Konstruksjon av små innebygde system basert på mjukprosessor

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

9. mars 2012

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

ISSN 1504-4939
ISBN 978-82-7644-475-9
Notater fra Universitetet i Stavanger, nr. 35

Forord

Dette kompendiet omhandlar konstruksjon av små datamaskinar[1] for overvaking og styring av ulike prosessar. Ein prosess kan f.eks. vera ein vaskemaskin, ein harddisk eller drivstoffinnsprøytinga i ein bilmotor.
I slike system 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")[2] system, (ibs).
Det motsette er f.eks. ein personleg datamaskin, PC, som har eit breitt bruksområde utan å vera tilknytta anna utstyr. Ein PC kan sjølvsagt også byggjast inn og då bli til f.eks. ein minibank, ein fotoautomat for passbilete eller eit kassaapparat.

Kompendiet er opprinneleg skrive for bruk i eit emne på feltet digital og analog elektronikk ved Universitetet i Stavanger, UiS, men kan brukast til å gi ei innføring i mikroprosessoren sin virkemåte og konstruksjon av den digitale kjernen av eit innebygd system. Kompendiet tar ikkje opp tema som handtering av analoge signal, konstruksjon av kraftforsyningar med meir. Dette blir dekt av anna læremateriell i emnet.

Digital og analog elektronikk er ein obligatorisk del av bachelorstudiet i elektro[3] ved UiS. Eit av **måla** for dette bachelorstudiet, er at studentane skal kunne

- forstå oppbygginga av,
- kunne analysera og
- konstruera enkle innebygde system.

Utstyrsplattforma for emnet har skifta fleire gonger. I 2005 gjekk ein over på bruk av **mjukprosessorar**, dvs. mikroprosessorar implementert i programmerbar elektronikk. Ein bruker her mjukprosessoren MicroBlaze frå Xilinx Inc. og utviklingsutstyret Embedded Development Kit. I dette PC-baserte verktøyet utviklar ein oppsettet, dvs. konfigurasjonsdata for den mjuke datamaskinen samt programmet som han skal utføra, og programmerer så ein FPGA-krets med alt dette.

Den første mjukprosessoren kom i 2000, men det gjekk nokre år før verktøy og tilgjengeleg elektronikk var tilstrekkjeleg utvikla.
Det er ennå lite lærestoff som er laga på feltet. Difor blei dette kompendiet utarbeidd. Kompendiet blir endra og utvida frå år til år, så tilbakemeldingar blir mottatt med takk.

[1] Med små datamaskinar meiner ein her datamaskinar med ein mikroprosessor og med moderate minnekrav, dvs. typisk $<$ 1MByte.

[2] Kompendiet er norsk. Engelske nemningar vil bli sett i hermeteikn viss dei blir nytta åleine og i parentes når dei blir viste saman med norsk utgåve av nemninga.

[3] Frå og med hausten 2012 vil dette studiet ha spesialiseringane Elektronikk og kommunikasjon og Industriell automatisering.

Innhald

Nokre vanlege forkortingar innan digital og analog elektronikk

Alle engelske nemningar 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.
anf	Antinedfaldingsfilter, "anti-aliasing filter".
BJT	"Bipolar Junction Transistor", bipolar transistor.
BPS	"bits per second", bit per sekund (bps), bitrate.
BRAM	"Block RAM", SRAM-blokk i FPGA. Minne i FPGA blir sett opp av mange BRAM.
BSB	Base System Builder, del av utviklingsverktøyet EDK.
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.
CMRR	"Common Mode Rejection Ratio", undertrykking av fellesnivå, dvs. gjennomsnittsnivå.
CPLD	"Complex PLD", ligg mellom PLD og FPGA i kompleksitet.
C/T	"Counter/timer", teljar/timer-modul.
D/A, DAC	"DA Converter", DA-omformar, dvs. omformar frå digital til analog verdi.
d-inv	Differensiell forsterkarkobling basert på 1 operasjonsforsterkar (opf) og med eigenskapar som inv.
d-i-inv	Instrumenteringsforsterkar, dvs. diff. forsterkarkobling basert på 3 opf.
DRAM	Dynamisk RAM, flyktig ("volatile") minneteknologi.
EDK	Embedded Development Kit, Xilinx sitt utviklingsverktøyet for mjukC.
EMI	"ElectroMagnetic 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.
GBW	"Gain BandWidth product", produkt av forsterkning og bandbreidde.
GPIO	"General Purpose Input Output port", parallellport.
IA	"Instumentation Amplifier", instrumenteringsforsterkar, sjå og d-i-inv.
ibs	Innebygd system, "embedded system".
IC	"Integrated Circuit", Integrert krets.
i-inv	Ikkje-inverterande forsterkarkobling.

IntC	"Interrupt Controller", kontrollermodul for avbrot.
inv	Inverterande forsterkarkobling.
IP-modul	"Intellectual Property", logikkmodul, perifermodul.
ISR	"Interrupt Service Rutine", avbrotsprogram.
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.
mjukC	mjuk mikrokontroller, "soft microcontroller".
mjukP	mjukprosessor, "soft microprocessor".
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.
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.
opf	Operasjonsforsterkar.
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.
PD	"PhotoDiode", 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).
PSRR	"Power Supply Rejection Ratio", undertrykking av variasjonar i forsyningsspenninga.
PWM	"PulseWidth 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 PIC-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.

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.
stm	Stegmotor, "step motor".
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.
UiS1	Øvingsmaskin basert på hovudkortet Spartan 3 Starter Board frå Xilinx og det UiS-konstruerte grensesnittkortet UiS_AD.
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.
vt	Vektortabell, "interrupt vector table".
XAPP	"Xilinx Application note", nyttige skriv om og eksempel på ibs.
XMD	"Xilinx Microprocessor Debugger", avlusingsverktøy i EDK.
XPS	Xilinx Platform Studio, del av utviklingsverktøyet EDK.
μP, μC	Mikroprosessor, mikrokontroller.

Kapittel 1

Innleiing

Mikroprosessorar treff ein på fleire gonger dagleg, ofte utan å vera klar over det. Ein eller fleire slike "hjernar" finst mellom anna i mobiltelefonen, bilen, vaskemaskinen og PC-en, der dei utrøytteleg utfører sine program, dvs. lister av instruksjonar. I dei førstnemnde tre produkta er mikroprosessoren del av eit innebygd system, dvs. ein datamaskin der funksjonane er det viktige som nemnt tidlegare. PC-en er derimot ein generell datamaskin som kan brukast til f.eks. utvikling av innebygde system. Kapittel 1 i boka [1] gir ei god innføring i temaet innbygde system.

Sidan den første mikroprosessoren kom i 1971, har det vore ei rivande utvikling. Han blir i dag produsert som elektroniske brikker i alle slags utgåver der kvar utgåve har sine faste eigenskapar. Eit nytt steg i utviklinga er at du kan laga din eigen mikroprosessor ved å laga ei oppskrift som du lastar 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. Dette kompendiet omhandlar konstruksjon av små innebygde system med utgangspunkt i mjukprosessoren MicroBlaze frå Xilinx Inc., men sjølve framgangsmåten kan også brukast på system basert på andre typar mikroprosessorar.

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 og graden av hyllevarebruk framfor eigenkonstruksjon.
3. Vurdering og val av maskinvare- kontra programvareomfang.
4. Vurdering og val av komponentar/modular[1].
5. Konstruksjon av grensesnitt mellom komponentar/modular.

[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.

6. Utvikling av maskinvareprototype.
7. Programmering, testing og modifisering.

Fleire av aktivitetane kan gå delvis parallelt. Om ein for eksempel tidleg i prosessen utarbeider test-rutiner, så kan dette medføra at ein avdekkjer brester i spesifikasjonen til systemet.
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. Kapittel 12 i boka [1] gir ei god innføring i temaet systemkonstruksjon.

Dette kompendiet vil i tillegg til generelt innføringsstoff, halda seg innanfor punkt 5 - 7 i lista over. Ein tar utgangspunkt i ei bestemt plattform og ser på korleis ein kan byggja eit ibs på denne.
Fleire detaljar om kompendiet er gitt under:

- I kapittel 2 ser ein på mikroprosessorbaserte system generelt, men avgrensar dette til ein-prosessorsystem ("single processor systems").
- Kapittel 3. omhandlar programmerbar elektronikk generelt og går så over på mjukprosessoren. Vekta blir her lagt på prosessoren MicroBlaze og utviklingsverktøyet Embedded development kit (EDK) frå Xilinx Inc.
- Grensesnittkonstruksjon generelt er omhandla i kapittel 4 men med eksempel frå MicroBlaze-baserte system.
- I vedlegg A og B er det vist framgangsmåte for å laga, køyra og endra eit enkelt innebygd system.
- Vedlegg C omhandlar det å utvikla program for mjukkontrolleren.
- I vedlegg D blir det vist framgangsmåte for å programmera Flash-minnet på den maskinvareplattformen ein nyttar her.

Kapittel 2

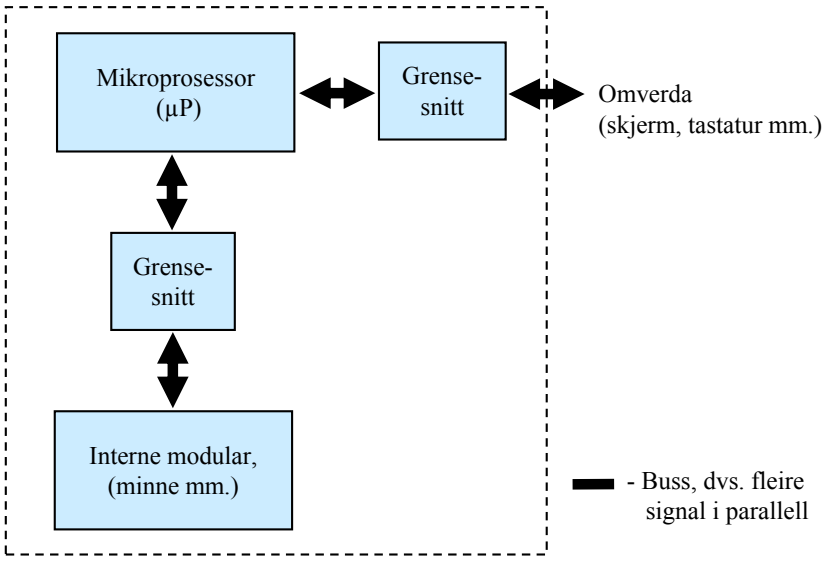
Mikroprosessorbaserte system

Dette kapitlet vil handla om følgjande:

- Litt om korleis mikroprosessorbaserte system er bygde opp.
- Litt om korleis mikroprosessoren er bygd opp og virkar.
- Litt frå historia til mikroprosessorbaserte system.

2.1 Generelt oppsett

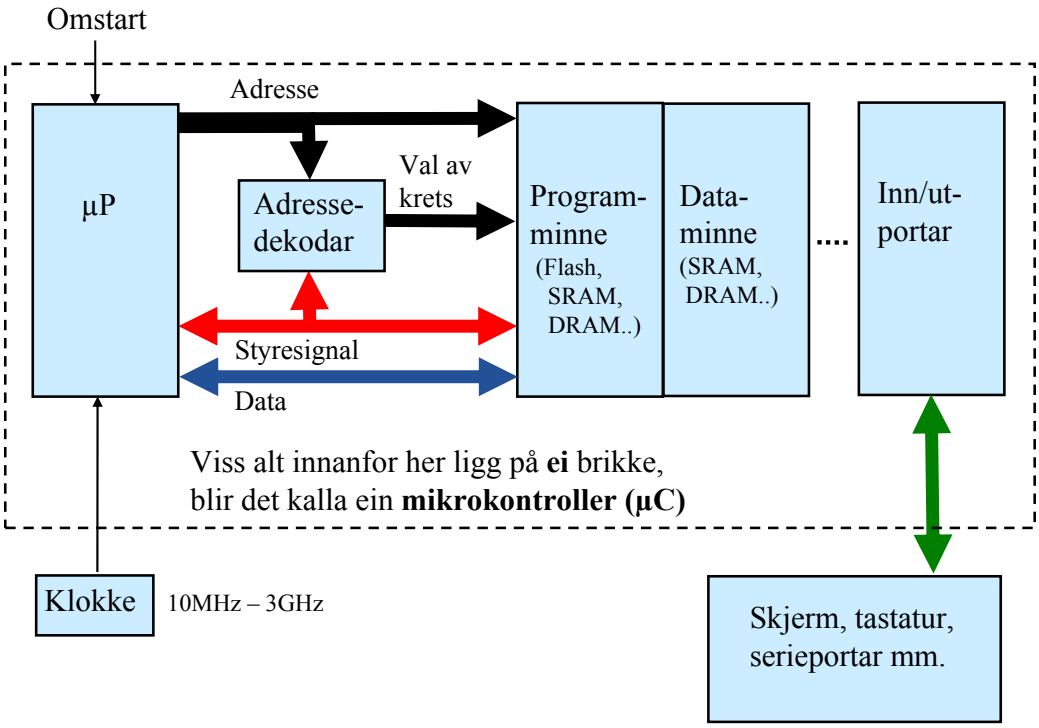
Oppbygginga av eit generelt mikroprosessorbasert system, dvs. ein datamaskin, er vist i figur 2.1.



Figur 2.1: Generelt mikroprosessorbasert system.

Eit **grensesnitt**[1] er her ein modul bygd opp av digital og analog elektronikk 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.

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 velja dei rette modulane til systemet. Typiske modular i eit mikroprosessorbasert system er viste i figur 2.2.



Figur 2.2: Typisk oppsett av eit mikroprosessorbasert system.

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

[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 programrutiner i eit bibliotek.
3) Ved kobling av ein laserskrivar til ein PC med kabel, ser ein ofte på sjølve kontaktane som fysiske grensesnitt.

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 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 i forhold til RAM.
Meir om desse minnetypane kan ein finna i feks. Wilmshurst, [1], kapittel 4.

Alle modulane i eit mikroprosessorbasert system som vist i figuren, er knytta saman vhja. eit **buss**-system, som inneheld tre underbussar:

- **Databuss**
  Over databussen hentar μP-en inn programinstruksjonar eller data[2], dvs. **lesing**, eller μP-en overfører data til ein annan modul, dvs. **skriving**.
  Breidda på bussen er avhengig av kva slags prosessor som blir brukt, og er vanlegvis på 8, 16 eller 32 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[3] eller overførast til.

  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 om overføringa er ei lesing eller skriving.

Oppsettet i figur 2.2 er ein såkalla Von Neumann-arkitektur. Mikroprosessoren både hentar instruksjonar og overfører data over eitt og same bussystem. Dette gir ein enkel arkitektur, men bussystemet kan lett bli ein flaskehals, den såkalla Von Neumann-flaskehalsen.
Von Neumann-arkitekturen var lenge den vanlege arkitekturen i datamaskinar. Unnatak var mellom anna system basert på digitale signalprosessorar (DSP). Mange av desse samt ein god del nyare prosessorar og det mjukprosessorsystemet som ein skal sjå på i kapittel 3, er basert på eit alternativt oppsett, nemleg Harvard-arkitekturen. Her har ein separate bussystem 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.

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.

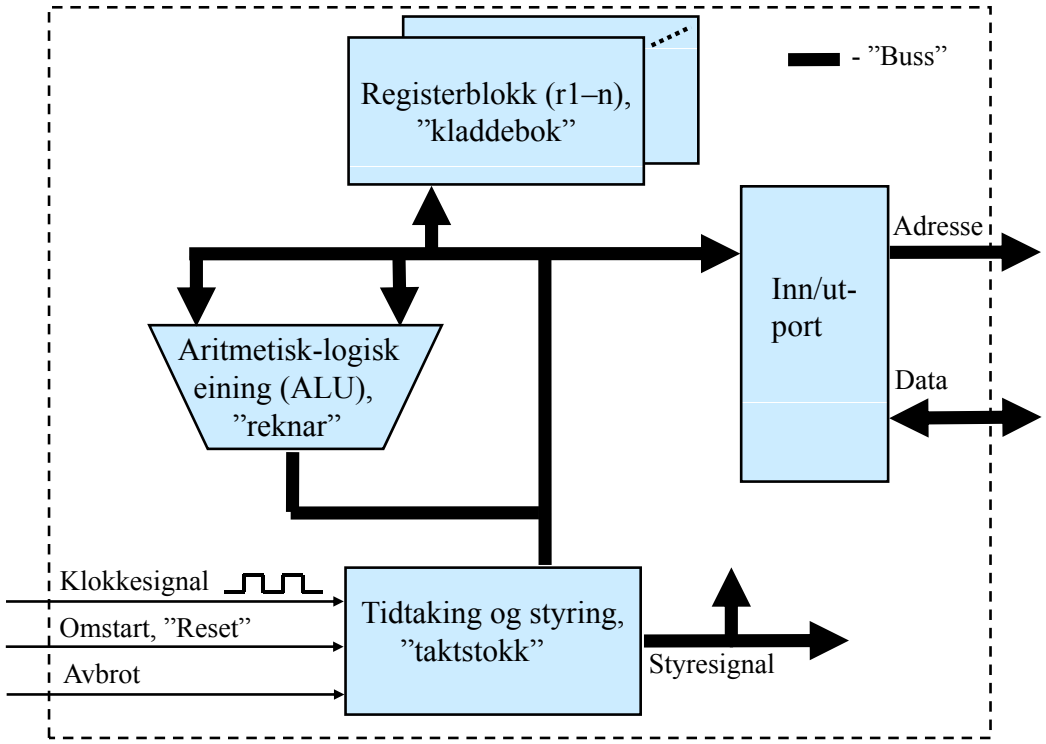
[2] Databussen kan også vera delt i to; ein databuss for programinstruksjonar, og ein databuss bare for data.

[3] eigentleg kopierast frå

2.2 Litt om mikroprosessoren

2.2.1 Generelt oppsett

Sjølve **mikroprosessoren** er 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** for mellombels, dvs. kortvarig lagring av data og adresser. Her har f.eks. mjukprosessoren MicroBlaze 32 register med breidde på 32 bit.
For å kunne henta **inn** data og levera **ut** resultat av ALU-operasjonane, treng μP-en ein **port** mot dei andre modulane i systemet. Eit anna ord for port er grensesnitt, sjå kapittel 2.1.
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 i området vist i figur 2.2.

Mikroprosessoren finn ikkje opp arbeidsoppgåvene sine sjølv, men les inn **instruksjonar** frå **programminnet**. Etter **innlesing** ("fetch") blir instruksjonen **dekoda**, dvs. tolka, og så **utført** ("executed"). Det er styremodulen som gjennomfører desse operasjonane også.
Eit eige register i mikroprosessoren, nemleg **programteljaren** (PC), viser til ei kvar

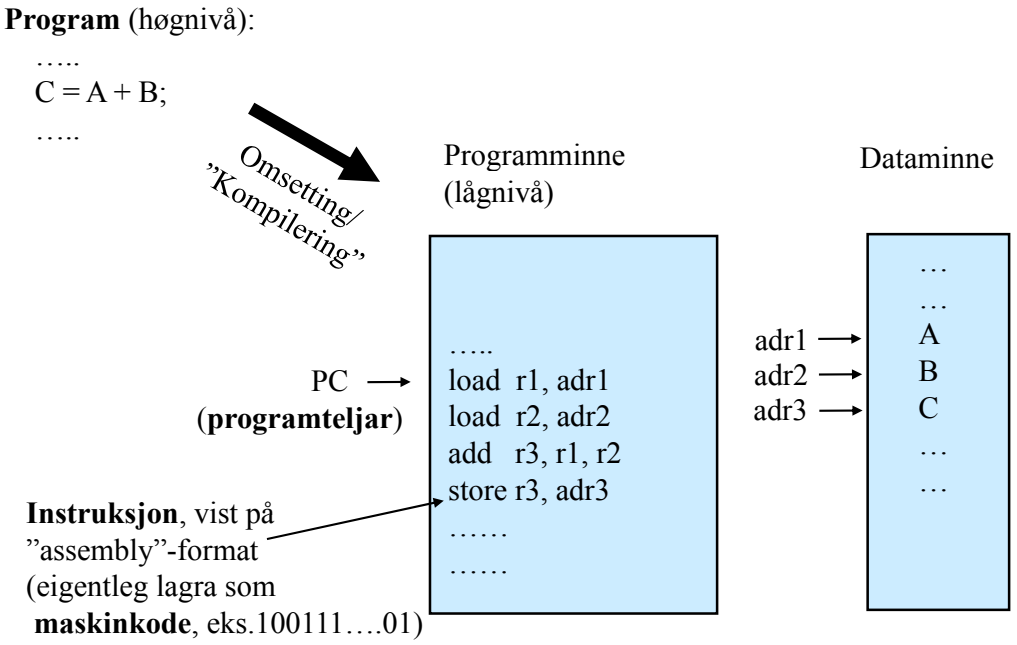
tid kor langt ein er kome i programmet. PC inneheld altså adressa til instruksjonen som nå blir køyrt.

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 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 feks. Linux, Unix og Windows.

2.2.2 Virkemåte

Virkemåten til mikroprosessoren kan illustrerast med programeksemplet vist i figur 2.4.



Figur 2.4: Programeksempel.

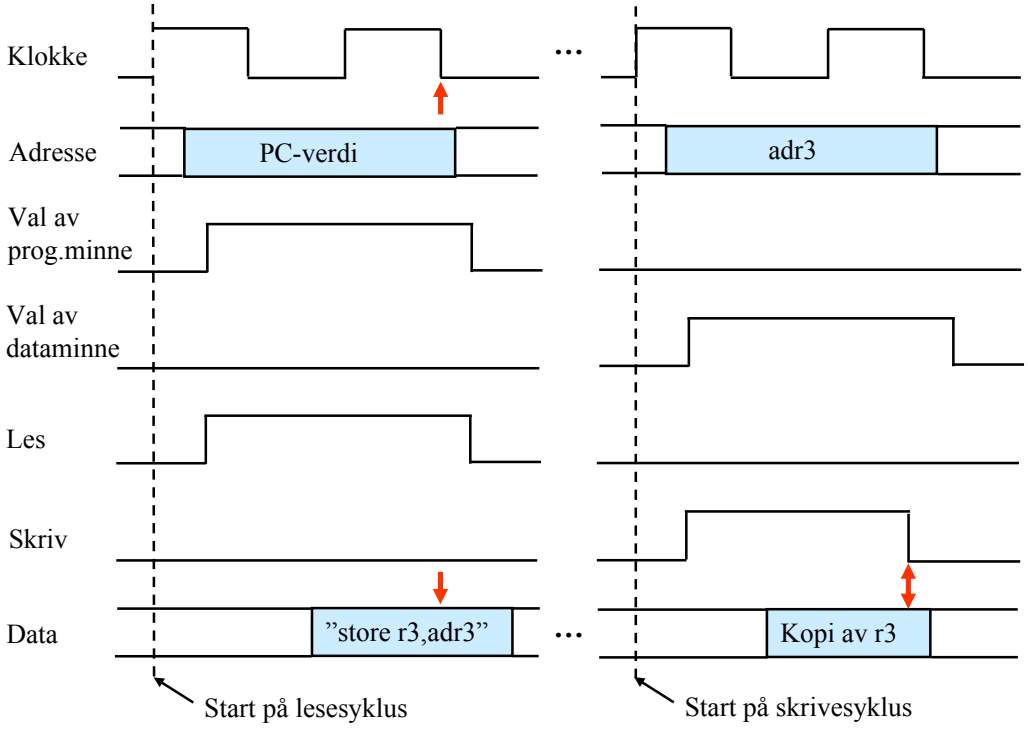
Dei fleste innebygde system blir programmerte vhja. høgnivåspråk som f.eks. C eller C++, og ein høgnivåinstruksjon kan vera som vist i figuren. For at μP-en skal forstå programmet, må det omsetjast/kompilerast til **maskinkode** som altså er kode på lågaste nivå. For å gjera slik kode lesbar for oss, vil kompilatoren mellom anna laga ei fil der programmet er i såkalla assemblyformat.
Figur 2.4 viser korleis assemblyutgåva av høgnivåinstruksjonen 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 frå 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 flyttar resultatet ut til 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 **verdien** til variabel A i variabel A si **minnecelle** på **adresse** 1 som vist i figur 2.4.

Henting av instruksjonar og overføring av data kan gå føre seg som vist i figur 2.5. Her er det først vist lesing av instruksjonen *store r3,adr3* i programmet i figur 2.4. Så blir instruksjonen utført, dvs. den nye verdien blir skriven til adressa for variabel *C*.
Instruksjonar blir overførte på databussen i form av maskinkode, dvs. som bitmønster. Kvar instruksjon har sitt eige bitmønster som mikroprosessoren *dekodar* etter henting for å finna ut kva han skal gjera.
Mikroprosessoren gjennomfører ein lese- eller skrivesyklus i løpet av eit visst an-



Figur 2.5: Eksempel på lese- og skrivesyklar.

tal klokkesyklar (Ks), dvs. klokkeperiodar, og styresignala er **synkroniserte** med klokkesignalet. Figuren tar utgangspunkt i mjukprosessoren MicroBlaze, som bruker typisk 2 Ks[4] på å gjennomføra ein syklus mot program- eller dataminnet. Oppsettet er forenkla[5], men viser sentrale styresignal som *les* og *skriv*, samt to av styresignala

[4]Med typisk klokkefrekvens $f_{KLK} = 50\text{MHz}$ blir $t_{KS} = 20nsek$.
[5]I tillegg til at ikkje alle styresignala er med, er alle flankar loddrette. Ein stigande eller positiv

som adressedekodaren gir ut. Det er vanleg å teikna bussar med dobbel strek som vist i figuren. Dette signaliserer at verdien er gitt av eit bitmønster, mens enkle signal er høge eller låge.

Gangen i ein typisk **lesesyklus** er fylgjande:

1. μP-en set ut ein adresseverdi på adressebussen ved første klokkeflanke[6].

   Når adressedekodaren får inn ein ny adresseverdi, vil han automatisk aktivera eit signal for val av den kretsen som er sett opp med adresseområdet som adresseverdien ligg i.
   I figuren er det programteljarverdien som blir lagt ut, og dekodaren vil då aktivera programminnet.

2. μP-en aktiverer lesesignalet.

3. Dei to aktive styresignala og adressesignala[7] vil gjera at programminnet etter ei viss tid[8] 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[9].

Gangen i ein typisk **skrivesyklus** er fylgjande:

1. μP-en set ut ein adresseverdi på adressebussen ved første klokkeflanke.

   Adressedekodaren vil som i ein lesesyklus automatisk aktivera signal for val av rett krets.
   I figuren er det verdien *adr3* som blir lagt ut, og dataminnet blir då valt.

2. μP-en aktiverer skrivesignalet.

3. Dei to aktive styresignala vil aktivera dataminnet, og adressesignala vil visa kor dataverdien skal leggjast i minnet.

4. μP-en set ut dataverdien på bussen, i dette tilfellet den nye verdien til variabelen *C*.

5. μP-en deaktiverer så skrivesignalet.

6. Dataminnet låser inn dataverdien på negativ flanke av skrivesignalet. Dette er som for lesing vist med loddrett pil i figuren.

---

flanke vil i praksis ha ei viss stigetid og ein fallande eller negativ flanke vil ha ei falltid.

[6] Pga. tidsforseinkingar eller etterslep ("propagation delay") i logikken kjem alltid signala litt etter den klokkeflanken dei er synkroniserte med.

