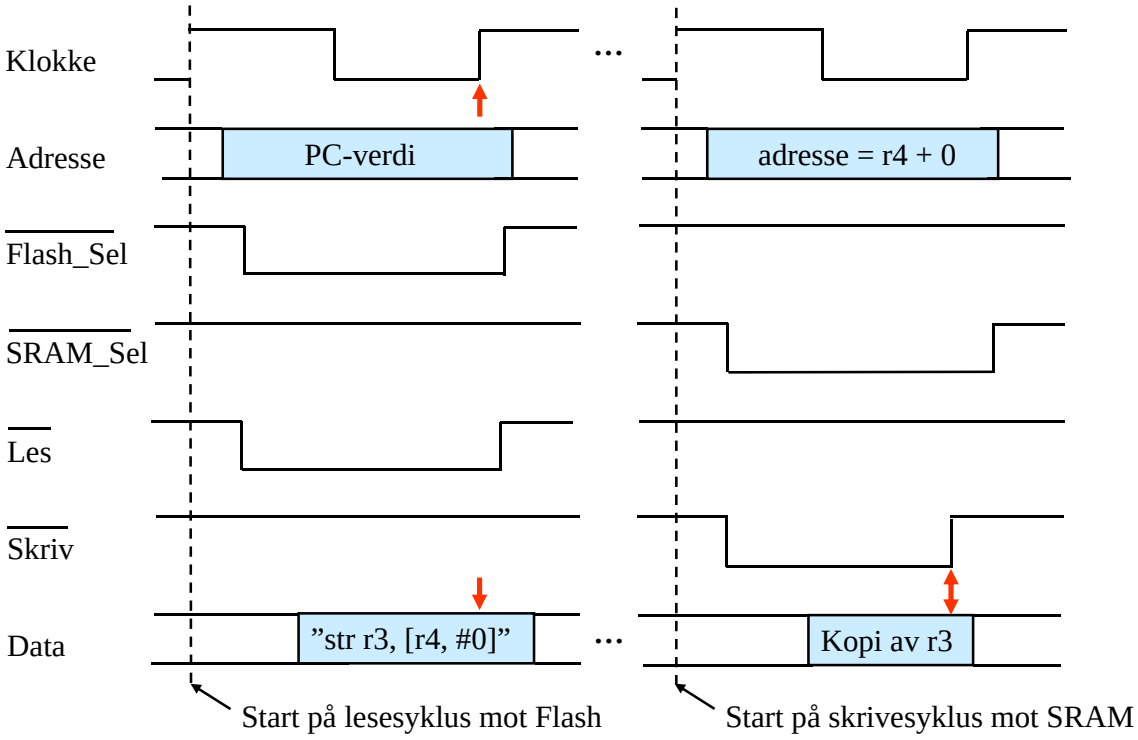
Prosessoren er ein kompleks tilstandsmaskin. Mellom anna vil logikken som styrer instruksjonshentinga, gå gjennom fleire tilstandar for å kunne realisera ei overføring mellom programminnet og hentebufferet i prosessoren.
Me skal sjå litt på slike overføringar i neste delkapittel.

[120]Det er også meir enn ein hentekø. Prosessoren driv som kjent med såkalla spekulativ henting ("speculative fetch"), sjå kapittel 2.2.4. Dette gir redusert utføringstid for hoppinstruksjonar.

3.10.3 Litt om mikroprosessoren sine lese- og skrivesyklar.

3.10.3.1 Typiske tidsdiagram basert på eit enkelt programeksempel

Henting av ein instruksjon og overføring av data kan gå føre seg som vist i figur 3.80.



Figur 3.80: Eksempel på lese- og skrivesyklar.

Utgangspunktet for figuren kan f.eks. vera følgjande høgnivåprogram:

```
  uint32_t k // Global variabel.
  ...
void main(void) {
  ...
  k++;
  ....
}
```

Ein tenkjer seg nå at
- k blir plassert i register $r3$ under inkrementeringa og
- adressa til k er plassert i register $r4$

Det siste som skjer under inkrementeringa av k, er at den nye k-verdien blir skriven tilbake

til dataminnet (SRAM) vhja. ein instruksjon som her er vist på assemblyformat:

```
str r3, [r4, #0]
```

For å kunne utføra denne skriveoperasjonen, må som kjent prosessoren først **lesa** sjølve instruksjonen frå programminnet (Flash). Dette er vist i venstre del av figur 3.80. Så blir instruksjonen utført, dvs. den nye verdien blir skriven til adressa for variabel *k*.
Instruksjonar blir overførte på databussen i form av maskinkode, dvs. som bitmønster. Kvar instruksjon har sitt eige bitmønster. Mikroprosessoren *dekodar* som vist i kapittel 3.10.2, dette bitmønsteret etter henting for å finna ut kva han skal gjera.
I dei neste underkapitla skal ein gå nærare inn på styresignala og kva som skjer i overføringssyklane.

3.10.3.2 Litt om synkronisering og styresignal

Ein mikroprosessor gjennomfører ein lese- eller skrivesyklus i løpet av eit visst antal klokkesyklar (KS), dvs. klokkeperiodar. Styresignala er **synkroniserte** med klokkesignalet.
Figur 3.80 tar utgangspunkt i prosessoren ARM Cortex-M3, som bruker 1 KS[121] på å gjennomføra ein syklus mot program- eller dataminnet. Oppsettet er forenkla[122], men viser sentrale styresignal som

$\overline{Les}$ og $\overline{Skriv}$,

samt to av styresignala som adressedekodaren[123] gir ut, nemleg

- $\overline{SRAM\_Sel}$ for aktivering av dataminnet og
- $\overline{Flash\_Sel}$ for aktivering av programminnet.

Det er som vist i kapittel 3.9.6, vanleg å teikna bussar med dobbel strek. I 3.80 gjeld dette adresse- og databussen. Denne teiknemåten signaliserer at verdien er gitt av eit bitmønster, mens enkle signal er høge eller låge.

3.10.3.3 Ein lesesyklus plukka frå kvarandre

Gangen i ein typisk **lesesyklus** som vist til venstre i figur 3.80, er følgjande:

1. μP-en set ut ein adresseverdi på adressebussen ved første klokkeflanke[124].

[121] Med typisk klokkefrekvens $f_{KLK} = 24$ MHz blir som kjent $t_{KS} \approx 41.7$ nsek.
[122] I tillegg til at ikkje alle styresignala er med, er alle flankar loddrette i tidsdiagrammet. Ein stigande eller positiv flanke vil i praksis ha ei viss stigetid og ein fallande eller negativ flanke vil ha ei falltid.
[123] Sjå kapittel 2.1. I ein mikrokontroller slik som vår, vil det truleg vera ein adressedekodar i kvar av minnemodulane og i kvar av perifermodulane.
[124] Pga. tidsforseinkingar eller etterslep ("propagation delay") i logikken kjem alltid signala litt etter den

Når adressedekodaren i minnemodulen får inn ein ny adresseverdi, og denne er innanfor adresseområdet for modulen, vil han automatisk aktivera eit internt styresignal. I figuren er det programteljarverdien som blir lagt ut, og dekodaren vil då aktivera sjølve programminnet vhja. styresignalet $\overline{Flash\_Sel}$.

2. μP-en aktiverer lesesignalet.

3. Dei to aktive styresignala og adressesignala[125] vil gjera at programminnet etter ei viss tid[126] legg ut rett verdi på databussen, i dette tilfellet **instruksjonen** som ligg på den gitte adressa.

4. μP-en **låser inn** dataverdien ("latching") på ein gitt klokkeflanke. Dette er vist med **loddrett pil** i figuren[127].

3.10.3.4 Detaljar i ein skrivesyklus

Gangen i ein typisk **skrivesyklus** som vist til høgre i figur 3.80, er følgjande:

1. μP-en set ut ein adresseverdi på adressebussen ved første klokkeflanke.

   Adressedekodaren i minnemodulen som blir adressert, vil som i ein lesesyklus automatisk aktivera modulen vhja. eit internt styresignal.
   I figuren er det verdien *adresse = r4* som blir lagt ut, og dekodaren i dataminnet vil aktivera sjølve minnet vhja. styresignalet $\overline{SRAM\_Sel}$.
   Ein del av adressebussen vil her også gå til ein dekodar i minnemodulen for å aktivera akkurat dei minnecellene som det skal skrivast til.

2. μP-en aktiverer skrivesignalet.

3. μP-en legg ut dataverdien på bussen, her den nye verdien til variabelen k.

4. μP-en deaktiverer så skrivesignalet.

5. Dataminnet låser inn dataverdien på positiv flanke av skrivesignalet. Dette er som for lesing vist med **loddrett pil** i figuren.

Lese- og skrivesyklusane blir avslutta med at styresignala blir sett passive, og dette fører til at program- eller dataminnet blir inaktivt.
Tilsvarande har ein dataaksessar, dvs. lesing og skriving, mot perifermodulane inne i ein mikrokontroller. Korleis grensesnittet mot desse er ordna, skal me sjå på i neste delkapittel.

klokkeflanken dei er synkroniserte med.

[125] Ein del av adressebussen vil gå rett til ein dekodar i minnemodulen for å aktivera dei minnecellene som data nå skal hentast frå. Viss n adressebit går til minnet, kan ein adressera 2^n minnelokasjonar. Omvendt vil f.eks. eit 1kB minne trenga 10 adressebit og eit 1MB minne 20 adressebit for å kunne adressera ein vilkårleg lokasjon.

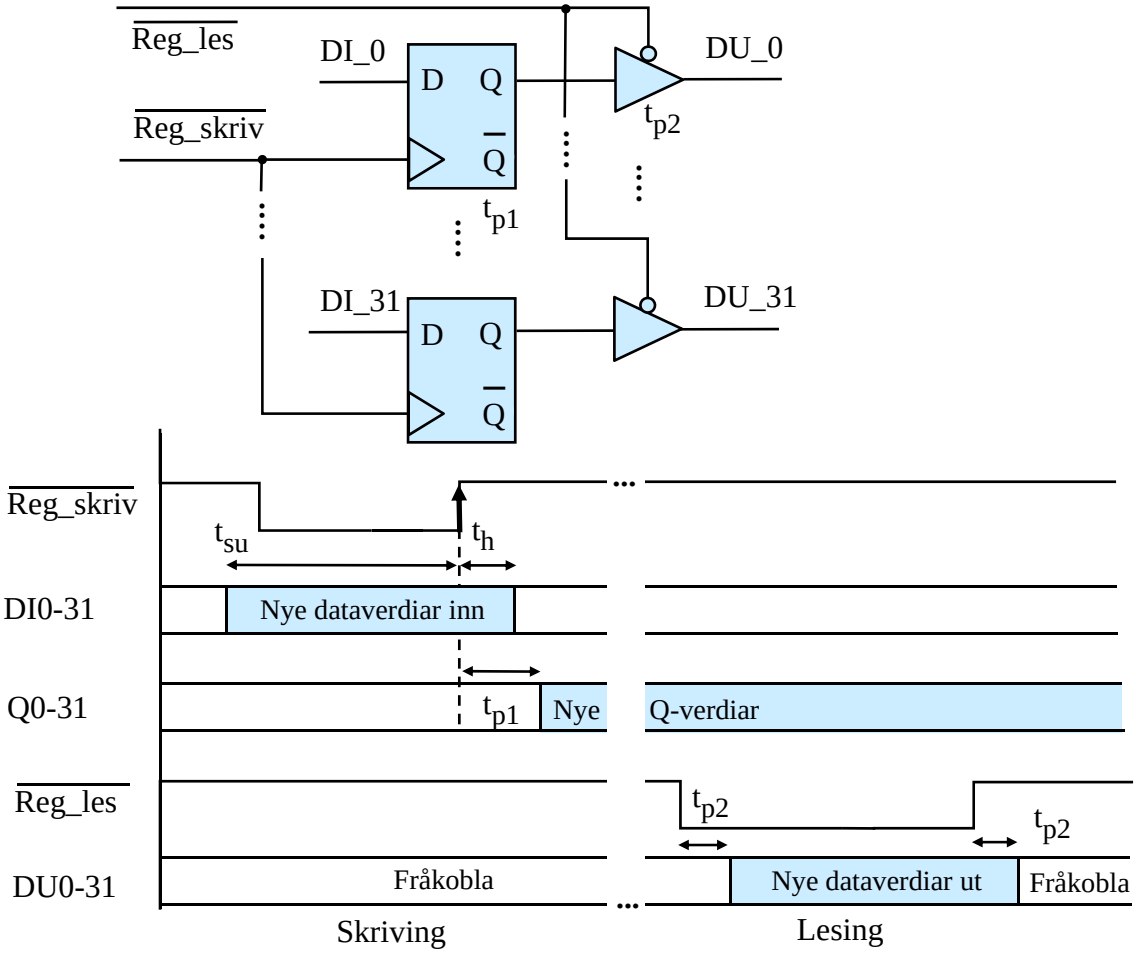
[126] Denne tida blir kalla **aksesstida**, og seier noko om kor raskt eit minne er.

[127] Det er altså viktig at aksesstida til minnet passar med klokkefrekvens og sykluslengde. Viss minnet f.eks. var for tregt, ville ikkje dataverdien frå minnet vera klar på bussen på innlåsingstidspunktet. Dette har ein sjølvsagt tenkt på ved konstruksjon av ein mikrokontroller.

3.10.4 Generelt om bussgrensesnitt for perifermodular i ein mikrokontroller

3.10.4.1 Eit gjensyn med registeraksessar

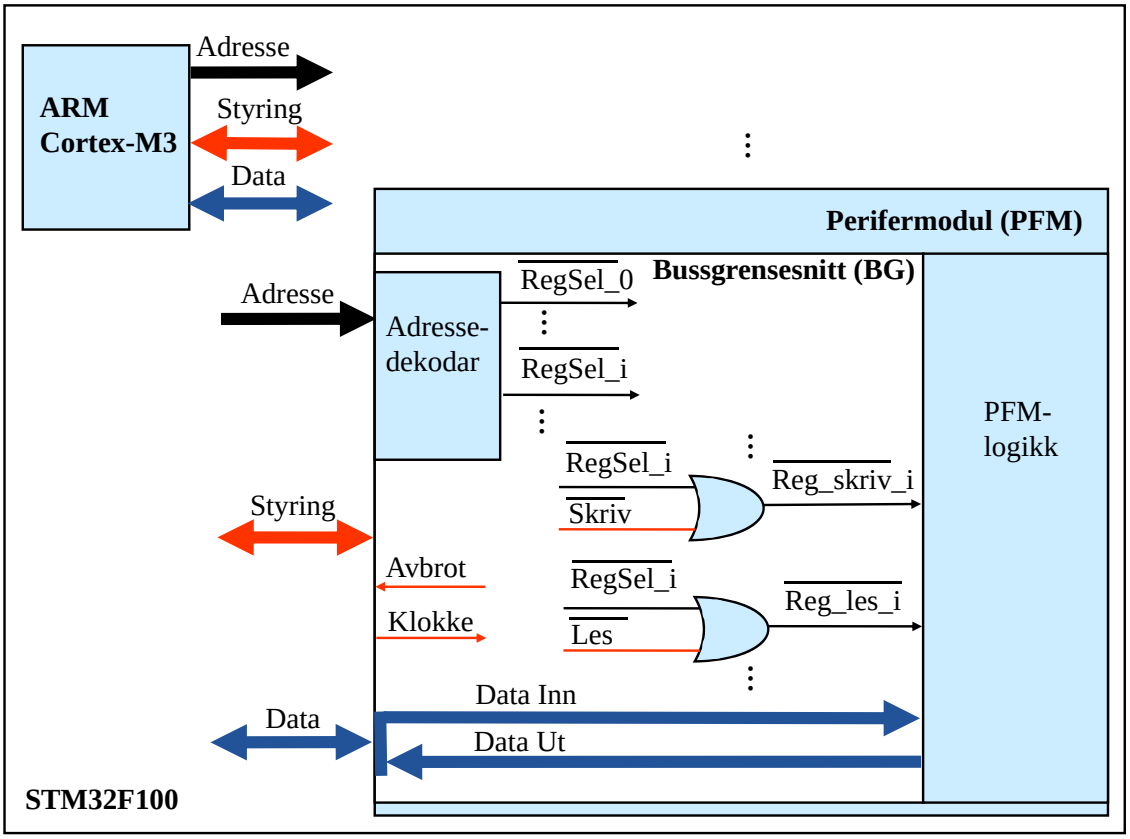
Som kjent innheld både mikroprosessorar og perifermodular kvar sine registersett. I figur 3.74 på side 215 blei det illustrert korleis aksessar, dvs. skriving og lesing, av slike register går føre seg. Dette er vist om igjen i figur 3.81.
Ein skal her mellom anna sjå på korleis styresignala $\overline{Reg\_les}$ og $\overline{Reg\_skriv}$ blir til.



Figur 3.81: Registeraksessar.

3.10.4.2 Systembussgrensesnitt med vekt på adressedekoding

Ein generell perifermodul i ein mikrokontroller vil innehalda logikk som realiserer sjølve hovudfunksjonane til modulen, dvs. taimerfunksjonar, parallell- eller seriekommunikasjon mm. I tillegg vil modulen innehalda logikk som realiserer eit standardisert grensesnitt mot mikroprosessoren sin systembuss. Signala $\overline{Les}$ og $\overline{Skriv}$ er f.eks. ein del av styrebussen. Bussgrensesnittet ("bus interface") er skissert i figur 3.82.



Figur 3.82: Bussgrensesnitt.

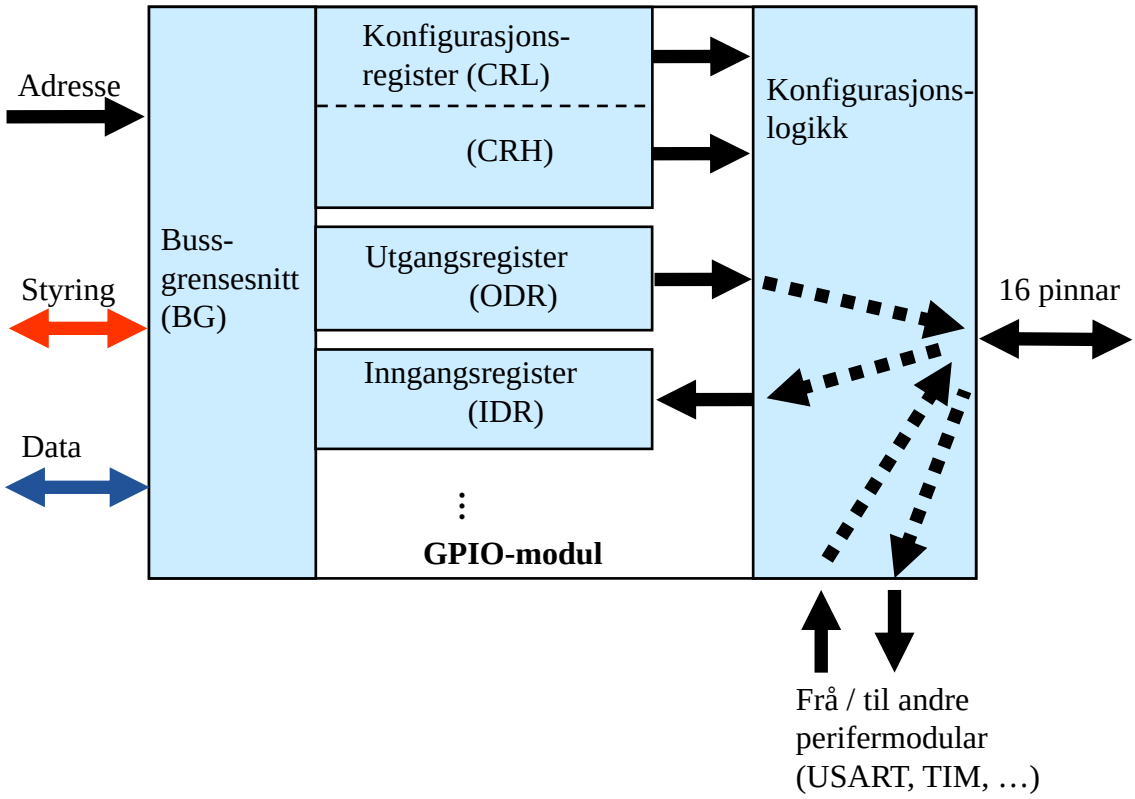
Ein sentral del av dette er **adressedekodaren**. Når denne får inn ein ny adresseverdi, og denne er innanfor adresseområdet for register nr.i i perifermodulen, vil han automatisk aktivera eit internt styresignal som her er kalla $\overline{RegSel\_i}$.

Saman med det aktuelle styresignalet for den aksessen som skal gjerast, dvs. $\overline{Les}$ eller $\overline{Skriv}$, blir dette til eit styresignal for å lesa frå eller skriva til det aktuelle registeret i perifermodulen, dvs. dette blir til eitt av styresignala
$\overline{Reg\_les\_i}$ eller $\overline{Reg\_skriv\_i}$.

3.10.5 Parallellport

3.10.5.1 Generell struktur

Strukturen til ein GPIO-modul ("General Purpose Input Output") er vist i figur 3.83.



Figur 3.83: GPIO-modul.

Konfigurasjonsblokka inneheld logikk for å kunne setja ei GPIO-linje som utgang eller inngang. Her finst det fleire typar å velja mellom.
Det vanlegaste er å styra utgangane frå utgangsregisteret ODR i GPIO-modulen som vist i eksempel 2.13 på side 64, eller å lesa inngangsverdiar inn til inngangsregisteret IDR i GPIO-modulen.
I figuren ser ein at nokre av pilene går gjennom konfigurasjonslogikken og til eller frå andre perifermodular. Slike inn- eller utgangar er av ein type som blir kalla **alternativ funksjon** (AF).
Grunnen til at **alle** inn- og utsignal går gjennom konfigurasjonslogikken til GPIO-modular, er at funksjonen til ein pinne på mikrokontrolleren skal vera mest mogleg **fleksibel**.
Konsekvensen av dette er at ein **alltid** må konfigurera aktuelle GPIO-modular sjølv om ein bare skal bruka serieportar, taimerar mm.

Ved å skriva rette bitmønster til konfigurasjonsregistra vist i figuren, vil ein velja ein viss type utgang eller inngang. Bitane i konfigurasjonsregistra styrer konfigurasjonslogikken

direkte. Inne i konfigurasjonslogikken vil det i tillegg til inngangs- og utgangsbuffer av ulike slag vera multipleksarar som vel kva signal som skal sleppast ut på eller inn frå kvar av pinnane.
Meir om dei ulike typane inn- og utgangar kjem i neste underkapittel, og så vil ein sjå på typiske konfigurasjonssekvensar.

3.10.5.2 Ulike typar inngangs- og utgangssignal

Dei ferdigdefinerte mønstra vist under, kan brukast ved val av signaltype. Desse er henta frå hjelpefila *stm32f10x_gpio.h*:

```
typedef enum  {
  GPIO_Mode_AIN    = 0x0,  // Analog inngang.
  GPIO_Mode_IN_FLOATING    // Flytande inngang, dvs. vanleg inngang
                   = 0x04, // utan opp- eller nedtrekk.
  GPIO_Mode_IPD    = 0x28, // Inngang med nedtrekksmotstand (''Pull-Down'').
  GPIO_Mode_IPU    = 0x48, // Inngang med opptrekksmotstand (''Pull-Up'').
  GPIO_Mode_Out_OD = 0x14, // Utgang, type ''Open Drain'', dvs. 1 transistor.
  GPIO_Mode_Out_PP = 0x10, // Utgang, type ''Push-Pull'' (vanleg CMOS-utgang).
  GPIO_Mode_AF_OD  = 0x1C, // Alternativ funksjon av OD-type.
  GPIO_Mode_AF_PP  = 0x18  // Alternativ funksjon av PP-type.
}GPIOMode_TypeDef;

typedef enum  {
  GPIO_Speed_10MHz = 1,
  GPIO_Speed_2MHz,
  GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;
```

Når det gjeld dei nedste definisjonane, kan ein bruka desse til å spesifisera kor rask ein utgang skal vera. Viss ein vel den raskaste utgåva, kan ein få ein del ringing[128] i signalet viss tilkoblingsleidninga eller -banen ikkje er tilpassa[129] drivaren i mikrokontrolleren. Den raskaste utgåva vil sjølvsagt også gi høgast effektforbruk.
Vanlegvis vil den tregaste utgåva gi eit fint signal og med skarpe/bratte nok flankar.

Konfigurasjonsblokka i figur 3.83 kan ein finna meir detaljert presentert i figur 11 og 12 på s. 102 i referansemanualen for mikrokontrolleren vår, [17].
Ein kan også lesa om konfigurasjon i kapittel 4 hjå Brown, [2].

[128] Sjå "en.wiki Ringing (signal)".
[129] Store banelengdar og høg impedans gir generelt mest ringing.

Typiske konfigurasjonssekvensar kan sjå ut som vist i dei neste underkapitla. Alle sekvensane er laga med basis i plattformkortet *STM32VLDiscovery*.

3.10.5.3 Konfigurasjon av to utgangar som styrer lysdiodar

```
//GPIOC
//-----------------------------------------
  //Deklarasjon av initialiseringsstrukturen.
    GPIO_InitTypeDef GPIO_InitStructure;

  //Slepp først til klokka paa GPIO-modul C.
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  //Konfigurer GPIO_pinnane PC8 og 9 som er kobla til LED-ar på kortet.
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOC, &GPIO_InitStructure);
```

3.10.5.4 Konfigurasjon av ein inngang som er kobla til ein brytar

```
//GPIOA
//-----------------------------------------
  //Deklarasjon av initialiseringsstrukturen.
    GPIO_InitTypeDef GPIO_InitStructure;

  //Slepp først til klokka på GPIOA-modulen.
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  //Konfigurer GPIO-pinne PA0 som er kobla til USER-brytaren.
    GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOA, &GPIO_InitStructure);
```

3.10.5.5 Konfigurasjon av to pinnar med alternativ funksjon, nemleg som sende- og mottakssignal for USART1-modulen

```
//GPIO-pinnane PA9 og 10 brukt mot intern USART1-modul
//----------------------------------------
  //Deklarasjon av initialiseringsstrukturen.
    GPIO_InitTypeDef GPIO_InitStructure_UART1;

  //Slepp først til klokka på GPIOA-modulen
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  //Sett USART1 Tx (PA9) som alternativ funksjon og ''push-pull''.
    GPIO_InitStructure_UART1.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure_UART1.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure_UART1.GPIO_Speed = GPIO_Speed_2MHz;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOA, &GPIO_InitStructure_UART1);

  //Sett USART1 Rx (PA10) som flytande inngang (''input floating'')
    GPIO_InitStructure_UART1.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure_UART1.GPIO_Mode = GPIO_Mode_IN_FLOATING;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOA, &GPIO_InitStructure_UART1);
```

3.10.6 Taimer

3.10.6.1 Litt om ulike typar taimerar i mikrokontrolleren

Det er ulike typar taimerar i mikrokontrolleren vår:

- **Systemtaimeren SysTick**
  Alle ARM-baserte mikrokontrollerar har ein eigen systemtaimer. Det spesielle med denne er at han er ein del av sjølve prosessorkjernen, i tilfellet vårt
  ARM Cortex-M3.
  Som me såg i kapittel 2.4.6.2 på side 103, så blir denne brukt til å generera periodiske avbrot slik at ein kan svitsja mellom ulike aktivitetar i eit system. Med denne eine funksjonen blir dette ein enkel taimer.

- **Generelle taimerar**
  Det er mange av desse såkalla TIM-modulane i mikrokontrolleren. Desse har mange ulike funksjonar. I tillegg til å generera periodiske avbrot, kan dei setjast opp til å generera ulike typar signal eller reagera på ulike typar inngangssignal.

- **Vakthund**
  Denne blir i referansemanualen, [17], kalla "Window watchdog" (WWDG).
  WWDG-modulen blir brukt til å sikra systemet mot feil som kan skje under programkøyring.
  I enklaste tilfellet lar ein programmet skriva regelmessig til eit register i modulen. Viss programmet feilar slik at denne skrivinga ikkje skjer, vil vakthunden resetja prosessoren.

- **Sanntidsklokke**
  Denne blir kalla "Real-time clock" (RTC), og er eit teljeverk som kan gi tid i sekund, minutt, timar osv. I tillegg har modulen alarmfunksjonar mm.
  RTC-modulen blir driven av ei klokke med frekvens 32.768 Hz. Ein teljar på 15 bit driven av denne klokka vil då ha ei omløpstid på eksakt 1 sek, noko som blir utnytta i teljeverket.

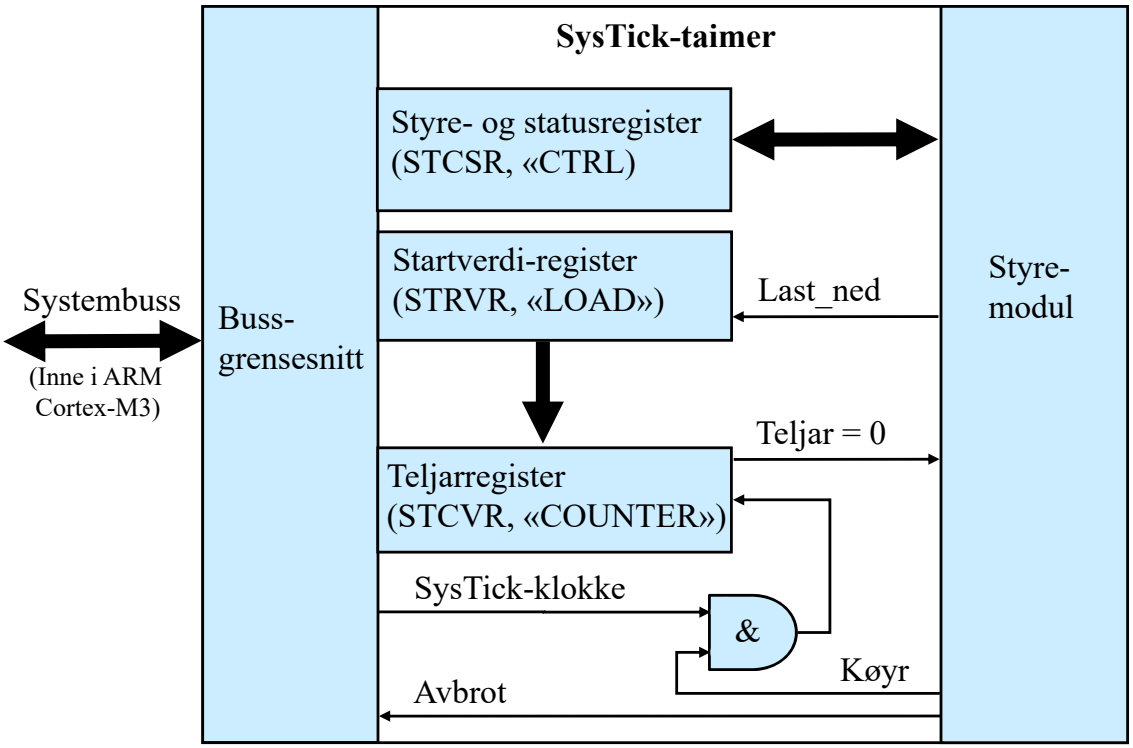
Ein ser alle desse modulane i blokkskjemaet på side 8 i brukarmanualen på plattformkortet vårt, [15]. Klokkesignalet til RTC-modulen er også markert.

Me skal nå gå djupare inn i struktur og konfigurasjon av taimerar og startar då med den enklaste, nemleg SysTick.
Det blir ikkje gått nærare inn på vakthunden og sanntidsklokka her, men ein kan finna detaljane på desse i referansemanualen, [17].

3.10.6.2 Systemtaimeren

Me skal her igjen sjå på systemtaimeren SysTick[130], som altså ein del av sjølve prosessorkjernen i mikrokontrolleren vår, ARM Cortex-M3.

Strukturen til systemtaimeren SysTick
Det er altså SysTick-modulen som er sjølve systemtaimeren i ARM-baserte mikrokontrollerar, og denne er som nemnt plassert inne i prosessorkjernen. Strukturen er vist i figur 3.84.



Figur 3.84: SysTick-taimeren.

Som vist tidlegare, er taimermodulen bygt opp av eit startverdiregister, ein teljar og litt styrelogikk. I tillegg vil det som kjent vera eit bussgrensesnitt. Ein taimer er altså også å sjå på som ei systemblokk inne i prosessorkjernen.

[130]SysTick-taimeren blei brukt i eit eksempel i kapittel 2.4.2.3.

Konfigurasjon av SysTick
Ein typisk konfigurasjonssekvens for SysTick-taimeren kan sjå ut som i det følgjande:

```
//Oppsett av SysTick-registra
  SysTick->CTRL = 0;  // Stopp teljaren
  SysTick->LOAD = 2400; // Legg inn startverdi, som her gir 100usek intervall.
  SysTick->VAL = 0;  // Nullstill teljaren
  SysTick->CTRL =
      (SysTick_CTRL_ENABLE | SysTick_CTRL_TICKINT | SysTick_CTRL_CLKSOURCE);
                // (0x7), Start teljaren, opna avbrotet og bruk intern klokke.
```

Når ein skal setja opp SysTick-taimeren, må ein altså gjera følgjande:

1. Stansa taimeren ved å skriva verdi 0 til kontrollregisteret ("SysTick Control Status Register", STCSR).
2. Leggja inn rett verdi i startverdiregisteret ("SysTick Reload Value Register", STRVR).
3. Nullstilla sjølve teljarregisteret ("SysTick Counter Value Register", STCVR).
4. Starta opp taimeren med rett spesifikasjon av avbrotsmodus og klokkekjelde.

Litt om SysTick-intervall og -avbrot
Når ein køyrer i gong taimeren, vil startverdien automatisk lastast inn først sidan teljarverdien er null.
Som vist i figuren, vil klokkesignalet bare driva teljaren når *køyr*-signalet er aktivert, dvs. når mønsteret/koden *SysTick_CTRL_ENABLE* er lagt inn i kontrollregisteret.

Ved i tillegg å setja opp taimeren til å gi avbrot, vil startverdien ein har sett opp, bestemma avbrotsintervallet T_{Int}. Samanhengen er slik:

T_{Int} = *LOAD*-verdi / f_{clk} der f_{clk} er klokkefrekvensen.

Avbrotssignalet ("timer interrupt") blir i ARM-baserte system brukt til å realisera konstante sampleintervall eller faste tidsintevall for veksling mellom ulike trådar i eit fleirtrådssystem med meir. Kapittel 2.4.6.2 handla nettopp om dette.
Sjølve avbrotshandteringa i mikrokontrollerar baserte på ARM Cortex-M3 blei tatt opp i kapittel 2.4.4.11 - 2.4.6.

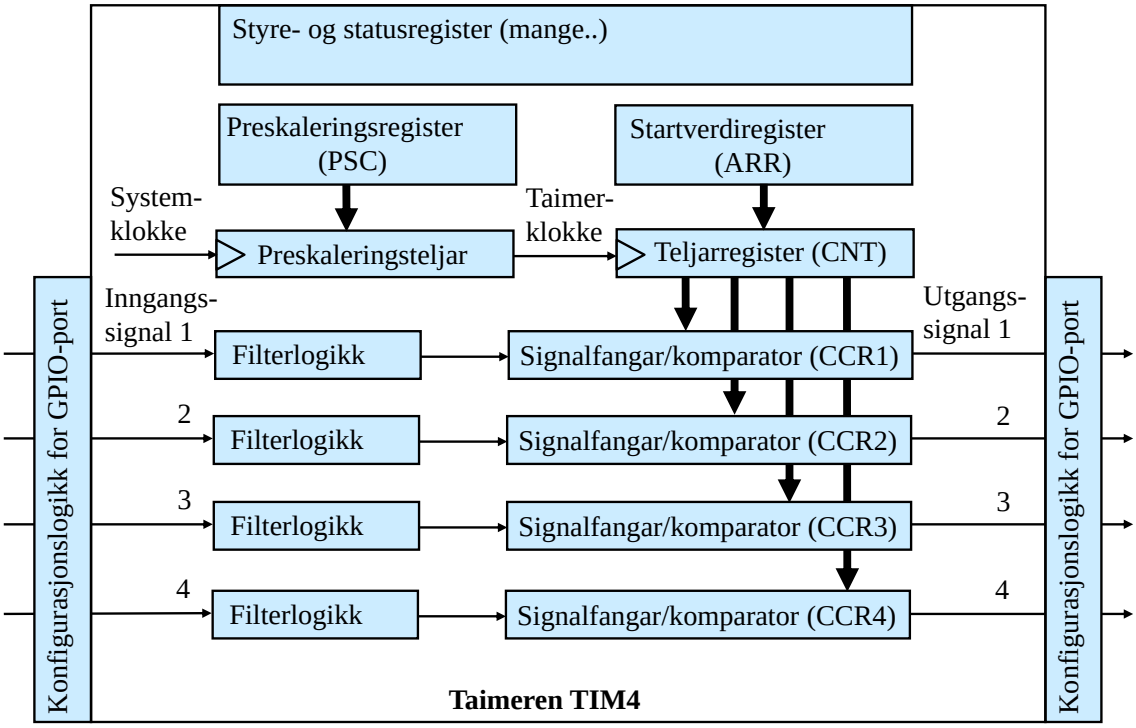
SysTick-taimeren er nærare presentert i kapittel B.3.3 i arkitekturreferansemanualen for ARM Cortex-M3, [21].

3.10.6.3 Generell taimer

Me skal her sjå på taimerar av typen TIM.

Strukturen til ein generell taimer av typen TIM
I STM32F-familien av mikrokontrollerar er det opptil 20 generelle (”general purpose”) taimerar, men ikkje alle desse er med i kvar familiemedlem.
Mikrokontrolleren vår har modulane TIM1-7 og TIM12-17[131].
Det er litt skilnad på funksjonalitet og talet på inn- og utgangar. Me skal her sjå på ei vanleg utgåve, nemleg perifermodulen TIM4[132].

Strukturen til TIM4-modulen er vist i figur 3.85.



Figur 3.85: Den generelle taimeren TIM4.

Taimeren har mange **status- og styreregister**[133], som indikert øvst i figuren.

Preskaleringsdelen er eigentleg ein enkel taimer som består av eit preskaleringsregister og ein preskaleringsteljar. Teljaren blir driven av systemklokkefrekvensen, og preskaleringsverdien avgjer kva systemklokkefrekvensen skal delast med. Ein kan visa at taimer-

[131] Desse modulane er viste figur 6 i brukarmanualen på plattformkortet vårt, [15].
[132] TIM2-5 er heilt like.
[133] Omgrepet styreregister inkluderer her både konfigurasjons- og dataregister.

klokkefrekvensen som går inn på sjølve teljarregisteret *CNT*, blir følgjande:

$$f_{TIMClk} = \frac{f_{SysClk}}{Preskalering + 1} \tag{3.35}$$

Viss ein då som med SysTick-taimeren legg inn ein startverdi inn i startverdiregisteret *ARR* ("Automatic Reload Register"), vil ein få ei omløps- eller **periodetid** gitt av:

$$t_{Periode} = startverdi \cdot t_{TIMClk} = \frac{startverdi}{f_{TIMClk}} \tag{3.36}$$

Denne taimeren har som vist fire **kanalar**, *CH1-4*. Funksjonaliteten til desse kan skiljast i to **hovudgrupper**.

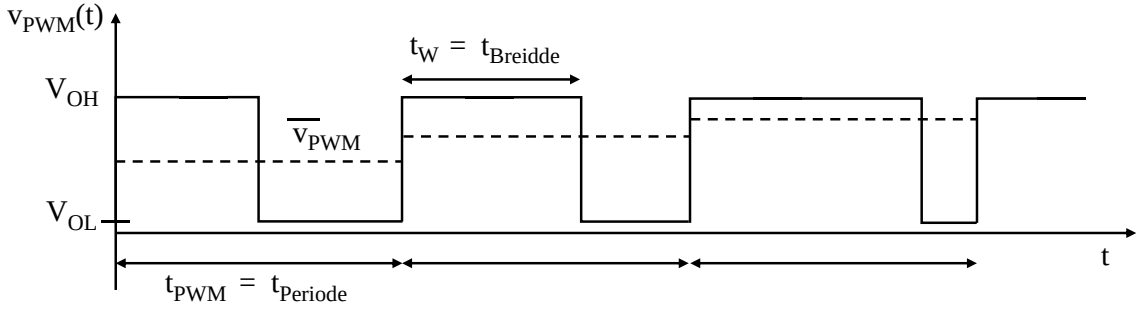
- **Fangst ("capture") av inngangssignal**
  Dette kan f.eks. brukast til tidsmåling. Ein kan setja opp taimeren slik at teljarverdien blir lagra i signalfangarregisteret *CCR* ("Capture/Compare Register") når det kjem ein flanke på inngangssignalet. Viss programmet les teljarverdien for kvar flanke, kan ein måla differansen mellom flankane i signalet.
  Ein kan også spesifisera at kvar flanke skal gi eit avbrot, og kan då lesa teljeverdien når ein er inne i avbrotsmetoden (*handler*).
- **Generering av utgangssignal**
  Ein kan f.eks. setja opp taimeren til å gi ut eit høgt nivå på ein kanal når teljaren *CNT* startar på ein ny periode. Ein kan så i tillegg leggja inn ein verdi i komparatorregisteret *CCR* for kanalen og setja opp taimeren slik at utgangen går låg når teljarverdien i *CNT* blir lik komparatorverdien i *CCR*.
  Slik kan ein styra kor brei ein perodisk puls skal vera. Dette er prinsippet bak pulsbreiddemodulasjon, sjå neste underkapittel.

Her har ein bare vist to eksempel frå det store spekteret av operasjonsmodi eller funksjonar i dei to hovudgruppene. Referansemanualen gir detaljane på alt dette.

I begge endane av kanalane ser ein at signala går gjennom konfigurasjonslogikken til ein GPIO-modul[134]. Av tabell 4 - 5 i brukarmanualen på plattformkortet, [15], ser ein at det er GPIOB som er aktuell GPIO-modul for dei fire kanalane til TIM4 som er vist i figur 3.85.

[134]Dette blei som kjent tatt opp i kapittel 3.10.5.1.

Brukseksempel for ein generell taimer: Pulsbreiddemodulasjon
Som nemnt i førre underkapittel, så kan ein bruka ein taimer til f.eks. å generera eit pulsbreiddemodulert signal ("Pulse Width Modulation", PWM). Her styrer ein perioden, dvs. avstanden mellom den positive flanken til pulsane i signalet, i tillegg til å styra pulsbreidda. Eit typisk PWM-signal er vist i figur 3.86.



Figur 3.86: Pulsbreiddemodulert utgangssignal.

Vanlegvis held ein perioden konstant og styrer bare breidda som vist i figuren. Viss me nå ser på **gjennomsnittsspenninga** for ein periode, $\overline{v}_{PWM}$, er denne gitt av:

$$\overline{v}_{PWM} = \frac{t_{Breidde}}{t_{Periode}}(V_{OH} - V_{OL}) + V_{OL}$$

PWM er nemleg ein veldig vanleg metode for **digital/analog-omforming** ("D/A conversion"). Ein gjer altså her ein digital breiddeverdi i området

$$0 \leq Breidde \leq Periode$$

om til ein analog spenningsverdi i området

$$V_{OL} \leq \overline{v}_{PWM} \leq V_{OH}$$

Her er $Breidde$ den verdien ein legg inn i komparatorregisteret CCR i figur 3.85,
og $Periode$ den verdien ein legg inn i startverdi- eller perioderegisteret ARR i same figuren.
Tidsparameteren $t_{Breidde}$ kan reknast ut på same måte som $t_{Periode}$ i likning 3.36.

For å gjera denne omforminga med god nok presisjon, er det viktig at V_{OL} og V_{OH} er tilstrekkjeleg støyfrie og stabile. Dette blir ofte realisert vhja. ein eigen rask drivarkrets med eiga og stabilisert spenningsforsyning der spenningsområdet ut er $0 \leq v_{ut} \leq V_{Ref}$.
V_{Ref} er den såkalla referansespenninga eller maksimalspenninga for utgangen.

Dette blir brukt til styring av ulike komponentar, f.eks. varmeelement[135], motorar, svitsja kraftforsyningar[136] mm., eller til å produsera lydsignal[137].

For at styringa av ein komponent ikkje skal opplevast som opphakka[138], er det viktig

[135] F.eks. ein induksjonskomfyr.
[136] Litt om slike såkalla DC/DC-omformarar står i eksempel 3.5 på side 128.
[137] PWM-baserte audioforsterkarar er ein eigen klasse av forsterkarar, nemleg klasse D, sjå Sjå "en.wiki Class-D amplifier". Desse er svært effektive og dominerer i dag som forsterkartype i mobiltelefonar og USB-høgtalarar for å ta nokre få eksempel.
[138] Som ein ser av figuren, endrar signalet seg sprangvis.

at periodetida for PWM-signalet er mykje kortare enn respons- eller reaksjonstida til komponenten som skal styrast.
Då vil styringa opplevast som glatt.

Typiske periodetider med tilhøyrande PWM-frekvens er vist under.

Komponent	t_{PWM}	f_{PWM}
Varmeelement	$>$ 1 sek	$<$ 1 Hz
Motorar	10 - 100 μsek	10 - 100 kHz
Svitsja kraftforsyningar	1 - 10 μsek	100 - 1000 kHz
Forsterkarar av klasse D	$<$ 10 μsek	$>$ 100 kHz

Konfigurasjon av ein TIM-modul for PWM-køyring
Ein kan dela opp konfigureringa av ein perifermodul av typen TIM som følgjer:

- Konfigurasjon av sjølve TIM-modulen, dvs. oppsett av reine taimerparametrar.
- Konfigurasjon av dei GPIO-pinnane som taimeren er kobla til.
- Oppstart av taimeren.

Her skal ein konfigurera taimeren TIM4 for PWM-køyring.

1. Oppsett av taimerparametrar
Ein vil her at periodetida skal vera $t_{Periode} = 10$ μsek. som gir ein PWM-frekvens på 100 kHz. I tillegg skal ein starta opp med ei PÅ-tid, dvs. pulsbreidde, lik 25% av perioden. Oppsettet av dette er vist under.

```
// Her er det bare TIM4 CH4 som skal initialiserast, og denne går ut på PB9,
// sjå tabell 5 i brukarmanualen.

//Først konfigurasjon av sjølve taimeren
//---------------------------------------------------------------------
  //Deklarasjon av taimerstrukturar.
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

  //Slepp først til klokka paa TIM4.
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

  // Her bare kanal CH4: PWM-signal med periode lik 10usek og på-tid 25%
  // Ein skal ha full oppløysing, dvs. ingen preskalering.
  // 1. Preskalering blir rekna ut slik:
  //  Preskalering = (f_SysCLK / f_TIM4) - 1 = (24MHz/24MHz)-1 = 0 her.
  // 2. Periodeverdien ein skal ha her, blir rekna ut slik:
  //  PWM_periode = t_periode/t_SysCLK = f_SysCLK/f_PWM = 24MHz/100kHz = 240.
  // 3. Breiddeverdien er gitt av på-tida (duty-cycle):
  //  PWM_breidde_4 = PWM_periode * duty_cycle/100 = 240 * 0.25 = 60 her.
  // Variablane her blir initialiserte i ei eiga deklarasjonsfil.

  //Konfigurasjon av preskalering og taimerperiode.
    TIM_TimeBaseStructure.TIM_Period = PWM_periode;
    TIM_TimeBaseStructure.TIM_Prescaler = Preskalering;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

  //Last ned konfigurasjonen i modulen.
    TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);

  //Konfigurasjon av modus (PWM1) og breidde for kanal: Ch4
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = PWM_breidde_4;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;

  //Last ned konfigurasjonen i modulen.
    TIM_OC4Init(TIM4, &TIM_OCInitStructure);
    TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable);
```

2. Oppsett av GPIO-pinnar og oppstart av taimeren
Dei siste stega i initialiseringa er vist under.

```
//Så oppsett av GPIO-pinnen PB9 som blir brukt av TIM4-modulen sin CH4,
//sjå tabell 5 i brukarmanualen.
//-------------------------------------------------------------------------
  //Deklarasjon av initialiseringsstrukturen.
    GPIO_InitTypeDef GPIO_InitStructure_TIM4;

  //Slepp først til klokka paa GPIOB, som taimerutgangen går gjennom
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

  //GPIOB-oppsett:TIM4 kanal 4 som alternativ funksjon ''push-pull''.
    GPIO_InitStructure_TIM4.GPIO_Pin = GPIO_Pin_9; //Tim4, kanal4 ut på PB9.
    GPIO_InitStructure_TIM4.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure_TIM4.GPIO_Speed = GPIO_Speed_2MHz;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOB, &GPIO_InitStructure_TIM4);

//Start til slutt TIM4
//-------------------------------------------------------------------------
    TIM_Cmd(TIM4, ENABLE);
```

3. Eksempel på typisk metode for å styra pulsbreidda

```
//Metode for å styra pulsbreidda når perioden er fast:

void PWM_sett_breidde_TIM4_k4(uint16_t breidde)  {

  //Konfigurasjon av modus (PWM1) og breidde for kanal: CH4
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = breidde;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;

  //Last ned konfigurasjonen i modulen
    TIM_OC4Init(TIM4, &TIM_OCInitStructure);
    TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable);
}
```

3.10.7 Serieport

3.10.7.1 Litt om seriell kontra parallell kommunikasjon

Skal ein overføra data mellom to system, kan dette gjerast **parallelt** eller **serielt**. Ved parallelle overføringar blir databitar overført parallelt eller samtidig, mens ved seriell overføring blir databitane overført sekvensielt, dvs. etter kvarandre.