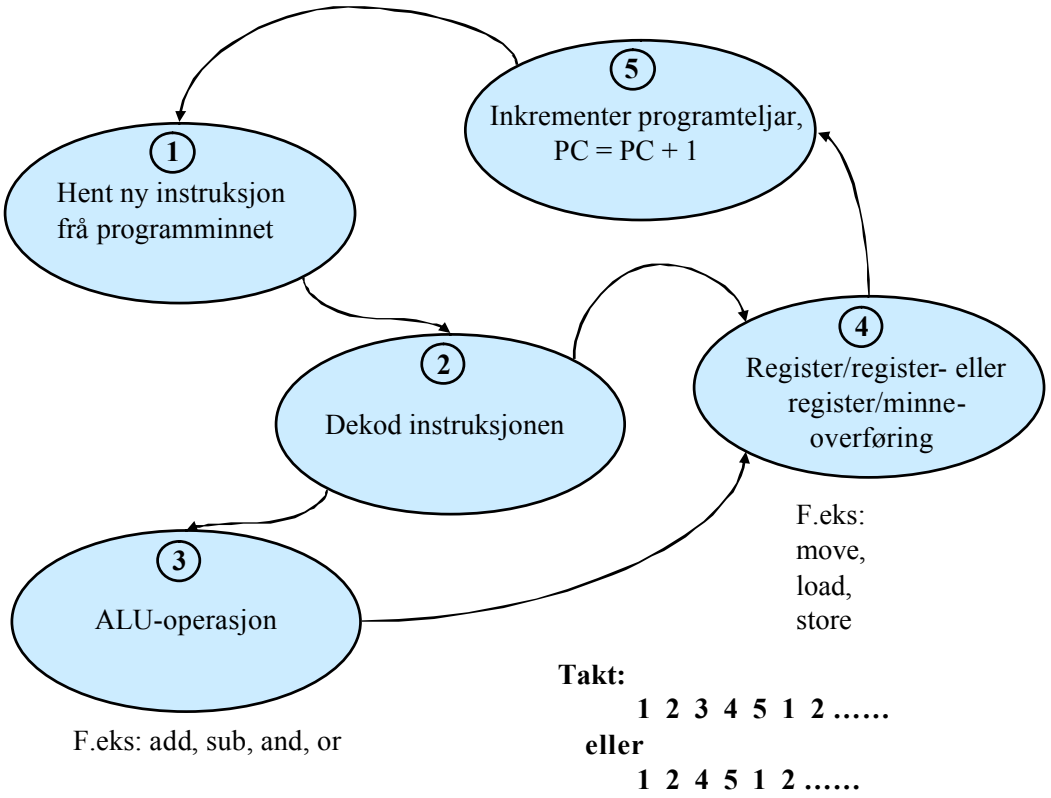
[7] Ein del av adressebussen vil gå rett til minnet for å visa kor i minnet data i dette tilfellet skal hentast frå. Viss n adressebit går til minnet, kan ein adressera 2^n minnelokasjonar. Omvendt vil feks. eit 1kB minne trenga 10 adressebit og eit 1MB minne 20 adressebit for å kunne adressera ein vilkårleg lokasjon.

[8] Denne tida blir kalla **aksesstida**, og seier noko om kor raskt eit minne er.

[9] Det er altså viktig at aksesstida til minnet passar med klokkefrekvens og sykluslengde. Viss minnet feks. var for tregt, ville ikkje dataverdien frå minnet vera klar på bussen på innlåsingstidspunktet.

Lese- og skrivesyklusane blir avslutta med at styresignala blir sett passive, og dette fører til at program- eller dataminnet blir inaktivt.

Køyringa av eit program, dvs. ei liste av instruksjonar, vil gå føre seg som vist i figur 2.6.

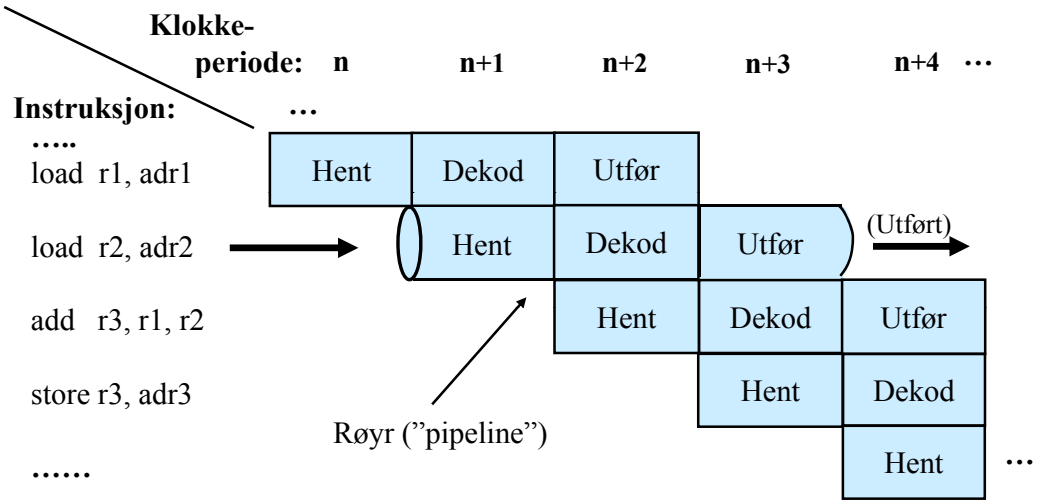


Figur 2.6: Mikroprosessoren sin tilstandsmaskin.

Køyringa blir styrt av ein **tilstandsmaskin** inne i μP-en som går med fast takt gitt av klokkefrekvensen som nemnt før. Avhengig av kva instruksjon som skal dekodast og utførast, blir sekvensane som vist i figuren. Viss det f.eks. er ein overførings-instruksjon, går ein rett frå tilstand 2 til tilstand 4.

I praksis går ikkje alle deloperasjonane i figur 2.6 sekvensielt, dvs. etter kvarandre, men parallelt. Dette blir kalla parallellkøyring ("pipelining"). Kapittel 1.3.3 i [1] omhandlar dette, og viser eit eksempel på eit **2-stegsrøyr** ("pipeline").
Mjukprosessoren **MicroBlaze** som ein skal sjå nærare på i kapittel 3, har eit **3-stegsrøyr** som vist i figur 2.7.

I si ferd gjennom røyret, går ein instruksjon som vist 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 gir ei mykje meir effektiv programkøyring. Programeksemplet i figur 2.7 er



Figur 2.7: Tre-stegs parallellkøyring.

henta frå figur 2.4.

Merk at ein har redusert effekt av slike røyr på stader i programmet der ein gjer hopp framover eller bakover. Meir om dette kjem i kapittel 3.2.2.

2.3 Litt historie

Hovudtrekka i utviklinga av mikroprosessorbaserte system er illustrert i figur 2.8. Her er også vist dei ulike øvingsmaskinane brukt i ingeniørutdanninga på Ullandhaug og i kva tidsrom dei blei brukte. Første maskinen var ein standard datamaskin av type AIM65, mens etterfølgjarane var heilt eller delvis eigenproduserte.

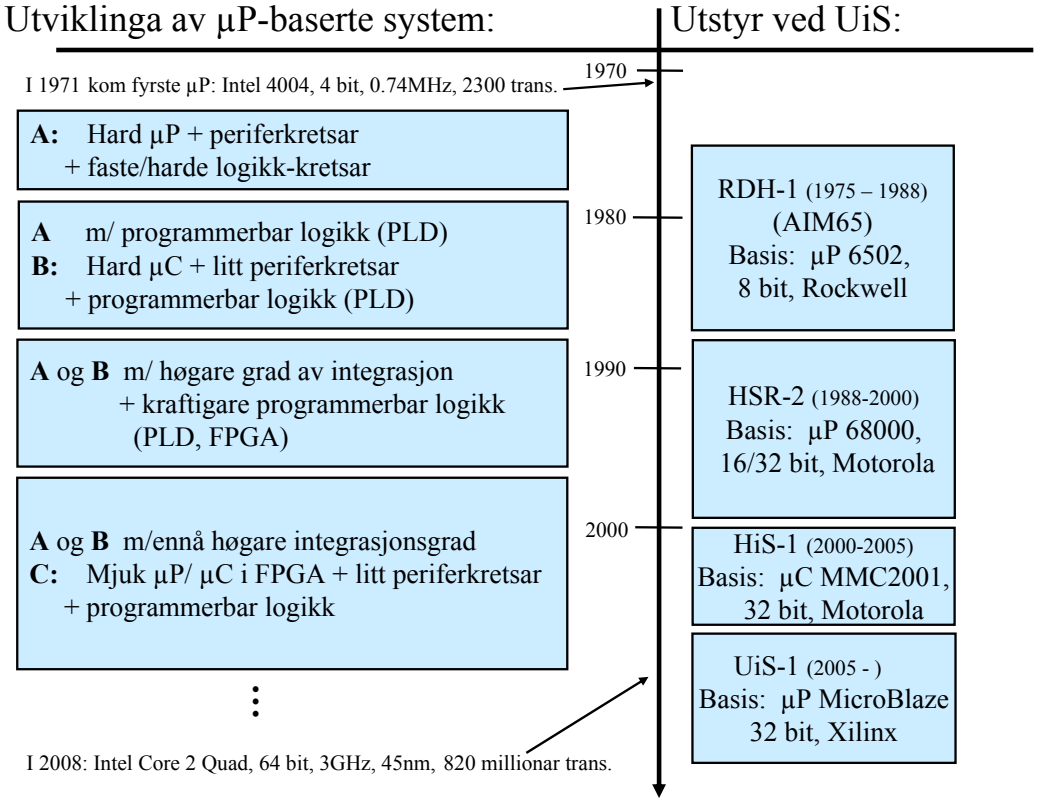
Utviklinga har gått mot stadig færre, men kraftigare kretsar og aukande bruk av programmerbare logiske kretsar. Dette gjeld også dei viste øvingsmaskinane.

Første PC-en kom i 1980. Merk at slike maskinar ennå er av type A. Grunnen til at ein ikkje integrerer det meste i så kraftige datamaskinar på ei brikke og dermed får eit system av type B, er følgjande:

- Effektforbruket på ei slik brikke ville sjølv med teknologien i dag blitt så stort at brikka ville havarera pga. for høg kjernetemperatur.

- Kompleksiteten ville 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 har byrja å gå nye vegar. I staden for å auka klokkefrekvensen og talet på transistorar, lagar ein nå doble eller firdoble kjerner eller prosessorar som f.eks. Intel Core Duo vist i figuren. Utfordringa er her å laga kompilatorar som greier å utnytta slike parallelle prosessorar.



Figur 2.8: 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 -)

Kapittel 3

Innebygde system basert på mjukprosessor i programmerbar logikk

I dette kapitlet skal ein først gå nærare inn på programmerbar logikk generelt og så gå inn på mjukprosessoren MicroBlaze, som kan implementerast i større programmerbare kretsar.
Siste del av kapitlet omhandlar oppsett av MicroBlaze-baserte mjukkontrollarar i programmerbare kretsar og det som må til av tilleggsmodular for å kunne realisera eit innebygd system.

3.1 Programmerbar logikk

3.1.1 Litt om klassifisering av logiske kretsar

Den første kommersielt tilgjengelege integrerte kretsen (IC) blei produsert i 1961 av Fairchild Semiconductor og hadde 15 transistorar[1]. IC-ar blir ofte klassifisert som vist i tabell 3.1.
Ein typisk **port** som det blir vist til i tabellen, inneheld pr. definisjon fire transistorar. Med fire transistorar kan ein feks. realisera ein nog-port ("NAND"). 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 som sa at talet på transistorar pr. arealeining ville doblast i løpet av 18 månader[2]. 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.

[1] IC-en frå 1961 inneheldt ein enkel logisk funksjon, nemleg ei 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.

[2] **Prøv å rekna ut** kva transistortal pr. $100mm^2$ ein skulle ha i 2009 i følgje den modifiserte Moore-lova viss det i IC-en produsert i 1961 var 15 transistorar pr. $100mm^2$

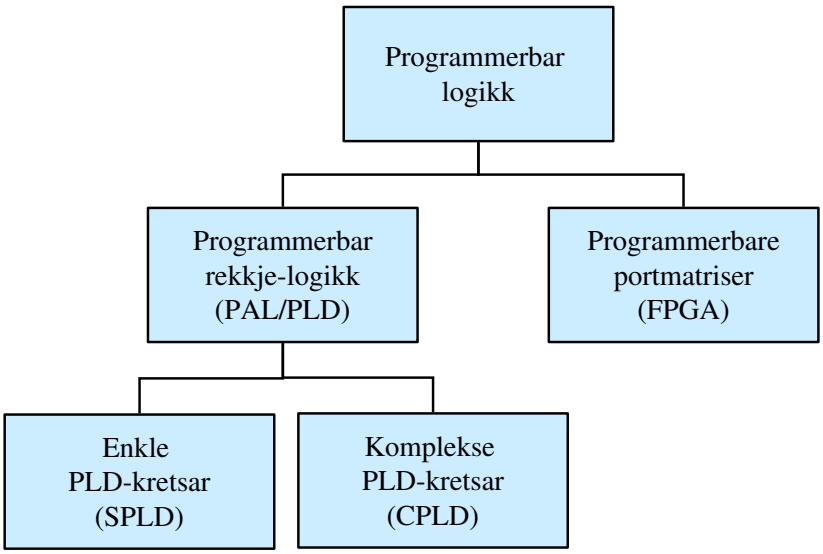
Kom første gong	Kategori	Portar ("Gates")	Transistorar
1961	Small Scale IC (SSIC)	1 - 25	< 100
Slutten av 1960-åra	Medium Scale IC (MSIC)	25 - 250	< 1000
ca.1975	Large Scale IC (LSIC)	250 - 2500	< 10.000
ca.1980	Very Large Scale IC (VLSIC)	Over 2500	> 10.000
ca.1986	Ultra Large Scale IC (ULSIC)		$> 1.000.000$

Tabell 3.1: Klassifisering av integrerte kretsar, IC.

Nokre av dei tettaste kretsane nå (2009) er DRAM[3]-minne produsert med 0.050μm-teknologi, der dei minste transistordelane då har breidder ned til 0.050μm (dvs. 500 Ångstrøm eller He-atomdiametrar).
På desse minnekretsane kan det vera opptil 4 milliardar transistorar på typisk $100mm^2$ areal.

3.1.2 Hovudtypar av programmerbar logikk

Programmerbar logikk kan delast inn som vist i figur 3.1.



Figur 3.1: Ulike typar programmerbar logikk.

Dei første programmerbare kretsane som kom på 1970-talet var av typen MSIC,og blei kalla "Programmable Logic Array", PLA. Neste generasjon blei kalla PAL eller GAL, der GAL er ein fleirgongsprogrammerbar PAL.

Desse kretsane blir også kalla **enkle programmerbare kretsar**, "Simple PLD", som vist i figur 3.1.

[3]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 f.eks. i kapittel 4 i [1].

Komplekse PLD-kretsar, CPLD, har ein meir omfattande og fleksibel innmat. Desse er av typen LSIC eller VLSIC.

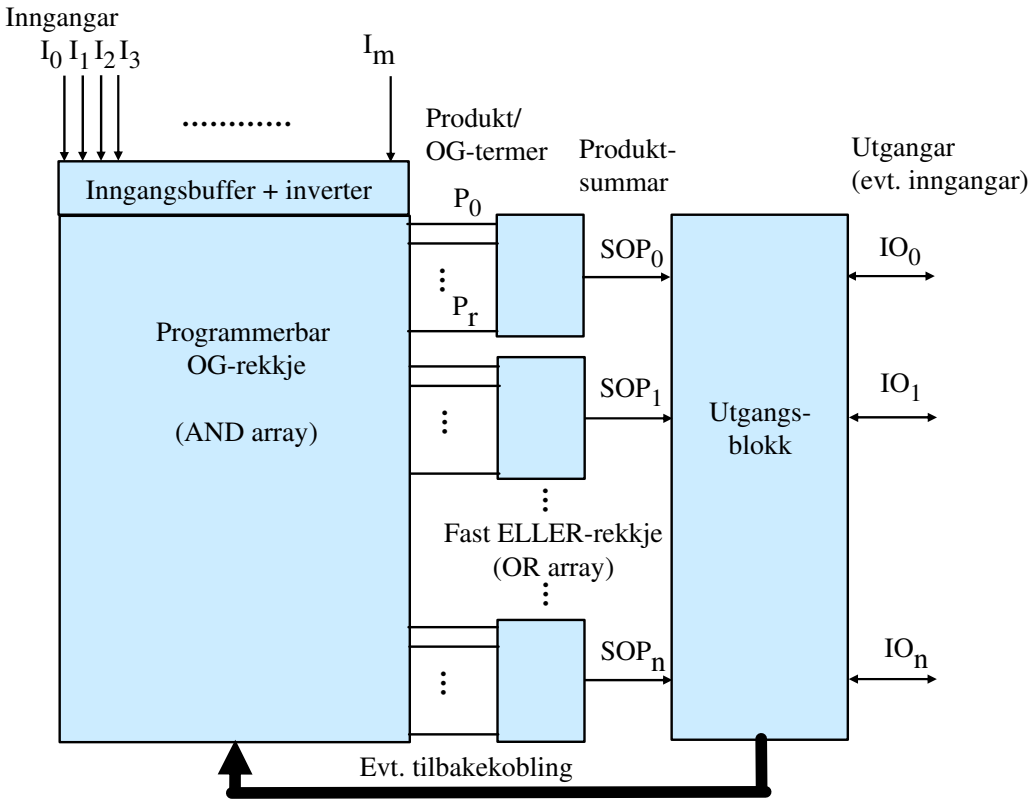
Dei aller største kretsane er såkalla felt-programmerbare portmatriser, FPGA, der alle er av typen VLSIC eller ULSIC.
Ordet **"felt"** siktar til at kretsane kan programmerast "i felten", dvs. ute hjå brukaren. Det motsette vil vera å programmera kretsen i fabrikken, dvs. hjå kretsprodusenten.

Ein vil nå gå nærare inn på desse hovudtypane av programmerbare kretsar.

SPLD og CPLD

Dei første programmerbare kretsane, PLA, hadde to påfølgjande trinn eller rekkjer ("array") der begge var programmerbare. Kretsane var dyre å produsera samt trege.

Neste generasjon blei effektivisert ved å gjera den såkalla ELLER-rekkja fast. Dette er vist i figur 3.2.



Figur 3.2: Oppbygginga av enkle programmerbare kretsar, SPLD.

Kretsar av typen PAL og GAL, dvs. **SPLD**-kretsar av i dag, har ein slik struktur.

Eksempel 3.1
>
Viss ein med kretsen i figur 3.2 vil realisera den logiske funksjonen

$IO_0 = I_1 I_2 + I_3 I_4$,

kan dette realiserast ved å la dei såkalla **produkttermene** eller OG-termene P_0 og P_1 bli som følgjer:

$P_0 = I_1 I_2, \;\; P_1 = I_3 I_4$

Dette blir realisert ved å programmera samband mellom inngangar og tilhøyrande OG- eller produktrekkjer. Ein har også som vist i figur 3.2, dei inverterte inngangsnivåa tilgjengelege for bruk i OG-rekkja.
Dei to produkta P_0 og P_1 er som vist i figuren, 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 3.2.
  Kretsen GAL22R10 er feks. ein krets med til saman 22 inn- og utgangar der 12 av desse er I-linjer og 10 er IO-linjer.

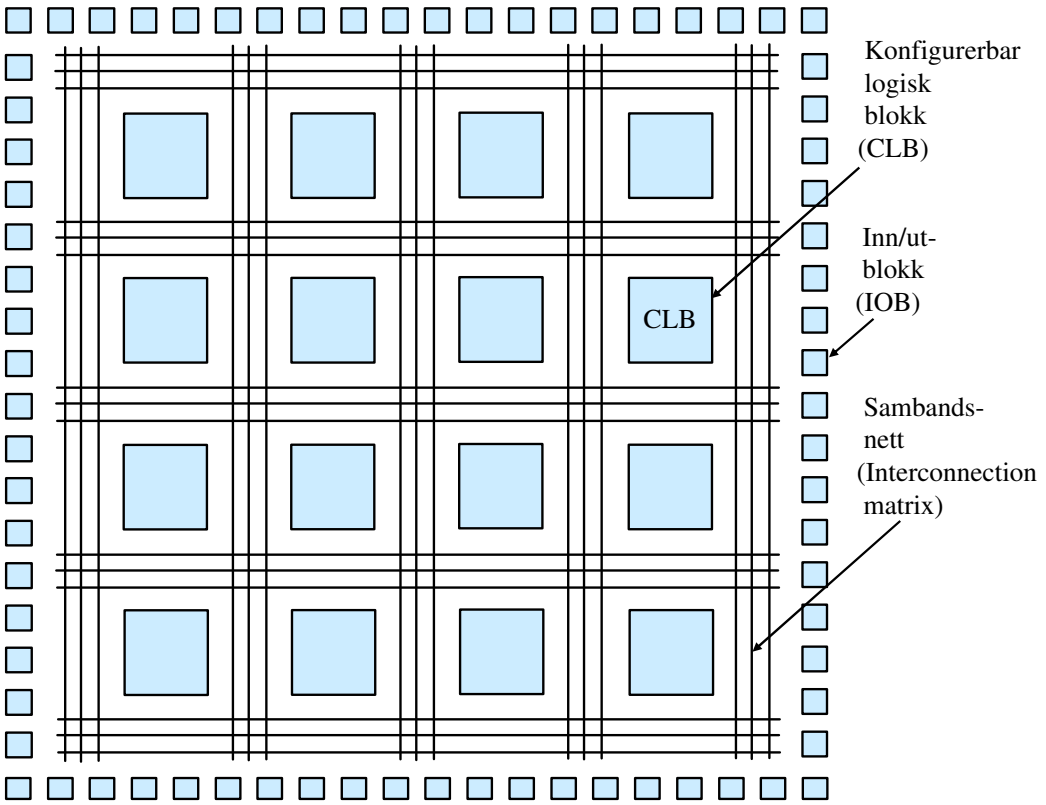
- Ei D-vippe for kvar utgang.
  Utgangen på sjølve vippa kan koblast tilbake til OG-rekkja slik at ein kan realisera ein tilstandsmaskin som f.eks ein teljar.
  Bokstaven "R" i GAL22R10 vil seia at ein har eit register i utgangsblokka, dvs. D-vippe på kvar av utgangane.

Adressedekoderen i mindre datamaskiner, jfr. figur 2.2, er eit eksempel på ein logisk modul som ofte er blitt realisert i SPLD-kretsar.

CPLD-kretsane er som nemnt kraftigare kretsar. Desse er grovt sett bygd opp av mange SPLD-modular som er knytt saman vhja. eit programmerbart grensesnitt ("programmable interconnection array", PIA).

FPGA

Strukturen til FPGA-kretsar er vist i figur 3.3.



Figur 3.3: Oppbygging av tradisjonelle FPGA-kretsar.

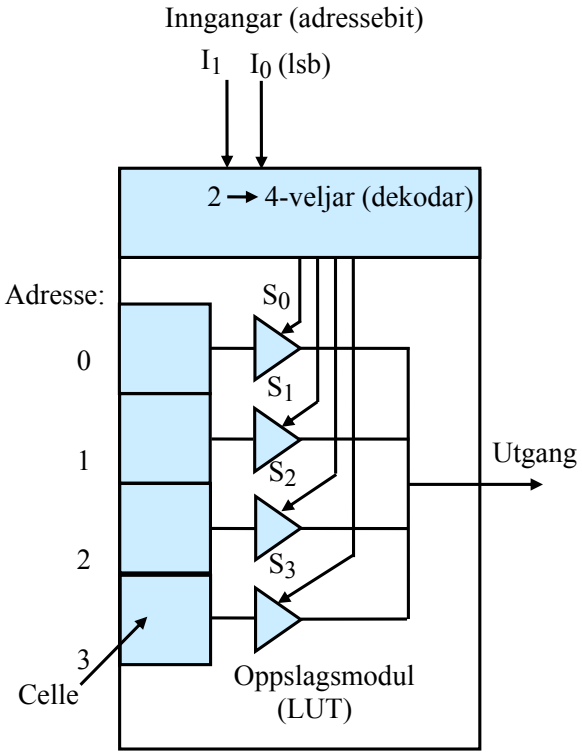
Dei konfigurerbare, dvs. programmerbare logiske blokkene, CLB, er kvar for seg mindre enn blokkene i ein CPLD, men det er mange fleire av desse. Det er også eit meir omfattande sambandsnettverk i ein FPGA.
Slik blir FPGA-kretsane generelt sett mykje kraftigare og fleksible enn CPLD-kretsar.

Kvar CLB i ein FPGA er bygd opp av fleire mindre logiske blokker. I kvar av desse blir ofte dei logiske funksjonar generert vhja. **oppslagstabellar** ("look-up table", LUT). LUT er eit alternativ til SOP-prinsippet som altså blir brukt i SPLD-kretsar, jfr. figur 3.2.

Virkemåten til LUT er illustert vhja. ein enkel logisk modul med to inngangar og ein utgang vist i figur 3.4.

Inngangane $I0-1$ utgjer her samla sett ei 2-bits adresse, og vhja. veljarlogikk vil ein på utgangen få ut innhaldet i den cella som har den valde adressa. 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 sanningstabellen i figuren.

Vanlegvis er talet på adressebit mykje større enn vist i figur 3.4. Modulen gir ei



I_1	I_0	Celleinnhald ved feks. XOR-funksjon
0	0	0
0	1	1
1	0	1
1	1	0

Figur 3.4: Modul som realiserer ein enkel logisk funksjon basert på oppslagstabell.

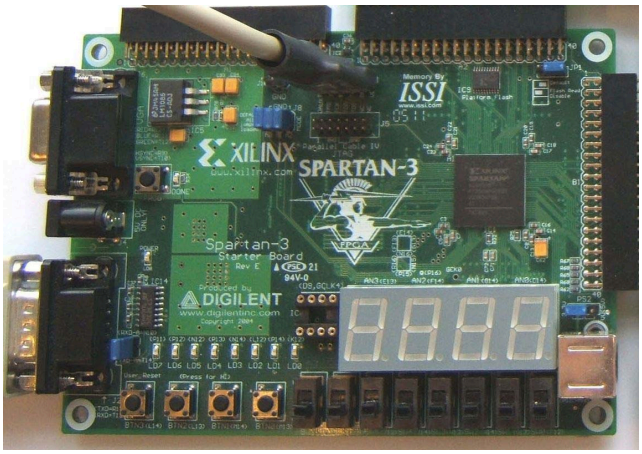
fleksibel og effektiv realisering av logiske funksjonar.
Merk at ein slik **LUT-modul** ikkje er noko anna enn ein rein **SRAM**-modul, og eignar seg difor godt i FPGA der kvar CLB ofte har mange små RAM-område.

Figur 3.3 viser ein tradisjonell FPGA-kretsar. Mange FPGA-kretsar av i dag kan i tillegg til CLB-ane og inn/ut-blokkene innehalda følgjande:

- "Block"RAM-modular, som kan brukast til å realisera data- og programminne for ein μP.
- Multiplikasjonsmodular.
- Faste mikroprosessorar.
- Raske kommunikasjonsmodular, dvs. med ratar $>$ 1Gbit/sek.

Hovudkortet til øvingsmaskinen vår, UiS1, inneheld ein FPGA-krets av typen **Spartan 3** frå Xilinx, sjå figur 3.5.

Utgåva heiter XC3S200, og denne har 200.000 portar.
Desse portane er fordelte på 480 CLB-ar, ca. 32kByte SRAM og 12 multiplikasjonsmodular.
Kraftigare kretsar frå Xilinx, som Virtex-familien, har i tillegg ein eller fleire faste kjerner, dvs. mikroprosessorar, av typen PowerPC. Ein kan då realisera ein datamaskin inne i FPGA-kretsen. Som vist i figur 2.2, kallar ein dette for ein mikrokontroller.



Figur 3.5: Hovudkortet Spartan-3 Starter Board frå Digilent, [4]

.

Den einaste modulen som ein til nå ikkje har hatt til disposisjon i FPGA, er **permanentminne**, som feks. Flash. Slike minne held som kjent på innhaldet sjølv om ein slår av kraftforsyninga.
FPGA-en som ein bruker, er som nemnt basert på SRAM-teknologi og er difor tom 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. Hovudkortet til øvingsmaskinen UiS1 har eit Flash-minne på 2Mbit, sjå modulen rett over Spartan-kretsen i figur 3.5.

Som eit alternativ til fast prosessorkjerne i FPGA eller ein mikroprosessorkrets kobla 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[4].
Meir om programmering kjem i neste kapittel. Meir om sjølve mjukprosessoren kjem i kapittel 3.2.