Parallelle overføringar har følgjande eigenskapar:

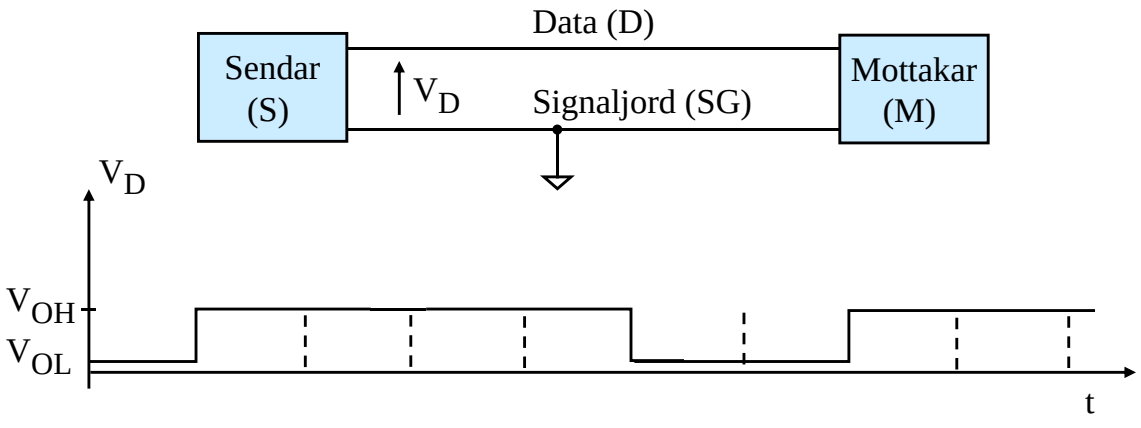
- Kan vera veldig raske, >10 GBytar/sek.
- Krev mange signallinjer og er derfor tungvint over typisk >1 meter.
- Kretsar og kretskort med parallelle grensesnitt blir store.

Serielle overføringar har følgjande eigenskapar:

- Kan også vera raske, >1 Gbit/sek.
- Krev få signallinjer og kan derfor lett realiserast over store avstandar.
- Kretsar og kretskort med serielle grensesnitt kan gjerast små, dvs. miniatyriserast.
- Serielle overføringar kan realiserast trådlaust og i optisk fiber.

3.10.7.2 Hovudutfordring ved seriell kommunikasjon

Eit enkelt oppsett for seriell dataoverføring mellom ein sendar (S) og ein mottakar (M) er vist i figur 3.87.



Figur 3.87: Seriell dataoverføring.

Ein tenkjer seg at signalspenninga V_D, dvs. spenninga på datalinja i forhold til signaljord, varierer med tida som vist nedst i figuren.

Ei **hovudutfordring** for mottakaren er å finna ut kor ein bit sluttar og kor neste startar, dvs. kor **bitgrensene**[139] er. Spesielt vanskeleg er dette når mange etterfølgjande bit har same verdi.

Ein har to hovudmetodar for seriell kommunikasjon, nemleg **synkron** og **asynkron** kommunikasjon. Desse løyser hovudutfordringa på ulike måtar.

3.10.7.3 Generelt om synkron kommunikasjon

Som namnet synkron tilseier, er eit klokkesignal involvert i dataoverføringa, nemleg eit klokkesignal generert av sendaren. Her er det igjen to hovudmetodar, nemleg:

- Kommunikasjon der klokkesignalet er ei ekstra signallinje.
- Kommunikasjon der klokkeinformasjonen er koda inn i datasignalet, dvs. i bitstraumen.

3.10.7.4 Synkron kommunikasjon der klokkesignalet er ei ekstra signallinje

Metodeeksempel: SPI
Eit eksempel på slik kommunikasjon er metoden "Serial Peripheral Interface", SPI. Dette er ein veldig vanleg metode som er realisert vhja. eigne perifermodular i dei fleste mikrokontrollerar. Svært mange aktuelle eksterne modular som ulike typar sensorar, aktuatorar og minne mm. kjem også med SPI-grensesnitt.

Eit typisk SPI-oppsett for mikrokontrolleren vår er vist i figur 3.88.

SPI-modulen i mikrokontrolleren er vanlegvis master og styrer kommunikasjonen. Dei ulike slavane kan aktiverast vhja. av separate signal. Sjølve SPI-modulen har følgjande signal:

***SCLK*, "Serial Clock":** Klokkesignal som databitstraumen er synkronisert med.

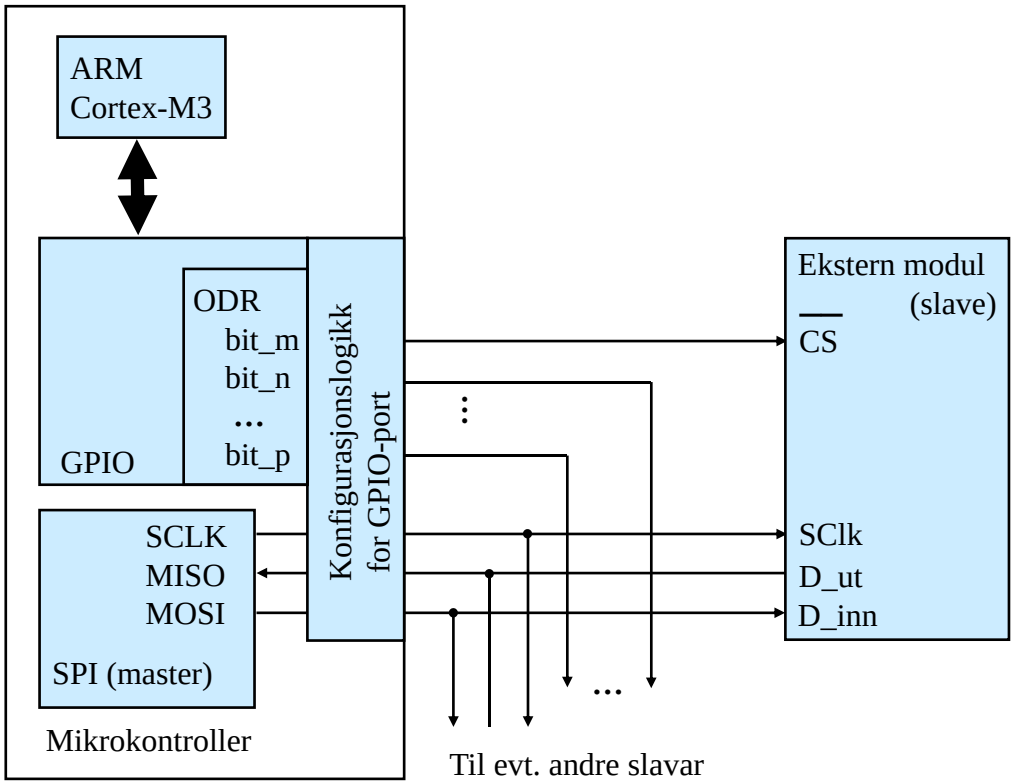
***MOSI*, "Master Out Slave In":** Serielle data ut frå master.

***MISO*, "Master In Slave Out":** Serielle data inn til master.

***SS*, "Slave Select":** Aktiveringssignal for ein slave.

Viss det bare er ein slave, kan ein kobla det aktivt låge *SS*-signalet til $\overline{CS}$-inngangen[140] på slaven.
Eit vanleg **alternativ** til dette er som vist i figuren, å bruka vanlege GPIO-utgangar. Ein

[139] Dei stipla bitgrensene i figuren er sjølvsagt ikkje synlege i eit slikt spenningssignal.
[140] "Chip Select".

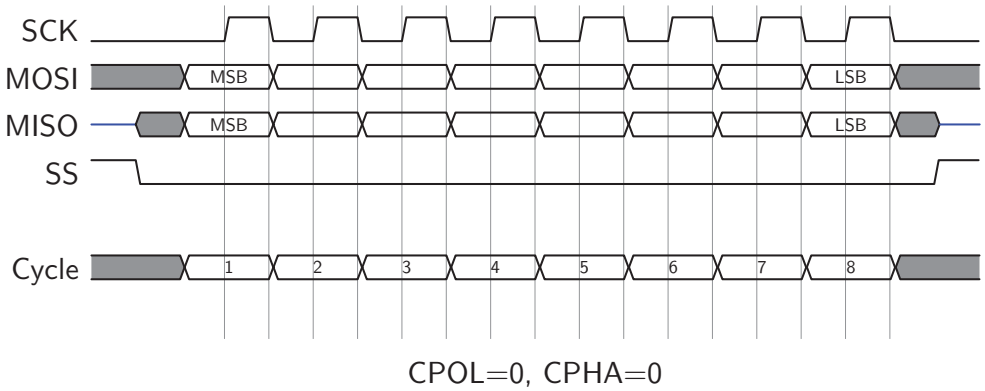


Figur 3.88: SPI-oppsett.

kan då aktivera dei ulike slavane vhja. bit i ODR-registeret til ein GPIO-modul.
Det er då lettare å utforma overføringssyklusen slik slavane vil ha denne. Ein vel så stor fordel er at ein då kan ha fleire slavar tilknytta ein og same SPI-modul. Ein kan altså realisera ein **SPI-buss**, som kan vera nyttig i mange system.

I figuren ser ein at SPI-signala går gjennom konfigurasjonslogikken til ein GPIO-modul. Av tabell 4 - 5 i brukarmanualen på plattformkortet vårt, [15], ser ein at det er GPIOA som er aktuell GPIO-modul for modulen SPI1 og GPIOB for SPI2.

Typisk overføringssyklus ved SPI-kommunikasjon
Ei innføring i virkemåte er gitt i kapittel 6 hjå Brown, [2]. Eit typisk tidsdiagram frå Brown for ei overføring i **full duplex**[141] er gjengitt i figur 3.89.



Figur 3.89: Typisk tidsdiagram for ei SPI-overføring.
(Frå Brown, [2], kapittel 6.1, under lisensen *Creative Common BY-NC-SA 3.0.*)

I denne overføringa er signalet *SS* brukt for å aktivera slaven. Ved bruk av ein vanleg GPIO-utgang for å styra $\overline{CS}$-inngangen på ein slave, vil overføringa sjå likeeins ut.

Oppførselen til SPI-grensesnitt frå ulike produsentar er ikkje heilt lik. Med innstillingane *CPOL* og *CPHA* kan ein tilpassa ein perifermodul i ein mikrokontroller til den oppførselen som ein slave har.
Innstillingane vist i figuren betyr følgjande:
CPOL = 0: Klokkesignalet er lågt i tomgang ("idle")[142].
CPHA = 0: Databitane er synkroniserte med 1. klokkeflanke, dvs. den positive her.

Då eit SPI-grensesnitt har relativt mange signallinjer, blir ikkje SPI brukt over store avstandar. Metoden er svært vanleg mellom komponentar på eit kretskort og mellom komponentar i eit innebygt system der avstandane ikkje er for store.

[141] Der ein sender og mottar samtidig.
[142] Forklaringane her er henta frå kap. 21.3 i referansemanualen [17].

Konfigurasjon av ein SPI-modul
Ein typisk konfigurasjonssekvens er vist under. Denne er delt opp i steg på same måte som for den generelle taimeren tidlegare i dette kapitlet.
I dette eksemplet er det bare ein slave, og denne blir aktivert vhja. eit $\overline{CS}$-signal styrt via ODR i ein passande GPIO-modul.

1. Oppsett av SPI-parametrar

```
//Oppsett av sjølve SPI-modulen, her SPI1.
//----------------------------------------
  //Først deklarasjon av initialiseringsstrukturen.
    SPI_InitTypeDef SPI_InitStructure;

  //Slepp først til klokka paa SPI1.
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

  //Oppsett av SPI1.
  // 1. Uni- eller bidireksjonal datamodus? Her bidireksjonal.
        SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
  // 2. SPI-modus, dvs. master eller slave.
        SPI_InitStructure.SPI_Mode      = SPI_Mode_Master;
  // 3. Databreidde.
        SPI_InitStructure.SPI_DataSize  = SPI_DataSize_8b;
  // 4. Tomgangsnivå for klokkesignalet.
        SPI_InitStructure.SPI_CPOL      = SPI_CPOL_Low;
  // 5. Kva klokkeflanke databitane er synkroniserte med.
        SPI_InitStructure.SPI_CPHA      = SPI_CPHA_1Edge;
  // 6. Maskin- eller programvarestyrte SS-signal? Her irrelevant.
        SPI_InitStructure.SPI_NSS       = SPI_NSS_Soft;
  // 7. Preskalerar for bitraten. Vil ha SPI-frekvens = 1.5 MHz. Har:
  //    f_spiclk = (f_sysklk/2)/preskalering = 1.5MHz (f_sysclk = 24 MHz)
        SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
  // 8. Kjem MSb eller LSb først? Her MSb.
        SPI_InitStructure.SPI_FirstBit  = SPI_FirstBit_MSB;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    SPI_Init(SPI1, &SPI_InitStructure);
```

2. Oppsett av GPIO-pinnar og oppstart av SPI-modulen

```
//Oppsett av GPIO-pinnane PA5 (SCLK), 6 (MISO) og 7 (MOSI),
//   som blir brukte av SPI1-modulen, sjå tabell i brukarmanualen.
//-----------------------------------------
  //Deklarasjon av initialiseringsstrukturen.
    GPIO_InitTypeDef GPIO_InitStructure_SPI1;

  //Slepp til klokka paa GPIO-portA.
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  //Konfigurer PA5 og 7 som er utgangar med alternativ funksjon.
    GPIO_InitStructure_SPI1.GPIO_Pin   = GPIO_Pin_5 | GPIO_Pin_7;
    GPIO_InitStructure_SPI1.GPIO_Mode  = GPIO_Mode_AF_PP;
    GPIO_InitStructure_SPI1.GPIO_Speed =  GPIO_Speed_2MHz;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOA, &GPIO_InitStructure_SPI1);

  //Sett PA6 som flytande inngang (''input floating'').
    GPIO_InitStructure_SPI1.GPIO_Pin  = GPIO_Pin_6;
    GPIO_InitStructure_SPI1.GPIO_Mode = GPIO_Mode_IN_FLOATING;

  //Initialiser, dvs. last ned konfigurasjonen i modulen.
    GPIO_Init(GPIOA, &GPIO_InitStructure_SPI1);

//Så oppsett av GPIO-pinnen PA4 (xCS, aktiv låg) som blir brukt
// i staden for SS-signalet i SPI1-modulen.
//-----------------------------------------
  //Deklarasjon av initialiseringsstrukturen.
    GPIO_InitTypeDef   GPIO_InitStructure;

  //Konfigurer som ein vanleg GPIO_pinne.
    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

  //Initialiser, dvs. last ned konfigurasjonen i modulen
    GPIO_Init(GPIOA, &GPIO_InitStructure);

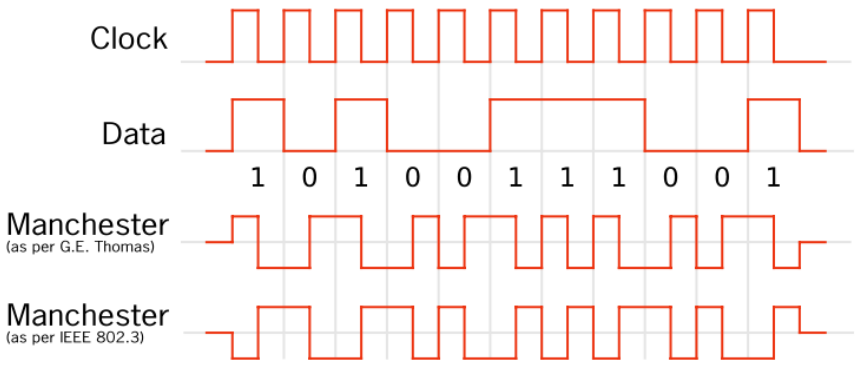
  //Deaktiver xCS. (Ved sending: xCS = 0 ved start og xCS = 1 etterpå.)
    GPIOA->ODR = GPIOA->ODR | GPIO_Pin_4;  //xCS = 1

  //Aktiverer SPI1.
    SPI_Cmd(SPI1, ENABLE);
```

3.10.7.5 Synkron kommunikasjon der klokkeinformasjonen er koda inn i datasignalet

Eit eksempel her er Ethernett-protokollen som bruker to signallinjer for kvar **datakanal**, dvs. som i figur 3.87. Den eine linja er her ikkje jorda, då overføringa er differensiell[143].
Eit typisk Ethernett-samband mellom ein **node**, f.eks. ein datamaskin, og ein **svitsj** eller **ruter**, har to datakanalar, ein for sending av datapakkar ut frå noden og ein for mottak av datapakkar. Ein har altså full duplex overføring her også.
For kabla samband bruker ein ofte to tvinna linjepar[144] for å realisera dei to kanalane.

Sidan ein ikkje har eiga klokkesignallinje for ein datakanal, kodar ein klokkeinformasjonen inn i bitstraumen. Her skal me sjå på metoden som blei brukt i *10BASE-T* Ethernet (IEEE-standard 802.3), dvs. ei utgåve med bitrate 10 Mbit/sek. Metoden heitte *Manchester*-koding og er eit godt eksempel på slik koding. To variantar av metoden er vist i figur 3.90.



Figur 3.90: Manchester-koding.
(Ref.: "Wiki Manchester_encoding_both_conventions.svg".)

Kodinga gir som vist ein **garantert transisjon** midt i kvart bit som mottakaren kan detektera. Då det vil vera ein uregelmessig forekomst av flankar mellom bitane, må mottakarlogikken vera relativt avansert.
Dette har vore ein populær metode som blei brukt i andre protokollar også.

Ein ulempe er at metoden krev ei bandbreidde på det doble av bitfrekvensen. Protokollar med høgare bitratar bruker bl.a. derfor andre metodar, f.eks. *MLT-3*[145] i *100BASE-T*.

Ved *USB*-kommunikasjon blir også klokkeinformasjon koda inn i datastraumen. Protokollen bruker ein variant av metoden "Non Return to Zero", NRZ, med såkalla "bit stuffing"[146] for koding av klokkeinformasjonen.

[143] Sjå "en.wiki Differential signaling".
[144] Sjå "en.wiki Ethernet over twisted pair".
[145] Sjå "en.wiki MLT-3 encoding".
[146] Sjå "en.wiki Non-return-to-zero".

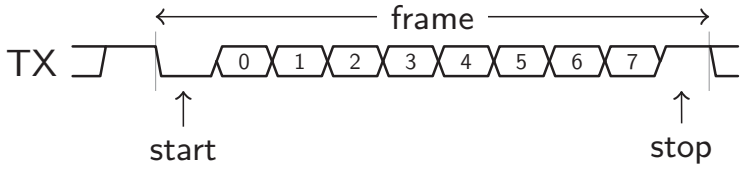
USB bruker bare ein datakanal[147] for overføring, og denne er differensiell som i Ethernet-protokollen.
USB-kablar har i tillegg til dei to differensielle signallinjene *D+* og *D-*, ei linje for jord og ei for 5Volt-forsyning.

3.10.7.6 Asynkron kommunikasjon

Ved asynkron kommunikasjon får ikkje mottakaren informasjon om klokkesignalet til sendaren. Sendar og mottakar har uavhengige klokker, men eit krav er at desse klokkene har nær same frekvens. For at mottakaren skal forstå sendaren, må dei i tillegg ha same protokolloppsett, dvs. følgjande:

- Talet på databit: Kan vera typisk 7 eller 8 bit.
- Paritet: Oddetals-, partalsparitet eller ingen paritet.
- Talet på stoppbit: Kan vera 1, 1.5 eller 2 bit.

Brown viser i kapittel 5, [2], ei ramme utan paritetsbit. Denne ramma er gjengitt i figur 3.91.



Figur 3.91: Rammeeksempel for asynkron kommunikasjon.
(Frå Brown, [2], kapittel 5, under lisensen *Creative Common BY-NC-SA 3.0.*)

Alle rammer startar med ein startbit, der ein har ein garantert transisjon frå 1 til 0. Så kjem databitane der LSb vanlegvis er først.

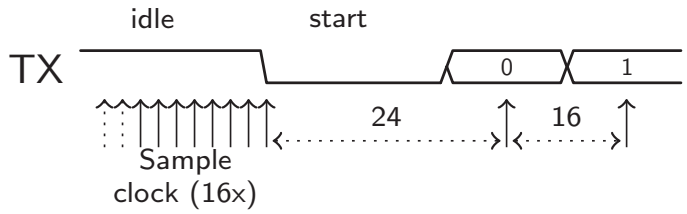
Viss ein spesifiserer at det skal brukast paritetsjekk, kjem det eit paritetsbit rett etter databitane.
Paritetssjekk av ei mottatt ramme kan brukast til å avdekka om ein av bitane har endra verdi under overføring. Viss odde paritet skal brukast, vil sendaren setja verdien av paritetsbiten slik at paritetsbiten og databitane til saman inneheld eit oddetal 1-arar.
Viss mottakaren detekterer eit partal 1-arar, vil han setja ein statusbit som viser at det har oppstått ein paritetsfeil[148].

Til slutt i ramma kjem stoppbiten, evt. stoppbitane. Ei ramme for sending av ein byte vil altså samla sett innehalda 11 bit med paritet og 1 stoppbit, og 10 bit utan paritet.

[147] USB er altså bare **halv duplex**.

[148] Viss to bitar har endra verdi, kan ikkje dette oppdagast med paritetssjekk, men ein 2-bitsfeil er mykje mindre sannsynleg enn ein 1-bitsfeil.
I kraftigare protokollar der meldingane er lengre og bitraten høgare, er feilsjekkinga mykje meir avansert.

For å kunne oppdaga startbit-transisjonen og kunne posisjonera seg til midten på kvart bit der mottakaren les bitverdien, **samplar** mottakaren signalet som kjem inn. Vanlegvis bruker mottakaren ein samplefrekvens som er 16 x bitfrekvensen.
Samplinga er vist i figur 3.92. Denne er gjengitt frå kapittel 5 hjå Brown, [2].



Figur 3.92: Sampling av bit ved asynkron kommunikasjon.
(Frå Brown, [2], kapittel 5, under lisensen *Creative Common BY-NC-SA 3.0.*)

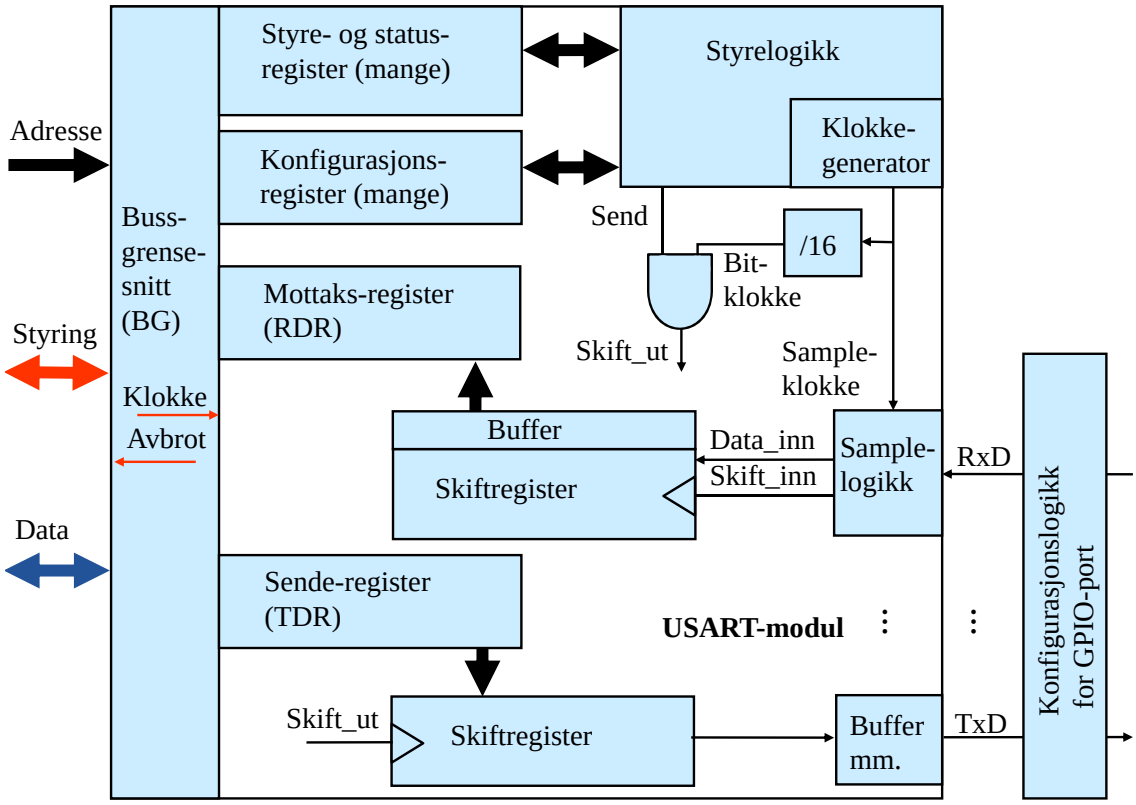
Ein kan lesa meir om virkemåte i kapittel 5 hjå Brown.

Det at ein samplar bitane med ei oppløysing på 16X, gjer at bitraten for slik kommunikasjon ikkje kan bli veldig høg. Andre eigenskapar gjer i tillegg at ein asynkron standard som RS-232 var avgrensa til 20 kBit/sek og 15 meter. Med moderne drivarkretsar kan ein i praksis køyra høgare bitratar.
Andre meir robuste standardar som RS-422 og RS-485[149] bruker differensiell overføring. Desse kan køyra bitratar opp til 1 Mbit/sek over avstandar typisk rundt 100 m, eller over avstandar opp til 1 km med bitrate typisk rundt 100 kbit/sek.

Den perifermodulen som handterer asynkron kommunikasjon i mikrokontrolleren vår, heiter *USART* "Universal Asynchronous and Synchronous Receiver and transmitter". Meir om denne kjem i dei neste underkapitla.

[149] Alle desse såkalla EIA-standardane kan ein finna meir om hjå "en.wiki RS-232", ".. RS-422" og ".. RS-485.

Eksempel på perifermodul: USART
Strukturen til ein USART-modul er vist i figur 3.93.



Figur 3.93: Den asynkrone delen av ein USART-modul.

Som namnet til perifermodulen tilseier, kan denne realisera både synkron og asynkron kommunikasjon. I figur 3.93 er det bare vist grovstrukturen til den delen som realiserer **asynkron** kommunikasjon.

Som ein ser, er **skiftregisteret** ei sentral funksjonsblokk i ein slik perifermodul.
Mottaksregisteret (”Receive Data Register”, RDR) dannar saman med eit skiftregister ein **”serie-til-parallell”-omformar** (S2P).
Likeeins dannar **senderegisteret** (”Transmit Data Register”, TDR) saman med eit skift-register ein **”parallell-til-serie”-omformar** (P2S).

Sjølve sendinga går med ein spesifisert bitrate, mens mottakslogikken samplar mottaks-signalet med ein frekvens som vanlegvis er 16 x bitraten som forklart i førre delkapittel.

Eit detaljert blokkskjema av USART-modulen er vist i figur 243 på s. 570 i referansema-nualen for mikrokontrolleren vår, [17]. Her er også vist ei liste over alle registra i tabell 131 på side 612.

Konfigurasjon av ein USART-modul
Ein typisk konfigurasjonssekvens for USART-modulen USART1 brukt til rein asynkron kommunikasjon, kan sjå ut som i det følgjande:

```
//USART1
//-----------------------------------------------
  //Først deklarasjon av UART-strukturar
    USART_InitTypeDef USART1_InitStructure;
    USART_ClockInitTypeDef  USART1_ClockInitStructure;

  //Slepp først til klokka på modulen
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

    USART_ClockStructInit(&USART1_ClockInitStructure);
    USART_ClockInit(USART1, &USART1_ClockInitStructure);

    USART1_InitStructure.USART_BaudRate = 9600;
    USART1_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART1_InitStructure.USART_StopBits = USART_StopBits_1;
    USART1_InitStructure.USART_Parity = USART_Parity_No ;
    USART1_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    USART1_InitStructure.USART_HardwareFlowControl =
                                        USART_HardwareFlowControl_None;
  //Initialiser, dvs. last ned konfigurasjonen i modulen
    USART_Init(USART1, &USART1_InitStructure);

//Så oppsett av GPIO-pinnane PA9 og 10 som blir brukte mot USART1
//-------------------------------------------------------------------------
  Dette er vist i kapitlet om systemblokka ''Parallellport'', sjå også
  kommentaren rett etter dette eksemplet.

//Start til slutt USART1
//-------------------------------------------------------------------------
    USART_Cmd(USART1, ENABLE);
```

Konfigurasjonskoden for sjølve GPIO-pinnane blei vist i kapittel 3.10.5.5 på side 232.
Når det gjeld sjølve klokkeinitialiseringa, kjem det meir om dette i kapittel 3.10.8.3.

I koden over ser ein sentrale kommunikasjonsparametrar bli sett opp, nemleg:

- Bitrate, her 9600 bps (bit pr. sek.).
- Ordlengde, her 8 bit.
- Talet på stoppbit, her 1 bit.
- Paritet, her ingen.
- Modus, her basert på sende- og mottakssignala Tx og Rx.
- Styring av datastraumen, her inga styring.

Den eksterne modulen som det skal kommuniserast med, må sjølvsagt vera sett opp på same måte for at dei to skal forstå kvarandre. Elles blir det generert feilmeldingar.

Ein kan også merka seg at samplelogikken, sjå figur 3.93, bruker ein samplefrekvens som er 16 x bitraten, her $9600 \cdot 16 = 153,6$ kHz.

3.10.8 Generell framgangsmåte ved konfigurasjon av perifermodular

Me har nettopp sett på sjølve konfigurasjonskoden for ein USART-modul. Det skal nå presenterast ein generell framgangsmåte for å koma fram til den rette konfigurasjonskoden for ein perifermodul. Her vil ein ta USART-modulen som eksempel.
Framgangsmåten er delt opp punktvis i det følgjande.

3.10.8.1 Rammeverk

Det kan vera lurt å laga ei eiga fil for kvar perifermodul. Denne kan då innehalda alle metodane for denne modulen inkludert initialiseringsmetoden. Denne metoden konfigurerer og startar opp modulen.
Fila for USART1-modulen kan heita *USART1_metodar.c*.

Øvst i denne fila må ein inkudera dei nødvendige CMSIS-filene. Her kan kapittel 2.4.2 vera til hjelp.
Dei nødvendige filene for USART-modulen kan inkluderast som vist under.

```
#include ''stm32f10x.h''
#include ''stm32f10x_gpio.h''
#include ''stm32f10x_rcc.h''
#include ''stm32f10x_usart.h''
```

I punkta under blir det vist kva ein bruker dei ulike filene til.

3.10.8.2 Klokke

I blokkskjemaet, dvs. figur 6, i brukarmanualen på plattformkortet vårt, [15], ser ein at modulen USART1 er knytt til periferbussen *APB2*[150].
I fila *stm32f10x_rcc.h* kan ein finna den rette metoden for å setja klokkesignal på USART1, nemleg

```
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph,FunctionalState NewState);
```

I same fila kan ein finna det rette argumentet *RCC_APB2Periph_USART1*.
I tillegg skal nå tilstanden vera *ENABLE*.

[150]Litt om denne står i kapittel 2.4.2.4 på side 63.

3.10.8.3 Parametrar for sjølve perifermodulen

Her må ein gå til fila *stm32f10x_usart.h* for å finna relevante strukturdefinisjonar. Desse gir saman med forklaringane i fila eit godt utgangspunkt for å setja opp perifermodulen på rett måte.
Når det gjeld USART-en, er det strukturane *USART_ClockInitTypeDef* og *USART_InitTypeDef* som må setjast opp.

Strukturen *USART_ClockInitTypeDef* er vist under.

```
typedef struct
{
  uint16_t USART_Clock;
  uint16_t USART_CPOL;
  uint16_t USART_CPHA;
  uint16_t USART_LastBit;
} USART_ClockInitTypeDef;
```

Me kjenner igjen nokre av dei parametrane som blei sette opp for ein SPI-modul.
Då RS-232-overføringar faktisk er meir standardiserte enn SPI-overføringar, kan ein bruka eit standard oppsett.
Kode for deklarasjon av struktur og konfigurasjon med eit standard oppsett er vist under.

```
USART_ClockInitTypeDef  USART1_ClockInitStructure;

USART_ClockStructInit(&USART1_ClockInitStructure);   //Sett opp strukturen.
USART_ClockInit(USART1, &USART1_ClockInitStructure); //Last ned i modul.
```

Strukturen *USART_InitTypeDef* er vist under.

```
typedef struct
{
  uint32_t USART_BaudRate;
  uint16_t USART_WordLength;
  uint16_t USART_StopBits;
  uint16_t USART_Parity;
  uint16_t USART_Mode;
  uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;
```

Her vil det vera store variasjonar i konfigurasjon mellom ulike applikasjonar.
I same fila som før, dvs. *stm32f10x_usart.h*, kan ein også finna definisjonar av bitmønster

for konfigurasjon. Her kan f.eks. det symbolske bitmønsteret *USART_Parity_No* brukast til å konfigurera paritetslaus overføring.

Hugs til slutt her å leggja inn kall av metode for nedlasting av strukturen i modulen.

3.10.8.4 Konfigurasjon av GPIO-pinnane brukt av modulen

Først av alt må ein finna ut kva signal ein har bruk for inn til og ut av modulen.
I dette eksemplet skal ein ha den enklast moglege to-vegsoverføringa (full duplex), og treng då bare mottakssignalet *RxD* og sendesignalet *TxD*.

I tabell 4 - 5 i brukarmanualen på plattformkortet vårt, [15], kan ein då finna ut kva GPIO-pinnar som kan formidla desse signala[151].

Så gjer ein på same måten som før for ein aktuell GPIO-modul, nemleg:

- Finn periferbusstilknytting og metode samt argument for klokkestart.
- Finn initialiseringsstruktur i fila *stm32f10x_gpio.h* og deklarer denne.
- Sett opp strukturen vhja. bitmønster frå same fil og last så ned.

3.10.8.5 Oppstart av perifermodul

I fila *stm32f10x_usart.h* kan ein finna oppstartsmetoden. For USART-modulen er den som vist under.

```
void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
```

Argumenta blir her *USART1* og *ENABLE*.

[151] I nokre tilfelle er det etter studering av denne tabellen at ein vel kva for ein perifermodul ein vil bruka. Det kan vera at elektronikken som skal koblast til og behovet for andre typar perifermodular tilseier at ein bruker f.eks. USART2 i staden for USART1.

3.10.8.6 Litt om det å laga metodar for modulen

Når ein i dette eksemplet så skal laga metodar for kommunikasjon, kan ein finna ferdig deklarerte metodar i fila *stm32f10x_usart.c*. Det kan vera metodar for sending og mottak som er deklarerte slik:

```
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);
uint16_t USART_ReceiveData(USART_TypeDef* USARTx);
```

Viss ein har behov for å laga eigne metodar, må ein kjenna registerstrukturen for perifermodulen. Når det gjeld USART-modulen, er denne strukturen deklarert i fila *stm32f10x.h* og ser ut som vist under.

```
typedef struct
{
  __IO uint16_t SR;
  uint16_t  RESERVED0;
  __IO uint16_t DR;
  uint16_t  RESERVED1;
  __IO uint16_t BRR;
  uint16_t  RESERVED2;
  __IO uint16_t CR1;
  uint16_t  RESERVED3;
  __IO uint16_t CR2;
  uint16_t  RESERVED4;
  __IO uint16_t CR3;
  uint16_t  RESERVED5;
  __IO uint16_t GTPR;
  uint16_t  RESERVED6;
} USART_TypeDef;
```

Eksempelvis vil C-instruksjonen `USART1->DR = '1';` plassera ASCII-koden for talet *1* i senderegisteret til USART-modulen.

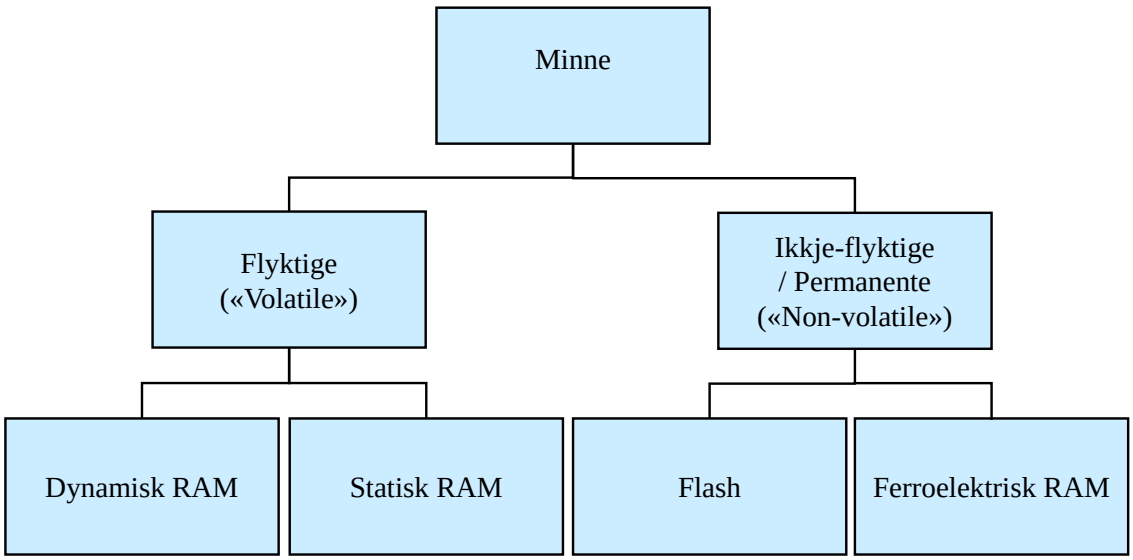
3.10.8.7 Litt om informasjonskjelder

Viss forklaringane i CMSIS-filene for ein perifermodul ikkje er klare nok, kan ein finna ut av den detaljerte oppførselen til modulen og registerdetaljar ved å studera referansemanualen for mikrokontrolleren vår, [17].
Det kan sjølvsagt også vera ein god del ferdig biblioteksstoff på nettet som kan brukast. Her gjeld det likevel **å vita kva ein gjer**, då ting som andre har laga, kan avvika frå detaljar i det oppsettet du vil ha.

3.10.9 Minne

3.10.9.1 Hovudgrupper

Hovudgruppene innanfor minneteknologi er vist i figur 3.94.



Figur 3.94: Hovudgrupper av minne.

Ein vil her først sjå på korleis eit minne er bygd opp, dvs. minnestrukturen, og så vil dei fire vanlegaste minnetypane av i dag bli presenterte. Desse dannar botnlinja i figur 3.94.

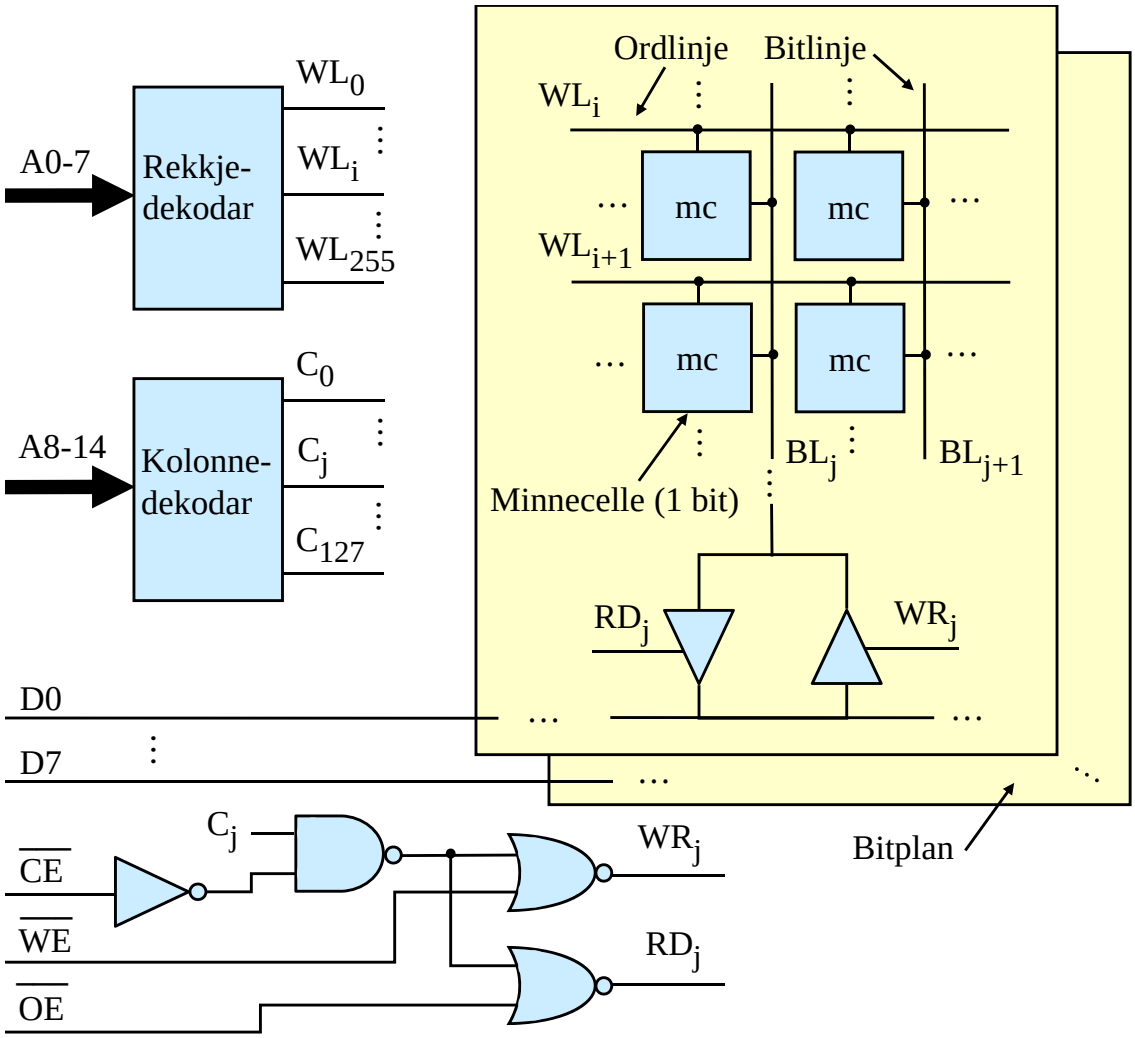
3.10.9.2 Struktur

Generell struktur til eit minne er vist i figur 3.95.

Det er her tatt utgangspunkt i eit 256kbitsminne organisert som 32k x 8 bit.
For å adressera 32kByte = $2^5 \cdot 2^{10}$ bytar, treng ein 15 adresselinjer, dvs. *A0-14*.

Eit minne har som vist i figuren, **matrisestruktur**. Nokre av adresselinjene blir brukt til å velja ei viss **rekkje**, også kalla **ordlinje** (WL).
Resten av adresselinjene[152] blir brukt til å velja ei viss **kolonne** (C). Saman med styresignala vil kolonnesignalet aktivera ei såkalla **bitlinje** (BL).
Avhengig av verdien på styresignala kan ein då **skriva** til den **minnecella** som ligg i skjæringspunktet mellom ordlinja og bitlinja eller **lesa** frå denne.

[152]Kva linjer som blir brukte og kor mange rekkjer det er i forhold til kolonner, varierer mellom ulike kretsar og produsentar.



Figur 3.95: Strukturen til eit minne på 32k x 8 bit.

Når ein aktiverer ei ordlinje, vil alle minnecellene langs denne opnast, og ein databit kan då generelt skrivast eller lesast som vist via bitlinja.

Litt om styresignala:

- $\overline{CE}$ ("Chip Enable") er eit signal for aktivering av minnet. Viss $\overline{CE} = 1$, vil ein ikkje kunne lesa frå eller skriva til minnet. Minnet vil då bruka minimal effekt.
- $\overline{WE}$ ("Write Enable") er eit skrivesignal. For å kunne opna skrivebufferet inn mot ei bestemt bitlinje BL_j, må ein som vist i figuren ha både $\overline{CE} = 0$ og $\overline{WE} = 0$. I tillegg må kolonnesignalet C_j vera aktivert.
- $\overline{OE}$ ("Output Enable") er eit lesesignal. For å kunne opna lesebufferet knytt til ei bestemt bitlinje, må ein som vist i figuren ha både $\overline{CE} = 0$ og $\overline{OE} = 0$ samt aktivt kolonnesignal.

Me kan sjå på $\overline{OE}$ som eit signal som opnar lesebufferet, dvs. sjølve *tristate*-bufferet ut mot databussen.

Sjølve styrelogikken vist i figuren er forenkla samanlikna med tilsvarande logikk i verkelege kretsar. Dette er gjort for å kunne illustrera funksjonen til dei ulike styresignala på ein mest mogleg tydeleg måte.

Som vist i figuren, er dei ulike **datalinjene** kobla til kvart sitt **bitplan**. Dette er ikkje fysiske etasjar inne i kretsen, men ein måte å visualisera strukturen på.
Når det gjeld datalinjene, kan dei vera **bi-direksjonale** som vist i figuren. Andre minne kan ha **separate** linjer for data inn til og ut frå minnet.

Det er bare minne av typen RAM ("Random Access Memory") som både kan lesast frå og skrivast til som vist. Permanentminne som Flash kan bare lesast frå på denne måten. Programmering skjer vhja. ein spesiell metode og over lengre tid.

Sentrale funksjonsblokker her er dei **to dekodarane** her. Matrisestrukturen gjer at ein sparar både logikk og linjer. I eksemplet her blir talet på linjer lik $256 + 128 = 384$. Viss ein brukte bare ein dekodar og ein-dimensjonal struktur, ville ein måtte ruta 32.768 linjer gjennom minnet!

Forskjellen mellom ulike minnetypar dreier seg litt om struktur, men mest om at minnecellene i desse er basert på ulik teknologi. Me skal sjå litt på det i det følgjande.

3.10.9.3 Litt om flyktige minne

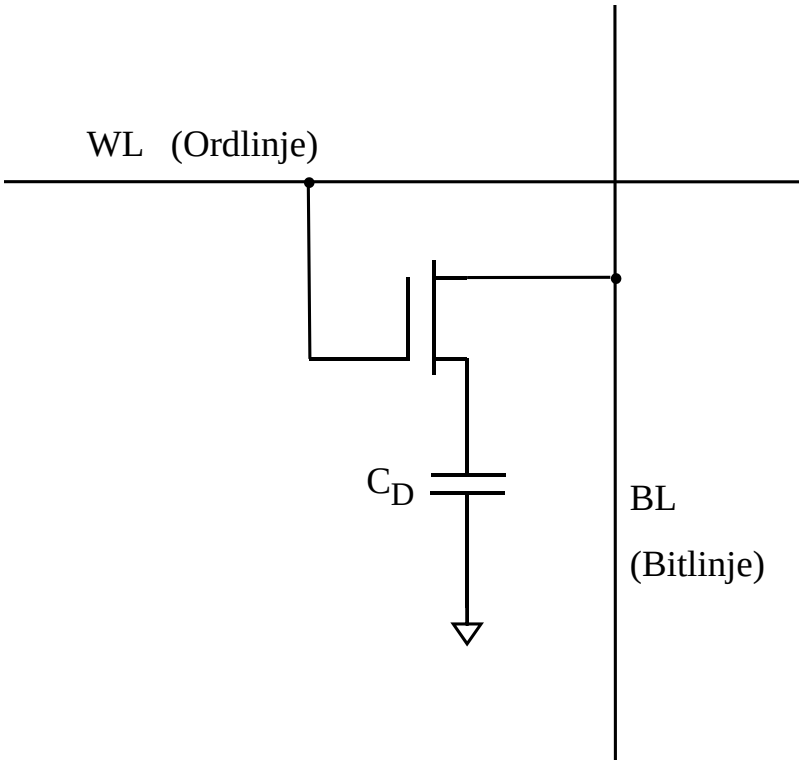
Flyktige ("volatile") minne mistar innhaldet når ein slår av kraftforsyninga til minnet. Her skal me sjå på dei to dominerande minnetypane DRAM og SRAM.

3.10.9.4 Dynamisk RAM

Ei minnecelle i eit dynamisk RAM-minne er vist i figur 3.96.

Dette er den **tettaste**, dvs. den minst plasskrevjande minnecella som finst i vanlege minne i dag. Det er årsaken til at denne teknologien blir brukt i store minnesystem, som f.eks. hovudminnet i PC-ar.

Det er ei stor ulempe med denne minnecella. Kondensatoren blir sjølutlada, dvs. at ladningane lek ut over tid. Slike minne krev derfor **oppfrisking** ("refresh"). Dette blir vanlegvis gjort av ein eigen DRAM-kontroller. Kvart 10. - 100. msekund vil kontrolleren aktivera ordlinje for ordlinje gjennom heile minnet og for kvar aktivert ordlinje tilføra nok ladning på bitlinjene til at ein '1' i minnecellene held seg til neste gong, men ikkje så mykje ladning at ein '0' blir til ein '1'.



Figur 3.96: Ei DRAM-celle.

Det er utvikla mange variantar av DRAM. Dei vanlegaste i dag er synkrone og med avanserte grensesnitt der ein oppnår høge overføringsratar.
Hjå "en.wiki Dynamic random-access memory" kan ein lesa meir om slike minne.

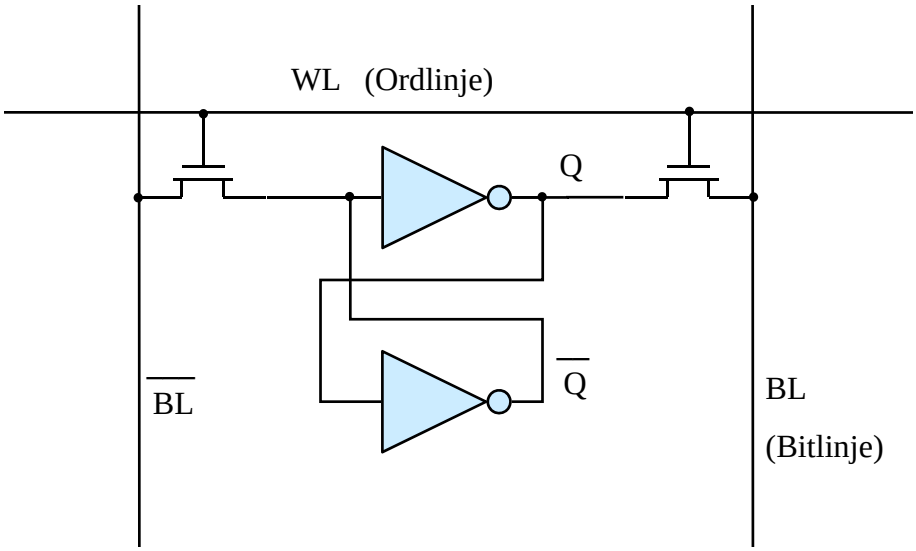
3.10.9.5 Statisk RAM

Ei minnecelle i eit statisk RAM-minne er vist i figur 3.97.