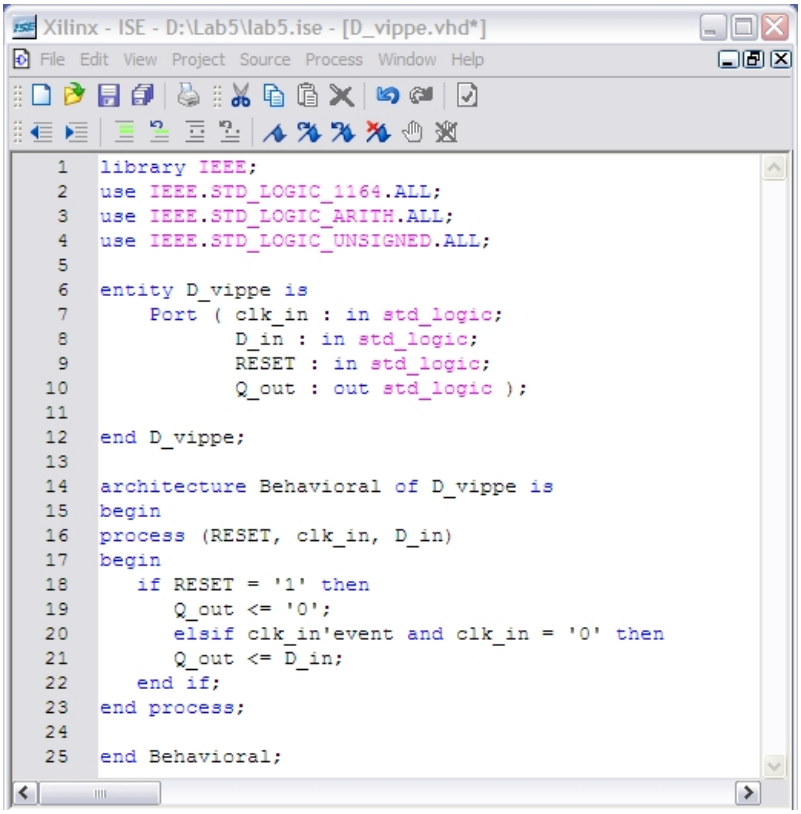
3.1.3 Programmering av logikk

Når ein skal programmera logikken i ein programmerbar krets, dvs. realisera ein logisk funksjon, bruker ein vanlegvis eit passande **programmeringsspråk** for å spesifisera dette. Logikk av lite omfang kan alternativt spesifiserast vhja. eit logikkskjema.
Eit vanleg språk for maskinvareprogrammering er **VHDL**[5].

[4] "Intellectual Property" 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 timer eller ein grensesnittmodul mot eit eksternt minne.

[5] "VHSIC Hardware Description Language", VHDL, der VHSIC - "Very High Speed Integrated Circuit".

Eit eksempel på oppsett av ein enkel funksjon i dette språket, her ei D-vippe, er vist i figur 3.6. Andre språk er mellom anna Verilog og ABEL.

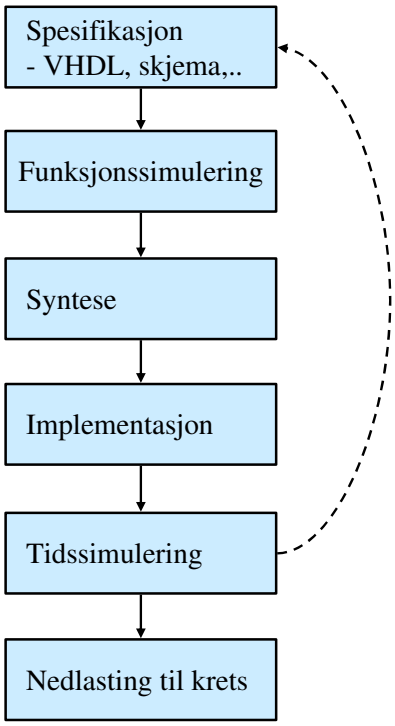


Figur 3.6: Eksempel på oppsett av ei D-vippe i VHDL.

Som ein ser i figuren, set ein først opp **inn- og utgangar** og spesifiserer så **arkitekturen**, dvs. dei logiske funksjonane.

Ved programmering av logikk med stort omfang, bruker ein ofte verktøy av høgare generasjon.
Xilinx har 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.

For å kunne leggja dette inn i ein programmerbar krets, må programkoden **kompilerast**, dvs. omsetjast. Vanlege trinn i ei slik omsetting er vist i figur 3.7.



Figur 3.7: 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 eksemplet i figur 3.6.
- **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 der ein fjernar overflødig logikk og bruker dei mest effektive portkombinasjonane. Til slutt blir det produsert ei nettliste, der ein 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 gliper eller andre tidsproblem pga. etterslepa ("propagation delay") gjennom portane, må ein endra spesifikasjonen.
- **Nedlasting**
  Implementasjonstrinnet gir som resultat ei såkalla **bitfil** som her blir overført

som ein **bitstraum** til kretsen via eit standard grensesnitt. I kretsen vil programkoden bli lagt i eit programminne, vanlegvis ei SRAM-blokk. Resten av bitane vil grovt sett bli ruta til oppslagstabellar, sjå figur.3.4, slik at dei logiske modulane i FPGA-en samla sett realiserer den spesifiserte mjukC-en.

3.1.4 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 og mjukprosessor.
- 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.
  Prosessoren er i hovudsak lik, mens det bare er sjølve kretskapasiteten og klokkefrekvensen som varierer frå system til system.
- Oppdateringar av både program- og maskinvare kan overførast til sluttbrukar og utførast der.

Nokre ulemper er det også ennå som vist under:

- Pga. fleksibiliteten har kretsane høgt effektforbruk. Dette er eit problem i mobilt utstyr, men lågeffekts-utgåver er på veg. Konstruksjonen kan alternativt overførast til ASIC, men dette gir lenger utviklingstid og krev omtanke.
- Utviklingsverktøya er på nokre felt ennå litt primitive, f.eks. på avlusing, men blir stadig utvikla.
- Ein FPGA må konfigurerast på nytt ved kvart straumpåslag. Dette gjer systemet sårbart med omsyn på piratkopiering.

3.2 Mjukprosessor

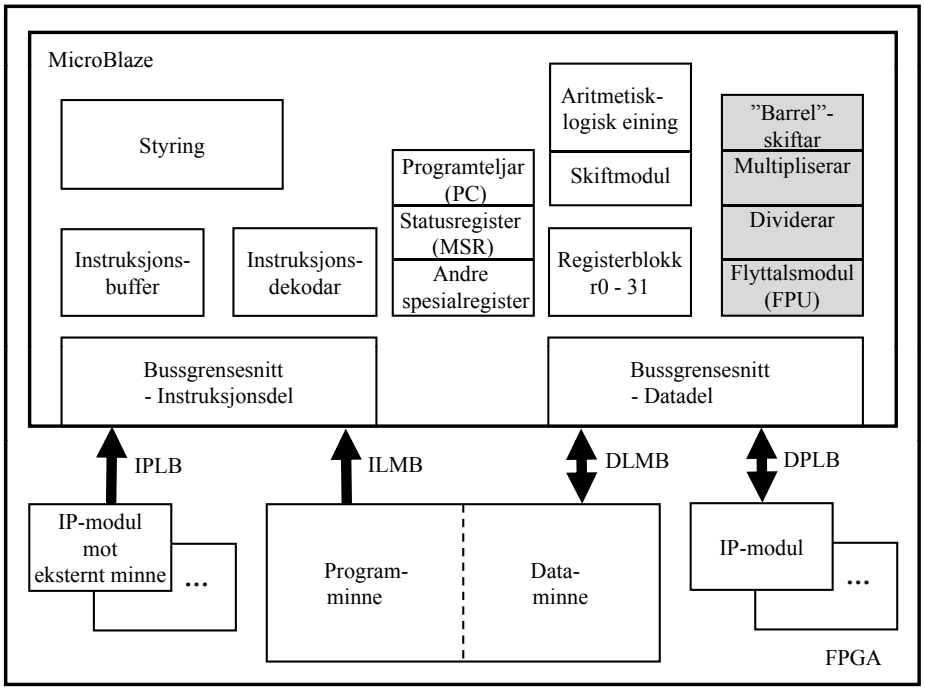
Den første mjuke prosessoren ("soft processor") kom i år 2000.
I dag fins mellom anna følgjande mjukprosessorar på marknaden:

- 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 blir brukt ved UiS.

3.2.1 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 3.8.



Figur 3.8: 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. Som ein ser av figur 3.8, er OPB likevel framleis tilgjengeleg i MicroBlaze.

- **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, [3].

Som nemnt i kapittel 2.2.1, har MicroBlaze ei registerblokk med dei 32 registra *r0-31*. Alle registra er på 32 bit. I tabell 3.2 i referansemanualen, [3], 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 subrutinekall.
- *r15* er lenkeregister, dvs. inneheld returadressa frå ei subrutine.

I tillegg har MicroBlaze mange spesialregister. Ein 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.

Nokre eksempel på registerbruk samt bruk av stakk blir gitt i neste delkapittel.

Merk at nokre av blokkene i MicroBlaze er valfrie, sjå figur 3.8. Dei kvite delane utgjer minimumsutgåva av mikroprosessoren, mens dei grå må spesifiserast ut frå behov. Desse, som feks. 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 3.8, vil funksjonane til desse vera rimeleg klare ut frå det som er skrive om mikroprosessorar generelt i kapittel 2.

3.2.2 Litt om MicroBlaze sitt instruksjonssett

Instruksjonssettet til MicroBlaze er vist i tabell 3.2. Ved å spesifisera bruk av tilleggsmodular som feks. dei som er gråfarga i figur 3.8, kan ein få tilgang til delar av dei skråstilte instruksjonane i tabellen.
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 3.2: Nokre sentrale MicroBlaze-instruksjonar.

Detaljar på desse samt resten av instruksjonssettet kan finnast i kapittel 4 i referansemanualen til MicroBlaze, [3].

To spesielle detaljar skal likevel 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, sjå figur 2.6.
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 3.2
>
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 Miroblaze 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 3.2, 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.

Ved bruk av **stakk** og lenkeregister, kan ein også realisera **nesta** rutinekall. Dette er vist i eksempel 3.3.

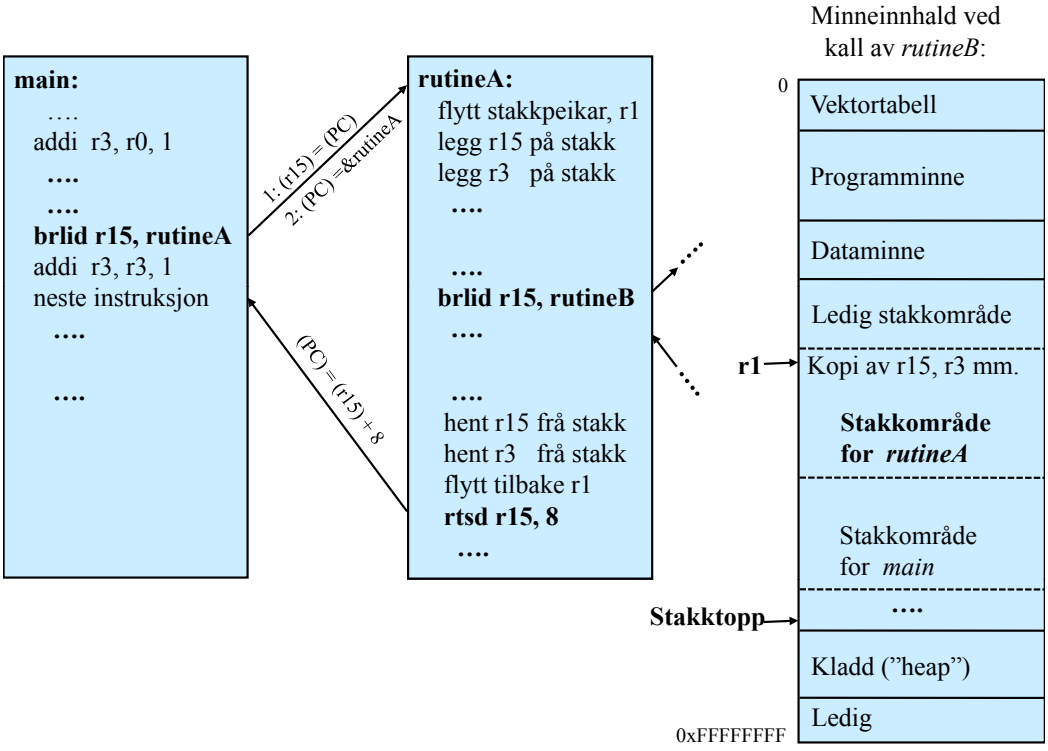
Eksempel 3.3
>
Hovudprogrammet

```
main() {
   .....
   Xuint32 i=1; //lokal variabel som f.eks. blir lagt i register r3.
   .....
   i++;
   rutine_A();
   .....
}
```

innheld nesta subrutinekall av *rutine_ A* og *rutine_ B*, dvs. at *rutine_ A* vil kalla *rutine_ B*. Førstnemnde rutine er vist i assembly-format øvst på neste side. Ein tenkjer seg som vist at *rutine_ A* også gjer bruk av register *r3*.

```
rutine_A() {
    .....
    brlid   r15,rutine_B
    addi    r3,r3,1
    .....
}
```

Nestinga gjer det nødvendig å bruka stakk slik at ein ikkje mistar det "gamle" innhaldet i lenkeregisteret ved kall av *rutine_B*. I tillegg bruker ein i hovudproggrammet *r3* til lagring av ein lokal variabel. Ved bruk av dette registeret i ei subrutine, må ein kopi av den "gamle" verdien også lagrast på stakken.
Det siste som blir gjort før retur frå ei rutine, er henting av dei gamle verdiane frå stakken.
Alt dette er illustrert i figur 3.9.



Figur 3.9: Register- og rutinehandtering ved subrutinekall.

Prosessorregister *r1* er som nemnt stakkpeikar. Kvar rutine får tildelt eit stakkområde under køyring, og *r1* peikar på starten av dette området. Som vist i figuren, er det første som skjer ved køyring av *rutine_A*, at stakkpeikaren blir flytt eit visst stykkje nedover[6] i minnet. I det området som då oppstår, blir dei viktige registerverdiane lagt inn, som forklart over.
Det siste som skjer før retur frå rutina, er altså henting av registerverdiar og så flytting av stakkpeikaren tilbake til stakkområdet for den rutina ein returnerer til, som her er *main*. Ein seier då at stakkområdet for *rutine_A* er tømt, dvs. at det ikkje eksisterer lenger.

[6] Mot ei lågare minneadresse.

I figuren er det også vist kva som skjer med programteljaren (PC) ved kall av subrutiner. Etter at adressa til hoppinstruksjonen er blitt plassert i *r15* ved kall av *rutine_A*, blir altså startadressa[7] for subrutina lagt inn i PC, og det fører til at mikroprosessoren går i gong med å køyra *rutine_A*. Merk at ved retur blir den verkelege returadressa *(r15)*[8] + *8*[9], då både hoppinstruksjonen og den etterfølgjande instruksjonen er utførte.

>

3.2.3 Avbrotshandtering

I førre seksjon blei det mellom anna vist korleis mikroprosessoren handterer kall av ei subrutine. Ein nokså tilsvarande situasjon oppstår viss mikroprosessoren får eit **avbrotssignal** ("interrupt"). Det kan feks. 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[10].
μP-en må då køyra ei såkalla avbrotsrutine ("interrupt service routine", ISR), som inneheld alt som må gjerast når eit slikt avbrot kjem.

Det som skjer ved utføring av ei avbrotsrutine, er vist i figur 3.10. Her er det tatt eksempel nettopp i ein taimermodul frå Xilinx, [6], som er sett opp til å gi avbrot når han har talt seg ned til null frå ein gitt startverdi[11].
μP-en køyrer først ferdig den instruksjonen han held på med, og hoppar så til avbrotsrutina. Ved avbrot er det som vist registeret *r14* som blir brukt til lagring av returadressa.

Ein ser også at avbrotsbiten IE i MicroBlaze sitt statusregister, MSR, blir sett til null under sjølve avbrotshandteringa. Andre avbrot uavhengig av prioritet må altså venta. I meir avanserte system vil avbrot med høgare prioritet kunne bryta av køyringa av ei avbrotsrutine. Eit slikt **nesta** avbrotssystem kan realiserast her også, men vil krevja at ein lagar noko passande assemblykode i starten og slutten på kvar avbrotsrutine.

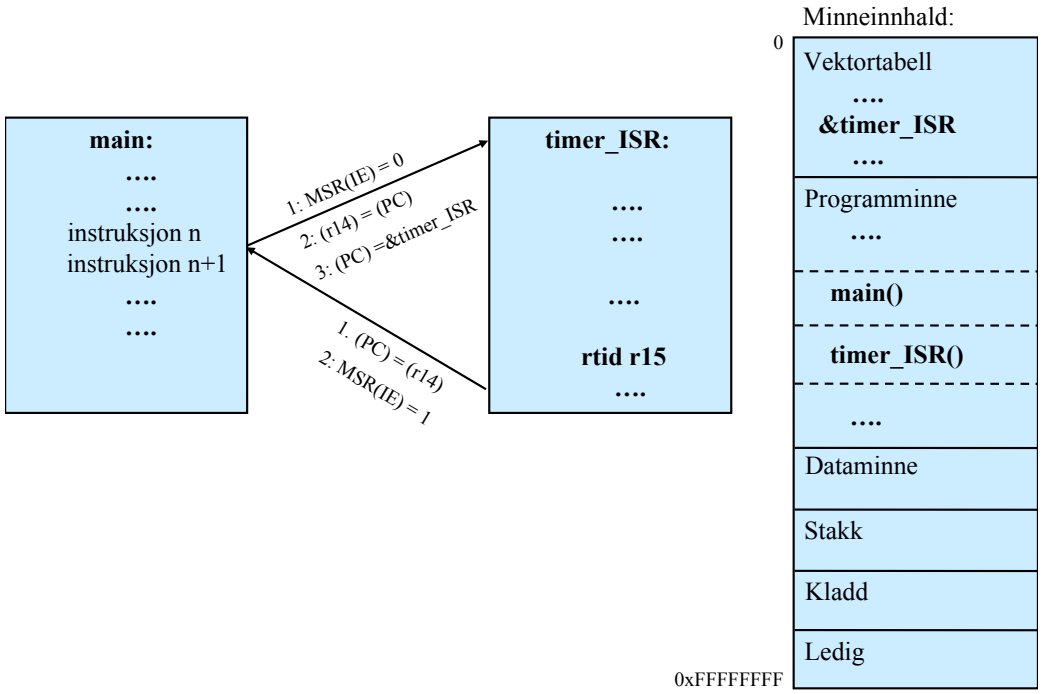
[7] Teiknet "&" framfor eit variabel- eller rutinenamn betyr "adressa til" i C

[8] (r15) betyr innhaldet i register r15.

[9] Alle instruksjonane er på 32 bit, dvs. at dei tar opp 4 byte kvar i minnet.

[10] 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 handtering av ei eiga rutine. Meir om dei ulike unntaka finn ein i kapittel 1 i referansemanualen til MicroBlaze, [3].

[11] Ved i tillegg å setja opp taimeren slik at han automatisk startar på nytt frå same startverdi, kan ein realisera eit konstant sampleintervall i systemet. Prosessoren kan då ved kvart avbrot gjera dei tinga som må skje regelmessig, som feks. innlesing og prosessering av data, og setting av styresignal.



Figur 3.10: Køyring av ei avbrotsrutine.

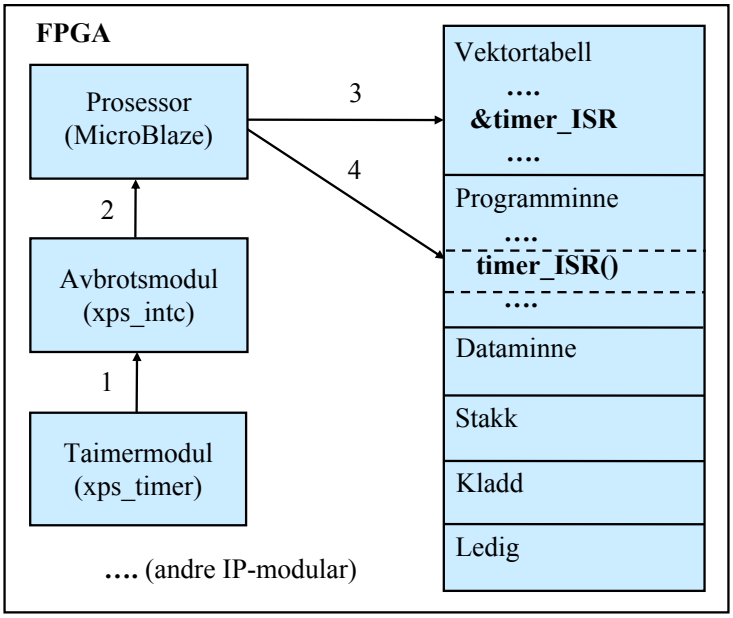
For at ei avbrotsrutine skal bli køyrt etter at ei kjelde har gitt eit avbrotssignal, må mikrokontrolleren vera konfigurert eller sett opp på rett måte. Figur 3.11 viser handteringsvegen for eit avbrot frå taimermodulen i eksemplet over. Viss alle dei 4 vegane i figuren skal vera opne, må følgjande vera oppfylt:

1. Kontoll- og statusregisteret i taimeren, **Timer Status and Control Register**, må vera sett opp til å gi avbrot ("enable interrupt").

2. Hovudregisteret Master Enable Register i avbrotsmodulen ("interrupt controller") må vera sett opp til å sleppa gjennom avbrot. Viss ikkje, vil ingen avbrot koma vidare frå denne modulen.
   I tillegg må ein i maskeringsregisteret Interrupt Enable Register fjerna maskene for dei avbrotskjeldene ein vil sleppa gjennom. Her har kvar kjelde sin bit, og den minst signifikante plasseringa i registeret gir høgaste prioritet.

3. For at μP-en skal bry seg om eit avbrot, må biten Interrupt Enable i statusregisteret til prosessoren, MSR, vera sett lik 1. Prosessoren vil då gå til den såkalla vektortabellen for å leita opp startadressa eller vektoren til den aktuelle avbrotsrutina.

4. For å få oppstart av ei avbrotsrutine, må altså startverdien for denne vera plassert på rett plass i vektortabellen. Dette blir normalt gjort som ein del av initialiseringa av maskinvaren etter ein oppstart.



Figur 3.11: Oppsett og handtering av eit taimeravbrot.

3.3 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.

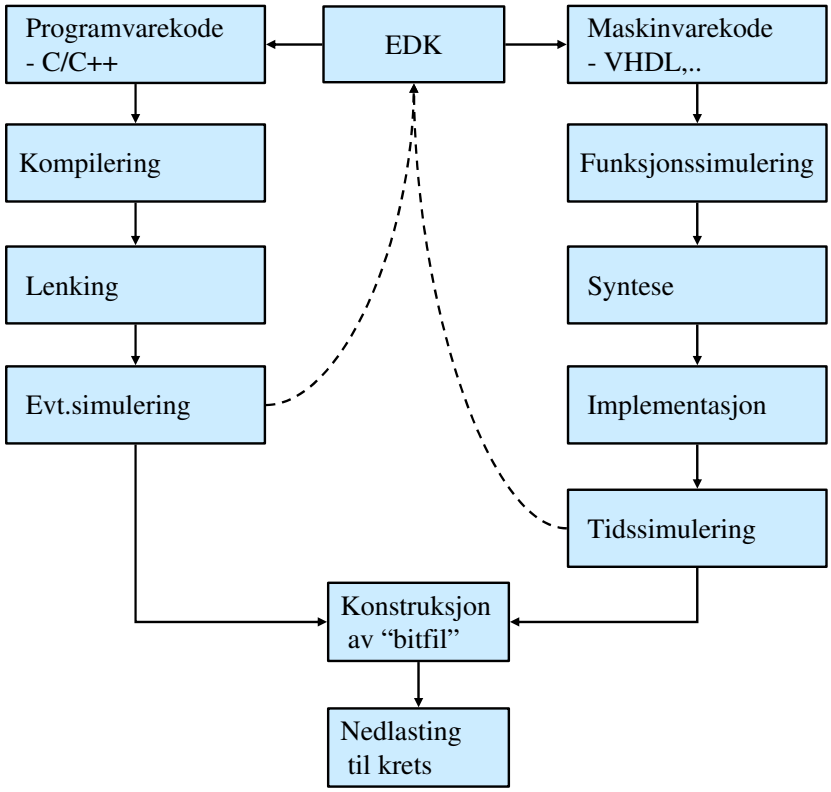
3.3.1 Maskinvarespesifikasjon og programmering

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 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 i fleire trinn før nedlasting som vist i figur 3.12.
Maskinvarekoden eller -spesifikasjonen blir då omsett til VHDL-kode som nemnt i kapittel 3. Dei ulike trinna i kompileringa av denne er som vist i kapittel 3.1.3.



Figur 3.12: Omsetting av kode generert i verktøyet Embedded Development Kit

Meir om handtering av **programkode** er vist i eksemplet i vedlegg A, kapittel A.4. I tillegg tar vedlegg C opp både generelle sider og verktøydetaljar ved programmering av ein MicroBlaze-basert mikrokontroller.

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 3.1.2 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.
Framgangsmåte for programmering av Flash-minnet på kortet "Spartan-3 Starter Board" frå Xilinx er vist i vedlegg D.

Kapittel 4

Grensesnittkonstruksjon

Nemninga **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 realisert i FPGA. I tillegg har ein grensesnitt mellom denne mikrokontrolleren og perifermodular utanfor FPGA-en, dvs. **eksterne** modular.

Dette kapitlet omhandlar konstruksjon av grensesnitt generelt men med eksempel henta frå ein MicroBlaze-basert mjukkontroller som skal koblast til eksterne modular.

4.1 Hovudtypar av grensesnitt

Ein har to hovudtypar av grensesnitt, nemleg dei som her blir kalla **programmerte** grensesnitt og **dekoda** grensesnitt.
I eit programmert grensesnitt realiserer ein kvart steg i ein overføringssyklus vhja. programkode, mens i eit dekoda grensesnitt vil mikroprosessoren sjølv generera ein overføringssyklus når han skal utføra ein lagre- ("store") eller lastinstruksjon ("load")[1].

Ein vil nå først seia litt om overordna krav til grensesnitt og så gå nærare inn på dei to hovudtypane. Her vil ein også gå gjennom detaljerte konstruksjonseksempel for slike grensesnitt.

[1] 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 blir ofte realisert vhja. programmerbar logikk. Viss mikrokontrolleren er mjuk, kan signala som regel skreddarsyast slik at koblinga kan gjerast direkte mot perifermodulen.

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.

Ein vil i kvart av konstruksjonseksempla under avsnitta for programmerte og dekoda grensesnitt gå gjennom desse tre punkta.

4.3 Programmert grensesnitt

Dette kapitlet tar utgangspunkt i eit grensesnitteksempel, nemleg ein LCD-modul kobla mot ein mjuk MicroBlaze-basert mikrokontroller. Først ser ein på det programmerte grensesnittet generelt og så blir dette realisert i detalj.

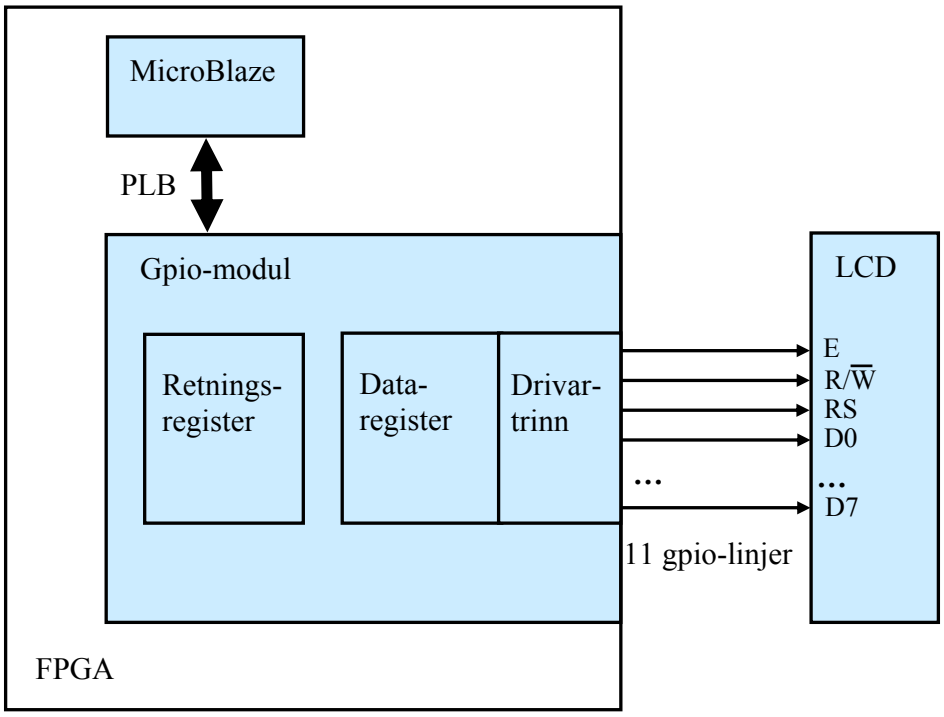
4.3.1 Innleiing

Figur 4.1 viser eit eksempel på eit programmert grensesnitt mellom ein MicroBlaze-basert mikrokontroller i FPGA og ein LCD-modul[2]. Som vist realiserer ein grensesnittet vhja. ein GPIO-modul[3].

Signala inn og ut av ein GPIO-modul må tilordnast kvar sin bit i dataregisteret i modulen. I tillegg blir då signala tilordna tilsvarande bit i eit retningsregister ("data direction register"). Dei bitverdiane ein legg inn her, bestemmer om signalet skal gå ut av eller inn i modulen. I eksemplet i figuren går alle signala ut til LCD-modulen. Det er difor i figuren også vist eit drivartrinn som gir standardiserte spenningsnivå og tilstrekkjeleg drivstraum ut mot den eksterne modulen.

[2] Denne er bygd opp av eit skjermelement og ein kontrollerkrets som styrer alle punkta på skjermen.

[3] "General Purpose Input/Output"-modul, også kalla parallellport.



Figur 4.1: Eksempel på eit programmert grensesnitt.

Inn-signal vil gå gjennom eit inngangsbuffer før verdiane hamnar i dataregisteret. Dette er også ein del av GPIO-modulen, men er ikkje vist i figuren.

I eit programmert grensesnitt legg ein vhja. programkode verdi etter verdi inn i dataregisteret i parallellporten slik at signala går i ein sekvens som spesifisert i databladet for den eksterne modulen som er tilknytta parallellporten.

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 dataregisteret i GPIO-modulen. Viss det er meir enn 32 signal, må ein bruka begge dataregistra i ein dobbel GPIO-modul[4] eller fleire enkle GPIO-modular.

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 signala.

3.4. Lag detaljert programkode.

[4]GPIO-modulane frå Xilinx, [8], kan gjerast doble ved bygging av maskinvaren for ein mjuk mikrokontroller, jfr. vedlegg A og B.

Punkt 3 i kapittel 4.2 er altså delt opp i fire delar her.

4.3.2 Realisering av eit grensesnitt for overføring av teikn til LCD-skjerm

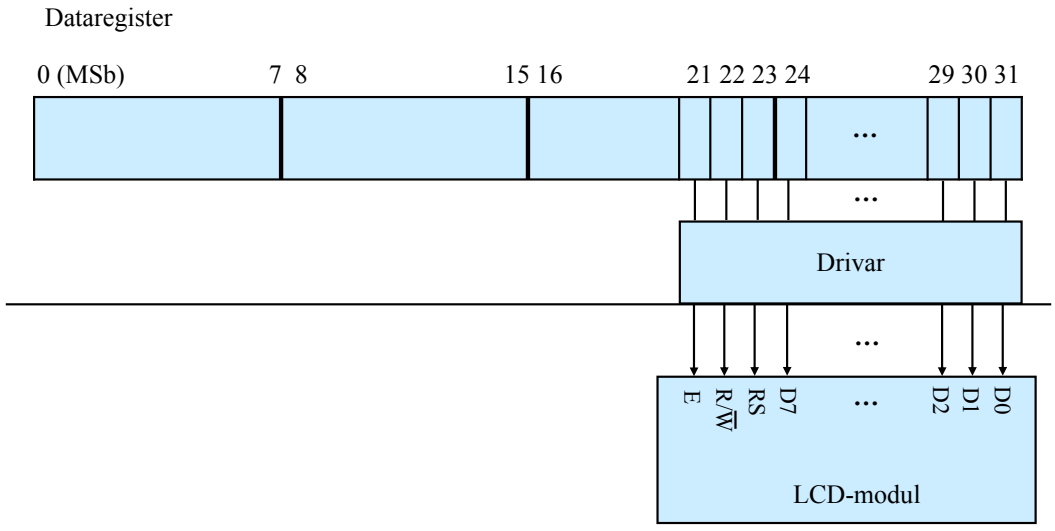
I det følgjande eksemplet skal ein sjå i detalj på korleis ein kan realisera ei overføring av eit teikn til LCD-skjermen vist i figuren over. I eksemplet her har ein valt teiknet A[5].
For å skriva ut A-teiknet i neste ledige posisjon på skjermen i figur 4.1, må ein skriva ei rekkje etterfølgjande bitmønster til dataregisteret i GPIO-modulen. Dette skal realiserast gjennom punkta 1 - 3 på førre sida.

Signaltilkobling

Signaltilkobling skjer altså ved å tilordna bit i dataregisteret til dei ulike signala. Dette er punkt 1 ved realisering av eit programmert grensesnitt.

Dei 11 LCD-signala skal då knyttast til kvar sin bit i dataregisteret til GPIO-en. Funksjonen til desse 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[6].
- D(ata) 0 - 7 inneheld teiknet eller kommandoen som skal overførast.

Tilordning av bit i dataregisteret kan gjerast som vist i figur 4.2.



Figur 4.2: Tilordning av bit i GPIO-en sitt dataregister.

[5]Ein kan lett generalisera eksemplet til å skriva ut ein variabelverdi.
[6]Ein kommando kan feks. vera å sletta heile skjerminnhaldet, eller å flytta til ein viss posisjon.

Rekkjefølgja av signala er ikkje kritisk, men det kan vera lurt å la datadelen okkupera den minst signifikante delen av dataregisteret[7]. Då unngår ein skiftoperasjonar når data, dvs. teikn eller kommandoar, skal leggjast inn i registeret. Meir om dette kjem i pkt.3.3-4.

Dei ulike signala vil som nemnt vera tilordna same bitnummer i GPIO-en sitt retningsregister.
Merk at data- og retningsregisteret alltid har ei breidd på 32 bit. Drivaren, sjå figur 4.1, vil derimot i ein mjuk mikrokontroller få ei breidd gitt av talet på signal.

Støymarginar og drivekapasitet

Dette er punkt 2 ved realisering av eit grensesnitt. Framgangsmåten blir som ved realisering av dekoda grensesnitt og er tatt opp der, sjå kapittel 4.4.

Tidskrav

Dette punktet kan som vist i innleiinga, delast opp i 4 passande underpunkt ved realisering av programmerte grensesnitt. Punkta blir gjennomgått i det følgjande.

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 LCD-modulen, [9]. Diagramma fortel oss kva krav som må oppfyllast for at modulen skal greie å motta eller senda data på rett måte. I datablad vil det i tillegg vera tabellar som viser typiske, samt maksimums- og minimumsverdiar for dei tidene som er med i diagramma.
Figur 4.3 viser eit forenkla diagram for ein skrivesyklus mot LCD-en, der bare det som er viktig for overføringa vår, er med[8].

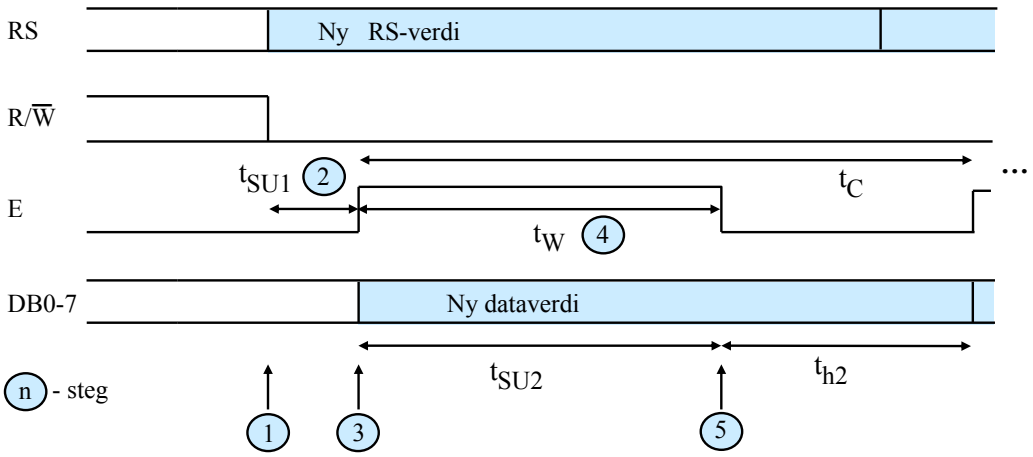
Ei overføring av eit teikn til LCD-en skjer ved at signala som vist i figuren, går i ein viss sekvens. Ved å dela opp ei overføring i passande **steg**, kan ein oppfylla tidskrava som LCD-modulen set. Eit steg representerer ein viss signalkombinasjon eller -tilstand.
I dette tilfellet bør ein som vist i figuren, 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), $t_W \geq 400$ ns og $t_{SU2} \geq 140$ ns.

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.

[7] Merk at MicroBlaze er bit-reversert, dvs. at bit 0 alltid er MSbit. Meir om dette står i tabell 1.2-4 med tilhøyrande tekst i [3]. Dei fleste periferkomponentar slik som LCD-modulen, er ikkje bit-reverserte, så det gjeld å passa på når ein lagar grensesnitt her.

[8] Eit diagram i eit datablad innheld svært mange opplysningar. Det er ei utfordring å sila ut det vesentlege.



Figur 4.3: Forenkla tidsdiagram for ein skrivesyklus.

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 tillegg er det slik at LCD-modulen treng ei viss tid på å skriva eit teikn ut på skjermen eller å utføra ein kommando. Desse tidene er gitt i tabell 5 i [9].

Punkt 3.2: Pseudokode for overføringsprogrammet
Pseudokoden for overføringa blir som følgjer:

1. Skriv eit mønster der $E = 0$, $RS = 1$, $R/\overline{W} = 0$.
2. Vent minst like lenge som kravet til oppsettingstid ("setup") for RS og $R/\overline{W}$.
3. Skriv eit mønster der $E = 1$, $RS = 1$, $R/\overline{W} = 0$ og $D0 - 7 =$ 'A' [9].
4. Vent minst like lenge som kravet til pulsbreidde ("pulse width") for E[10].
5. Skriv eit mønster der $E = 0$[11], $RS = 1$, $R/\overline{W} = 0$ og $D0 - 7 =$ 'A'.

[9] Denne skrivemåten gir ASCII-koden til teiknet A. Dette er ein vanleg standard for koding av tekst.

[10] Ein vil då samstundes også oppfylla kravet til oppsettingstid for data, som er mindre enn pulsbreiddekravet for E.

[11] 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 aldri endrar databussverdien før ved neste overføring, vil kravet til haldetid vera oppfylt med svært god margin.

Punkt 3.3: Bitmønster for aktivering og deaktivering av styresignal
Basert på tilordninga vist i figur 4.2 kan ein setja opp bitmønster som vil aktivera[12] og deaktivera styresignala. Ein innfører her følgjande definisjonar:

E_1[13] = 0100 0000 0000 binært = 0x400[14], E_0 = 0000 0000 0000 binært = 0x000
RW_1 = 0010 0000 0000 binært = 0x200, RW_0 = 0000 0000 0000 binært = 0x000
RS_1 = 0001 0000 0000 binært = 0x100, RS_0 = 0000 0000 0000 binært = 0x000

I eit C-program vil desse definisjonane sjå slik ut:

```
#define E_1     0x400
#define E_0     0x000
#define RW_1    0x200
#define RW_0    0x000
#define RS_1    0x100
#define RS_0    0x000
```

Merk at ein her ser på signala kvar for seg. Ved å bruka **eller-operasjonar** i programkoden kan ein få til eit vilkårleg mønster for styresignala, sjå neste punkt.

[12] For eit aktivt høgt signal som feks. E vil aktivering vera det same som å setja signalverdien til '1', dvs. høg.
[13] Altså det bitmønsteret som set signalet E til '1'.
[14] "0x" er notasjon for heksadesimal representasjon i språket C.

Punkt 3.4: Programkode
Som regel kan ein dra nytte av eit bibliotek med ferdige drivarfunksjonar når ein skal laga program for slike grensesnitt. Her tenkjer ein seg at dei nødvendige drivarfunksjonar heiter:
- *sett_ datareg(baseadr,data)*
- *sett_ retnreg(baseadr,data)*[15]
- *vent(nanosekund)*

Med desse rutinene kan ein leggja inn eit bitmønster i data- og retningsregisteret for ein GPIO-modul som har baseadressa *baseadr*. I tillegg kan ein venta eit viss tid målt i nanosekund[16].
Basert på pseudokoden og bitmønstra over blir då programkoden for å overføra teiknet A til LCD-skjermen, følgjande:

....
sett_retnreg(baseadr, 0x000[17]);[18].
....
sett_datareg(baseadr, $(E\_0|RW\_0|RS\_1)$);
vent(60);[19]
sett_datareg(baseadr, $(E\_1|RW\_0|RS\_1|'A')$);
vent(400); [20]
sett_datareg(baseadr, $(E\_0|RW\_0|RS\_1|'A')$);
....

Denne programkoden kan ein med fordel plassera i ei eiga subrutine for skriving av teikn der ein overfører teiknet i subrutinekallet. Tilsvarande kan ein laga ei rutine for skriving av kommandoar, der altså $RS = 0$.
Med dette er realiseringa av grensesnittet fullført.

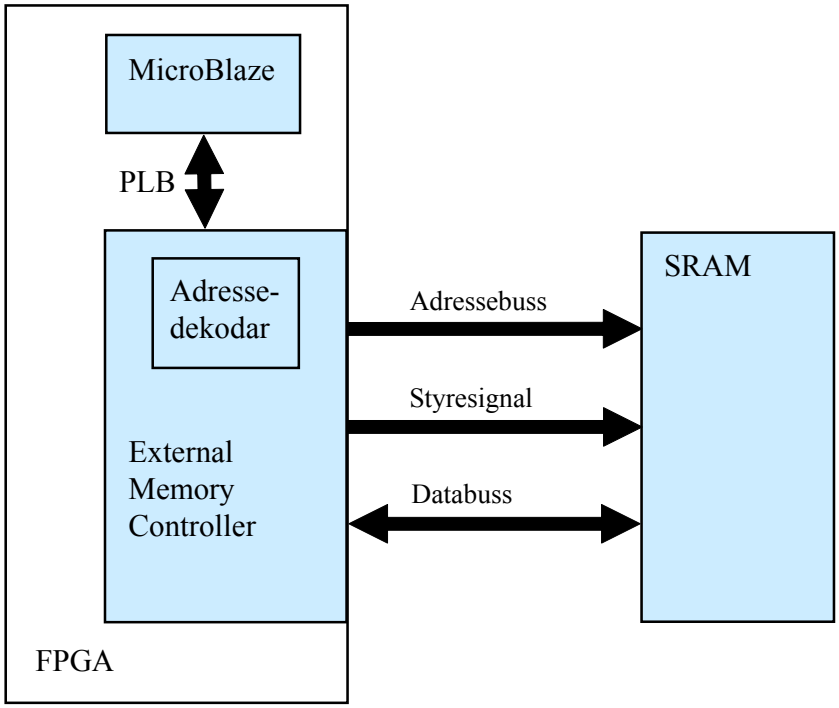
[15] I ein MicroBlaze-basert mikrokontroller vil desse heita *Xgpio_ mSetDataReg()* og *Xgpio_ mSetDataDirection()*.
[16] Ei presis rutine vil vera taimerbasert og med ein klokkefrekvens på 50MHz vil ein i praksis aldri greia å realisera venting kortare enn nokre hundre nanosekund.
[17] Alle signala er konfigurert som utgangar. Basis her er spesifikasjonen til Xilinx sin GPIO-modul.
[18] Viss signala alltid skal feks. vera utgangar, kan dette gjerast ein gong for alle i ei initialiseringsrutine.
[19] Denne kan sannsynlegvis sløyfast då tida det tar MicroBlaze å utføra programinstruksjonane frå steg 1 til steg 2, truleg vil overstiga tidskravet her.
[20] Her vil ein altså også oppfylla kravet til oppsett av data.

4.4 Dekoda grensesnitt

Dette kapitlet tar også utgangspunkt i eit grensesnitteksempel, nemleg eit SRAM-minne kobla mot ein mjuk MicroBlaze-basert mikrokontroller. Først ser ein på det dekoda grensesnittet generelt og så blir dette realisert i detalj.

4.4.1 Innleiing

Figur 4.4 viser eit dekoda grensesnitt mellom ein MicroBlaze-basert mikrokontroller i FPGA, dvs. ein **mjukC**, og eit eksternt SRAM-minne.



Figur 4.4: Eksempel på eit dekoda grensesnitt.

Bussgrensesnittet mot SRAM kan som vist realiserast vhja. perifermodulen "External Memory Controller", EMC[21], [7].

I dekoda grensesnitt er **adressedekodaren**[22] ein sentral komponent. I eksemplet vist i figur 4.4 er denne ein del av EMC-modulen.
Adressedekodaren lyttar på den interne adressebussen. Når feks. MicroBlaze i figur 4.4 vil gjera ein aksess mot den eksterne komponenten, dvs. når adressene på bussen er innanfor adresseområdet til den eksterne komponenten, køyrer EMC-modulen automatisk ein overføringssyklus mot SRAM-minnet.

[21] Denne perifer- eller IP-modulen heiter eigentleg Multi CHannel External Memory Controller då han støttar både PLB- og såkalla MCH-grensesnitt inne i mjukC-en.
[22] Sjå også kapittel 2.1.

Eksempel 4.1
>
Viss adresseverdien *adr1* ligg i register *r3*, og høyrer til adresseområdet for den eksterne SRAM-kretsen i figur 4.4, så vil instruksjonen

sw r7, r3, r0

generera ein skrivesyklus der ordet i register *r7* blir overført til adresse *adr1* i SRAM-minnet.
>

4.4.2 Realisering av eit grensesnitt mellom EMC og SRAM

Utgangspunktet ved all grensesnittkonstruksjon er altså punkt 1 - 3 i kapittel 4.2.

Signaltilkobling

Viss signala frå ein mikroprosessor eller -kontroller skal kunne koblast direkte til ein perifermodul, må desse ha både rett **funksjon** og **polaritet**. Viss feks. aktiveringssignalet til ein perifermodul er **aktivt lågt**, må mikroprosessoren eller -kontrolleren kunne gi ut eit tilsvarande signal.

Figur 4.5 viser detaljert signaltilkobling i grensesnitteksemplet frå figur 4.4.

SRAM-minnet er identisk med minnet som er plassert på baksida av hovudkortet til øvingsmaskinen vår, UiS1. Dette kortet er vist i figur 3.5.
Minnet er oppbygd av to kretsar organiserte som 256K x 16 bit, [10]. Samla sett gir dette ein databussbreidde på 32 bit og ein lagringskapasitet på 1MByte.

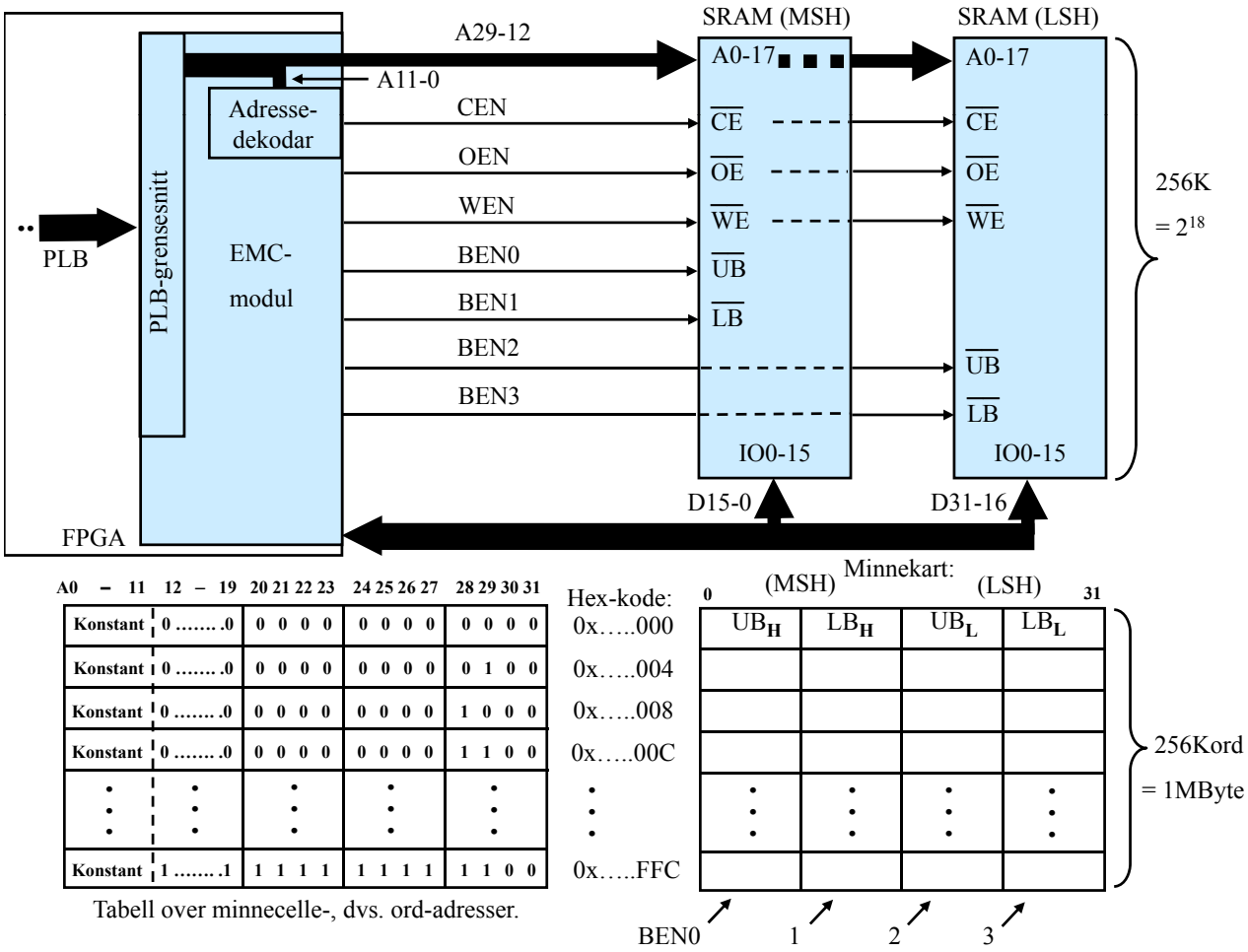
I figuren er det også vist eit minnekart ("memory map") for nedste del av minnet samt kva verdi dei ulike adressesignala har ved aksessar, dvs. lesing eller skriving, av dei nedste adressene. I det følgjande skal ein gå nærare inn på dei ulike signala som er viste i figuren:

- **Adressebussen**
  Minnestorleiken og organiseringa avgjer kor mange adresselinjer eller -signal ein treng. Her er minnet samla sett organisert som 256K x 32 bit,
  dvs. 256K-ord ("word"). For å kunne adressera eit vilkårleg ord av totalt $256K = 2^8 \cdot 2^{10} = 2^{18}$ ord, treng ein 18 adresselinjer.

  Då MicroBlaze som nemnt før, er bit-reversert[23], blir det **A29-12**[24] som skal koblast til A0-17 på dei to SRAM-kretsane.

  Merk også her at det alltid er den minst signifikante delen av adressebussen som er kobla direkte til minnemodulen, mens den mest signifikante delen går

[23] Jfr. tabell 1.2-4 med tilhøyrande tekst i [3].
[24] Som ein ser i tabellen nedst til venstre i figur 5, er A31-30 irrelevant når ein adresserer ord.



Figur 4.5: Dekoda grensesnitt mellom ein MicroBlaze-basert μC og ekstern SRAM.

til adressedekodaren. Denne delen blir altså brukt til å velja kva for ein systemkomponent, feks. SRAM-minnet, som skal veljast, jfr. kapittel 4.4.1.

- **Styresignala**
  Merk aller først at styresignal som er **aktivt låge** har endinga **N**. EMC-modulen tilbyr begge utgåvene. Her må ein velja N-utgåvene då alle styresignala til SRAM-kretsane er aktivt låge, sjå figur 4.5.

  CEN ("Chip Enable"): EMC-modulen har 4 ferdig dekoda CE-signal som kan brukast til å velja 1 av 4 ulike minnebankar. Her har ein bare 1 minnebank eksternt, nemleg SRAM-minnet. CE-signalet, her CEN, går som vist til begge kretsane, som er kobla parallelt.

  OEN ("Output Enable"): Aktivering av dette lesesignalet saman med CEN får minnet til å leggja ut data på databussen.

  WEN("Write Enable"): Aktivering av dette skrivesignalet saman med CEN får minnet til å lagra det som ligg på databussen.

  BEN3-0 ("Byte Enable"): Desse 4 signala styrer kva for byte som skal lesast eller skrivast. EMC-modulen genererer desse basert på verdien til adresselinjene A31-30 og på kva type overføring som skal gjerast, dvs. om det er ei ord-, halvord- eller byte-overføring, sjå [3], tabell 2.4-5.

- **Databussen**
  Når minnet har ei databussbreidd på 32 bit, treng ein alle datalinjene som EMC-modulen gir oss, nemleg D31-0.
  Merk at D31-D16 skal koblast til same RAM-krets, og at D15-D0 skal koblast til den andre RAM-kretsen. Det vil vera naturleg, men strengt tatt ikkje nødvendig, å kobla D31 til IO0 på den SRAM-kretsen som inneheld den minst signifikante halvdelen av kvart ord. Likeeins vil det vera naturleg å kobla D0 til IO15 på den andre SRAM-kretsen, sjå figur 4.5.

Meir om signala står i [3], tabell 2.2-5 med tilhøyrande tekst.

Eksempel 4.2
>
Ein tenkjer seg at baseadressa til SRAM-minnet ligg i register *r5* i MicroBlaze, at verdien 0x12345678 ligg i register *r6* og at verdien 0xAB ligg på relativ adresse[25] 9 i SRAM-minnet.

Instruksjonen *sw r6, r5, r0*
vil då generera ein skrivesyklus der ordet i register *r6* blir lagt inn i ord 0 i SRAM-minnet.
Signala vil ha følgjande verdiar under skrivinga:
A12-29 = 00...00, CEN = 0, OEN = 1, WEN = 0, BEN0-3 = 0000 og D0-31 = 0x12345678

Instruksjonen *shi r6, r5, 6*
vil generera ein skrivesyklus der LSH(alvord) i register *r6* blir lagt inn i relativ adresse 6 i SRAM-minnet, dvs. i ord 1.
Signala vil ha følgjande verdiar under skrivinga[26]:
A12-29 = 00...01, CEN = 0, OEN = 1, WEN = 0 og BEN0-1 = 11, BEN2-3 = 00 og D16-31 = 0x5678.

Instruksjonen *lbui r7, r5, 9*
vil generera ein lesesyklus der byten i adresse 9 i SRAM-minnet, dvs. i ord 2, blir plassert på databussen og overført.
Signala vil ha følgjande verdiar under lesinga:
A12-29 = 00...02, CEN = 0, OEN = 0, WEN = 1 og BEN0 = 1, BEN1 = 0, BEN2-3 = 11 og D8-15 = 0xAB.
(r7) = 0x000000AB etter at instruksjonen er utført.