Som ein ser, så er ei minnecelle rett og slett ein enkel lås ("latch"). Sidan ein inverterande buffer som kjent er bygd opp av to transistorar, vil det samla sett gå med 6 transistorar for å realisera ei slik minnecelle.
Det er innført ei ekstra bitlinje her, og nivået på denne vil vera det inverse av den andre som vist. Ved skriving til denne minnecella, overfører ein databitverdien på BL-linja samtidig som ein overfører den inverse verdien på $\overline{BL}$-linja.
Dette gir eit rask innstilling av låsen.

Låsen vil som kjent halda på informasjonen, dvs. Q-verdien, sålenge kraftforsyninga er på. Minnet blir derfor kalla **statisk**.
Slike minne treng då ikkje tilleggslogikk slik som DRAM, og vil vera den føretrekte typen i mindre og mellomstore innebygde system. Inne i store prosessorbrikker[153] og mikrokon-

[153]Dette blir kalla **hurtigminne** ("cache memory"). Det er ofte fleire nivå på dette. F.eks. kan ein ha dei



Figur 3.97: Ei SRAM-celle.

trollerar er også denne typen som regel nødvendig.

Dette har også vore den raskaste minnetypen med aksesstider < 10 nsek. Avanserte utgåver av DRAM kan rett nok også gjerast like raske i dag.

Meir om SRAM kan finnast hjå "en.wiki Static random-access memory".

3.10.9.6 Litt om permanente minne

Permanente ("non-volatile") minne held på innhaldet når ein slår av kraftforsyninga. Her skal me sjå på den dominerande minnetypen Flash samt ein relativt ny minnetype, nemleg FRAM. Denne siste er veldig aktuell som minne for permanent lagring av moderate datamengder i innebygde system.

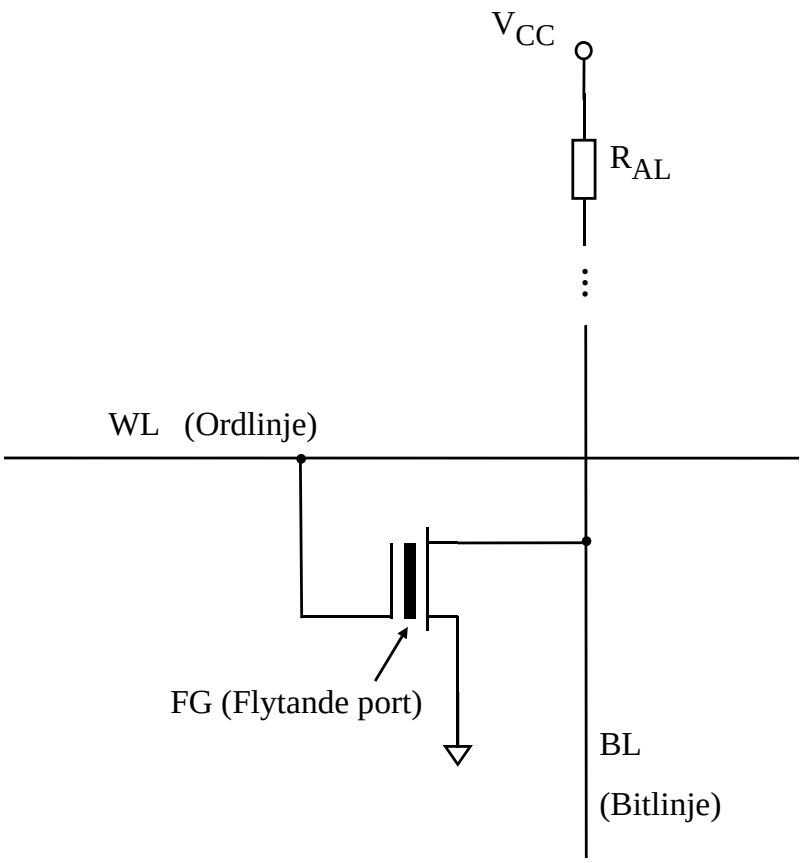
3.10.9.7 Flash

Ei minnecelle i eit Flash-minne er vist i figur 3.98.

Dette er som DRAM ein av dei tettaste minnetypane. Minnecella er bygt opp av bare ein transistor, nemleg ein spesiell transistor med ein såkalla flytande port ("floating gate").

Ved **programmering** av cella tilfører ein ladningar til den isolerte porten vhja. metoden

tre nivåa *L(evel)1 - 3* i ein PC-prosessor. Nettstaden "no.wiki Hurtigminne" gir ein liten start på dette og så kan ein gå over på dei engelske sidene etterpå.



Figur 3.98: Ei Flash-celle.

HEI ("Hot Electron Injection"). Terskelspenninga V_{Th}[154] vil då stiga så høgt at ein ikkje greier å slå transistoren på ved å aktivera ordlinja.
Opptrekksmotstanden[155] vil gjera at lesing av ei programmert minnecelle i dette tilfellet vil gi ein '1'.

I dei cellene som **ikkje** blir programmerte, vil transistoren slå seg på når ein aktiverer ordlinja. Ein vil då som ein ser av figuren, lesa ein '0' .

Skal ein **reprogrammera** minnet, må ein først **sletta** ("erase") minneinnhaldet. Eit Flash-minne er delt opp i **sektorar**[156]. Sletting må gjerast av ein heil sektor i slengen. Sjølve slettinga går ut på å lada ut alle dei opplada isolerte portane, noko ein gjer med metoden **QT** ("Quantum Tunnelling").
Flash-minne kan reprogrammerast typisk 100.000 til 1.000.000 gongar.

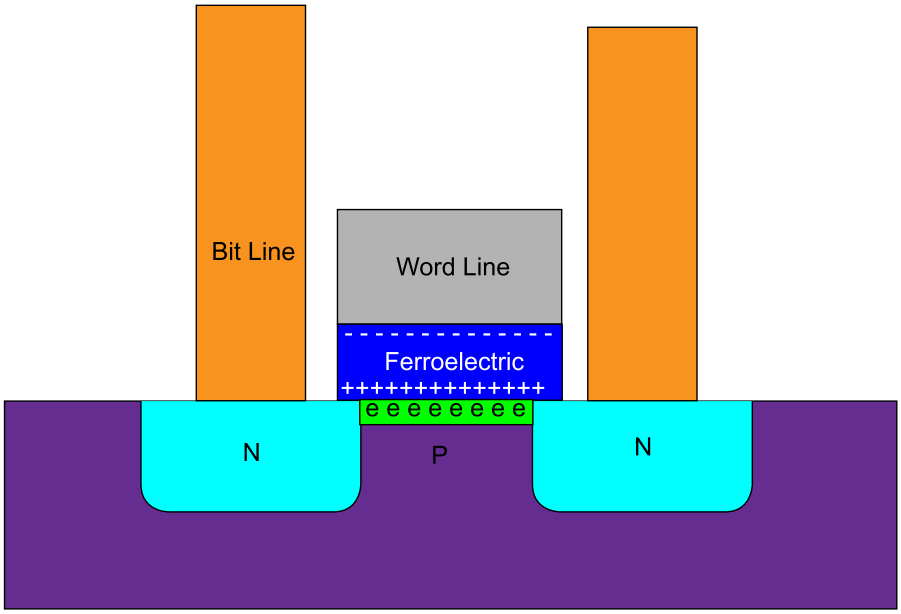
Flash-minne er tilgjengeleg i form av brikker med kapasitet typisk opp til 1 Gb og diskar på i alle fall 128 GB i dag.

[154] Jfr. kapittel 3.4.2 på side 116.
[155] Eigentleg er opptrekksmotstanden ein felteffekttransistor i eit spesielt oppsett der G(ate) er kobla saman med D(rain). Dette blir kalla ei aktiv last ("active load").
[156] Desse er typisk på 64, 128 eller 256kB.

Det er to hovudtypar Flash-minne, nemleg NOR- og NAND-Flash. Det er bare den førstnemnde som har ein såkalla ”Random Access”-oppførsel som nemnt tidlegare. Ved bruk av NAND-flash kan ein bare lesa heile blokker om gongen. Det blir likevel produsert meir av denne typen enn av NOR-flash, då denne blir brukt til lagring av bilete og video i mobiltelefonar og kamera mm. Men som programminne må ein altså velja NOR-flash.
Du kan finna meir hjå ”en.wiki Flash memory”.

3.10.9.8 Ferroelektrisk RAM

Strukturen til ei ferroelektrisk minnecelle er vist figur 3.99.



Figur 3.99: Ferroelektrisk minnecelle
(Ref.: ”Wiki 1T_FeRAM_cell_structure.svg”.).

Ei ferroelektrisk minnecelle liknar på ei DRAM-celle der kondensatoren er erstatta av magnetisk materiale. Ved skriving set ein orienteringa av dei magnetiske dipolane i ein retning gitt av bitverdien. Orienteringa er permanent[157], men kan altså endrast med ein vanleg skriveoperasjon tilsvarande skriveoperasjonar mot SRAM-minne.

Når ein les ei FRAM-celle, vil ein avhengig av orienteringa av dipolane få ein straumpuls eller ikkje.
Leselogikken må altså vera litt spesiell samanlikna med dei minna ein har sett på til nå. Sett utanfrå oppfører minnet seg likevel som ein SRAM-krets.
Akkurat dette gjer FRAM til eit veldig aktuelt minne for permanent lagring av moderate datamengder i innebygde system. Skal ein laga ein liten dataloggar eller bare har behov for automatisk lagring av nokre brukarinnstillingar som ikkje må forsvinna når systemet

[157] Sjølv om cella liknar ei DRAM-celle, er det ingen form for ”lekkasje” her.

blir slått av, er FRAM det enklaste alternativet.
Ved bruk av Flash må ein implementera ein tyngre skriveprosedyre, men har då rett nok eit minne med mykje større kapasitet.

Alle typane permanentminne kan ein få med serielt grensesnitt, f.eks. SPI. Serielle grensesnitt er veldig vanleg i slike tilleggsminne, og gjer at kretsane blir små og elektronikken kompakt.
Ein eksempelkrets er *FM25CL64*[158]. Han er organisert som 8k x 8 bit, har SPI-grensensnitt og totalt bare 8 pinnar.

Meir detaljar om oppbygging og virkemåte av FRAM er å finna hjå "en.wiki Ferroelectric RAM".

[158] Guglar du dette namnet, får du opp ei lenke til produsenten og databladet.

3.10.9.9 Ei lita samanlikning av dei fire minnetypane

Ei lita og grov samanlikning av dei fire minnetypane som er presenterte her, er vist i tabell 3.16.

	DRAM	SRAM	Flash (NOR)	FRAM
Tett	Ja	Nei	Ja	Nei
Rask skriving	Ja	Ja	Nei	Ja
Rask lesing	Ja	Ja	Ja	Ja
Kretsstørrelse	$\geq$ 4Gb	$\geq$ 64MB	$\geq$ 1Gb	$\geq$ 2Mb
Permanent	Nei	Nei	Ja	Ja
Treng tilleggslogikk	Ja	Nei	Nei	Nei
Serielt grensesnitt	Uvanleg	Vanleg	Vanleg	Svært vanleg

Tabell 3.16: Samanlikning av minnetypar.

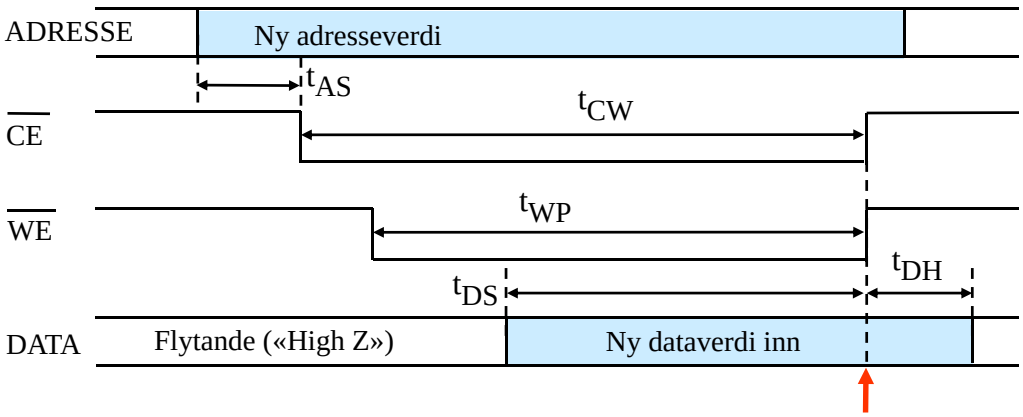
3.10.9.10 Typiske tidsdiagram for dataoverføring til og frå minne

For at eit minne skal kunne oppfatta data på rett måte, må styre-, adresse- og datasignala bli køyrde i rett sekvens. I tillegg vil det vera mange tidskrav som må oppfyllast.
Dette gjeld både for ein **skrive**-sekvens eller -**syklus** (”write cycle”) mot eit minne og når ein vil at eit minne skal levera data, dvs. i ein **lesesyklus** (”read cycle”).

Ein skal her sjå på typiske tidsdiagram for dataoverføring i begge desse tilfella. Tidsdiagramma vil gjelda for skriving til og lesing frå vanlege minne som SRAM og FRAM samt for lesing frå Flash-minne.

Som konkret minneeksempel er her brukt *AMIC A623308A-70SF*. Dette er eit SRAM-minne på 8K x 8 bit. Både namna på styresignala og signalsekvensane som er viste vhja. tidsdiagram i databladet, [27], er typiske for slike minne.
Sjølve minnestrukturen er som vist i figur 3.95 på side 259, og funksjonen til dei ulike styresignala er som forklart i teksten rundt denne figuren.

Skrivesyklus
Typisk signalsekvens og sentrale tidsparametrar for skriving er vist i figur 3.100.



Figur 3.100: Typisk skrivesyklus.

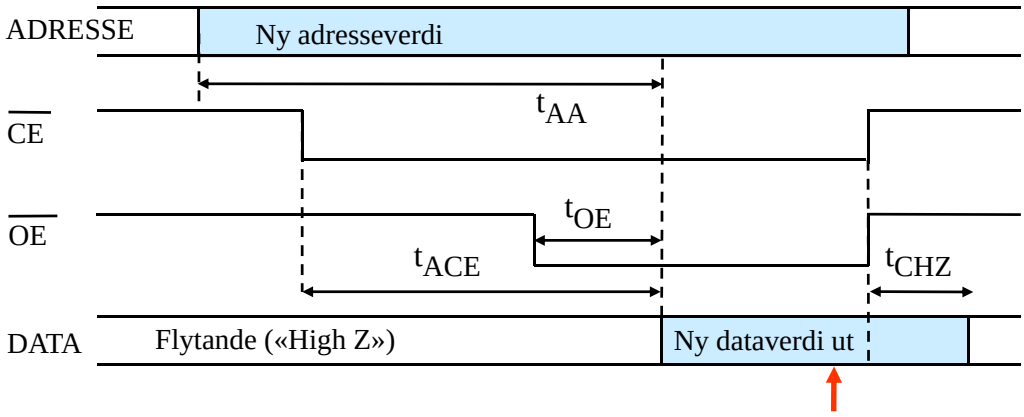
Dette er ei forenkla utgåve av diagrammet "Write Cycle 2" på side 8 i det nemnde databladet, [27]. Tidskrava står i tabellen på side 5.
Sentrale tidskrav er her knytt til:

- Adresseoppsett, t_{AS}.
  Ny adresseverdi må stå ei stund på bussen før ein kan aktivera kretsen.
- Aktiveringstid, t_{CW}.
  Det må gå ei viss minimumstid frå ein aktiverer kretsen til skrivepulsen blir deaktivert.
- Skrivepulsbreidde, t_{WP}.
  Sjølve skrivepulsen må vara ei viss minimumstid.
- Dataoppsett, t_{DS}.
  Data må stå ei stund på bussen før ein kan deaktivera skrivesignalet.
  Merk: Det er ved deaktivering, dvs. på positiv flanke av $\overline{WE}$, at minnet låser inn data. Dette er vist med **raud pil** i figuren.
- Datahalding, t_{DH}.
  Data må haldast ei stund på bussen etter deaktivering av skrivesignalet for at innlåsinga skal bli fullført.

Alle desse typane tidskrav botnar i at portar har etterslep og at me har med sekvensiell logikk å gjera. Grunnlaget for tidskrava kan me finna i kapittel 3.9.5 om datalåsen.

Viss ein skal laga eit grensesnitt mot eit slikt minne, må alle tidskrava oppfyllast.
I kapittel 4 skal ein nettopp sjå på korleis slike grensesnitt kan lagast.

Lesesyklus
Typisk signalsekvens og sentrale tidsparametrar for lesing er vist i figur 3.101.



Figur 3.101: Typisk lesesyklus.

Dette er ei forenkla utgåve av diagrammet "Read Cycle 1" på side 6 i det nemnde databladet, [27]. Tidskrava står her også i tabellen på side 5.
Sentrale tidskrav er her knytt til:

- Adresseaksess, t_{AA}.
  Frå ny adresse er på plass vil det gå ei maksimumstid før nye data er gyldige ("valid") på utgangen. Ei tilleggsvilkår er at styresignala $\overline{CE}$ og $\overline{OE}$ er aktiverte tidleg nok, sjå under.
- Aktiveringsaksess, t_{ACE}.
  Det går ei viss maksimumstid frå ein aktiverer kretsen til nye data er gyldige og kan lesast av på utgangen. Som regel er $t_{ACE} = t_{AA}$ for eit minne.
- Opningstid, t_{OE}.
  Det går ei viss tid på å opna *tristate*-bufferet ut mot databussen. Denne tida er generelt og naturleg nok mykje kortare enn aksesstida for minnet.
- Fråkoblingstid, t_{CHZ}.
  Etter at minnet er deaktivert, går det ei viss tid før minnet er fråkobla databussen. Ein kan ikkje setja i gong nye dataoverføringar før dette skjer.

Sjølve innlåsinga av data, som ved lesing skjer i mikroprosessoren, er markert med raud pil i figuren.
Merk at denne markeringa bare er omtrentleg. Innlåsinga er synkronisert med klokkesignalet til prosessoren, og ein må ha djupare kjennskap til denne for å kunne plassera dette tidspunktet eksakt i forhold til tidspunkta for deaktiveringa av styresignala.

Ein sentral tidsparameter for eit minne er altså **aksesstida**. Minnet i dette eksemplet har ei aksesstid på 70 nsek, noko som er ikkje spesielt raskt. Det er likevel meir enn raskt nok som tilleggsminne i mange innebygde system.

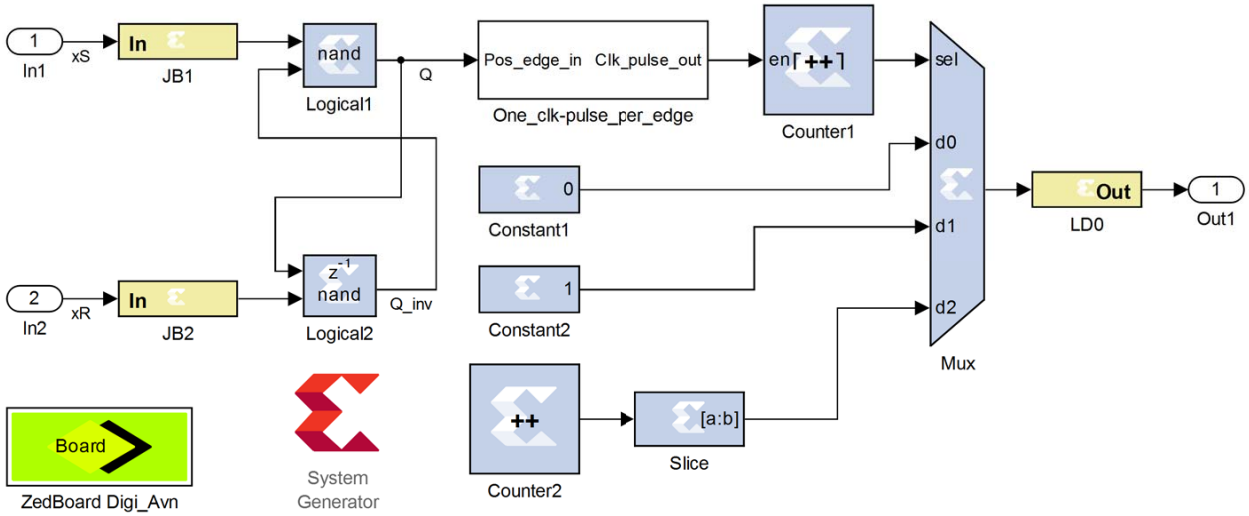
3.10.10 Systemblokkeksempel i FPGA: Sykkellyktstyring

Dei systemblokkene me har sett på til nå, har vore ein del av mikrokontrolleren vår.

Me skal nå sjå på ei systemblokk realisert i FPGA, nemleg sykkellyktstyringa frå eksempel 3.3 på side 111.
Sykkellyktstyringa vil bli realisert vhja. verktøyet *Simulink med System Generator* (SSG).

System Generator er som nemnt før, basert på *Matlab* og *Simulink* og er eigentleg laga for realisering av **signalbehandlingsmetodar** i FPGA. Verktøyet er likevel også fullt brukbart for realisering av generelle logikkfunksjonar i FPGA.
Eit lite utdrag frå biblioteka til SSG blei vist i figur 3.44 på side 172.

Sykkellyktstyringa kan realiserast vhja. SSG som vist i figur 3.102.
Plattformkortet brukt her er *ZedBoard*.



Figur 3.102: Sykkellyktstyring realisert i FPGA vhja. *Simulink med System Generator* (SSG).

Nokre detaljar om systemet:

- **Inngangssida** er basert på bruk av ein vekselbrytar. Oppkoblinga er som vist i figur 3.67 på side 207, men den SR-baserte avprellinga er altså nå realisert i FPGA.
- **Flankedetektor og flanketeljar:** For kvar positiv flanke i innsignalet vil flankedetektoren i den kvite boksen gi ein signalpuls til den etterfølgande teljaren[159] slik at teljeverdien aukar med 1, dvs. blir inkrementert.

[159]Eigentleg opnar flankedetektoren teljaren vhja. inngangen E i eit så kort intervall at teljaren bare får tid til å telja seg opp ein gong. Det er systemklokka på 100 MHz som driv teljaren, og E er altså aktivert i så kort tid at bare ein systemklokkeperiode slepp inn på teljaren.

Merk: Viss ein i dette verktøyet hadde ei teljarblokk med ein *CLK*-inngang, kunne ein ha kobla det avprella brytarsignalet inn på denne. Sidan teljarblokkene her blir drivne av systemklokka, blir det her ei litt spesiell realisering vhja. ein sjølvkonstruert flankedetektor.

Når det gjeld sjølve flanketeljaren ("Counter 1"), vil denne vera sett opp med ei breidde på 2 bit. I tillegg er maksimal teljeverdi lik 2.
Flanketeljaren er sjølve **tilstandsmaskinen** her og vil då veksla mellom tre ulike tilstandar.

- **Multipleksar:** Flanketeljarverdien på 2 bit og med maksimalverdi 2 vil gå inn på *Sel(ect)*-inngangane til multipleksaren. Verdien vil som kjent avgjera kva for eit av tre ulike signal som skal sleppast gjennom denne.
  Signala som held lyset av eller på, er realiserte vhja. kvar si *konstantverdiblokk* ("Constant").
  Det tredje signalet får lyset til å blinka.

- **Blinksignalgenerator:** Til å generera blinksignalet er det brukt ein **teljar**. Her passar det godt at teljaren blir driven av systemklokka.
  Teljaren er sett opp med ei bitbreidde som gir passeleg blinkefrekvens for den mest signifikante teljarbiten. MSb blir henta ut vhja. ei **bitplukkarblokk** ("Slice") og sendt vidare til mux-en.

- **Diode:** Ein lysdiode på plattformkortet er kobla til utgangssignalet frå multipleksaren.

Kapittel 4

Grensesnittkonstruksjon

Når ein skal konstruera eit innebygd system, dvs. ein datamaskin for ei styrings- og/eller overvakingsoppgåve, må ein grovt sett gjennom følgjande aktivitetar:

1. Utforming av detaljert spesifikasjon av systemet.
2. Vurdering og val av maskinvare- og programvareplattform.
3. Vurdering og val av maskinvare- kontra programvareomfang.
4. Vurdering av hyllevarebruk framfor eigenkonstruksjon.
5. Vurdering og val av komponentar/modular[1].
6. Konstruksjon av grensesnitt mellom komponentar/modular.
7. Utvikling av maskinvareprototype.
8. Programmering, testing og modifisering.
9. Klargjering for produksjon.

Fleire av aktivitetane kan gå delvis parallelt, og konstruksjonsprosessen går vanlegvis gjennom fleire iterasjonar. Under testing og modifikasjon må ein ofte tilbake og gjenta tidlegare aktivitetar.
Etter at protypekonstruksjonen er utført, går ein over i sjølve produksjonsfasen for systemet. Denne delen av produktutviklinga, som også vil vera iterativ, er ofte undervurdert.

[1] Her må ein i tillegg til komponentane sine eigenskapar ta omsyn til kor tilgjengelege komponentane er, kva støtte ein får frå produsentane i form av dokumentasjon mm., eigenskapane til aktuelle utviklingsverktøy og ikkje minst prisar.