>

[25] Dvs. relativt til baseadresssa.
[26] Detaljar om sjølve tidsdiagramma for skriving og lesing kjem i kapitlet om tidskrav.

Støymarginar og drivekapasitet

Ved overføring av digitale verdiar over korte avstandar, f.eks. mellom ulike kretsar på eit kretskort, blir det brukt eit visst spenningsområde[27] for logisk høgt nivå, '1', og eit anna spenningsområde for logisk lågt nivå, '0'. Ein kallar ofte dette overføring på **rå** eller direkte form.
Ved datakommunikasjon over lengre avstandar må verdiane kodast då støy mm. ellers vil føra til for høg feilrate. Ved koding eller modulasjon kan, avhengig av metoden, f.eks. ein viss frekvens eller ei viss faseforskyving indikera den digitale verdien.

Ein skal her sjå på overføringar på rå form. Ein går 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 innføring i viktige parametrar og så vil det bli gjort ein detaljert analyse av grensesnittet i figur 4.5.

Viktige parametrar

Ved analyse av om ei overføring vil gå greitt, må ein sjølvsagt sjå på overføring av både av logisk høge og logisk låge verdiar. I figurane 4.6-7 er det vist eit enkelt oppsett der ein krets driv eit signal gjennom utgangen sin og over til inngangane på ei rekkje kretsar av same type. I eksemplet er det brukt ein enkel kretstype, nemleg inverteren SN74ALS04B ("Advanced Low-Power Schottkey"). Teknologien er TTL, jfr. databladet [11].
Forsyningsspenninga er her $V_{CC} = 5$ Volt.

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 $V_{OH,min}$.
2. Kva er den lågaste spenninga som blir godtatt som ein '1' på inngangen? Denne parameteren blir kalla $V_{IH,min}$.

Støymarginen ("noise margin") i høg tilstand, NM_H, er gitt av:

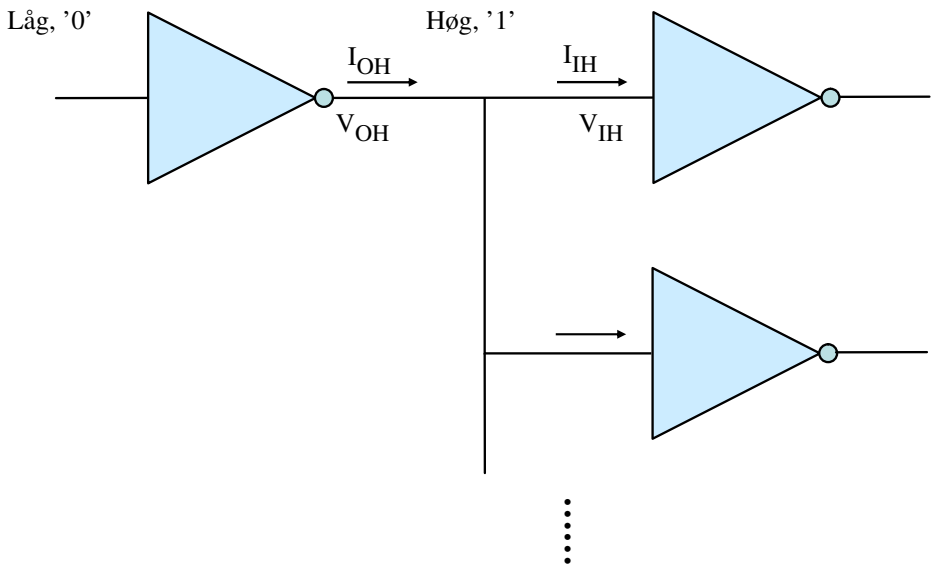
$$NM_H = V_{OH,min} - V_{IH,min} = (V_{CC}{}^{28} - 2) - 2 = 5 - 4 = 1\ Volt$$

På vegen frå sendaren/utgangen til mottakarane/inngangane kan altså spenninga få lov til å falla med opptil 1 Volt [29] utan at mottakarane feiltolkar den logiske verdien. Slik reduksjon kan skje pga. ohmsk spenningsfall eller pga. støy.

[27] Kva spenningsområde ein bruker, er avhengig av kva forsyningsspenninga er og kva teknologi dei integrerte kretsane er basert på. Vanlegast i dag er CMOS-teknologi, men TTL dominerte tidlegare.

[28] Eigentleg er det minimumsverdien av forsyningsspenninga som skal inn her. Dette vil gi den verste situasjonen, dvs. absolutt minst støymargin.

[29] Dette er ein stor støymargin. I elektronikk i dag er vanlegvis forsyningsspenningane lågare, og då er marginane tilsvarande låge. Å redusera V_{CC} er eit godt tiltak for å redusera effektforbruket i elektronikk, men ein får sterkare krav til støyimmunitet.



Figur 4.6: Sentrale parametrar ved analyse av drivekapasitet for logisk høg tilstand.

Verdiane i reknestykkjet over er henta frå databladet, [11]. Som ein ser av dette, er vilkåra for reknestykkjet at straumen gjennom utgangen ikkje overstig maksimumsverdien $I_{OH,max} = -0.4$mA. Dette er **drivekapasiteten** for utgangen i høg tilstand. Konsekvensen av å overskrida drivekapasiteten for ein utgang er at utgangen ikkje greier å halda seg innanfor spesifisert spenningsområde.
Ein negativ $I_{OH,max}$-verdi betyr her at straumen går ut av utgangen, sjå figur 4.6. Dette blir kalla straumleveranse ("source").

Ein inngang trekkjer maksimalt straumen $I_{IH,max} = 20\mu A = 0.02$mA. Ein kan altså teoretisk sett kobla 20 inverterar av same type til ein utgang utan å overskrida drivekapasiteten. Dette blir kalla **viftefaktoren** ("fan-out"). For kretsteknologien TTL, som kretsane her tilhøyrer, er det vanleg å setja viftefaktoren lik 10.

b) Låg tilstand

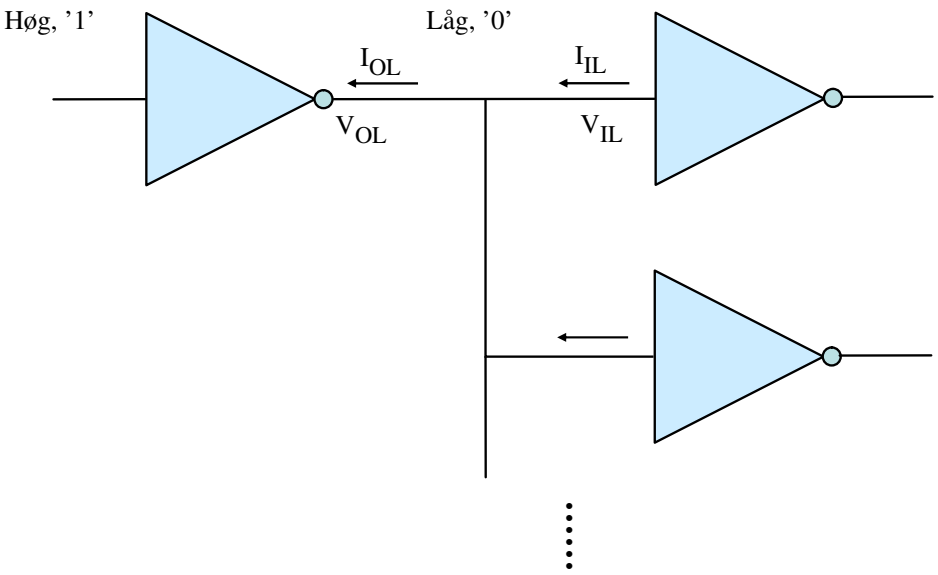
Ved overføring av ein logisk låg verdi melder det seg tilsvarande sentrale spørsmål:
1. Kva er den høgaste spenninga som utgangen i verste fall kan gi for ein '0'? Denne parameteren blir kalla $V_{OL,max}$.
2. Kva er den høgaste spenninga som blir godtatt som ein '0' på inngangen? Denne parameteren heiter $V_{IL,max}$.

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

$$NM_L = V_{IL,max} - V_{OH,max} = 0.8 - 0.4 = 0.4\ Volt$$

På vegen frå sendaren/utgangen til mottakarane/inngangane kan altså spenninga få lov til å auka med opptil 0.4 Volt utan at mottakarane feiltolkar den logiske verdien.

Som ein ser av databladet, er vilkåra for reknestykkjet over at straumen gjennom utgangen ikkje overstig maksimumsverdien $I_{OH,max} = 8$ mA. Ein positiv verdi betyr at straumen går inn i utgangen, sjå figur 4.7. Dette blir kalla straumsvelging ("sink").



Figur 4.7: Sentrale parametrar ved analyse av drivekapasitet for logisk låg tilstand.

Ein inngang vil her levera ein straum som er maksimalt lik $I_{IH,max} = -0.1$ mA. Ein kan altså her teoretisk sett kobla 80 inverterar av same type til ein utgang ved overføring av ein låg verdi.
Det er den **minste** viftefaktorverdien av dei to for høg og låg tilstand ein bruker. I tillegg legg ein inn litt margin og endar då som nemnt over, for TTL sin del på ein viftefaktor lik 10.

Analyse av støymarginar er ikkje kritisk når ein koblar saman kretsar av same familie slik som her, då produsenten har lagt inn tilstrekkjelege marginar ved utvikling av slike familiar. Ein må likevel overhalda krava til viftefaktor.

Ved samankobling av ulike typar kretsar er det derimot viktig å utføra analyse av støymarginar. Det er også viktig å ta med drivekapasiteten i analysen.
Dei utrekna støymarginane, som er uttrykk for den verste situasjonen, kan av og til bli små. Støymarginane kan likevel bli mykje betre i praksis. Viss drivekapasiteten til ein utgang er større enn det samla straumtrekket gjennom dei inngangane som er tilkobla, så vil utgangen vera i stand til å driva spenninga lenger opp i høg tilstand og lenger ned i låg tilstand enn det som analysen viser.

I det følgjande kjem ein detaljert analyse av støymarginane i grensenittet mellom EMC og SRAM.

Analyse av støymargin for grensesnittet mellom EMC og SRAM

Som vist i figur 4.5, vil dei fleste signala bli drivne av EMC-modulen. Meir presist sagt er det utgangar på inn/ut-blokkene (IOB) i FPGA-kretsen som gjer dette, sjå figur 3.3. Desse utgangane er igjen kobla til fysiske pinnar på FPGA-kretsen. Dei to eksterne SRAM-kretsane står for driving av databussen, men dette skjer bare under lesing.
Det må altså utførast ein analyse for kvar driveretning. Ein vil her først finna verdiane til sentrale parametrar og så gjera analyse for dei to retningane.

a) Parameterverdiar
Parameterverdiar for IOB-ane i FPGA-en er gitt av databladet for den typen FPGA som blir brukt her, nemleg Spartan 3, sjå [5]. IOB-ane kan setjast opp med spenningsområde for mange ulike familiar av logiske kretsar. Standardoppsettet er L(ow)V(oltage)CMOS33, som er basert på ei forsyningsspenning $V_{CCO} = 3.3$ Volt slik dei siste to siffera i namnet viser.
Parameterverdiar for SRAM-kretsane kan finnast i databladet, [10].

- **IOB i FPGA sett opp som LVCMOS33:**
  Når IOB-en driv ein utgangspinne:
  $V_{OL,max} = 0.4$ Volt, $V_{OH,min} = V_{CCO,min} - 0.4 = 3.0 - 0.4 = 2.6$ Volt.
  Drivekapasitetane $I_{OL,max}$ og $I_{OH,max}$ kan setjast til ulike nivå frå 2 mA og oppover til 24 mA.

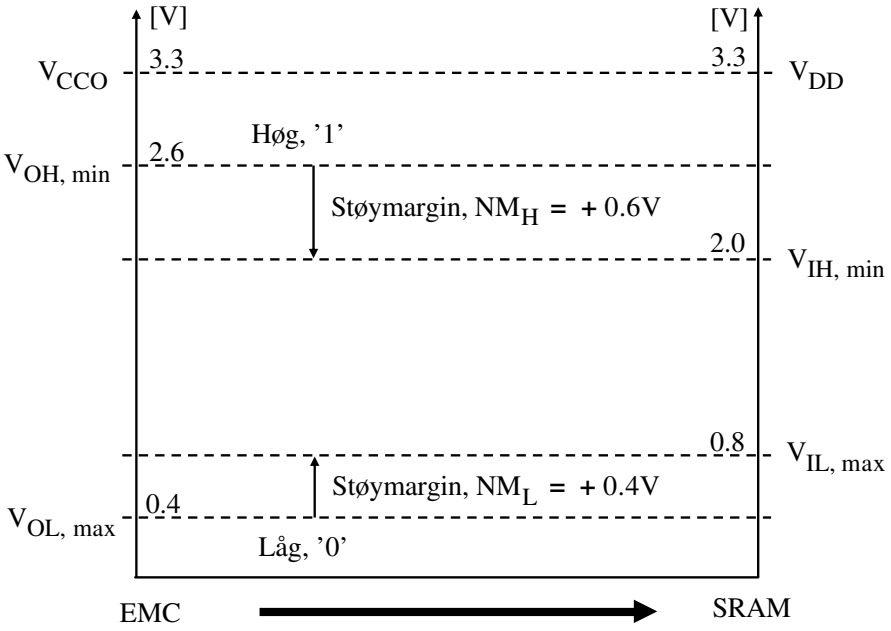
  Når ein IOB-inngang blir driven av ein tilkobla krets:
  $V_{IL,max} = 0.8$ Volt, $V_{IH,min} = 2.0$ Volt.
  Straumtrekket i inngangsdelen av ein IOB er samla i ein parameter, I_L, i databladet. Denne lekkasjestraumen er oppgitt til å vera maksimalt $\pm 25\mu$A.

- **SRAM:**
  $V_{OL,max} = 0.4$ Volt, $V_{OH,min} = 2.4$ Volt.
  Drivekapasitetane $I_{OL,max} = 8.0$ mA og $I_{OH,max} = -4.0$ mA.

  $V_{IL,max} = 0.8$ Volt, $V_{IH,min} = 2.0$ Volt.
  Straumtrekket i inngangsdelen er her også samla i ein parameter, I_{LI}, i databladet. Denne straumen, som blir kalla lekkasjestraum, er oppgitt til å vera maksimalt $\pm 5\mu$A.

b) Støymarginar når IOB i FPGA er drivar
Dette gjeld driving av følgjande signal:
CEN, OEN, WEN, BEN3-0, A29-12 samt D31-0 ved skriving.

Eit samla diagram for høgt og lågt nivå er vist i figur 4.8.

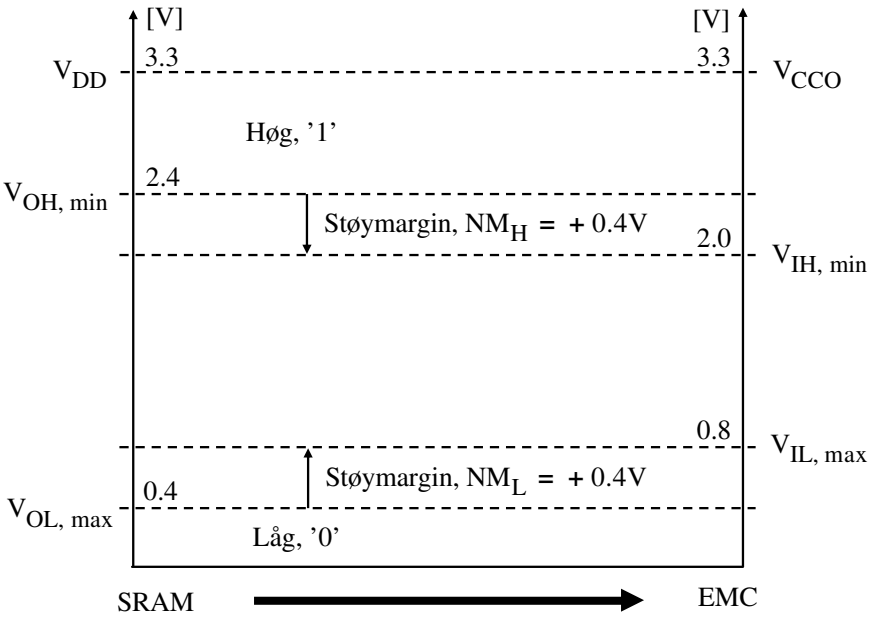


Figur 4.8: Eksempel på analyse av støymarginar under ein skrivesyklus i eit dekoda grensesnitt.

Som ein ser, er alle støymarginane positive og har greie verdiar. Det at drivekapasitetane er mykje større enn straumtrekket på inngangane, gjer som nemnt før, at forholda vil vera mykje betre i praksis.

c) Støymarginar når SRAM er drivar
Dette gjeld følgjande signal:
D31-0 når MicroBlaze les data frå RAM-en.

Eit samla diagram for høgt og lågt nivå er vist i figur 4.9.



Figur 4.9: Eksempel på analyse av støymarginar under ein lesesyklus i eit dekoda grensesnitt.

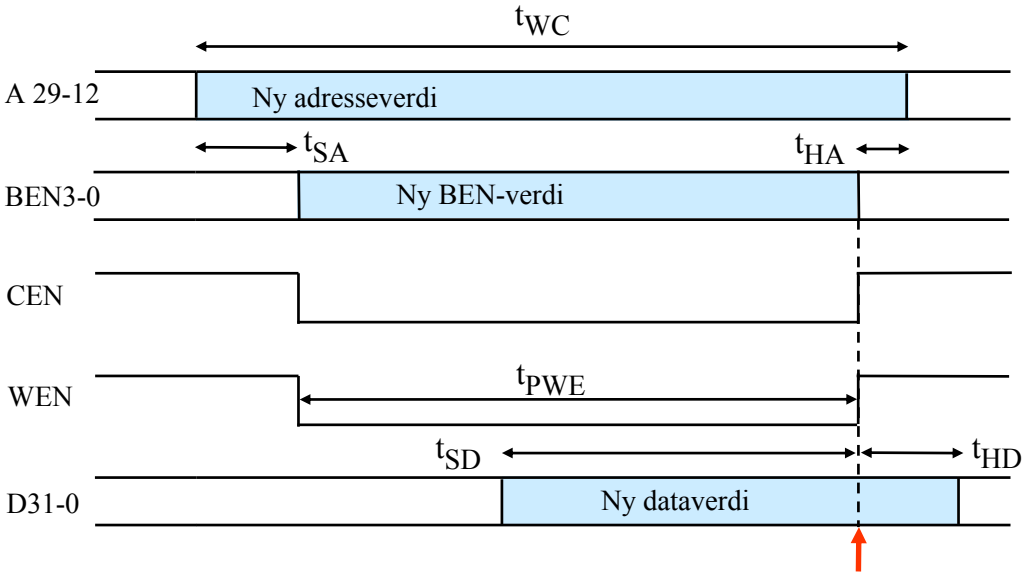
Konklusjonen frå b) vil gjelda her også.

Tidskrav ved dataoverføring

Då gjenstår det eitt punkt, nemleg det å realisera overføringssekvensar for skriving og lesing som tilfredsstiller tidskrava i begge endane. Signala må altså som i eksempelet på programmert grensesnitt i kapittel 4.3, gå i ein spesifisert sekvens og over tilstrekkjelege tidsrom for at data skal bli oppfatta rett.
Ein hovudskilnad mellom programmerte og dekoda grensesnitt er at det i sistnemnte tilfelle er ein eigen modul, nemleg EMC-en, som køyrer overføringssekvensane automatisk. Det som er opp til konstruktøren å gjera her under bygging av mjukkontrolleren, er å spesifisera signala inn og ut av EMC-en og lengder på visse tider i overføringssekvensane.
Signaltilkoblinga er forklart tidlegare. Tidsdiagram og verdiar vil bli delte i to, der ein ser på skriving til SRAM først.

Skriving til SRAM

Ein forenkla, men typisk skrivesyklus er vist i figur 4.10. Denne er basert på diagrammet for "Write Cycle no.1" i databladet til SRAM-kretsane, [10].



Figur 4.10: Typisk skrivesyklus mot ekstern SRAM.

Bokstavane "S" og "H"[30] står for oppsett og halding. Data må feks. stå ei viss tid før og etter **innlåsing** i SRAM-en. Tidspunkt for innlåsing er markert med den loddrette pila. I tillegg må pulsar vara visse minimumstider. For å sikra ei påliteleg dataoverføring, må heile skrivesyklusen også vara ei gitt minimumstid.
I databladet er det tabellar som viser SRAM-en sine krav til dei ulike tidene.

Relevante tidsparametrar som ein kan spesifisera i EMC-modulen, er:
- skrivesykluslengde t_{WC}[31]
- skrivepulsbreidde t_{PWE}[32]

Ved å setja desse til rette verdiar, kan ein tilpassa syklusen slik at krava frå det eksterne minnet blir tilfredsstilt. EMC-modulen vil også innføra visse oppsett- og haldetider som passar dei fleste minnetypar, sjølv om ikkje alle desse tidene kan justerast separat.

SRAM-kretsane i dette grensesnitteksemplet er veldig raske. Krava til sykluslengde og skrivepulsbreidde er følgjande, jfr. [10]:
$t_{WC} \geq 10$ nsek, $t_{PWE} \geq 8$ nsek.

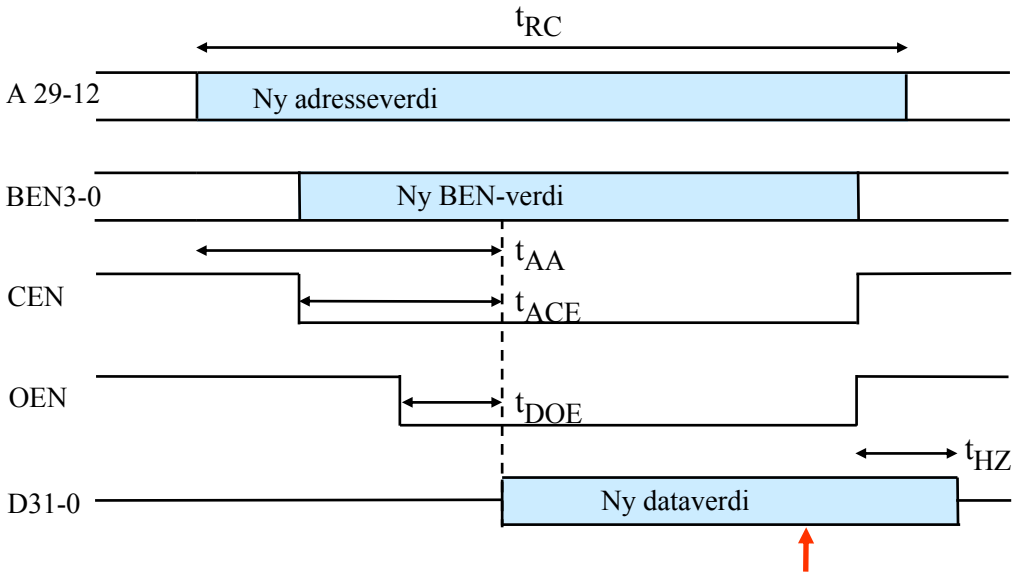
Klokkefrekvensen til mjukkontrolleren er som nemnt før, 50 MHz. Dette gir ein klokkeperiode på 20 nsek. MicroBlaze vil bruka fleire klokkesyklar eller -periodar på å overføra data via IP-modulen MChEMC og ut til eksternt minne[33]. SRAM-en har

[30] Fullstendig namn på alle tidsparametrane er viste i databladet.
[31] Denne tidsparameteren heiter C_TWC_PS_MEM i databladet for EMC-modulen, [7].
[32] Denne tidsparameteren heiter C_TWP_PS_MEM i EMC-databladet.
[33] På laboratoriet er dette målt til over 10 periodar. Dei aller raskaste overføringane går over

altså ingen problem med å henga med på syklusane til MChEMC-modulen.

Lesing frå SRAM

Ein tilsvarande forenkla, men typisk lesesyklus er vist i figur 4.11. Denne er basert på diagrammet for "Read Cycle no.2" i databladet til SRAM-kretsane.



Figur 4.11: Typisk lesesyklus mot ekstern SRAM.

Når MicroBlaze les data frå SRAM-en, så kjem data ut på bussen ei viss tid etter aktivisering[34]av SRAM-kretsen. Denne tida blir kalla **aksesstida** og seier noko om kor raskt minnet er. Etter ei viss tid vil data bli **innlåst** i EMC-en. Dette er illustrert med den loddrette pila. EMC-en vil så deaktivisera kretsen, som så vil stenga databussporten[35].

I databladet for SRAM-en vil ein sjå at det er tre ulike aksesstider. Verdiane for desse er:
$t_{AA} \leq 10$ nsek, $t_{ACE} \leq 10$ nsek og $t_{DOE} \leq 4$ nsek.

Signalet OEN blir vanlegvis aktivisert så tidleg at aksesstida her, t_{DOE}, blir ukritisk. Då CEN alltid blir aktivsert litt etter at ny adresse er sett ut på bussen, vil det vera t_{ACE} som bestemmer når data kjem ut på bussen.
Denne tidsparameteren kan også spesifiserast i EMC-modulen[36].

Ved å spesifisera dette seier ein frå til EMC-modulen om når data kan ventast ut på databussen og dermed om når EMC-en kan låsa inn desse. Slik sikrar ein seg at det er gyldige data som blir innlåste.

LMB-bussen mellom MicroBlaze og internt SRAM-minne og tar 1 - 2 periodar avhengig av optimaliseringsgraden. To periodar er vanlegast her.

[34]I tillegg til adressa må både CEN og OEN gå låge for å aktivisera kretsen.

[35]Databussen vil då flyta, dvs. vera i såkalla høgimpedans-tilstand ("High Z").

[36]Tidsparameteren heiter C_TCEDV_PS_MEM i databladet for EMC-modulen.

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 mjukprosessoren 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 det dekoda grensesnitt som rår grunnen.

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 IP-modular for slike serielle standardar[37], 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.

[37] Eks.: SPI og I2C på kretskort, CAN, USB og Ethernett mellom ulike system.

Referansar

[1] Tim Wilmshurst: "An introduction to the design of small-scale embedded systems". Palgrave, 2001.

[2] Thomas L. Floyd: "Digital fundamentals". Pearson, 9.utg., 2006.

[3] "MicroBlaze processor reference guide, Embedded development kit EDK 10.1" Xilinx, UG081 (v9.3), 2008.

[4] "Spartan-3 Starter Board User Guide".
Xilinx, inc., UG130 (v1.1), 13.mai 2005.

[5] "Spartan-3 FPGA Family: Complete Data Sheet".
Xilinx, DS099, 17.januar 2005.

[6] "XPS Timer/Counter (v.1.00a)".
Xilinx, inc., DS573, 21.april, 2008.

[7] "XPS Multi-Channel External Memory Controller (XPS MCh EMC)(v.2.00a)".
Xilinx, inc., DS575, 21.juli, 2008.

[8] "XPS General Purpose Input/Output (GPIO)(v.1.00a)".
Xilinx, inc., DS569, 22.juli, 2008.

[9] "KS0070B 16COM/80Seg driver & controller for dot matrix LCD". Samsung Electronics.

[10] "IS61LV25616AL 256K x 16 High speed asynchronous CMOS Static RAM with 3.3V supply". Integrated Silicon Solution, Inc., Rev.D 03/11/05.

[11] "SN54ALS04, SN54AS04, SN74ALS04B, SN74AS04 Hex Inverters".
SDAS063B-April 1982-Revised December 1994, Texas Instruments.

Vedlegg A

Bygging og oppstart av eit enkelt MicroBlaze-basert system

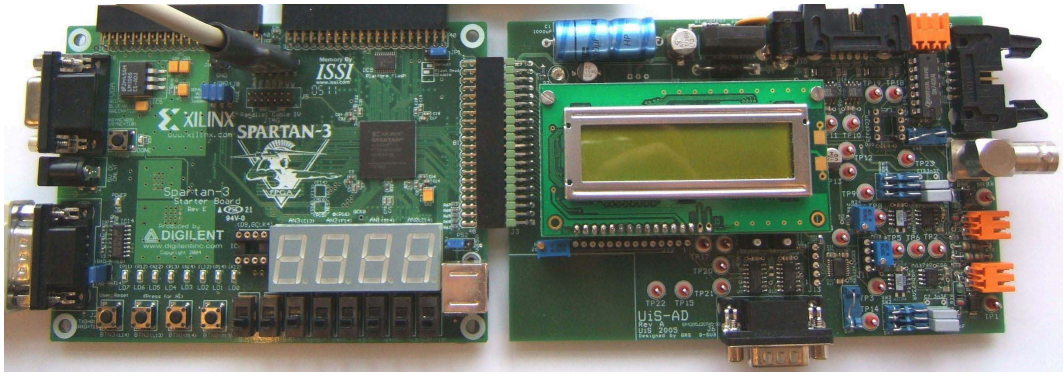
Det blir her vist eit eksempel på bygging og køyring av eit enkelt innebygd system basert på eit kretskort frå Xilinx.
I FPGA-kretsen på dette kretskortet blir det realisert ein MicroBlaze-basert mikrokontroller vhja. pakken **Xilinx Platform Studio**, (XPS). Denne pakken er ein del av utviklingsverktøyet **Embedded Development Kit** (EDK) frå Xilinx.
Programpakken XPS inneheld modulen **Base System Builder** (BSB), som ein bruker til sjølve bygginga av den mjuke mikrokontrolleren.

Først blir FPGA-kortet og -kretsen presentert og så blir systemoppsettet vist. Til slutt kjem sjølve framgangsmåten[1].

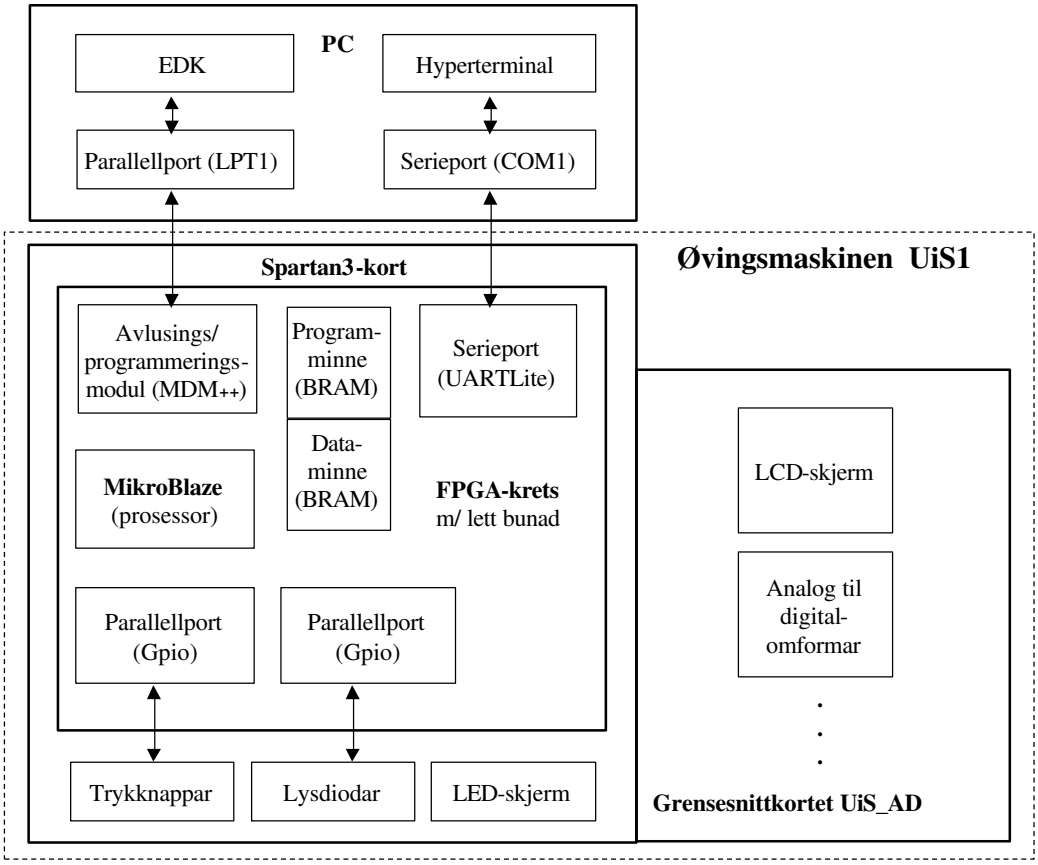
A.1 Målsystem

Målsystemet ("target"), dvs. maskinvareplattformen for det innebygde systemet, er øvingsmaskinen UiS1. Denne er vist i figur A.1.
Hovudkortet i denne er FPGA-kortet **Spartan-3 Starter Board**, [4]. Som figuren viser, inneheld øvingsmaskinen også eit tilleggskort, grensesnittkortet UiS_AD. Dette kortet inneheld LCD-skjerm, A/D-omformar mm., men har inga anna oppgåve i dette eksemplet enn å forsyna hovudkortet med kraft.
Sjølve FPGA-kretsen høyrer til Xilinx sin Spartan-familie og heiter XC3S200. Han har 200.000 logiske portar, dvs. rundt 800.000 transistorar, og er faktisk ein av dei minste Spartan3-kretsane.

[1] Framgangsmåten er basert på bruk av EDK-versjon 10.1.



Figur A.1: Målsystem.



Figur A.2: Liten datamaskin med mikrokontroller i FPGA.

A.2 Systemstruktur

Det innebygde systemet skal byggjast opp som vist i figur A.2.
Mikrokontrolleren skal køyra et program for lesing av brytarar, styring av lysdiodar og overføring av informasjon til PC via serieport.

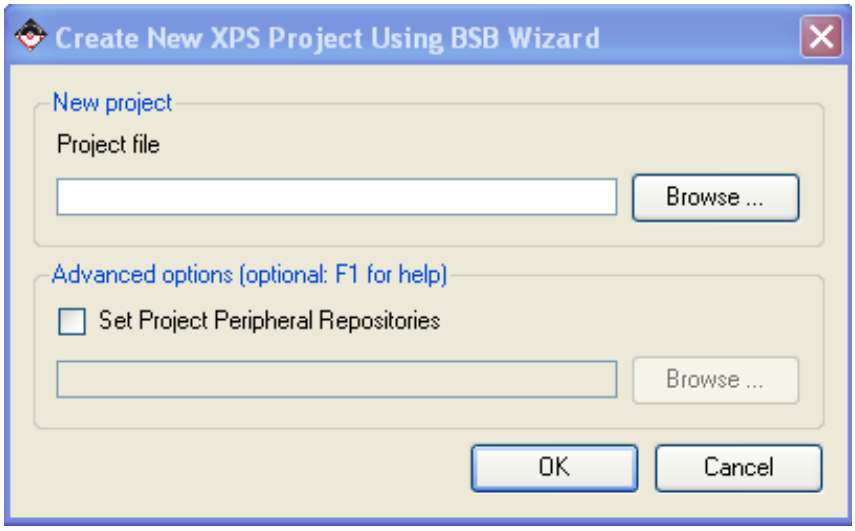
A.3 Framgangsmåte for bygging av mjukkontrolleren

A.3.1 Oppstart av Base System Builder

Start først opp Xilinx Platform Studio og vel BSB viss dette kjem opp automatisk eller gå inn på *File– >New project– >Base System Builder*.
BSB er XPS sitt verktøy for sjølve bygginga. Etter bygging kjem ein så tilbake til XPS der ein kan gjera eventuelle endringar eller utvidingar av maskinvaren, nedlasting mm.

A.3.2 Spesifikasjon av nytt prosjekt

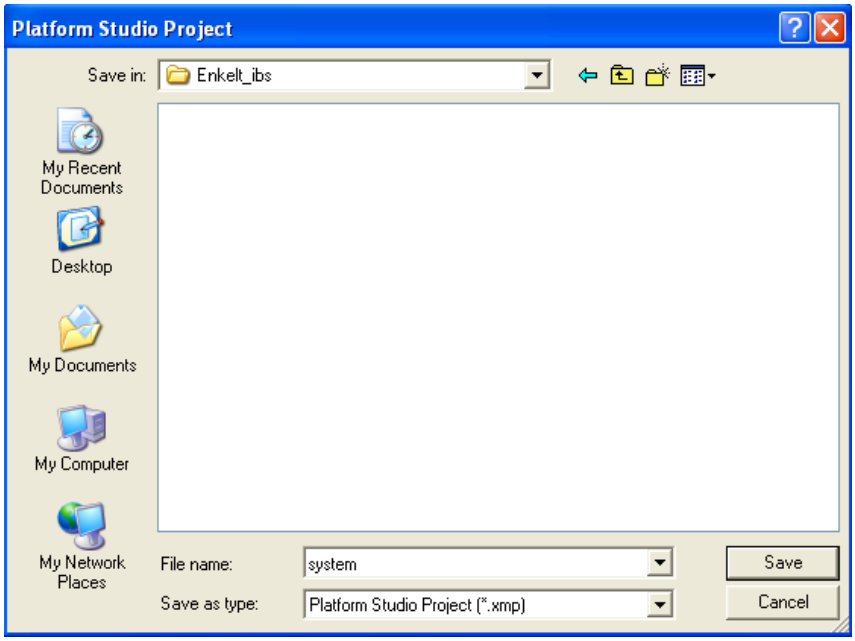
Etter oppstart får ein opp vinduet vist i figur A.3.



Figur A.3: Spesifikasjon av nytt prosjekt.

Trykk på ***Browse*** og lag ny mappe for dette prosjektet som skal heita *Enkelt_ ibs*.
Ein skal så trykkja ***Save*** i fila *system.xmp* som vist i figur A.4. Denne blir då generert av BSB.
Merk: XPS godtar ikkje mellomrom i katalog- eller filnamn.

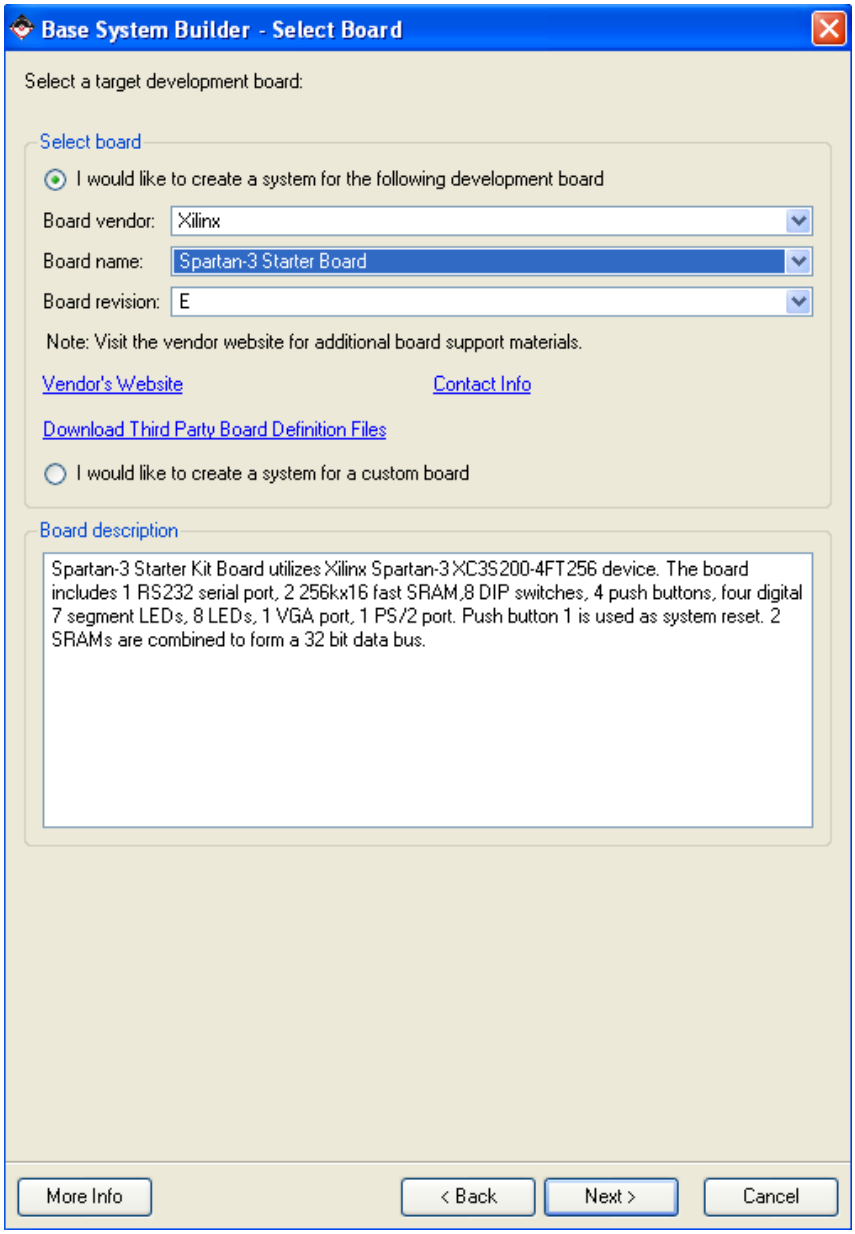
Trykk så ***OK***.



Figur A.4: Prosjektvindu.

A.3.3 Oppsett av målsystem

Ein får nå opp eit vindu der ein vel å laga ein ny konstruksjon.
Gå så til **neste** vindu. Det kjem då opp eit vindu som vist i figur A.5. Ei skal her



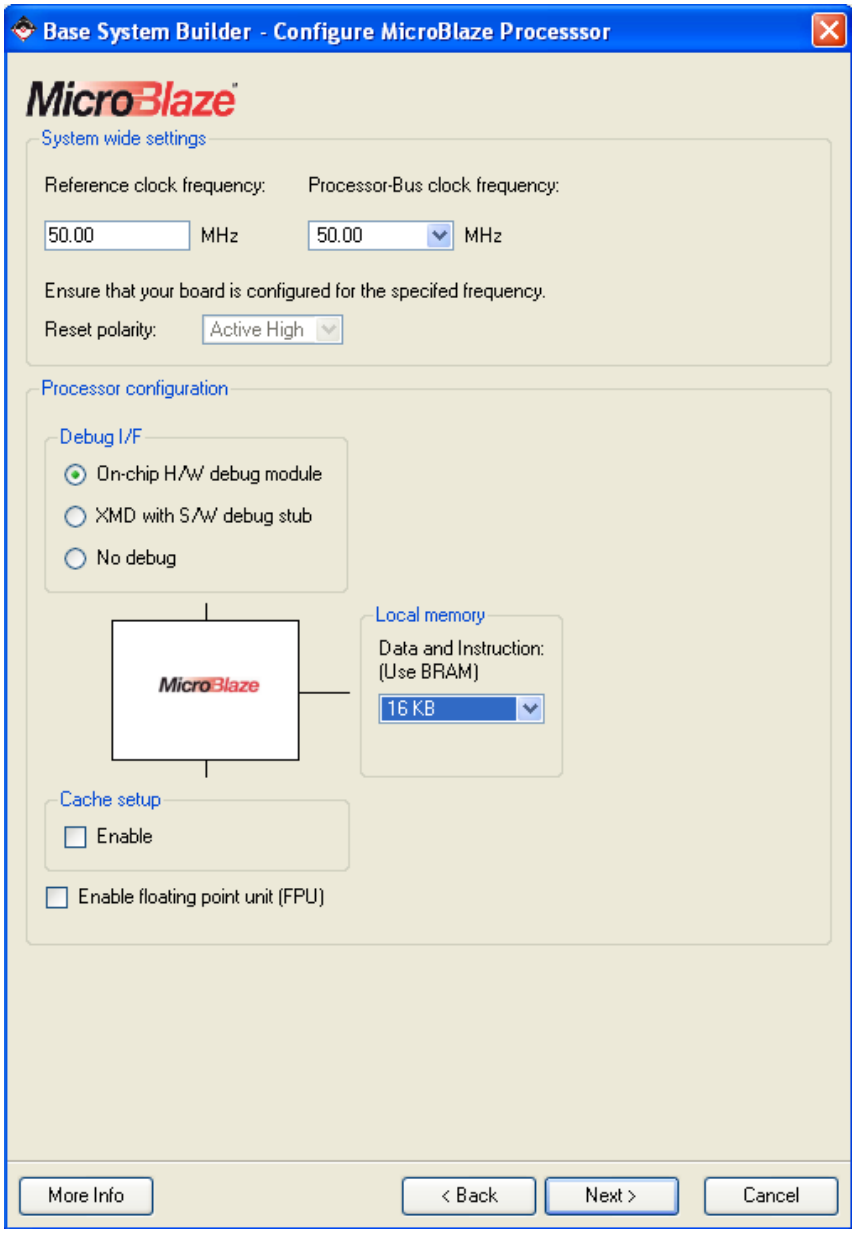
Figur A.5: Spesifikasjon av målsystem.

spesifisera målsystem som vist.
Gå så til **neste** vindu.

A.3.4 Oppsett av prosessor og perifermodular

I neste vindu skal ein så velja MicroBlaze som prosessor.
Gå så til **neste** vindu.

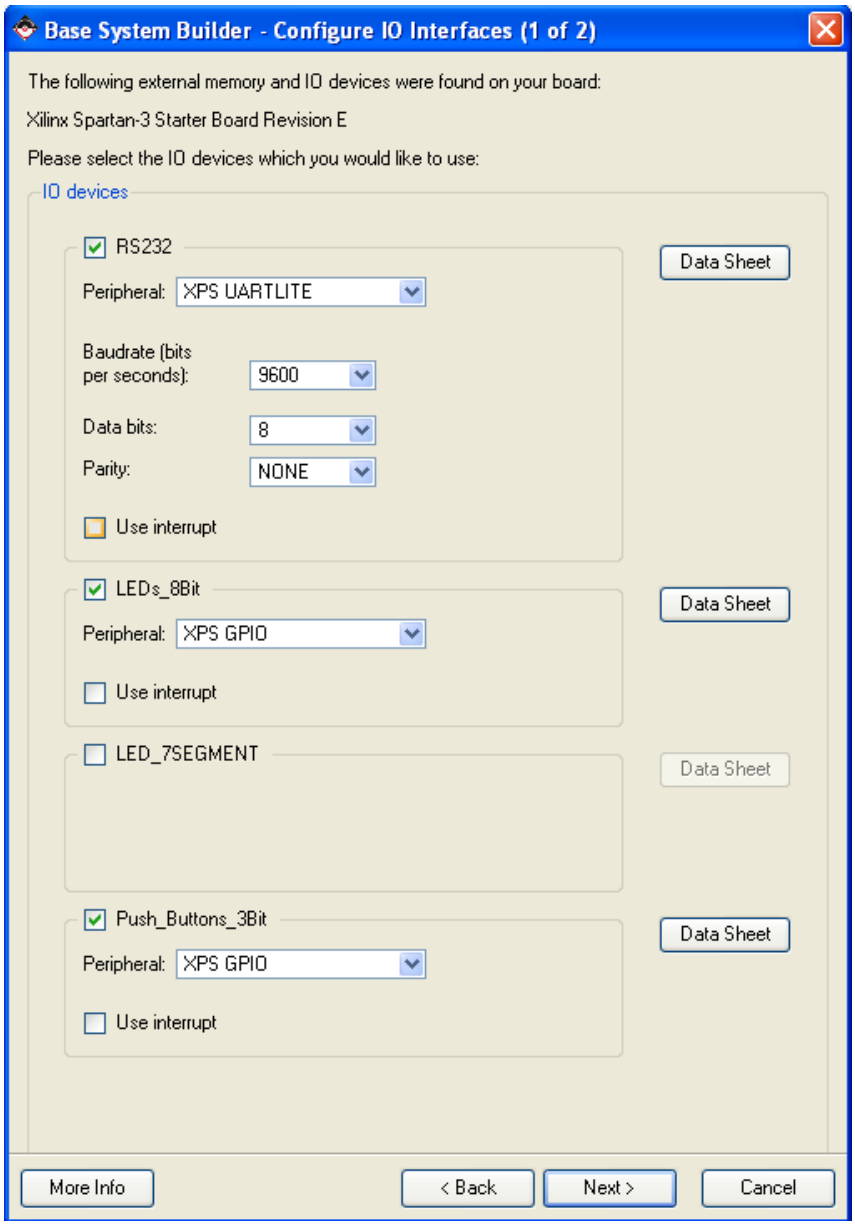
Neste vindu etter dette er vist i figur A.6.
Ein spesifiserer systemklokkefrekvens og minnestorleik som vist. I tillegg skal det



Figur A.6: Spesifikasjon av minne, klokke og støttefunksjonar.

vera fullt utbygd avlusingskapasitet ved å ha ein eigen modul for dette på brikka.
Hurtigminne, "cache", skal ein ikkje ta i bruk her.
Gå så til **neste** vindu.

I dei neste vindua spesifiserer ein perifermodulane til dette systemet som vist i figur A.7 og A.8.

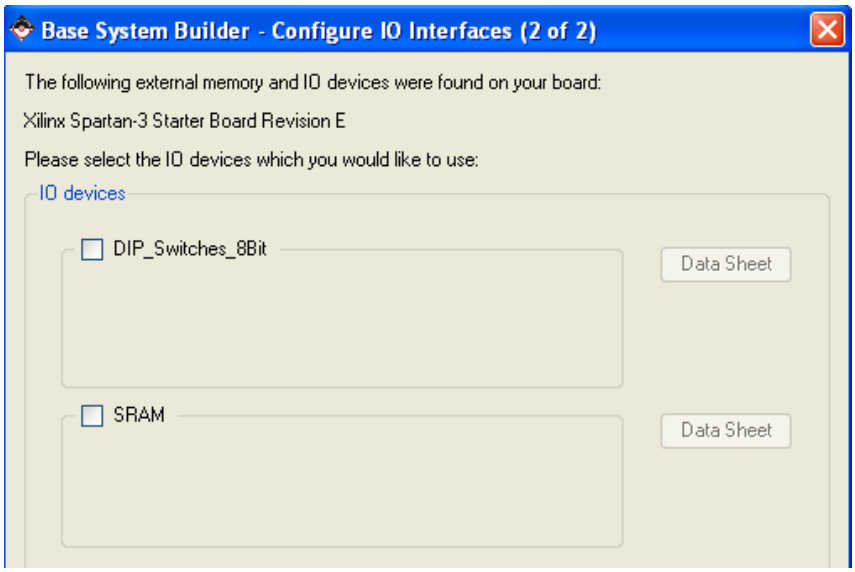


Figur A.7: Spesifikasjon av perifermodular.

Ingen av perifermodulane skal gi avbrot. Alle desse modulane er knytte til PLB-bussen.

Gå så til **neste** vindu.

I vinduet vist i figur A.8 skal ein **velja vekk** både DIP-brytarmodulen og bruk av ekstern SRAM. Ein har ikkje behov for dette minnet og unngår då at det blir brukt plass på ein perifermodul, "Multi CHannel External Memory Controller" (MCHEMC), som dannar grensesnitt mot det eksterne minnet, sjå kapittel 4.4.1.

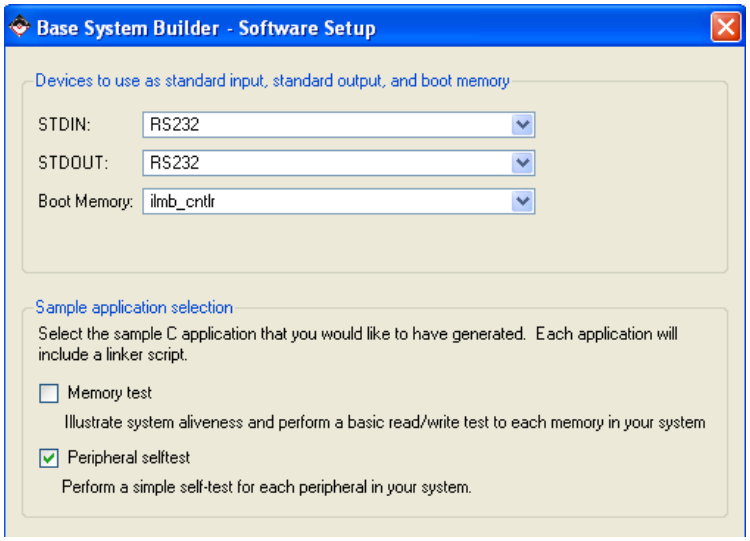


Figur A.8: Spesifikasjon av eventuelt tilleggsminne.

Gå så til **neste** vindu både her og så ein gong til. Ein har ikkje behov for andre perifermodular enn dei som alt nå er lagt inn.

A.3.5 Fullføring av bygginga

Det som gjenstår nå, er mellom anna å spesifera standard inn/ut-port som her vanlegvis er ein RS232-modul, sjå figur A.9.

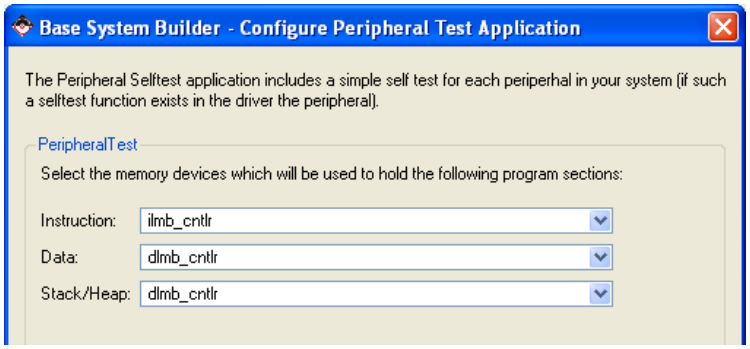


Figur A.9: Spesifikasjon av standard inn/ut-port og testprogram.

Så skal ein velja lokalminnet i FPGA-en, dvs. BRAM, som oppstartsplass for programmet. I tillegg kan ein få generert testprogram for maskinvaren. Ein skal her som vist i figuren bare velja testprogram for perifermodulane.

Gå så til **neste** vindu.

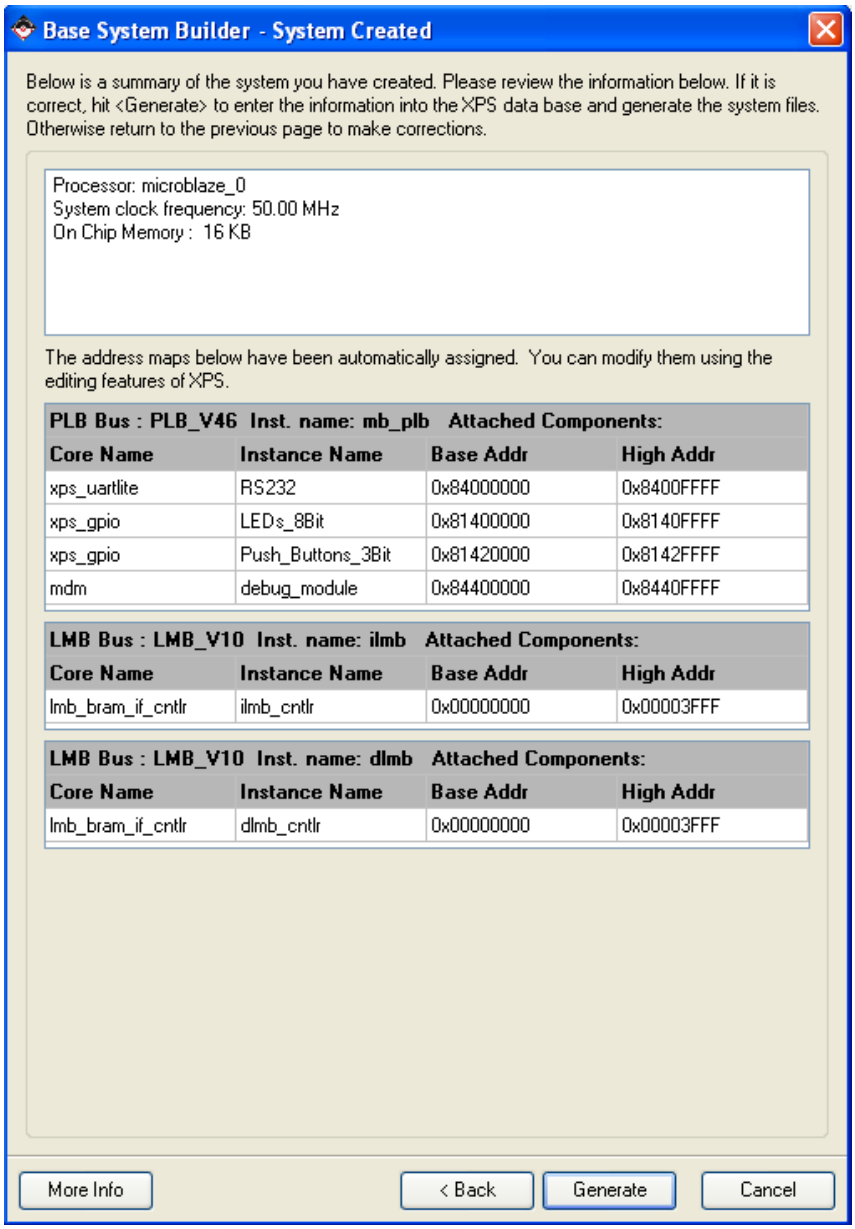
I figur A.10 er seksjoneringa av minnet vist. Ulike kontrollermoduler tar seg altså av overføring mellom mikroprosessoren og instruksjons- og dataseksjonane for eit applikasjonsprogram.