Her skal ein ta utgangspunkt i at plattformen er kretskortet *STM32VLDiscovery*[2]. Me skal i dette kapitlet sjå på punkt 6 over, nemleg grensesnittkonstruksjon basert på denne plattformen.

Omgrepet **grensesnitt** blei innført i kapittel 2 og vist i figur 2.1. Med grensesnitt meiner ein her alt som skal til av maskin- og programvare for at mikroprosessoren skal kunne kommunisera med ein perifermodul. Maskinvaren eller elektronikken i eit grensesnitt er realisert vhja. fysiske logikkblokker eller digitale kretsar.
Ein har slik sett mange grensesnitt inne i ein mikrokontroller.

I tillegg har ein grensesnitt mellom ein mikrokontroller og perifermodular utanfor denne, dvs. **eksterne** modular.

Dette kapitlet omhandlar konstruksjon av grensesnitt generelt men med eksempel henta frå mikrokontrollerar baserte på ARM Cortex-M3 der mikrokontrolleren skal koblast til eksterne modular.

4.1 Hovudtypar av grensesnitt

Ein har to hovudtypar av grensesnitt:

- **Programmerte grensesnitt**
  I eit slikt grensesnitt realiserer ein kvart steg i ein overføringssyklus vhja. programkode. Dette er den vanlegaste typen grensesnitt mellom ein mikrokontroller og ein ekstern modul.
- **Dekoda grensesnitt**
  I eit dekoda grensesnitt vil mikroprosessoren sjølv generera ein overføringssyklus når han skal utføra ein lagre- (”str”) eller lastinstruksjon (”ldr”)[3]. Alle grensesnitta mellom prosessoren og perifermodulane inkl. minnemodulane inne i ein mikrokontroller er dekoda. Ein kan og ha dekoda grensenitt mot eksterne modular.

Ein vil nå først seia litt om overordna krav til grensesnitt og så gå nærare inn på dei to hovudtypane. Hovudfokuset i dette kapitlet vil vera på konstruksjon av programmerte grensesnitt. Derfor vil ein også gå gjennom eit detaljert konstruksjonseksempel for eit slikt grensesnitt, nemleg eit grensesnitt mot ein LCD-modul.

[2]Plattformkortet er vist i figur 1.9 på side 12.
[3]Desse blir også kalla skrive- og leseinstruksjonar.

4.2 Krav til eit grensesnitt

For at eit grensesnitt skal fungera, må følgjande vera tilfredsstilt:

1. **Signaltilkoblinga**
   Mikroprosessoren eller -kontrolleren må ha signal som kan koblast direkte til perifermodulen eller evt. gjennom ein mellomliggjande modul. Slike modular blir ofte realiserte vhja. programmerbar logikk.
2. **Krava til støymarginar og drivekapasitet**
   Viss signalmottakaren skal oppfatta rett logisk nivå, må sendaren halda rett spenningsnivå på signalutgangen. I tillegg må sendaren ha tilstrekkjeleg kapasitet, dvs. han må kunne forsyna signallinjene med tilstrekkjeleg straum.
3. **Tidskrava ved dataoverføring**
   Under lesing og skriving må dei ulike signala gå i rett sekvens og med visse tidsluker seg i mellom for at data skal kunne mottakast feilfritt.

I kapittel 3.4.4.2 på side 121 såg me på korleis ein kan finna ut om krava knytt til punkt 2 er tilfredsstilte. Dette punktet blir derfor ikkje tatt med i konstruksjonseksemplet på programmerte grensesnitt her.

Før me går laus på detaljane i programmerte grensesnitt, skal me sjå litt meir på kva eit dekoda grensesnitt er.

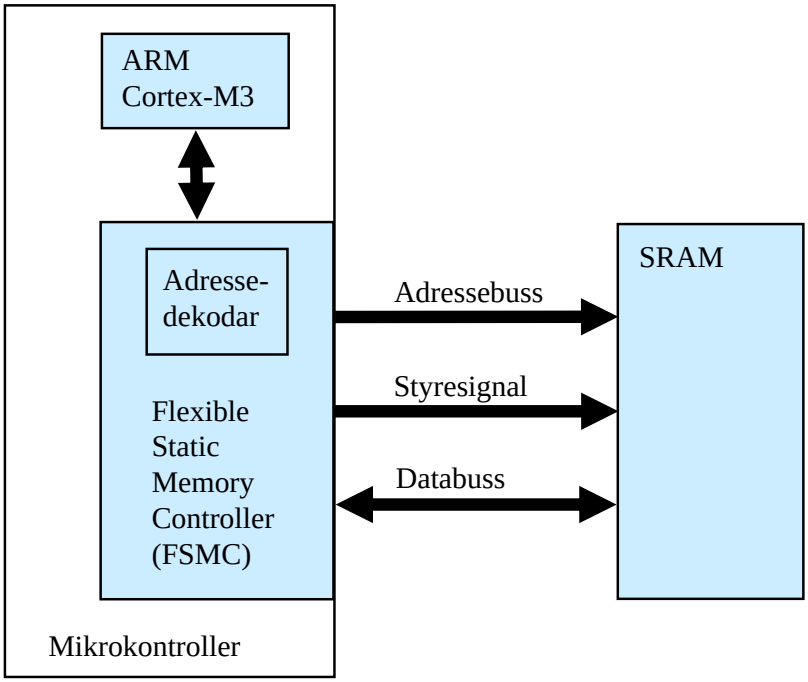
4.3 Litt om dekoda grensesnitt

Me tar utgangspunkt i eit grensesnitteksempel, nemleg eit eksternt SRAM-minne kobla til ein mikrokontroller basert på kjernen ARM Cortex-M3.

Som vist i minnekartet for ARM Cortex-M3 i tabell 2.2 på side 39, er det sett av eit adresseområde for eksterne minnemodular. Viss f.eks. det interne dataminnet i ein mikrokontroller ikkje er tilstrekkjeleg for det innebygde systemet som skal utviklast, kan ein henga på ein ekstern SRAM-modul.

For å kunne realisera eit slikt grensesnitt, må ein velja ein mikrokontroller som har ein eigen perifermodul for dette. I mikrokontrollerar frå STMicroelectronics baserte på ARM Cortex-M3 heiter denne perifermodulen "Flexible Static Memory Controller", **FSMC**.

Mikrokontrolleren vår har ikkje ein slik modul. Viss ein treng FSMC, kan ein f.eks. velja ein mikrokontroller frå familien STM32F103, sjå databladet [22].
Figur 4.1 viser eit dekoda grensesnitt mellom ein slik mikrokontroller og eit eksternt SRAM-minne.



Figur 4.1: Eksempel på eit dekoda grensesnitt.

Som ein ser, har kontrolleren for eksterne minne, FSMC, eit bussgrensesnitt mot SRAM-kretsen.

I dekoda grensesnitt vil også **adressedekodaren** vera ein sentral komponent og er ein del av FSMC-modulen.

Adressedekodaren lyttar på den interne adressebussen. Når f.eks. ARM-prosessoren vil gjera ein aksess mot den eksterne komponenten, dvs. når adressene på bussen er innanfor adresseområdet til den eksterne komponenten, køyrer FSMC-modulen automatisk ein overføringssyklus mot SRAM-minnet.
Slike minnekontrollerar inneheld altså ein eigen **tilstandsmaskin** som realiserer dataoverføringssyklar til og frå det eksterne minnet.

For å illustrera dette kan ein sjå på følgjande høgnivåprogram:

```
  uint32_t m // Global variabel.
  ...
void main(void) {
  ...
  m++;
  ....
}
```

Ein tenkjer seg at variabelen m er plassert i det eksterne SRAM-minnet.

Viss nå
- m blir plassert i register *r5* under inkrementeringa og
- adressa til m er plassert i register *r6*

så vil ei skriving av den nye m-verdien tilbake til det eksterne dataminnet (SRAM) skje vhja. instruksjonen under vist på assemblyformat:

```
str r5, [r6, #0]
```

Prosessoren vil utføra denne instruksjonen ved å generera ein automatisk skrivesyklus på den interne systembussen i mikrokontrolleren der ordet i register *r5* blir overført til adressa gitt av innhaldet i *r6*. Når FSMC oppfattar at adresseverdien på bussen er innanfor adresseområdet til den eksterne komponenten, køyrer så modulen automatisk ein skrivesyklus på den eksterne bussen som flyttar dataverdien frå den interne systembussen og over til det eksterne SRAM-minnet.

Når grensesnittet er dekoda, vil ein også i nokre mikrokontrollerar kunne ha **eksterne programminne** f.eks. i form av ein Flash-modul.
Prosessoren vil som kjent under instruksjonshenting leggja ut ein programteljarverdi (PC) på den interne adressebussen. Viss denne verdien høyrer til adresseområdet for det eksterne Flash-minnet, vil minnekontrolleren FSMC automatisk utføra ein leseoperasjon frå dette minnet. Etter innlesing vil minnekontrolleren flytta den aktuelle instruksjonen over på den interne systembussen der prosessoren kan låsa denne inn i hentebufferet.

4.4 Programmert grensesnitt

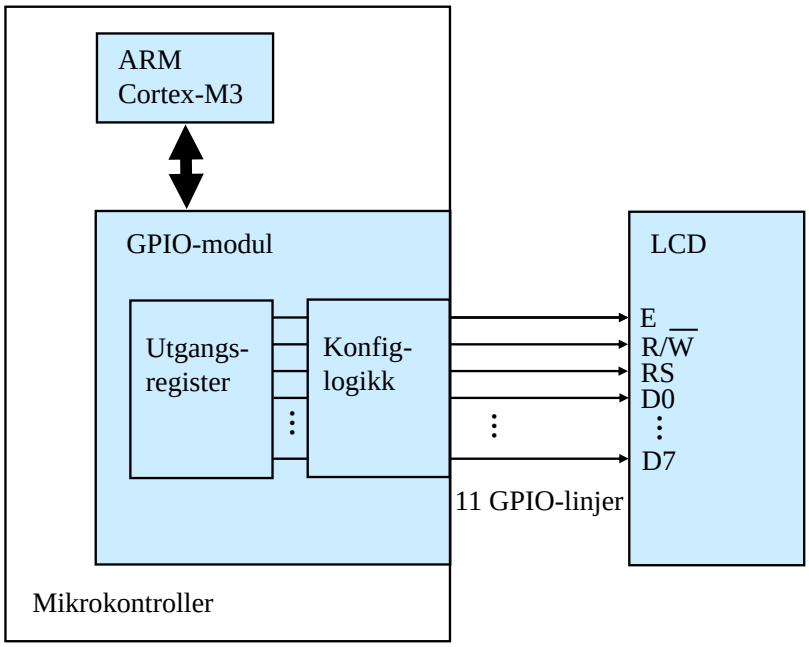
Dette kapitlet tar utgangspunkt i eit grensesnitteksempel, nemleg LCD-modulen[4] MDLS16265, [25], kobla mot mikrokontrolleren vår, STM32F100RB.
Først ser ein på det programmerte grensesnittet generelt og så blir dette realisert i detalj.

4.4.1 Innleiing

Figur 4.2 viser eit eksempel på eit programmert grensesnitt mellom ein mikrokontroller basert på ARM Cortex-M3 og ein LCD-modul.
Som vist, realiserer ein programmerte grensesnitt vhja. GPIO-modular[5].

I eksemplet her skal heile grensesnittet realiserast vhja. ein GPIO-modul, mens i andre

[4]Ein LCD-modul er bygd opp av eit skjermelement og ein kontrollerkrets som styrer alle punkta på skjermen.
[5]"General Purpose Input/Output"-modular, også kalla parallellportar.



Figur 4.2: Eksempel på eit programmert grensesnitt.

tilfelle må ein bruka fleire slike perifermodular.

Signala inn og ut av ein GPIO-modul må generelt tilordnast kvar sin bit i inngangsregisteret (IDR) eller utgangsregisteret (ODR) i modulane.
I grensesnittet som skal lagast her, går alle signala ut til LCD-modulen. Ein kan då styra alle signala ved å skriva passande data til utgangsregisteret i ein GPIO-modul.

Det er i figuren også vist at signala går gjennom konfigurasjonslogikk. I dette tilfellet skal altså alle signala konfigurerast som utgangar.
Konfigurasjonslogikken vil også innehalda **drivartrinn**[6] som gir standardiserte spenningsnivå og tilstrekkjeleg drivstraum ut mot den eksterne modulen, her LCD-modulen.

Dei 11 LCD-signala skal her knyttast til kvar sin bit i utgangsregisteret til GPIO-modulen som her skal vera GPIOB.
Funksjonen til signala er som følgjer:

- E(nable) er aktiveringssignal. $E = 1$ gir aktivering av skjermen.
- R(ead)/$\overline{W}$(rite) er eit retningssignal. $R/\overline{W} = 0$ gir skriving mot skjermen.
- R(egister)S(elect) er eit adresseringssignal. $RS = 1$ ved overføring av teikn og 0 ved kommandooverføring.
  Ein kommando kan f.eks. vera å sletta heile skjerminnhaldet, eller å flytta til ein viss

[6]Når det gjeld eventuelle inn-signal, vil dei gå gjennom eit inngangsbuffer i konfigurasjonslogikken før verdiane hamnar i inngangsregisteret.

posisjon. Alle kommandoar og dataoverføringar med tilhøyrande *RS*-verdi er vist i tabell 5 i databladet for sjølve kontrollerkretsen i LCD-modulen, [24].

- *D*(ata) 0 - 7 inneheld teiknet eller kommandoen som skal overførast.

I eit programmert grensesnitt legg ein vhja. programkode verdi etter verdi inn i utgangsregistra til GPIO-modulane. Slik vil signala ut til den eksterne modulen gå i ein tidssekvens som spesifisert i databladet for den eksterne modulen som er tilknytta GPIO-modulane.

4.4.2 Framgangsmåte for realisering av eit programmert grensesnitt

I tråd med punkt 1 - 3 i kapittel 4.2 kan ein realisera eit programmert grensesnitt mot ein ekstern modul ved å gå gjennom fylgjande punkt:

1. Tilordna signala kvar sin bit i inngangsregisteret (IDR) eller utgangsregisteret (ODR) i GPIO-modulane.

2. Finn støymarginar og krav til drivekapasitet for signala. Iverksett nødvendige tiltak. Revurder eventuelt valet av ekstern modul.

3.1. Studer tidsdiagramma for dei overføringssyklane ein skal realisera, og finn ut kor mange signaltilstandar eller steg desse må delast opp i.

3.2. Lag pseudokode for overføringssyklane.

3.3. Lag bitmønster for aktivering og deaktivering av dei ulike styresignala.

3.4. Lag detaljert programkode.

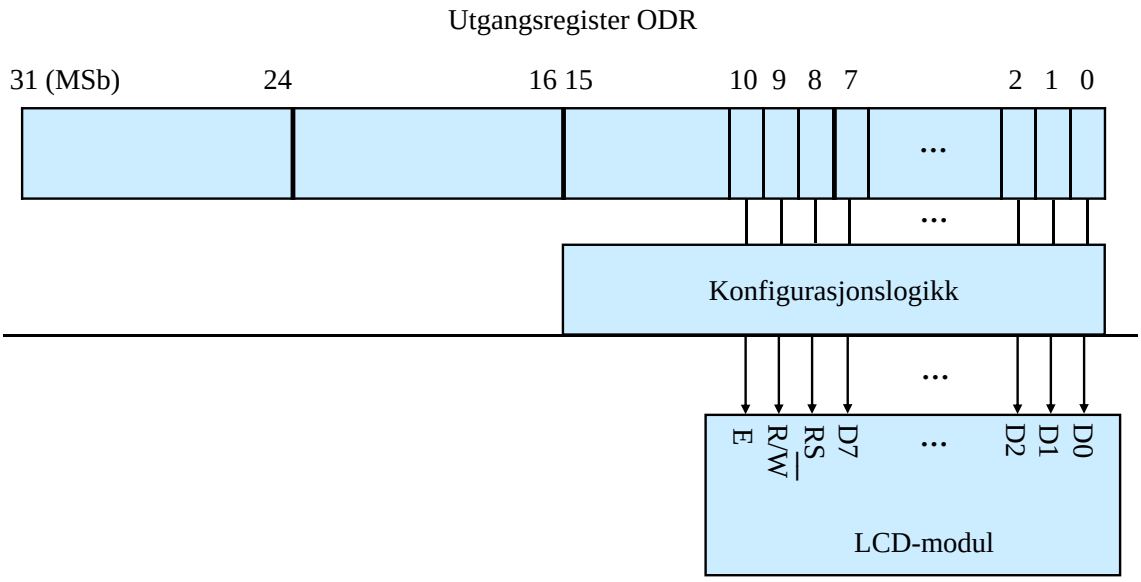
Punkt 3 i kapittel 4.2 er altså delt opp i fire delar her.

I det følgjande eksemplet skal ein sjå i detalj på korleis ein kan realisera skriving på ein LCD-skjerm som vist i figur 4.2. For å få til dette må ein realisera skrivesyklar for teikn og kommandoar til LCD-skjermen.

4.4.3 Realisering av LCD-grensesnittet, punkt 1: Signaltilkobling

Signaltilkobling skjer altså ved å tilordna bit i utgangsregisteret ODR til dei ulike signala. Dette er punkt 1 ved realisering av eit programmert grensesnitt.

Dei 11 LCD-signala kan knyttast til kvar sin bit i dataregisteret til GPIO-en som vist i figur 4.3. I eksemplet her skal me altså bruka GPIOB.



Figur 4.3: Tilordning av bit i GPIO-en sitt utgangsregister ODR.

Rekkjefølgja av signala er ikkje kritisk, men det kan her vera **lurt** å la datadelen okkupera den minst signifikante delen av registeret. Då unngår ein skiftoperasjonar når data, dvs. teikn eller kommandoar, skal leggjast inn i registeret. Meir om dette kjem under punkta 3.1-4.

4.4.4 Realisering av LCD-grensesnittet, punkt 2: Støymarginar og drivekapasitet

Dette punktet er som nemnt tidlegare, behandla i kapittel 3.4.4.2.

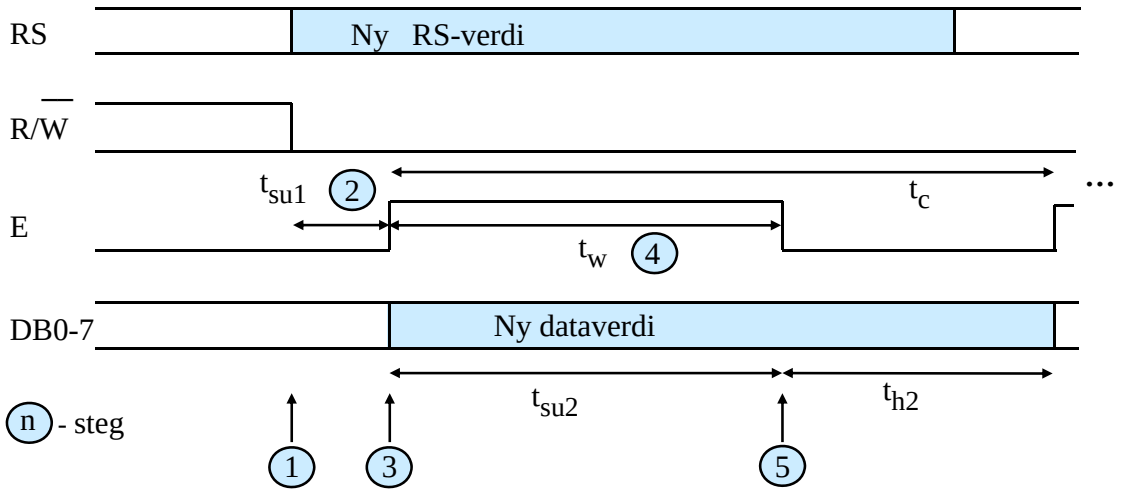
4.4.5 Realisering av LCD-grensesnittet, punkt 3: Tidskrav

Dette punktet kan som vist i kapittel 4.4.2, delast opp i 4 passande underpunkt ved realisering av programmerte grensesnitt. Punkta blir gjennomgått i det følgjande.

4.4.5.1 Punkt 3.1: Tidsdiagram og oppdeling i steg

For å finna ut korleis ein skal få til ei overføring, må ein studera **tidsdiagrammet** for denne. Slike diagram finn ein i databladet for sjølve kontrollerkretsen i LCD-modulen, [24]. Eit tidsdiagram for skriveoperasjonar mot LCD-modulen er vist i figur 6 på nest siste side i databladet. Diagrammet fortel oss kva krav som må oppfyllast for at modulen skal greie å motta data på rett måte.

I eit datablad vil det i tillegg vera tabellar som viser typiske, samt maksimums- og minimumsverdiar for dei tidene som er med i diagrammet.
Figur 4.4 viser eit forenkla diagram for skrivesyklusen i figur 6 i databladet, der bare det som er viktig for overføringa vår, er med[7].



Figur 4.4: Forenkla tidsdiagram for ein skrivesyklus.

Ei overføring av eit teikn til LCD-en skjer ved at signala som vist i figuren, går i ein viss sekvens. Ved å dela opp ei overføring i passande **steg**, kan ein oppfylla tidskrava som LCD-modulen set. Eit steg representerer ein viss signalkombinasjon eller -tilstand.
I dette tilfellet bør ein som vist i figuren, i utgangspunktet ha fem steg. I databladet for LCD-en finn ein følgjande tidskrav når forsyningsspenninga V_{DD} ligg mellom 2.7 og 4.5 Volt:

$t_{su1} \geq 60$ n(ano)s(ekund) - oppsettingstid for adresse, dvs. RS,
$t_w \geq 400$ ns - E-pulsbreidde, dvs. opningstid før innlåsing av data og
$t_{su2} \geq 140$ ns - oppsettingstid for data.

Med fem steg her vil ein både kunne gi signala rett sekvens og oppfylla tidskrava. Dette kjem klarare fram ved å studera pseudokoden i neste punkt.

Merk at det også er minimumskrav knytt til syklustida t_c. Det går altså ei viss tid før **inngangstrinnet** til LCD-en er i stand til ta imot ei ny overføring, for eksempel av ein kommando. I tilfellet her seier databladet at ein ny aksess mot LCD-modulen kan tidlegast skje etter $t_c = 1400$ ns $= 1.4$ μs i frå siste positive E-flanke.

Når så eit teikn eller ein kommando er overført og har kome seg inn døra i LCD-modulen, må LCD-modulen **prosessera** denne infomasjonen. LCD-modulen treng altså ei viss tid på å skriva eit teikn ut på skjermen eller å utføra ein kommando. Desse tidene er gitt i tabell 5 i [24].

[7]Eit diagram i eit datablad innheld svært mange opplysningar. Det er ei utfordring å sila ut det vesentlege.

F.eks. bruker LCD-modulen minst 43 μs på å skriva eit teikn til minnet[8] (og samstundes ut på skjermen). Skrivinga skjer til den skjermposisjonen som peikaren (*cursor*) har.
Viss neste teikn skal skrivast til ein heilt annan posisjon, må ein flytta peikaren ved å overføra kommandoen "Set DDRAM Address". LCD-modulen bruker som vist i tabellen, minst 39 μs på å utføra denne kommandoen.

Prosesseringa av eit teikn eller ein kommando inne i LCD-modulen tar altså mykje lenger tid enn sjølve overføringa over grensesnittet mot modulen.
Då me ikkje greier å oppfatta informasjon som blir presentert i kortare intervall enn typisk 100 ms, vil LCD-modulen vera meir enn rask nok, dette sjølv om heile skjerminnhaldet skulle skiftast ut med høgaste brukbare oppdateringsfrekvens.

4.4.5.2 Punkt 3.2: Pseudokode for overføringsprogrammet

Pseudokoden i fem steg for overføring av eit teikn, dvs. ein ASCII-verdi, blir som følgjer:

1. Skriv eit bitmønster til utgangsregister *ODR_B* der $E = 0$, $RS = 1$ og $R/\overline{W} = 0$.
2. Vent minst like lenge som kravet til oppsettingstid ("setup") for RS og $R/\overline{W}$.
   Merk: Då kravet her er så lite, dvs. 60 nsek som er litt i overkant av 1 klokkeperiode, kan ein sløyfa dette steget. Kvar C-instruksjon vil ha lengre utføringstid enn dette, slik at kravet blir automatisk oppfylt. Jfr. her målingane som blir gjort av utføringstid for ei enkel *while*-løkkje i laboratorieøving 2 i Datamaskinarkitektur.
3. Skriv eit bitmønster til *ODR_B* der $E = 1$, $RS = 1$ og $R/\overline{W} = 0$
   og $D0 - 7 =$ *ASCII-kode*.
4. Vent minst like lenge som kravet til pulsbreidde ("pulse width") for E[9].
5. Skriv eit mønster der $E = 0$[10], $RS = 1$ og $R/\overline{W} = 0$
   og $D0 - 7 =$ *ASCII-kode*.

Pseudokoden for overføring av ein kommando blir tilsvarande. Einaste forskjellane er at $RS = 0$ og at $D0 - 7 =$ *kommando*.
Denne pseudokoden kan lett programmerast i C, men først skal me sjå på korleis ein kan realisera aktivering og deaktivering av dei ulike styresignala.

[8] Dette heiter "Write Data to RAM" i tabell 5 i databladet.