Figur A.10: Spesifikasjon av minnebruk.

Gå så til **neste** vindu.

Ein får nå fram ei oppsummering av systemet med forslag til mellom anna adresseområde for dei ulike modulane, sjå figur A.11.

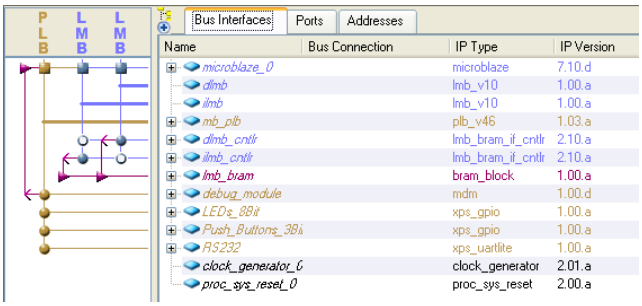


Figur A.11: Oppsummering.

Ein kan då bare trykkja ***Generate*** her og ***Finish*** i neste vindu for å laga systemet.

A.3.6 Ein liten sjekk av maskinvaren til systemet

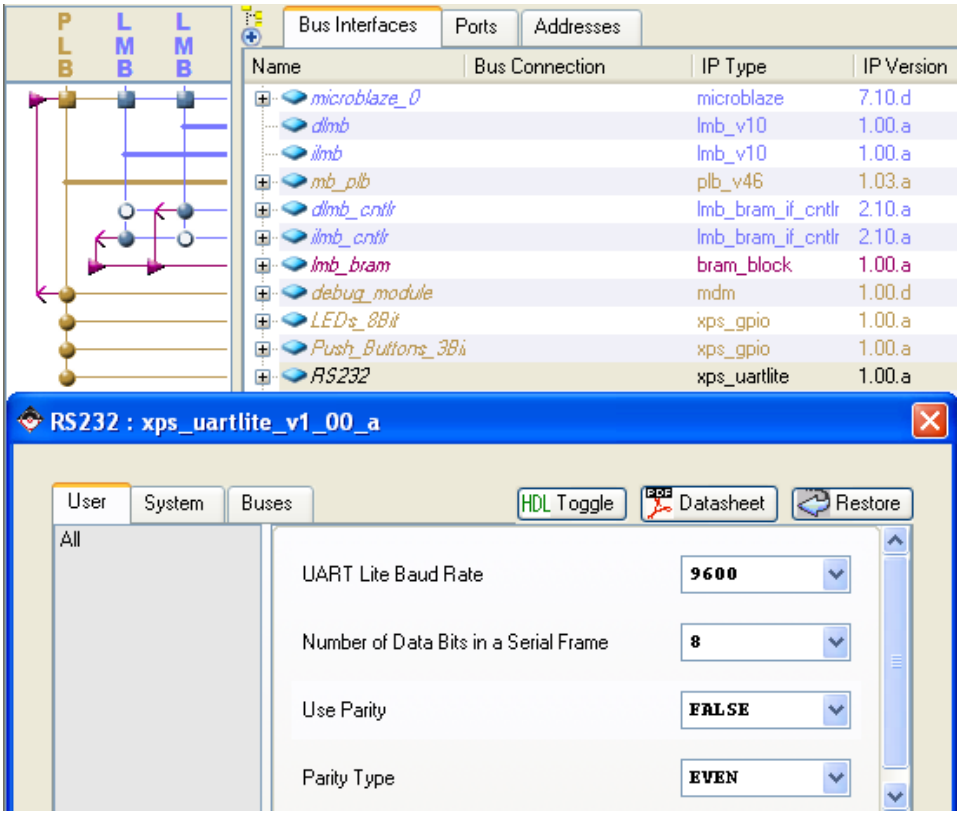
Eit systemoversyn som vist i figur A.12, kjem nå automatisk opp til høgre i Xilinx Platform Studio (XPS). Til venstre er det i tillegg vist kva bussar dei ulike modulane



Figur A.12: Systemoversyn.

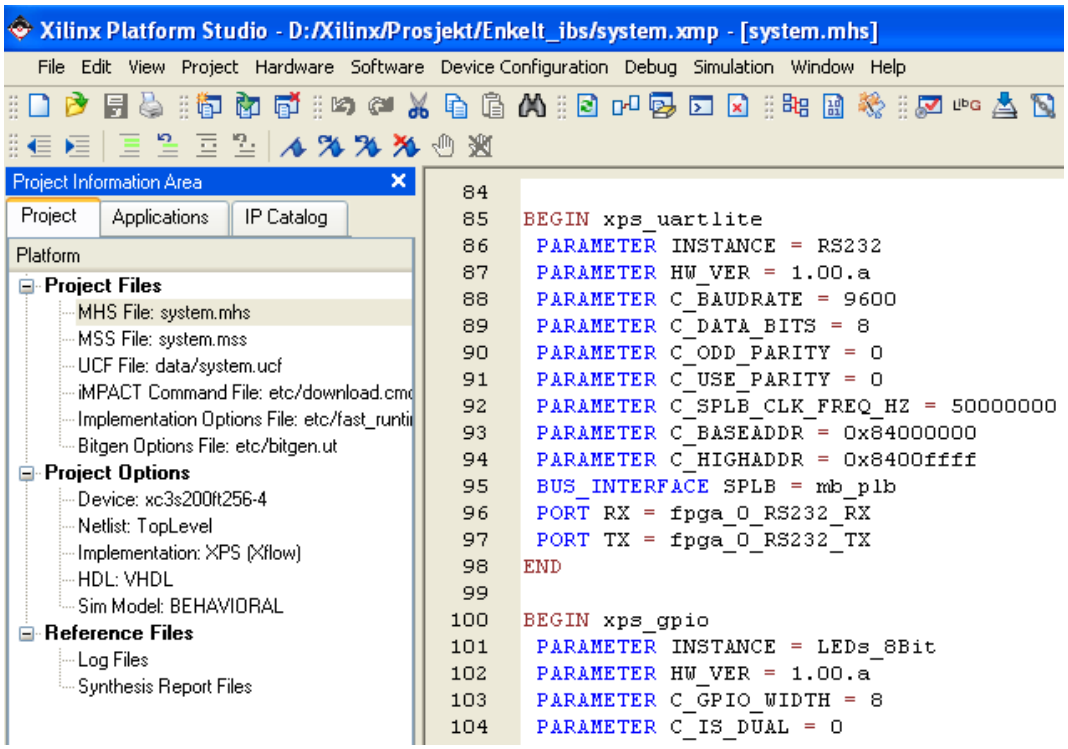
er knytte til.
Ved å dobbeltklikka på ein modul i systemoversynet, får ein fram parametrane for denne.

Figur A.13 viser parametervinduet for perifermodulen *xps_uartlite*, dvs. RS232-modulen i mikrokontrolleren.



Figur A.13: Parametersjekk av perifermodul.

Ein kan også gå til venstre i XPS og inn i mappa *Project* der ein finn fila *system.mhs*, ("microprocessor hardware specification"), sjå figur A.14. Denne



Figur A.14: Maskinvarespesifikasjon.

fila inneheld altså ein spesifikasjon av heile maskinvaren. I det viste utsnittet av fila kjenner ein igjen parametrane for perifermodulen *xps_uartlite* frå figur A.13.
Når ein har forvissa seg om at maskinvaren til systemet er slik ein ville ha det, er ein klar for kompilering av denne, sjå neste kapittel.

A.3.7 Kompilering av maskinvaren til systemet

Ved å trykka *Hardware− >Generate bitstream* vil maskinvareoppskrifta bli kompilert til ei såkalla **bitfil**. Denne skal seinare koblast saman med kompilert programvare og lastast ned i FPGA-kretsen.
Merk: Kompilering av maskinvare går gjennom mange fasar og tar **lang tid**. Mellom anna skal det syntetiserast ein ny mikrokontroller av alle dei logiske blokkene i FPGA-en, noko som er eit omfattande arbeid.

Viss kompileringa var vellukka, dvs. feilfri, er ein klar til å starta programutvikling, kompilering og nedlasting. Ein skal her nøya seg med å køyra testprogrammet for perifermodulane. Dette er nå generert automatisk av BSB-pakken, jfr. figur A.9. Ein framgangsmåte for køyring av dette blir vist i neste kapittel.

A.4 Testkøyring av mjukkontrolleren

Utviklingsverktøyet Embedded Develeopment Kit (EDK) er delt opp i desse to pakkane:

- **Xilinx Platform Studio (XPS)**
  Vhja. denne pakken utfører ein som kjent frå førre kapittel oppsett og endringar av maskinvaren til mikrokontrolleren.
- **Software Development Kit (SDK)**
  Her kan ein gjera sjølve programutviklinga for mikrokontrolleren og kan også i frå denne lasta ned i FPGA-en ein samla maskinvare- og programvarespesifikasjon i ei såkalla bitfil.
  Ein kan også gjera dette i XPS, men det "eclipse"-baserte Xilinx-verktøyet SDK er kraftigare og meir brukarvenleg.

Framgangsmåten vist i det følgjande, er basert på bruk av pakken SDK.

A.4.1 Kompilering av testprogram

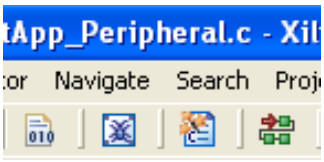
Før ein går over til SDK, gå inn i mappa *Application* og høgreklikk på prosjektet *TestApp_ Peripheral*. Det skal her vera haka av at det er dette prosjektet som skal leggjast inn i BRAM-minnet i FPGA-en (*Mark to initialize BRAM*).

I tillegg skal ein gjera klart for programutvikling ved å byggja opp ein støttepakke for testprogrammet. Dette gjer ein **i XPS** ved å trykka *Software − > Generate Libraries and BSPs*, ("Board Support Package").
Støttepakken er ei samling av nødvendige drivarar og bibliotek mm.

Trykk så *Software − > Launch Platform Studio SDK* for å starta opp programutviklingsverktøyet.
I det vesle vinduet som dukkar opp, trykk *Import XPS Application project*
og kryss her av for programvareprosjektet *TestApp_ Peripheral*.
Dette blir nå importert og oppretta som eit SDK-prosjekt slik at ein **neste gong** SDK blir starta opp, bare kan trykkja *Cancel* i dette vinduet.

Først skal ein her klikka på C-fila *TestApp_ Peripheral* og studera denne.

Kompilering og lenking av C-programmet kan nå utførast ved å trykka
Project − > Build All **eller** ved å trykkja på knappen til venstre i figur A.15. Hugs



Figur A.15: Sentrale knappar i SDK.

å lagra C-fila etter at du har gjort endringar i programmet. Viss
Project − > Build automatically er huka av, blir bygginga gjort automatisk etter lagring.

A.4.2 Oppsett av terminalprogram på PC

Testprogrammet for perifermodulane les dei tre brytarane, styrer lysdiodane og skriv meldingar ut på serieporten. For å sjå desse meldingane, skal ein starta opp Hyperterminal-programmet på PC med same bitrate og oppsett som i
perifermodulen *opb_ uartlite*, sjå figur A.13 eller i fila *system.mhs*. Ein kan då fanga opp det som mikrokontrolleren sender ut.
Terminalprogrammet kan startast opp ved å gå inn på
Start − > All Programs − > Accessories − > Communications − > HyperTerminal.

A.4.3 Nedlasting og køyring av testprogrammet

Før nedlasting **første gong** må ein trykka *Device Configuration − > Bitstream Settings* og til høgre i det vesle vinduet velja rett SDK-prosjekt og nedlastingsfil, nemleg *TestApp_ Peripheral.elf.*

Sjekk også at koblingsbrua merkt med nr.3 i figur A.1 står i fråkobla stilling ("disable").

Nedlasting til FPGA-en skjer ved å trykkja *Device Configuration − > Program FPGA* **eller** ved å trykkja på knappen til høgre i figur A.15.

Testprogrammet vil etter nedlasting starta opp automatisk og gi utskrift på Hyperterminalen samt eit lysmønster på lysdiodane.

Viss ein får feil når ein prøver å lasta ned konfigurasjonen til FPGA-en, kan det vera at kommunikasjonen blir blokkert av Windows sin brannmur. Typisk kan blokkeringar skje viss PC-en er sett opp på nytt.
Dette kan testast ved å velja *Device Configuration − > JTAG Settings*. Viss det er blokkering, vil ein få melding om dette før JTAG-oppsettet blir opna.
Kvitter med å fjerna blokkeringa. Lukk så JTAG-oppsettet og prøv å lasta ned på nytt.

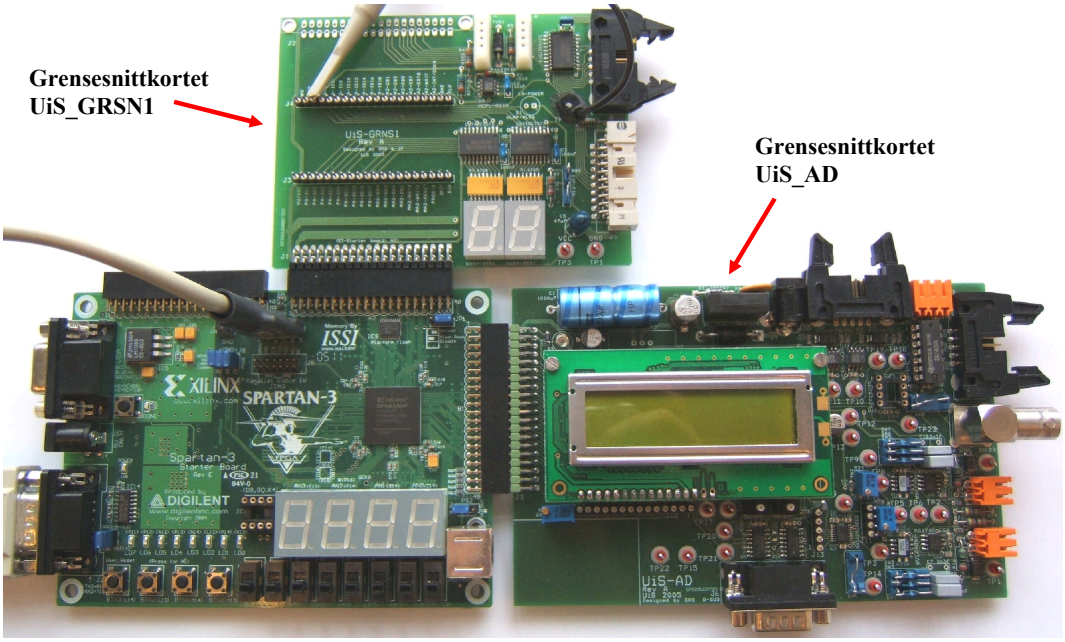
Vedlegg B

Utviding av eit enkelt MicroBlaze-basert system

Det er i vedlegg A vist eit eksempel på oppsetting av eit enkelt innebygd system basert på øvingsmaskinen UiS1. Hovudkortet i denne er FPGA-kortet "Spartan 3 Starter Board" frå Xilinx.
Nå er øvingsmaskinen utvida med eit enkelt grensesnittkort, sjå figur B.1. For å kunne kommunisera med dette kortet, må mikrokontrolleren i FPGA-en utvidast. Dette kan gjerast vhja. pakken **Xilinx Platform Studio** (XPS) som er ein del av utviklingsverktøyet **Embedded Development Kit** (EDK) frå Xilinx. I dette vedlegget blir det vist **framgangsmåte**[1] for utvidinga.

[1] Framgangsmåten er som i vedlegg A basert på bruk av EDK-versjon 10.1.

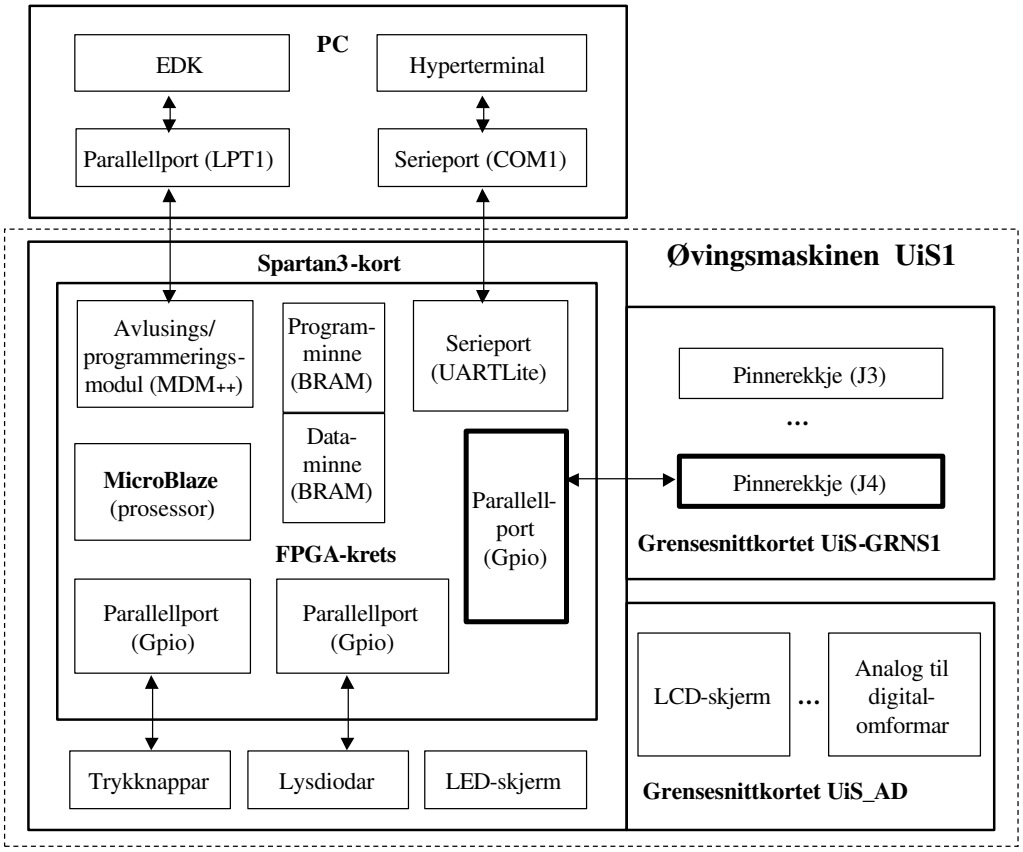
B.1 Systemstruktur



Figur B.1: Systemet i vedlegg A utvida med eit ekstra grensesnittkort.

Det nye grensesnittkortet heiter UiS_GRSN1 og er som vist i figur B.1 kobla til A2-pluggen på Spartan3-kortet. I tillegg er Spartan3-kortet som nemnt i vedlegg A, tilknytt grensesnittkortet UiS_AD, dette via pluggen B1. Grensesnittkortet UiS_AD inneheld mellom anna kraftforsyning for systemet.

Eit blokkskjema av det utvida systemet er vist i figur B.2.



Figur B.2: Blokkskjema av det utvida systemet.

Det nye grensesnittkortet inneheld 2 pinnerekkjer som vist i figuren, og ein ønskjer nå at mikrokontrolleren skal kunne styra pinne 3 og 4 på pinnerekkja J4.
Mikrokontrolleren må då utvidast med ein ny **perifer-** eller **IP-modul**, nemleg ein **2-bits parallellport**.

Slik kortet er konstruert, blir nå pinne 3 på pinnerekkja J4 kobla til FPGA-pinne *d5*.
Pinne 4 på pinnerekkja blir kobla til FPGA-pinne *d6*.

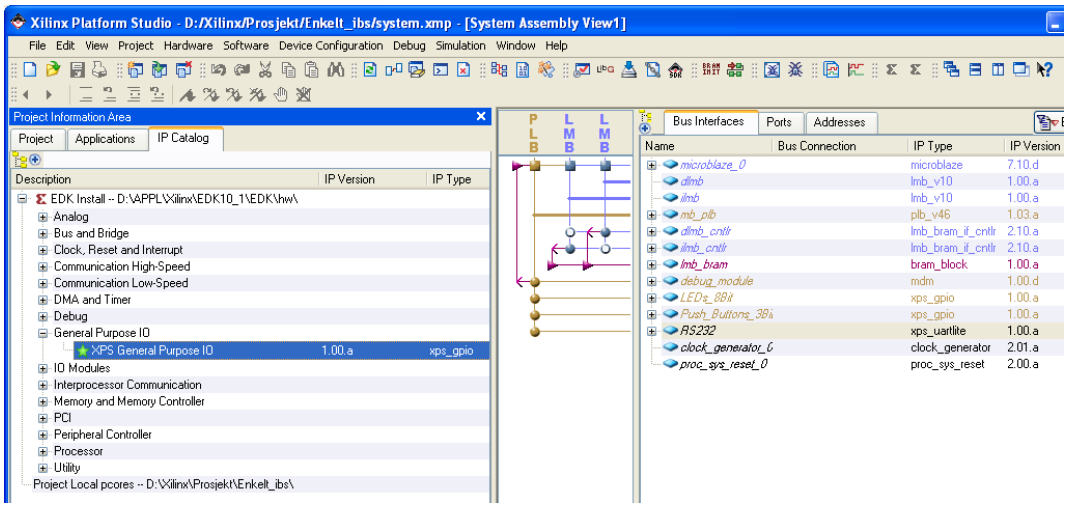
B.2 Framgangsmåte

Det skal altså leggjast til ein ***xps_gpio***-modul, dvs. ein parallellport, med breidde på 2 bit.

Instans-namnet skal vera *J4_pinne3_4*.

B.2.1 Oppretting av perifermodulen

Inne i XPS skal ein gå inn på IP-katalogen til venstre i vinduet, dvs. katalogen over tilgjengelege perifermodular. Her skal ein velja ein parallellport som vist i figur B.3. Ein legg denne inn i systemet ved å høgreklikka på modulen og velja **"Add IP"**. Ein kan også her få fram databladet på modulen.

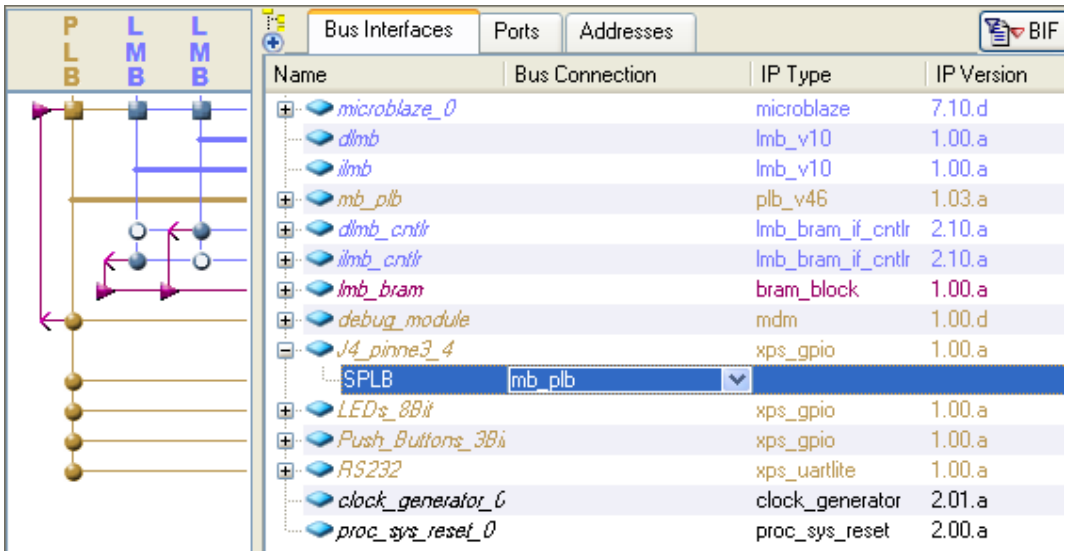


Figur B.3: IP-katalog.

Når den nye systemmodulen dukkar opp i vinduet *System Assembly* til høgre i XPS, kan ein venstreklikka ein gong på namnet til modulen og så leggja inn **instansnamnet**.

B.2.2 Tilkobling av perifermodulen til bussystemet

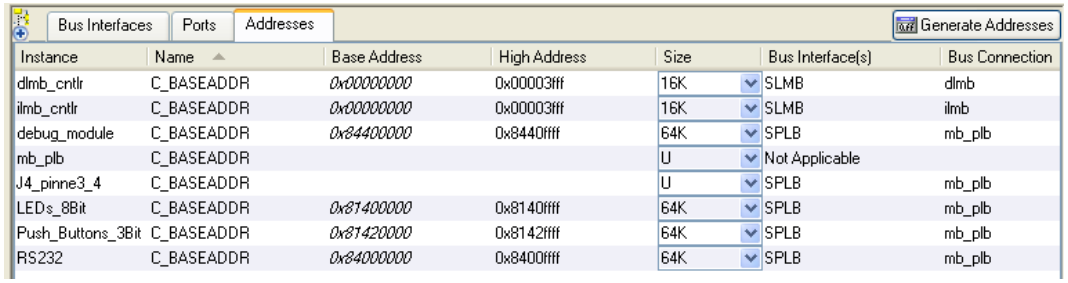
Til venstre i vinduet *System Assembly* ser ein kva bussar dei ulike modulane er knytta til.
Ved å klikka på "+"-teiknet for den nye modulen får ein fram tilkoblingsfeltet som vist i figur B.4. Knytt her den nye perifermodulen til MicroBlaze sin **Processor Local Bus**, *mb_plb*.



Figur B.4: Busstilkobling.

B.2.3 Generering av adresseområde for perifermodulen

Øvst i vinduet *System Assembly* vel ein *Adresses* og får då opp minnekartet for systemet som vist i figur B.5. Ved å trykkja på knappen *Generate Adresses* vil

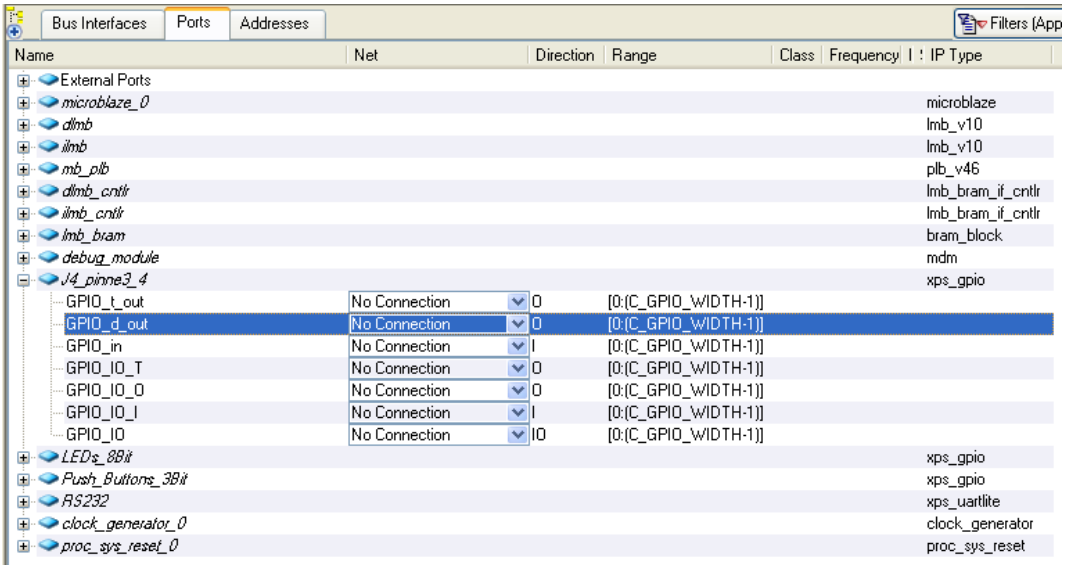


Figur B.5: Adresseområde for perifermodulane i mikrokontrolleren.

parallellporten få tildelt sin del av minneområdet.
(Merk: Både rekkjefølgje og adresseverdiar kan avvika frå det som er vist her.)