[9] Ein vil då samstundes også oppfylla kravet til oppsettingstid for data, som er mindre enn pulsbreiddekravet for E. Grunnen er altså at data blir plassert ut på bussen i punkt 3.

[10] Teiknet blir lasta inn i skjermen på den negative flanken av E. I figur 4.3 er også haldetida t_{h2} vist. Data må altså liggja ei stund på bussen etter at E har gått låg for at dei skal bli oppfatta rett. Då ein her ikkje endrar databussverdien før ved neste overføring, vil kravet til haldetid vera oppfylt med svært god margin.

4.4.5.3 Punkt 3.3: Bitmønster for aktivering og deaktivering av styresignal

Basert på tilordninga vist i figur 4.3, kan ein setja opp bitmønster som vil aktivera[11] og deaktivera styresignala. Ein innfører her følgjande definisjonar:

LCD_Enable	= 0100 0000 0000 binært = 0x400
$LCD\_Disable$	= 0000 0000 0000 binært = 0x000
LCD_Read	= 0010 0000 0000 binært = 0x200
LCD_Write	= 0000 0000 0000 binært = 0x000
LCD_RS_data	= 0001 0000 0000 binært = 0x100
$LCD\_RS\_kommando$	= 0000 0000 0000 binært = 0x000

F.eks. kan det symbolske namnet LCD_Enable forståast som det bitmønsteret som set signalet E til '1'.

I deklarasjonsfilene til eit programvareprosjekt for systemet vil desse definisjonane sjå slik ut:

#define LCD_Enable	0x400
#define $LCD\_Disable$	0x000
#define LCD_Read	0x200
#define LCD_Write	0x000
#define LCD_RS_data	0x100
#define $LCD\_RS\_kommando$	0x000

Merk at ein her ser på styresignala kvar for seg. Ved å bruka **eller-operasjonar** i programkoden, kan ein få til eit vilkårleg mønster for styresignala, sjå neste punkt.

[11] For eit aktivt høgt signal som f.eks. E vil aktivering vera det same som å setja signalverdien til '1', dvs. høg.

4.4.5.4 Punkt 3.4: Programkode

Basert på CMSIS-standarden kan ein nå lett laga instruksjonar i C for å leggja inn bitmønster i utgangsregisteret for GPIO-modulen.

I tillegg treng ein metodar som realiserer venting i eit viss tidsintervall slik at ein oppfyller tidskrava ved overføringa. Ein presis metode vil ofte vera taimerbasert, men her er det tilstrekkjeleg med metodar som inneheld ei lita venteløkkje der startverdien bestemmer tidsintervallet[12].

Me går nå ut frå at det er GPIO-modul B som skal brukast her, og at ingen andre metodar i systemet brukar denne porten.
Me går også ut frå at porten er konfigurert vhja. ein eigen oppstartsmetode.

Basert på pseudokoden og bitmønstra som er laga, blir då programkoden for å overføra teiknet *teikn* til LCD-skjermen, følgjande:

```
void skrivesyklys_data( uint8_t teikn)   {

  GPIOB->ODR = (LCD_Disable | LCD_Write | LCD_RS_data);   // Steg 1

  vent_60ns();  // Steg 2. Dette steget kan nok sløyfast, sjå merknaden under.

  GPIOB->ODR = (LCD_Enable | LCD_Write | LCD_RS_data | teikn);   // Steg 3

  vent_400ns(); // Steg 4. Dette vil også oppfylla kravet til oppsett av data.

  GPIOB->ODR = (LCD_Disable | LCD_Write | LCD_RS_data | teikn);   // Steg 5
}
```

Denne programkoden er som ein ser, laga som ein metode for skriving av teikn der ein overfører teiknet i metodekallet. Tilsvarande kan ein laga ein metode for skriving av kommandoar, der altså $RS = 0$.

Venting blir altså realisert som vist vhja. metodane over.
Merk: Det er med den klokkefrekvensen som er i bruk her, 24 MHz, ikkje behov for å leggja inn ekstra venting for tidskrav på mindre enn 100 nsek, då sjølve assemblykoden for setjing av bit i dataregisteret vil gi nok venting i slike tilfelle. Tida det tar ARM Cortex-M3 å utføra programinstruksjonane frå steg 1 til steg 3 utan ekstra venting, vil overstiga tidskravet på 60 nsek her.

[12] Dette kan fininnstillast vhja. oscilloskop-målingar. Det er også viktig å hugsa at viss ein endrar optimaliseringsnivå, så vil utføringstida for venteløkkja endrast!

Når ein skal bruka desse metodane for å skriva f.eks. ein tekststreng til skjermen, er det som nemnt før viktig å hugsa at skjermen etter ein skrivesyklus treng tid på å få teiknet ut på skjermen. Skjermkontrolleren er derfor ikkje mottakeleg for nye teikn før etter ei viss tid.
Tabell 5 i databladet for kontrolleren, [24], inneheld som nemnt alle kommandoar og dataoverføringar med tilhøyrande *RS*-verdi samt *utføringstid*.
Som vist i tabellen, treng kontrolleren typisk rundt 40 μsek på å utføra ein vanleg kommando eller ei skriva eit teikn til skjermen. Nokre kommandoar krev mykje lenger tid.

Med dette er realiseringa av grensesnittet fullført.

4.5 Litt om val av type grensesnitt

Ein har altså to hovudtypar av **parallelle** grensesnitt, nemleg **programmerte** og **dekoda** grensesnitt.

Programmerte grensesnitt er som vist enkle å konstruera, men gir pga. meir programkode generelt mykje tregare dataoverføring. Dette gjer at ein vanlegvis bruker dekoda grensesnitt mot eksterne minne og andre komponentar der ein ønskjer raske overføringar. Viss mikroprosessoren i tillegg skal kunne køyra program frå eit eksternt minne, **må** grensesnittet vera dekoda.

Programmerte grensesnitt blir brukt mot trege komponentar eller komponentar der ein ikkje er avhengig av raske overføringar. Dette kan f.eks. vera skjermar eller tastatur.

Mange mindre mikrokontrollerar har ikkje eit eksternt bussgrensesnitt, noko som gjer programmerte grensesnitt til det einaste moglege.
Programmerte grensesnitt dominerer nok som parallelt grensesnitt mellom mikrokontrollerar og eksterne komponentar, men inne i mikrokontrolleren og i mikroprosessorbaserte system som f.eks. ein PC, er alle grensesnitta dekoda.

Når det gjeld grensesnitt under eitt, har ein i lengre tid sett ein aukande bruk av raske **serielle** framfor parallelle overføringar mellom mikrokontrollerar og eksterne komponentar. Dette gir innsparingar i pinnetal og banetal på eit kretskort og gjer at ein kan redusera dimensjonar og kostnader. Behovet for kabling mellom system blir også sterkt redusert. Tilgjengelege perifermodular i mikrokontrollerar for slike serielle standardar som f.eks. SPI, I2C, CAN, USB og Ethernett, og eit aukande tilfang av kretsar med slike grensesnitt, forsterkar denne utviklinga. Det er likevel mange grensesnitt som ut frå ulike omsyn framleis vil vera parallelle, og av desse vil ein ha både dekoda og programmerte grensesnitt.

Referansar

[1] Morten Tengesdal: "Konstruksjon av små innebygde system basert på mjukprosessor". Notater fra UiS nr. 35, 9. mars 2012.

[2] Geoffrey Brown: "Discovering the STM32 Microcontroller". Utgåve: 5. juni 2016. Open lærebok frå *www.cs.indiana.edu/ geobrown/book.pdf*. Lasta ned sist 6. februar 2018.

[3] Joseph Yiu: "The definitive guide to the ARM Cortex-M3 and Cortex-M4 processors". Newnes, 3. utg., 2014.

[4] Joseph Yiu: "The definitive guide to the ARM Cortex-M3 processor". Newnes, 2. utg., 2010.

[5] Joseph Yiu: "The definitive guide to the ARM Cortex-M0". Newnes, 1. utg., 2011.

[6] Rob Toulson og Tim Wilmshurst: "Fast and effective embedded systems design - Applying the ARM mbed". Newnes, 2012.

[7] Nettsidene til ARM sitt informasjonssenter: *http://infocenter.arm.com/help/index.jsp*. Brukt bl.a. 27. mars 2014.

[8] David Patterson og John L. Hennesy: "Computer Organization and Design, The hardware/software interface". ARM edition 2010, Elsevier.

[9] Claude E. Shannon: "A Symbolic Analysis of Relay and Switching Circuits". Frå "Claude Elwood Shannon Collected Papers". Redigert av N. J. A. Sloane og AAron D. Wyner, AT & T Bell Laboratories. IEEE Press.

[10] Artikkel av Alexander Magoun og Paul Israel på internett: *http://theinstitute.ieee.org/tech-history/technology-history/did-you-know-edison-coined-the-term-bug*. Lasta ned sist 7. februar 2018.

[11] Jon Fidjeland: "Xilinx System Generator for ZedBoard". Institutt for data- og elektroteknikk - UiS, 7. oktober 2013.

[12] Trygve Eftestøl: "DIGITALTEKNIKK - Notater til BIE220 Digitalteknikk". Med handteikningar laga av Tom Ryen. UiS, 23.oktober 2009.

[13] M. Morris Mano og Michael D. Ciletti: "Digital Design". Pearson 2007, 4.utgåve.

[14] Thomas L. Floyd: "Digital fundamentals". Pearson, 9.utg., 2006.

[15] "UM0919 User Manual STM32 Value Line Discovery". Juni 2011, DocID 17217 Rev 2. STMicroelectronics, *www.st.com*.

[16] "STM32F100x4 STM32F100x6 STM32F100x8 STM32F100xB". Datablad, juni 2012, DocID 16455 Rev 7. STMicroelectronics, *www.st.com*.

[17] "RM0041 Reference manual STM32F100xx advanced ARM-based 32-bit MCUs". Oktober 2010, DocID 16188 Rev 3. STMicroelectronics, *www.st.com*.

[18] "PM0056 Programming manual STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual". Mars 2011, DocID 15491 Rev 4. STMicroelectronics, *www.st.com*.

[19] "STM32 32-bit ARM Cortex MCUs Releasing your creativity". STMicroelectronics, *www.st.com*.

[20] "Cortex-M3 Revision r2p1 Technical Reference Manual". Copyright 2005-2008, 2010 ARM Limited. ARM DDI 0337I (ID072410), *www.arm.com*.

[21] "ARM v7-M Architecture Reference Manual - Errata Markup". Copyright 2006-2010, 2010 ARM Limited. ARM DDI 0403D Errata 2010_Q3 (ID100710), *www.arm.com*.

[22] "STM32F103xC STM32F103xD STM32F100xE". Datablad, april 2011, DocID 14611 Rev 8. STMicroelectronics, *www.st.com*.

[23] "UM232R USB - Serial UART Development Module Datasheet". Document Reference No.: FT_000051, versjon 1.1, 2011-11-25. Future Technology Devices International Ltd, *www.ftdichip.com*.

[24] "KS0070B 16COM/80Seg driver & controller for dot matrix LCD". Samsung El.nics.

[25] "Datasheet Varitronix MDLS 16265". Avnet Embedded 07/2010, *www.avnet-embedded.eu*.

[26] "Data sheet 74HC/HCT85 4-bit magnitude comparator". Philips Semiconductors, desember 1990. Søkjeord "74hc85 nxp".

[27] "AMIC A623308A Series 8K X 8 BIT CMOS SRAM". AMIC Technology, Corp., juli 2006, Version 1.0.

[28] Tim Wilmshurst: "An introduction to the design of small-scale embedded systems". Palgrave, 2001.

[29] "MicroBlaze processor reference guide, Embedded development kit EDK 10.1". Xilinx, UG081 (v9.3), 2008.

[30] "Spartan-3 FPGA Family: Complete Data Sheet". Xilinx, DS099, 17.januar 2005.

[31] "IS61LV25616AL 256K x 16 High speed asynchronous CMOS Static RAM with 3.3V supply". Integrated Silicon Solution, Inc., Rev.D 03/11/05.

[32] "SN54ALS04, SN54AS04, SN74ALS04B, SN74AS04 Hex Inverters". SDAS063B-April 1982-Revised December 1994, Texas Instruments.

Vedlegg A

Litt om talsystem, aritmetikk og koding

Innhaldet i dette kapitlet er basert på [12][1].

A.1 Talsystem

A.1.1 Titalssystemet

I kapittel 1.2.1 på side 10 blei det vist med eit eksempel kva eit posisjonstalsystem er. Eit eksempel til frå titalssystemet, dvs. med desimaltal, er vist under.

$$\begin{aligned} 328,5_{10} &= 3 \cdot 100 + 2 \cdot 10 + 8 \cdot 1 + 5 \cdot 0.1 \\ &= 3 \cdot \underset{\uparrow}{10}^{2} + 2 \cdot \underset{\uparrow}{10}^{1} + 8 \cdot \underset{\uparrow}{10}^{0} + 5 \cdot \underset{\uparrow}{10}^{-1} \end{aligned} \tag{1.1}$$

Talet 10 er **grunntal/base/radix** i titalssystemet.

[1] Notat av Trygve Eftestøl med handteikna figurar av Tom Ryen.

A.1.2 Generelt om talsystem

I eit system med grunntal r vil koeffisientane a_i, dvs. siffera, multipliserast med potensar av r slik at talet

$$\underset{\substack{\uparrow \\ MSD}}{a_{n-1}}\ a_{n-2} \ldots a_2\ a_1\ a_0\ a_{-1}\ a_{-2} \ldots \underset{\substack{\uparrow \\ LSD}}{a_{-m}} \tag{1.2}$$

tilsvarer

$$\underset{\substack{\uparrow \\ MSD}}{a_{n-1} \cdot r^{n-1}} + a_{n-1} \cdot r^{n-2} + \ldots + a_2 \cdot r^2 + a_1 \cdot r^1 + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} \ldots \underset{\substack{\uparrow \\ LSD}}{a_{-m} \cdot r^{-m}} \tag{1.3}$$

eller meir kompakt

$$\sum_{i=-m}^{n-1} a_i \cdot r^i \tag{1.4}$$

der
- MSD er det mest signifikante og LSD det minst signifikante siffer ("digit"),
- n er talet på siffer til venstre for komma-punktet, og
- m er talet på siffer til høgre for komma.

Då sifferet eller koeffisienten nærast komma på heiltalssida er a_0, blir MSD $= a_{n-1}$.
Eksemplet i (1.1) viser også dette. Talet på siffer på heiltalssida er $n = 3$, og verdien av MSD er $a_{n-1} \cdot r^{n-1} = a_2 \cdot r^2 = 3 \cdot 100$.

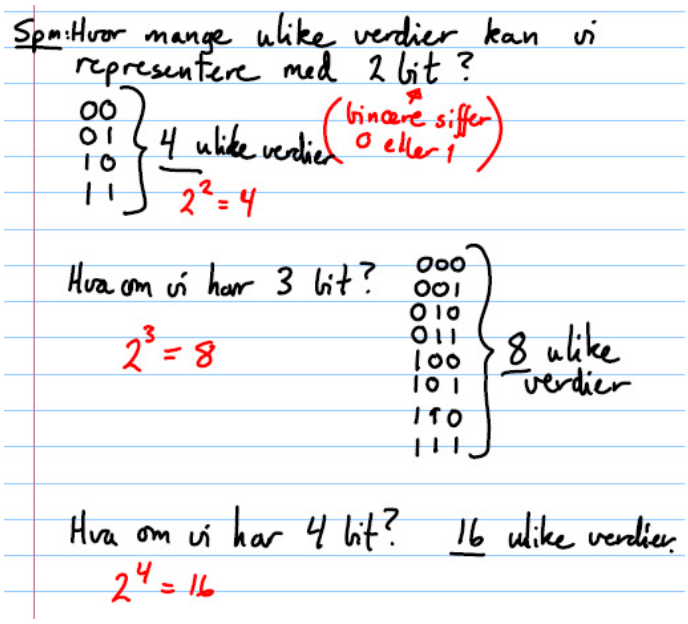
A.1.3 Totalssystemet

For totalssystemet, også kalla det **binære** talsystemet, har ein at $r = 2$. Her blir koeffisientane a_i kalla *bit* og kan bare ha to moglege verdiar: 0 og 1.
Eit eksempel er gitt under.

Eksempel A.1.

$$\begin{aligned} 1011,01\ \underset{\substack{\uparrow \\ r=2}}{_2} &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + +1 \cdot 2^{-2} \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned} \tag{1.5}$$

I eit binært tal med n bit, kan ein representera 2^n ulike verdiar som illustrert i figur A.1 for to og tre bit.



Figur A.1: Samanheng mellom talet på bit i eit bitmønster og kor mange verdiar som kan representerast. (Ref.: Tom Ryen, [12].)

Tabell A.1 viser samanhengen mellom bittalet og talet på ulike verdiar for $n = 1, \ldots, 32$.

n	2^n	n	2^n	n	2^n	n	2^n
0	1	8	256	16	65.536	24	16.777.216
1	2	9	512	17	131.072	25	33.554.432
2	4	10	1024	18	262.144	26	67.108.864
3	8	11	2048	19	524.288	27	134.217.728
4	16	12	4096	20	1.048.576	28	268.435.456
5	32	13	8192	21	2.097.152	29	536.870.912
6	64	14	16.384	22	4.194.304	30	1.073.741.824
7	128	15	32.768	23	8.388.608	31	2.147.483.648
						32	4.294.967.296

Tabell A.1: Kor mange tal som kan representerast vhja. n bit.

A.1.4 Talområde for binære tal

Talområdet for eit tal er gitt av:

$$S = [N_{min}, N_{max}]$$

der
- S er talområdet eller spennet,
- N_{min} er den minimale talverdien og
- N_{max} er den maksimale talverdien i talområdet.

Som me har sett, er talet på ulike verdiar som kan representerast, lik 2^n for eit tal på n bit. Når det gjeld det resulterande talområdet, vil det vera avhengig av om ein skal kunne representera både positive og negative verdiar eller om forteiknet skal vera likt for alle verdiane.
Det første talområdet er då **bipolart** og det sistnemnte **unipolart**[2].

A.1.4.1 Unipolare tal

For eit unipolart binærtal med n bit, er N_{min} gitt av at alle bitane er 0, dvs. $N_{min} = 0$.

Maksimalverdien N_{max} er gitt av at alle bitane er 1. Basert på summeformelen for ei geometrisk rekkje[3] får ein då følgjande:

$$N_{max} = 1 \cdot 2^{n-1} + 1 \cdot 2^{n-2} + \ldots + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^n - 1$$

For eksempel vil ein for $n = 2$ og $n = 3$ ha maksimalverdiane 3 og 7, noko ein ser greitt ut frå eksempla i figur A.1.

Talområdet for unipolare tal blir altså $S = [N_{min}, N_{max}] = [0, 2^n - 1]$.

A.1.4.2 Bipolare tal

Me skal her nøya oss med å seia at ved representasjon av bipolare tal vil ein bruka halve området på negative verdiar og halve på positive. Ein reknar vanlegvis 0 som eit positivt tal og får såleis følgjande talområde:

$$S = [N_{min}, N_{max}] = [-\frac{1}{2} \cdot 2^n, \frac{1}{2} \cdot 2^n - 1] = [-2^{n-1}, 2^{n-1} - 1]$$

.

[2] Ved programmering kallar ein ofte unipolare datatypar for "unsigned", dvs. utan forteikn, og bipolare datatypar for "signed", dvs. med forteikn. Dette er litt upresist som ein vil sjå i kapittel A.2.3.

[3] Sjå f.eks. "en.wiki Geometric series".

Eksempel A.2.

Ein mikrokontroller innheld ein A/D-omformar på 10 bit. A/D-en omformar eit analogt spenningssignal til digital kode, og ein kan setja omformaren opp til å ta inn spenningar i området 0 - 5 Volt eller -2.5 - +2.5 Volt.
I førstnemnte tilfelle blir den omforma digitale koden unipolar og i det andre tilfellet bipolar.
Talområdet for den digitale koden i dei to tilfella blir då følgjande:

Unipolart talområde: $S = [N_{min}, N_{max}] = [0, 2^{10} - 1] = [0, 1023]$.

Bipolart talområde: $S = [N_{min}, N_{max}] = [-2^{10-1}, 2^{10-1} - 1] = [-512, 511]$.

Meir om representasjon av negative og positive tal kjem i kapittel A.2.3.

A.1.5 Oktale og heksadesimale tal

For det **oktale** talsystemet har ein $r = 8$, dvs. at koeffisientane a_i kan ha 8 moglege verdiar:
0,1,2,3,4,5,6,7.

For det **heksadesimale** talsystemet[4] er $r = 16$, dvs. at koeffisientane a_i kan ha 16 moglege verdiar:
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

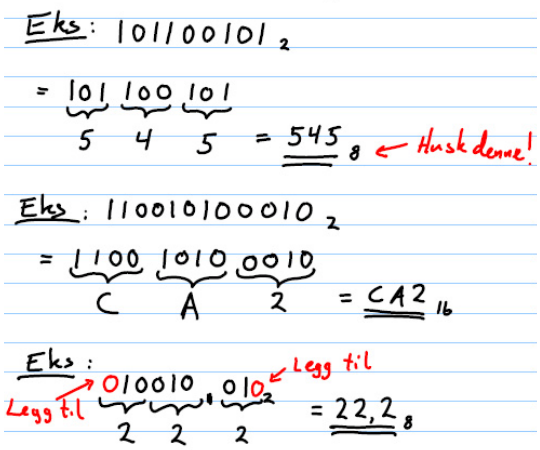
Ettersom $2^3 = 8$ og $2^4 = 16$, kan kvart enkelt oktale siffer representerast binært med tre bit og kvart enkelt heksadesimale siffer med fire bit. Dette er vist i tabellen A.2.

Det er lett å konvertera frå binære til oktale og heksadesimale tal og motsett som illustrert i figur A.2.

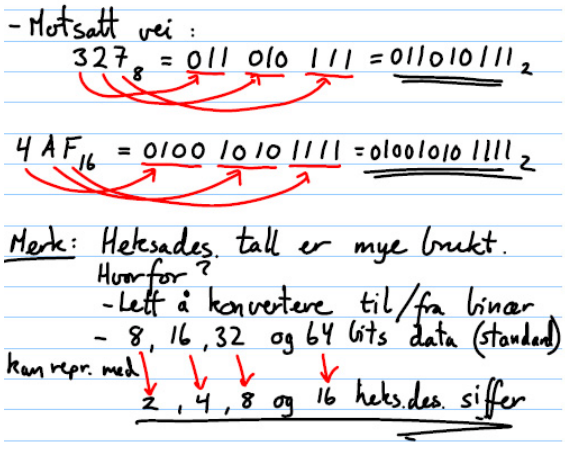
[4]Det heksadesimale talsystemet er som ein vil sjå gjennom det meste av dette skrivet, svært sentralt når ein skal forstå datamaskinar.

Talsystem: (Grunntal)	Desimal (10)	Binær (2)	Oktal (8)	Heksadesimal (16)
	00	0000	00	0
	01	0001	01	1
	02	0010	02	2
	03	0011	03	3
	04	0100	04	4
	05	0101	05	5
	06	0110	06	6
	07	0111	07	7
	08	1000	10	8
	09	1001	11	9
	10	1010	12	A
	11	1011	13	B
	12	1100	14	C
	13	1101	15	D
	14	1110	16	E
	15	1111	17	F

Tabell A.2: Samanhengen mellom binære, oktale og heksadesimale tal.



(a) Frå binære til oktale og heksadesimale tal



(b) Frå oktale og heksadesimale til binære tal

Figur A.2: Konvertering mellom tal på ulike totalsformat

A.1.6 Konvertering frå desimaltal til binærtal

Eit tal gitt med grunntal r kan altså konverterast til desimaltal D vhja. likninga under.

$$D = \sum_{i=-m}^{n-1} a_i \cdot r^i \tag{1.6}$$

Eit eksempel på konvertering fra heksadesimal til desimal er gitt i det følgjande.

Eksempel A.3.

$$\begin{aligned} B23A,0F_{16} &= B \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + A \cdot 16^0 + 0 \cdot 16^{-1} + F \cdot 16^{-2} \\ &= 11 \cdot 4096 + 2 \cdot 256 + 3 \cdot 16 + A + 0 + 15 \cdot \frac{1}{256} \\ &= 45626.05859375_{10} \end{aligned} \tag{1.7}$$

Kva med motsett veg, dvs. frå desimaltal til eit tal med grunntal $r \neq 10$?
Ei oppskrift for slik konvertering er som følgjer:

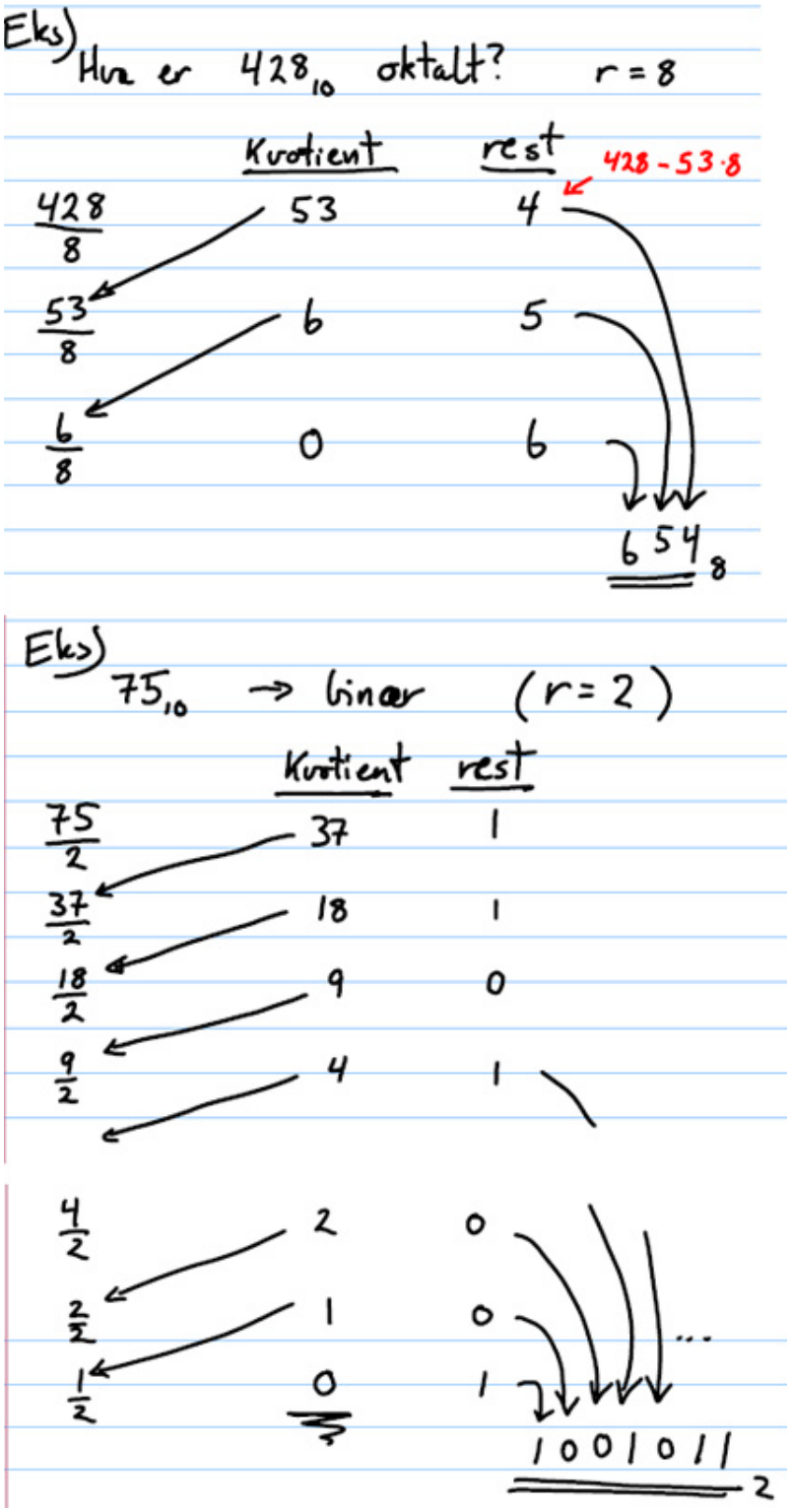
i. Divider talet D med grunntalet r.

ii. Ta vare på rest-verdien (LSD).

iii. Divider kvotienten fra forrige iterasjon på r og ta vare på rest-verdien (som skal leggast til til venstre for førre rest-verdi).

iv. Hald fram til kvotienten er 0 (MSD).

Nokre eksempel på konvertering frå desimaltal til både oktal- og binærtall er gitt i figur A.3.



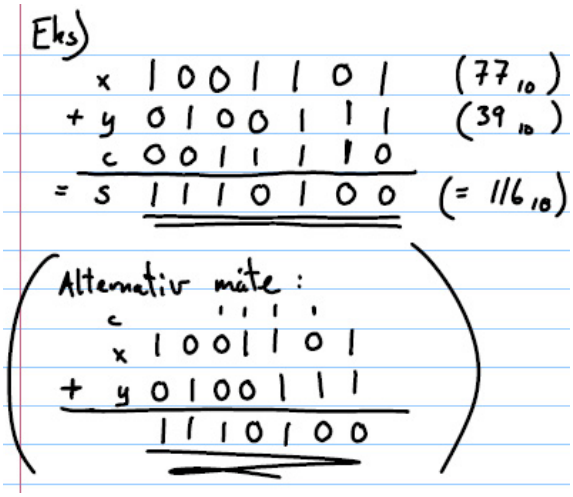
Figur A.3: Konvertering frå desimaltal til oktal- og binærtal.
(Ref.: Tom Ryen, [12].)

A.2 Aritmetikk

Ein vil her sjå på rekneoperasjonane addisjon og subtraksjon. I samband med subtraksjon vil det vera naturleg å ta med representasjon av negative tal.

A.2.1 Addisjon

Binær addisjon[5] kan gjerast på tilsvarande måte som ein har lært det i grunnskulen. Eit eksempel på addisjon med bruk av mente ("carry") er vist i figur A.4.



Figur A.4: Addisjonseksempel. Bokstaven c viser mentet. (Ref.: Tom Ryen, [12].)

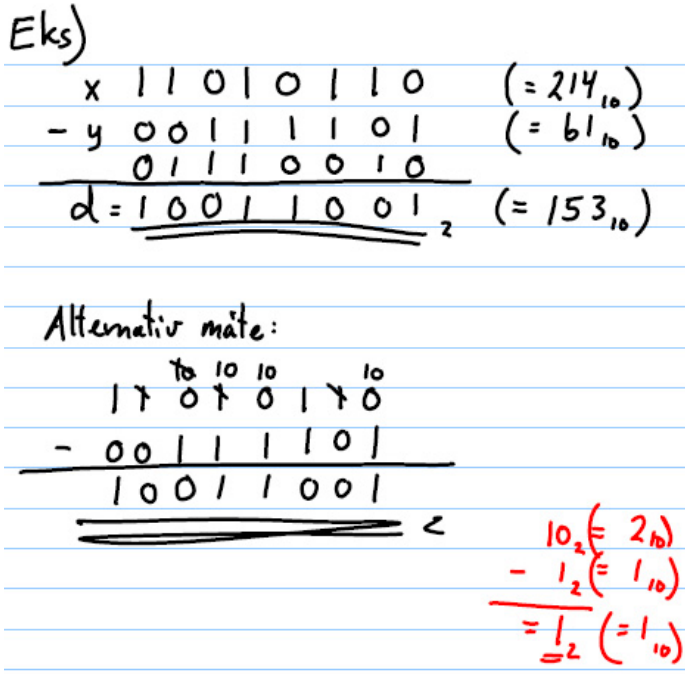
Slik blir også tala adderte siffer for siffer i ein datamaskin.

[5] addend+addend=sum

A.2.2 Subtraksjon

A.2.2.1 Tradisjonelt oppsett

Subtraksjon slik som ein lærte det i grunnskulen, gjer bruk av låning som vist i figur A.5.



Figur A.5: Subtraksjonseksempel. (Ref.: Tom Ryen, [12].)

A.2.2.2 Subtraksjon ved bruk av komplement

Det viser seg at subtraksjon kan realiserast på ein meir effektiv måte i digital logikk. Ved å bruka såkalla **komplement**, kan ein sleppe unna ved å utføra ein addisjon i staden for subtraksjon.
Det er to typar komplement:

Einar-komplement: Dette er det same som bitvis invertering[6] av eit tal og er ein vanleg logisk operasjon som vist før.

Toar-komplement: Dette er einar-komplementet + 1. Det er dette som blir brukt ved realisering av subtraksjon. Dette vil bli vist under.

[6]Det inverse eller komplementære av 1 er som kjent 0 og omvendt.

Me definerer einar-komplementet av talet X til å vera $\overline{X}$.
Viss talet X er på 8 bit og eksempelvis har verdien $X = 01100101$, så har me følgjande:

X	01100101
$+\overline{X}$	$+$ 10011010
$=$	11111111
$+$ 1	00000001
$=$	1 00000000

Me ser her bare på tal av ei viss lengde, her 8 bit, og forkastar då mentet som oppstod i siste addisjonen.
Me innser her ganske greitt at addisjonen mellom eit **generelt tal** og einar-komplementet av talet ville gitt bare 1-arar **uansett** utgangspunkt. I tillegg er det innlysande at den siste addisjonen gir mente lik 1 og ellers bare 0-ar.

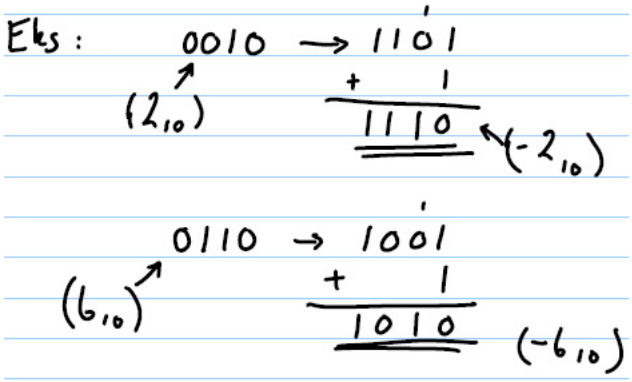
Dette kan skrivast slik:

$X + \overline{X} + 1 = 0$ som gir at $-X = (\overline{X} + 1)$

Skal me altså ha den negative utgåva av eit positivt tal X, er denne gitt av toar-komplementet til X, dvs. parentesuttrykket!
Ein subtraksjon av talet X frå talet Y kan då skrivast slik:

$Y - X = Y + (-X) = Y + (\overline{X} + 1)$

Ein kan altså bruka ei og same digitale adderarblokk til å utføra både addisjon og subtraksjon!
Forskjellen er at ved subtraksjon må toar-komplementet dannast først, og dette involverer som me har sett, ei bitvis invertering og så ein addisjon av talet 1.
Eit eksempel på korleis ein finn toarkomplementet er vist i A.6.



Figur A.6: Konvertering til toarkomplement. (Ref.: Tom Ryen, [12].)

A.2.3 Generelt om representasjon av tal med forteikn

I grunnskulen lærte me å bruka forteikn $\pm$ for å kunne skilja positive og negative tal. Ein måte å representera negative binærtal på er tilsvarande, nemleg å bruka MSbit som forteiknsbit slik at 1/0 i denne posisjonen angir $\pm$. Dette er brysomt ved aritmetiske operasjonar i ein datamaskin ettersom forteikn og talverdi må handterast kvar for seg. I staden brukar ein som me har sett, toar-komplementrepresentasjon ved aritmetiske operasjonar.

Tabell A.3 illustrerer ulike 4-bits representasjonar av tallrekkja $-8, -7, \ldots, +6, +7$.

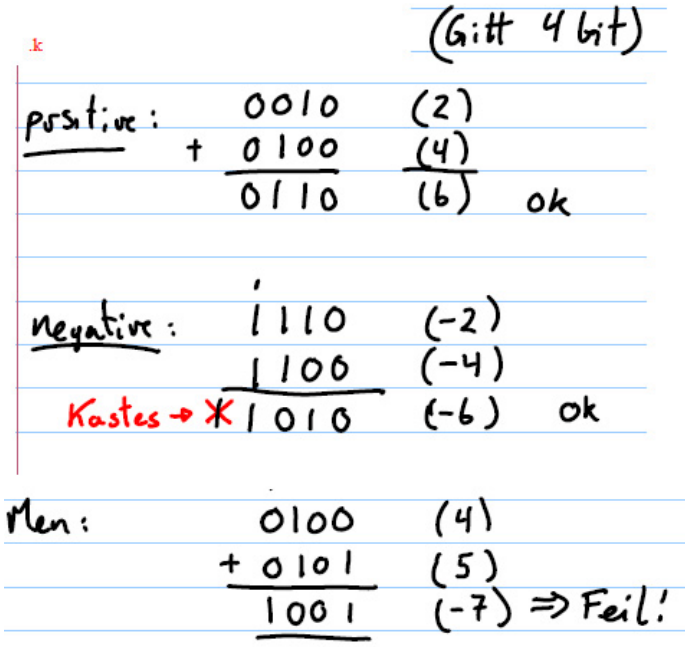
Desimal-verdi	Toar-komplement	Einar-komplement	Absoluttverdi m/forteikn
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000	0000	0000
-0	0000	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	-	-

Tabell A.3: Representasjon av tal med forteikn.

A.2.4 Litt meir om aritmetisk addisjon og subtraksjon

Ved bruk av toar-komplementrepresentasjon av tal er det bare å addera bitmønstra direkte. Det er då **viktig** at alle bitmønstra har same lengde, og eit eventuelt mente som går utover dette, må forkastast.
Talet på verdiar som kan representerast, er avgrensa til 2^n. Viss resultatet ligg utanfor dette området, blir det kalla overflyt ("overflow"). Resultatet får då feil forteiknsbit.

Eksempel på addisjonar utan og med overflyt er vist i figur A.7.

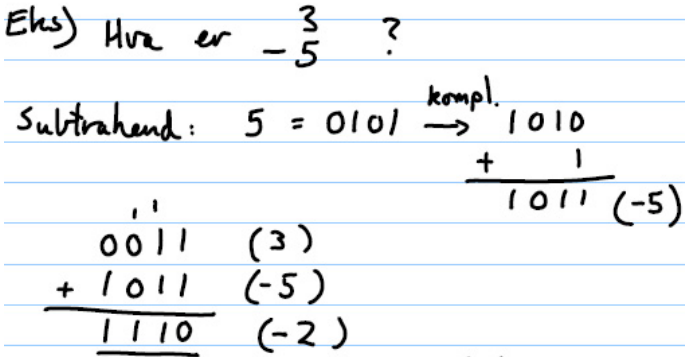


Figur A.7: Addisjon med toarkomplement. (Ref.: Tom Ryen, [12].)

Ved subtraksjon blir subtrahendens forteikn endra vhja. toar-komplementering for deretter å addera den til minuenden.

$$\begin{aligned}(\pm A) - (+B) &= (\pm A) + (-B) \\ (\pm A) - (-B) &= (\pm A) + (+B)\end{aligned} \tag{1.8}$$

Eksemplet i figur A.8 viser korleis framgangsmåten er i datamaskinen.



Figur A.8: Addisjon med toarkomplement. (Ref.: Tom Ryen, [12].)

A.3 Generelt om binærkodar

Digitale system prosesserer og lagrar all informasjon som binære mønster (bitmønster). Me kan representera informasjon (tal, bokstavar,...) på ulike måtar, dvs. ved bruk av ulike *kodar*. Den nødvendige bitbreidda, n, er avhengig av kor mange diskrete informasjonselement som skal kodast. Ei mengde med åtte element krev 3 bit ($2^3 = 8$). Ein kan ha kodar som bruker fleire bit enn nødvendig, men ikkje færre.

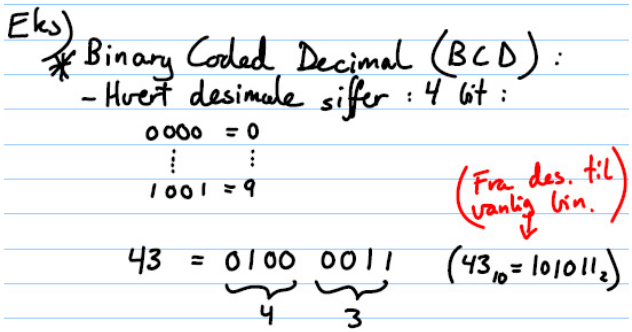
A.3.1 Binærkoda desimaltal ("Binary Coded Decimal", BCD)

BCD[7] er ein kode som blir brukt til å representera dei desimale sifra 1-9. Kodinga er vist i tabell A.4.

Desimalverdi	BCD-kode
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Tabell A.4: Desimaltala med tilhøyrande BCD-kode.

Eit tal med k desimale siffer vil då trenga $4 \times k$ bit til BCD-representasjon. Sjå eksemplet i figur A.9 for $k = 2$.



Figur A.9: BCD-representasjon av talet 43. (Ref.: Tom Ryen, [12].)

[7] Sjå "en.wiki Binary-coded decimal".

Viss ein f.eks. skal skriva eit tal som har så og så mange desimale siffer ut på ein enkel skjerm[8], kan det ofte vera aktuelt å skriva talet ut på BCD-form, dvs. siffer for siffer i titalssystemet.
Talet må då omformast frå binærkode til BCD-kode som eit første steg. Meir om dette kjem i kapittel A.3.3.

A.3.2 Gray-kode

Gray-koden[9] har den eigenskapen at bare ein bit endrast for kvart inkrement. Kodinga av 4-bitsverdiar er vist tabell A.5.

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

Tabell A.5: Gray-koding av 4 bit.

Dette vil vera ein fordel i system der ein kan få feil ved transisjon fra eitt tal til neste når fleire bit må skifta verdi samtidig. Risikoen for at slike feil skal oppstå kan reduserast ved å bruka Gray-kode. Ved bruk av binærtal vil f.eks. skiftet frå 7 til 8 representerast som $0111 \rightarrow 1000$, noko som medfører at alle fire bit endrar verdi, mens for Gray kode tilsvarer akkurat denne endringa $0100 \rightarrow 1100$.
I optiske måleinstrument som f.eks. såkalla optiske enkodarar for vinkelmåling, er denne kodinga vanleg.

[8]Det kan f.eks. vera ein 7-segmentskjerm, sjå "en.wiki Seven-segment display", med eit visst antal siffer. Viss denne tar imot BCD-kodar, vil det vera ein omformar internt i skjermmodulen som basert på BCD-kodane slår av og på dei ulike segmenta i kvart siffer.

[9]Sjå "en.wiki Gray code" som gir ein grundig bakgrunn for denne.

A.3.3 ASCII-kode

A.3.3.1 Standardutgåva

ASCII-kode står for "American Standard Code for Information Interchange" og er ein svært vanleg kode for binær representasjon av teikn ("character"). Kodane for ulike teikn er viste i tabellen i figur A.10.

ASCII Code Chart

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

Figur A.10: ASCII-tabellen. Det mest signifikante heksadesimale kodesifferet (MSD) finn du til venstre i tabellen.
(Ref.: "Wiki ASCII_Code_Chart.svg".)

Dette systemet bruker 7 bit for kvart teikn og er den opprinnelege ASCII-utgåva eller standardutgåva.

Frå tabellen kan ein lesa at koden for 'A' er $0x41 = 01000001_2$ [10] og
koden for '1' er $0x31 = 00110001_2$.
ASCII-tabellen er som vist laga slik at ein finn koden for eit desimaltal ved å leggja til talet $0x30$.

Eksempel A.4. *Utskrift av eit desimaltal på ein LCD-skjerm*

Variabelen A er på 8 bit. Variabelverdien $01111110_2 = 0x7E = 126$ er lagra på binær form i minnet.
I eit program skal verdien skal skrivast ut på ein liten LCD-skjermmodul[11].

Verdien må då først gjerast om til eit binærkoda desimaltal, dvs. på BCD-form, jfr. kapittel A.3.1. Verdien vil då bli representert ved BCD-kode for dei desimale siffera 1, 2 og 6. Ved så å senda siffer for siffer i ASCII-kode, vil sjølve

[10] *0x* er notasjonen for heksadesimale tal i språket C.
[11] I tillegg til skjermelementet er det i slike modular ein liten kontrollerkrets (grafisk prosessor + eit lite grafikkminne) som tar imot data og skriv desse ut.

teikna koma fram på skjermen[12].

Her vil ein då først senda koden $0x31$, så $0x32$ og sist $0x36$[13].

A.3.3.2 Utvida utgåve

Som ein ser av figur A.10, så er det ikkje plass til alle dei vanlege teikna i standardutgåva. Det blei derfor laga ein utvida ASCII-kode[14]. Denne koden er på 8 bit. Det er fleire utgåver av denne som vist i fotnotereferansen, der den mest populære er ISO-standarden 8859-1. Denne blir også kalla *Latin 1*, og dekkjer mellom anna alle dei vest-europeiske bokstavane[15].

Viss ein bruker meir enn dei 256 ulike teikna som ein 8-bitskode gir, må ein over på såkalla multibytekodar. Her er UTF-8[16] ein effektiv metode som gir ein kode på 1 byte for teikn i det utvida ASCII-settet, og meir enn 1 byte utanfor dette.

A.3.3.3 Litt om minnebehov ved lagring av tekst og bilete

Følgjande eksempel kan illustrera kor stor plass tekst og bilete tar opp ved lagring i eit minne.

Eksempel A.5. *Minnebehov for tekst og bilete*

Ein tenkjer seg at dette kompendiet skal utvidast, og at denne utvidinga er litt tekst samt eit bilete tatt med ein smarttelefon.
Kompendiet er på *pdf*-format, og sidetalet er rundt 300.
Ein vil nå sjå på tekst, figurar og bilete kvar for seg.

[12] Omforminga til BCD kan ein f.eks. gjera ved å først å finna talet på heile hundre ved å dela på 100, så gjenta det same for tiarane. Ein sit så igjen med 1-arane. Så finn ein ASCII-koden for kvart siffer ved å leggja til $0x30$, som vist før.
I C-biblioteket *stdlib.h* har ein metoden *itoa()* som gjer alt dette, dvs. konverterer eit heiltal ("integer") til ein ASCII-koda streng. I andre samanhengar kan ein ha behov for den motsette omforminga, og den kan gjerast med metoden *atoi()*.

[13] Her har ein gått ut frå at skjermen er sett opp med høgreskift slik at neste teikn automatisk blir plassert til høgre for det førre teiknet.

[14] Sjå "en.wiki Extended ASCII".

[15] Viss ein derimot kjøper ein japansk LCD-skjermmodul, som er veldig vanlege i innebygde system, vil ein sjå at den øvre delen av ASCII-tabellen brukt i modulen er dominert av japanske teikn.

[16] Sjå "en.wiki UTF-8".

- **Tekst:**
  Kvart teikn blir koda basert på den nemnde standarden *Latin 1*, som er på 8 bit, dvs. 1 byte. Viss ein tar litt i og reknar 40 linjer pr. side og 90 teikn pr. linje, får ein følgjande plassbehov:

  B_tekst = 300 x 40 x 90 x 1 byte = 1.080.000 byte ≈ 1 MByte.

  Det såkalla "Portable Document Format" vil i tillegg innhalda mellom anna kommandoar for oppsett av dokumentet. Tekstdelen vil likevel samla sett ta liten plass.

- **Teikna figurar:**
  Alle dei nummererte figurane i kompendiet er anten bilete eller teikna. Dei teikna figurane dominerer, og alle desse er **vektoriserte**[17]. Det betyr at i staden for å lagra informasjon om kvart bilet/figur-punkt, så lagrar ein kommandoar for utskrift av figuren. Å f.eks. lagra koordinatane for endepunkta til ei linje og korleis streken skal sjå ut er mykje mindre plasskrevjande enn å lagra alle biletpunkta for linja. I tillegg blir detaljane like skarpe same kor mykje ein bles opp figuren på ein skjerm.
  I pdf-formatet er då både teksten og desse teikna figurane vektoriserte.

  I gjennomsnitt tar kvar av dei teikna figurane i kompendiet rundt 30 KB. Alternativet, eit komprimert **rasterformat**[18] som JPEG[19] eller PNG[20], ville trass i komprimeringa ta ein god del større plass og blir som sagt uskarpt ved oppblåsing.

- **Bilete:**
  Bileta i kompendiet er lagt inn som *.jpg-* eller *.png*-filer. Oppløysinga er relativt låg, men ikkje lågare enn at dei er skarpe på ei A4-side. Bileta tar i gjennomsnitt rundt 100 KB.

  Det nye biletet som ein tenkjer å ha med i kompendiet, er tatt med ein IPhone 5 der kameraet er på 8 Megapikslar. Fargeoppløysinga er på 24 bit. Fargen i kvart biletpunkt har altså ein verdi som tar 3 Bytar.
  Eitt bilete i råformat, dvs. rasterformatet BMP ("bitmap") vil altså ta opp 24 MB!
  Bileta blir heldigvis lagra i jpg-format, og det aktuelle biletet var då ut frå kameraet på rundt 3 MB, altså komprimert rundt 8 gonger.

Samla sett tar kompendiet opp rundt 6 MB før utviding, og det aktuelle biletet tar opp like mykje plass som eit halvt kompendium!
Dette viser kor lett eit dokument kan bli blese opp ved ukritisk handtering av bilete. Det er i dei fleste tilfelle nødvendig med kraftig tilleggskomprimering[21] før ein tar dette inn i eit skriv. Skal ein bruka bileta for seg sjølv i store format, blir sjølvsagt saka ei anna.

[17] Sjå "en.wiki Vector graphics". Figurar på formatet *svg* ("Scalable Vector Graphics"), er vektoriserte. I kompendiet er all teikning gjort vhja. pakken *MS Power Point* og så skrive ut på *pdf*-format, noko som også gir eit vektorisert resultat.
[18] Sjå "en.wiki Raster graphics".
[19] Sjå "en.wiki JPEG".
[20] Sjå "en.wiki Portable Network Graphics".
[21] I f.eks. programmet *MSWord* kan ein under menyen *Biletverktøy* komprimera bilete.

April 2018
ISSN 1504-4939
ISBN 978-82-7644-767-5
Notat nr. 39, Universitetet i Stavanger

Universitetet i Stavanger
N-4036 Stavanger
Norge
www.uis.no

Utfordre.
Utforske.