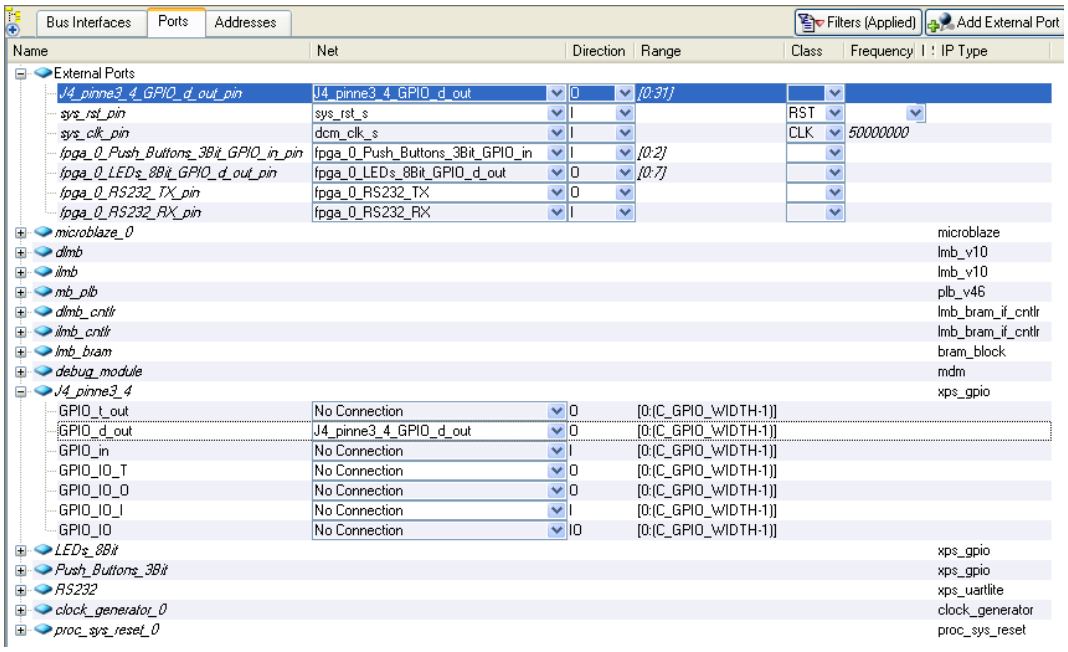
B.2.4 Spesifikasjon av portane til perifermodulen

Trykk nå på fliken *Ports*. Ved å klikka på "+"-teiknet for den nye modulen får ein fram alle moglege porttypar for modulen, sjå figur B.6.
Den nye modulen her skal vera ein rein utport. Dette får ein til ved å velja *Make external* under porttypen *GPIO_d_out*. (Tips: Strekk ut feltet *Net* slik at heile port-teksten viser.)



Figur B.6: Porttypar for den nye IP-modulen.

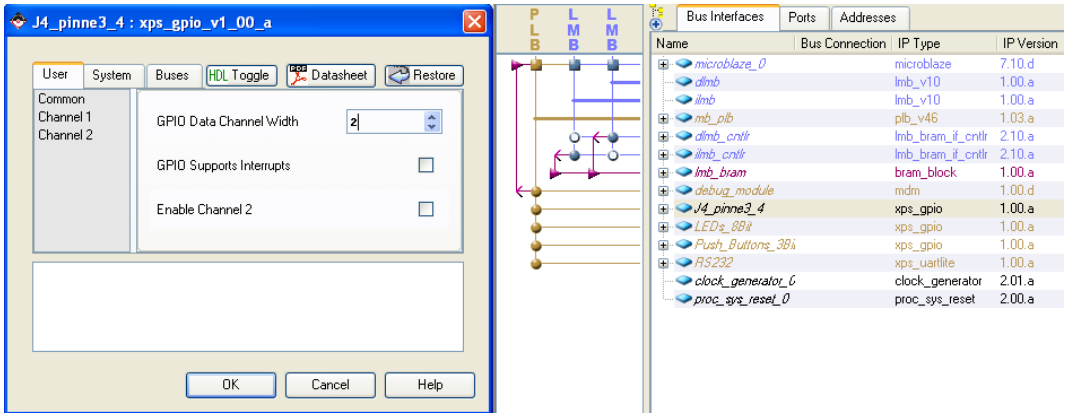
Oppsettet blir nå som vist i figur B.7. Øvst i figuren ser ein alle dei eksterne portane, dvs. dei som skal knyttast til fysiske pinnar på FPGA-kretsen.



Figur B.7: Oppsett av ekstern utport for den nye IP-modulen.

B.2.5 Parametrisering av perifermodulen

Ein skal nå fullføra utvidinga ved å setja viktige **parametrar** for parallellporten. Høgreklikk på modulen *J4_pinne3_4* og vel *Configure IP*.
Sett så opp viktige parametrar for denne som vist i figur B.8.



Figur B.8: Parameter-oppsett for den nye IP-modulen

Trykk så "OK".

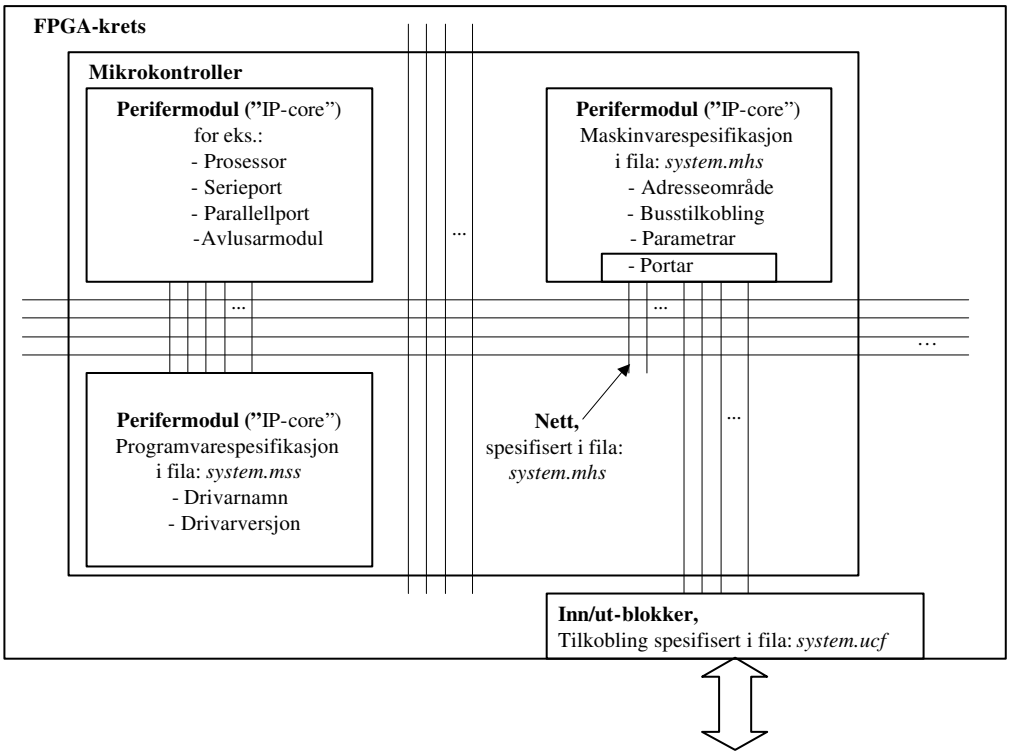
B.2.6 Fysisk tilkobling av perifermodulen

Ein skal til slutt kobla perifermodulen **fysisk** til pinnerekkja på grensesnittkortet.

I mappa *Project Files* i systemvinduet i XPS, skal ein først finna fram fila *system.mhs* og opna denne.
I port-lista øvst i fila skal det mellom anna stå
`"PORT J4_pinne3_4_GPIO_d_out_pin = J4_pinne3_4_GPIO_d_out, DIR = O, VEC = [0:1]"`

Det siste uttrykket er her namnet på sjølve **porten** ut av Gpio-modulen som vist i figur B.7. Det første uttrykket er namnet på sjølve **signalvegen** eller **nettet** ("net") ut mot ein fysisk pinne. Dette er illustrert i figur B.9.

Til slutt er breidda og retninga spesifisert, dvs. 2 bit og "ut" her.
Merk: Viss breidda er feil, så rett opp denne.
Lagre så fila.



Figur B.9: System-struktur og -filer

Det siste ein skal gjera, er å endra fila *system.ucf*, ("user constraints file"). I denne skal ein knytta nett til fysiske pinnar på FPGA-kretsen, sjå figur B.9.
Fila *system.ucf* finn ein også under *Project Files* i systemvinduet i XPS.
Legg her til følgjande:

```
"Net J4_pinne3_4_GPIO_d_out_pin<0> LOC=d5; ## J4,pinne 3,PA-IO2"
"Net J4_pinne3_4_GPIO_d_out_pin<1> LOC=d6; ## J4,pinne 4,PA-IO4"
```

Lagre så fila.

Utvidinga skal nå vera fullført!
For å sjekka om ein har gjort nokre feil på vegen, kan ein gå inn på
Hardware− >Generate Bitstream
i menyen på XPS.
Ein vil då kompilera sjølve maskinvaren, dvs. køyra gjennom høgre del av flytdiagrammet i figur 3.12.

Vedlegg C

Om å laga program for eit MicroBlaze-basert system

C.1 Innleiing

Programmeringa på laboratoriet vil bli utført vhja. **Embedded Development Kit (EDK)**, som er Xilinx sitt utviklingsverktøy for FPGA-baserte, dvs. mjuke mikrokontrollerar.
Utviklingsverktøyet er delt opp i desse to pakkane:

- **Xilinx Platform Studio (XPS)**
  Vhja. denne pakken utfører ein oppsett og endringar av maskinvaren til mikrokontrolleren. Programpakken XPS inneheld modulen **Base System Builder** (BSB), som ein bruker til sjølve bygginga.
- **Software Development Kit (SDK)**
  I dette "eclipse"-baserte verktøyet kan ein gjera sjølve programutviklinga for mikrokontrolleren. Verktøyet har både C-kompilator, assembler, lenkar og avlusar. I tillegg kan ein i SDK lasta ned i FPGA-en til målsystemet ("target system") ein samla maskinvare- og programvarespesifikasjon i ei såkalla bitfil. Etter nedlasting av desse konfigurasjonsdata frå PC-en til målsystemet startar programmet opp automatisk i FPGA-en.

Ein vil i dette notatet sjå litt nærare på sjølve programmeringa, som hovudsakleg vil bli i C. Språket C kan seiast å vera eit "subsett" av C++.

Arbeidsgangen frå utvikling av kjeldekode i C til køyrbart program i FPGA-kretsen kan delast opp i følgjande steg:

1. Utvikling av kjeldekode, dvs. kode i form av *.c*[1]- og *.h*[2]-filer.
2. Kompilering av kode, dvs. at *.c*- og *.h*-filer blir omgjort til objektkode i form av *.o*[3]-filer.
3. Lenking av objektkode. Filer av type *.o* blir lenka saman til absoluttkode, her i form av ei køyrbar *.elf*[4]-fil.
4. Nedlasting til krets og avlusing ("debugging").

Ein tek nå dette steg for steg i det følgjande.

C.2 Utvikling av kjeldekode

C.2.1 Litt om oppsett og programmering

Program bør vera modulære og oversiktelege. For å oppnå dette, kan det vera gunstig å dela opp programmet i ein **portabel** del og ein **maskinvarespesifikk** del.
I den portable delen vil ein finna hovudprogrammet *main()* samt dei funksjonane og definisjonane som ikkje er avhengig av maskintypen. Den maskinvarespesifikke programvaren knyter den portable delen saman med sjølve maskinvaren.
Denne oppdelinga er viktig då den gir ryddig programvare samt minimalt modifikasjonsarbeid ved overgang til ny maskinvare. Her må ein då bare endra den maskinvarespesifikke programvaren, men merk at heile programvaren likevel som oftast må kompilerast på nytt for den nye maskinvaren.
Ein skal nå gi eit eksempel på oppdeling av eit enkelt program for ein MicroBlaze-basert mikrokontroller med enkelt grensesnitt mot omverda.

[1] Ei *.c*-fil inneheld programkode og har halen *.c*, feks. *filnamn.c*.
[2] Ei *.h*-fil ("header"-fil) inneheld definisjonar og deklarasjonar og har halen *.h*, feks. *filnamn.h*.
[3] Ei objektfil har halen *.o*, feks. *filnamn.o*.
[4] Ei elf-fil ("executable and linking format"-fil) har halen *.elf*, feks. *filnamn.elf*.

Eksempel:
>
Mikrokontrolleren som blir bygd i vedlegg A i notat IIA har mellom anna ein 8bits parallellport (gpio-modul) for styring av lysdiodar og ein 3bits parallellport for lesing av brytarar.
Den maskinvarespesifikke eller drivardelen av eit C-program som bl.a. skriv til lysdiodane og les frå brytarane, vil trenga maskinvarespesifikke definisjonar og funksjonar som vist i det fylgjande.

Ved bygging av mikrokontrolleren i BSB/XPS blir det automatisk generert eit rammeverk med programmeringsfiler. Desse filene vil vera til hjelp når ein skal utvikla kjeldekode for mikrokontrolleren.
Ei nyttig definisjonsfil her heiter *xparameters.h*.
(Denne kan finnast i SDK ved å sjå i vinduet *C/C++ Projects* og der i mappa *microblaze_0_sw_platform + microblaze_0 + include*.)
I denne står det mellom anna, men merk at adresseverdiane kan variera:

" `#define XPAR_LEDS_8BIT_BASEADDR 0x81400000`" og
" `#define XPAR_PUSH_BUTTONS_3BIT_BASEADDR 0x81420000`"

Ved å studera databladet på ein gpio-modul, finn ein ut at dataregisteret ligg på same adresse som baseadressa over. (Databladet eller "PDF-doc" kan lett finnast ved å høgreklikka på perifermodulen i vinduet *IP Catalog* i XPS og så klikka på "View PDF-doc".)
Sekvensen

#define lysdiodar ((unsigned int *) XPAR_LEDS_8BIT_BASEADDR)*

lysdiodar = 0xFF;

ville såleis få alle lysdiodane til lysa viss gpio-modulen var sett opp som ut-port først.

Det er vanleg at ein i slike definisjonar også tek med tilleggsspesifikasjonen **"volatile"**, sjå eksemplet under:

#define brytarar ((volatile unsigned int *) XPAR_PUSH_BUTTONS_3BIT_BASEADDR)*

Denne tilleggsspesifikasjonen viser kompilatoren at verdien til variabelen "brytarar" kan bli endra av andre enn programkoden, nemleg av sjølve maskinvaren ved trykk på ein av dei fysiske brytarane. Dette forhindrar at kompilatoren optimaliserer vekk kode der ein f.eks. står og venter på ein viss brytarverdi utan at sjølve programkoden er inne og endrar denne variablen.

Ein har altså i definisjonseksempla over knytta samband mot fysiske adresser vhja. **peikarar** og **CAST-operasjon**.
Merk at slike definisjonar er innbakt i dei **ferdige drivarrutinene** ein nyttar ved programmering av MicroBlaze. Ein treng difor ikkje gå ned på det nivået som er vist her, men kan i programmeringa vår bruka meir fleksible og lesbare funksjonskall som finst i dei ferdiglaga drivarrutinene for kvar perifermodul.

Viss ikkje programmet er lite, bør dei maskinvarespesifikke **funksjonane** plasserast i eiga fil, f.eks. *UiS1_funk.h*.
Likeeins bør også dei globale **definisjonane** og **variabeldeklarasjonane** leggjast inn i eigne filer, her eksemplifisert med fila *UiS1_dekl.h*. Denne vil her bare sjå slik ut:

```
//————————————————————-
// Deklarasjonsfila UiS1_dekl.h
//————————————————————-


// Globale variablar

unsigned char diodeverdi = 0;
unsigned char brytarkode;

//————————————————————-
```

Merk: Alle funksjonsfiler må ha ein referanse til denne deklarasjonsfila. Det ville då vore naturleg å inkludera denne i alle funksjonsfilene, men dette gir feilmelding ved kompilering. Grunnen er at variablar bare kan deklarerast ein plass i programmet.
Løysinga på problemet er å laga ei såkalla ekstern deklarasjonsfil som kan inkluderast av funksjonsfilene.

Den **eksterne deklarasjonsfila** kan heita *UiS1_ekstern_dekl.h* og vil her sjå slik ut:

```
//————————————————————-
// Deklarasjonsfila UiS1_ekstern_dekl.h
//————————————————————-


// Ekstern deklarasjon av globale variablar

extern unsigned char diodeverdi;
extern unsigned char brytarkode;

//————————————————————-
```

I hovudprogrammet, her kalla *TestApp.c*, inkluderer ein då sjølve deklarasjonane, dvs. fila *UiS1_dekl.h*, og sjølve deklarasjonane skjer då bare ein plass.

Funksjonsfila, *UiS1_funk.h*, kan sjå slik ut:

```
//————————————————————-
// Funksjonsfila UiS1_funk.h
//————————————————————-


#include "xparameters.h"    // " " - Henting frå arbeids-katalog
#include "xgpio_l.h"     // Ferdiglaga drivarrutiner

#include "UiS1_ekstern_dekl.h"


// Fyrst kjem "funksjons-prototypane"

void maskinvare_init(void)
void skriv_til_lysdiodar(unsigned char diodeverdi)
unsigned char les_brytarar(void)

// Så kjem sjølve funksjonsdeklarasjonane

void maskinvare_init(void)
        {
        // Oppsett av retning på gpio-modular m.m.

        .......;
        }

void skriv_til_lysdiodar(unsigned char utverdi)
        {
        XGpio_mSetDataReg(XPAR_LEDS_8BIT_BASEADDR, 1,utverdi );
        }

unsigned char les_brytarar(void)
        {
        return XGpio_mGetDataReg(XPAR_PUSH_BUTTONS_3BIT_BASEADDR,
1);
        }

//————————————————————-
```

I **hovudprogrammet** *TestApp.c* vil ein sjå kall av dei maskinspesifikke funksjonane. All kode og definisjonar som ligg i hovudprogrammet blir då **portabel**. Denne fila kan sjå slik ut:

```
//————————————————————-
// Hovudprogrammet TestApp.c
//————————————————————-


#include "UiS1_dekl.h"
#include "UiS1_funk.h"

int main(void)
        {
        // Først eventuelle definisjonar og lokale deklarasjonar

        .
        maskinvare_init(); // initialisering av maskinvaren
        .
        while(1)  // endelaus løkkje
                {
                .
                skriv_til_lysdiodar( diodeverdi );
                .
                brytarkode = les_brytarar();
                .
                // Og så vidare
                .
                }
        return 0;      // Standard slutt
        }

>
```

C.2.2 Litt om korleis ein kan leggja inn assembly-stubbar i C-koden

Ein kjem hovudsakleg til å programmera i C, men kan i nokre tilfelle ha behov for assemblykode. Dette kan leggjast inn som såkalla "in-line assembly" i C-programmet og blir då kompilert saman med dette. Oppsettet er som vist i fylgjande eksempel:

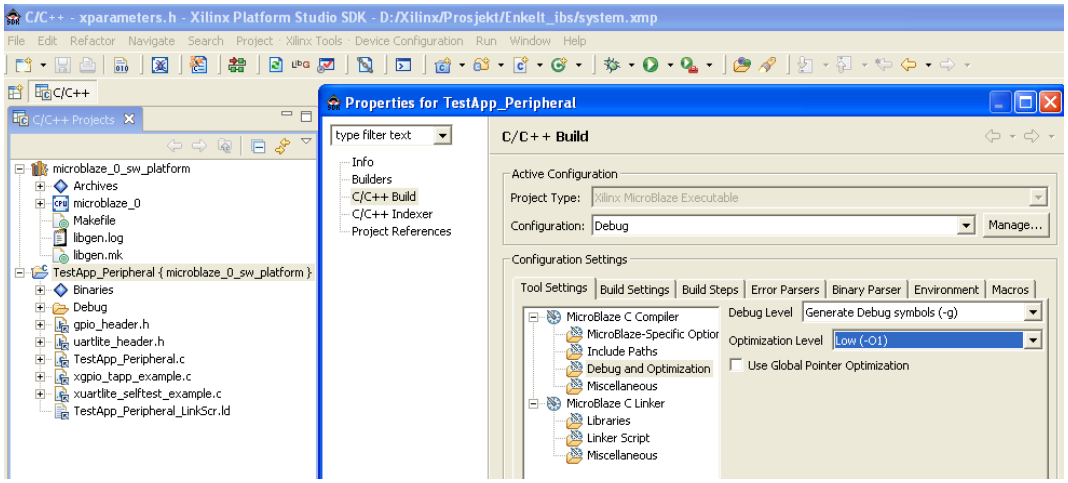
```
asm("         addi r7, r0, 0x100"); // Ein instruksjon pr linje
asm("loekke:  addi r7, r7, -10    "); // ":" etter knagg ("label").
asm("         bgti r7,loekke      ");
```

Når det gjeld registerbruk og instruksjonssett, sjå tabell 3-2 og kap.4 i referansemanualen for MicroBlaze, [3].(Denne kan også lett finnast ved å høgreklikka på Microblaze-modulen i vinduet *IP Catalog* i XPS og så klikka på *View PDF-doc.*)

C.3 Frå kjeldekode til køyrbart program

C.3.1 Litt om verktøyinnstillingar

For å få ut nok informasjon til å kunne avlusa eit program, må kompilatoren vera sett opp på rett måte. Standard oppsett er som vist i figur C.1 men med ei lita endring, nemleg at **optimaliseringa** akkurat her er sett til nivå 1, (*-O1*). Standardinnstillinga



Figur C.1: System-struktur og -filer.

er nivå 2.
Som ein del av standardinnstillinga er det som vist, spesifisert at det skal lagast tilstrekkjeleg informasjon for avlusaren, (*Generate Debug symbols (-g)*).

Ved å høgreklikka på mappa *TestApp_ Peripheral* i vinduet *C/C++ Projects* i SDK og så velja *Properties*, får ein fram oppsettet i figuren.

C.3.2 Kompilering av kjeldekoden

Eit trykk på bygg-knappen (*Build all*)[5] til venstre i figur C.2, vil føra til fylgjande:

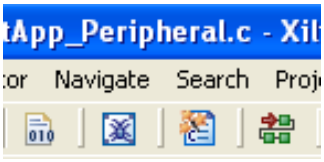
1. Dei inkluderte *.h*-filene blir lagde inn i kjeldefilene (*.c*), og desse blir kompilerte, dvs. gjort om til kvar sine objektfiler. Det blir også mellom anna laga *.s*-filer, som viser assemblyutgåva av *.c* + *.h*-filene.
   Merk at objektfiler er **relokerbare**, dvs. at det i objektkoden ikkje er nytta absolutte hopp- eller variabeladresser. Det er lenkaren, sjå kapittel C.3.4, som basert på spesifikasjonen vår, 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[6]. I tillegg får kode, variablar og stakk sine absolutte plasseringar i minneområdet.
   Lenkaren utfører dette ut frå spesifikasjonen gitt i **kommando-fila** *"TestAppLinkScr"* og produserer så den køyrbare **absoluttfila** *"executable.elf"*.

C.3.3 Generering av listefil

Ved å trykka på knappen rett til høgre for bygg-knappen i figur C.2, vil ein få opp eit kommandovindu for **Xilinx Microprocessor Debugger**, (XMD). Viss ein i dette legg inn kommandoen

mb-objdump -S
*D:/**Prosjektsti**/**Prosjektnamn**/SDK_projects/TestApp_Peripheral/Debug/*
TestApp_Peripheral.elf > asm.txt



Figur C.2: Sentrale knappar i SDK.

så vil ein få generert listefila *asm.txt* i rotkatalogen til prosjektet.
Denne fila inneheld mellom anna:

- Disassemblert maskinkode, dvs. assemblykode, for heile programmet.
- Absolutte adresser for programkoden.
- Adresser for globale variablar i koden der dei blir brukte.

[5] Viss *Build Automatically* er valt, så blir *Build All* køyrt kvar gong du lagrar ei programfil i SDK.

[6] Eit eksempel her er den ferdige drivarfila *xgpio_l.h* for parallellportar av typen *xps_gpio*, sjå programeksemplet i kapittel C.2.1.

C.3.4 Litt meir om lenkinga

Når ein skal utvikla eller modifisera mikrokontrollerbaserte system og så programmera desse, er det viktig at **ein veit kva ein gjer**. Ein må sjølv styra plassering av programkode og variablar i minnet, og det er her mange detaljar som må spesifiserast.

Sjølve kommandofila for lenkaren blir laga automatisk utfrå spesifikasjonen på det systemet ein har bygt, men det er likevel viktig å forstå oppsettet av ei slik fil.
For å hjelpa på forståinga skal ein fyrst sjå på viktige **variabeltypar i C** og så på nemninga **seksjon**.
Deretter skal ein sjå litt på innhaldet i ei typisk kommando-fil for lenking.

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 her lagde i nokre av dei faste prosessorregistra, evt. på stakken, og eksisterar berre ei tid. Dette fører til at viss fleire funksjonar kallar same funksjon, vil den same variabelen kunne eksistera på ulike stader i minnet til ulik tid. Dersom ein funksjon 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 funksjonen har blitt kalla. Då automatiske variable kjem og går med funksjonskall, held dei ikkje på verdien sin mellom funksjonskall. Dette gjer at dei må gjevast verdi for kvar gong funksjonen blir kalla, ellers vil innhaldet av variabelen innehalda ein tilfeldig verdi.
Eksempel:

```
int funksjon(...) {
    int v;  ......
}
```

Statiske variablar

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

```
int v;
static char str[10] = "ABC";
void main(void) { ..... }
```

eller

```
int funksjon(...) {
```

```
    static int v = 5;  ......
}
```

Initialiserte variablar

Initialiserte automatiske variablar er automatiske variablar som får verdi under start av ein funksjon, når programmet har starta. Når funksjonen 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 variabelen i minnet. Det blir og lagra ein konstant av variabelen på ein annan stad i minnet, og den får oppstartverdien til variabelen.
Oppstartsprogrammet, i dette tilfellet *crtinit*, kopierer konstantane inn i dei respektive variablane slik at desse får rett startverdi. (Dette programmet er ferdiglaga og vil etter kopiering m.m., kalla opp hovudprogrammet, *main()*.)
Eksempel:

```
int v = 2;
void main(void) { ..... }
```

eller

```
int funksjon(...) {
    static int v = 5;  ......
}
```

Seksjonar

Ein seksjon er eit område i minnet som blir avsett til å innehalda ein type data. I hovudsak har ein fylgjande seksjonar:

- ***text***, der **programkoden** blir lagt.
- ***rodata*** ("read only data"), seksjon for **konstantar** som f.eks. tekststrengar.
- ***data***, der **initialiserte statiske variablar** blir lagt, som f.eks. *diodeverdi* i eksemplet i kapittel C.2.1.
- ***bss*** ("block starting with symbol"), seksjon for **ikkje-initialiserte statiske variablar** som f.eks. *brytarkode* i eksemplet i kapittel C.2.1.
- ***bss_stack*** eigen seksjon for kladdeområde ("heap") og stakk.

Spesifikasjonen av desse seksjonane kan sjå ut som i lenkefil-eksemplet under (kommentarane "//...." er lagt til etterpå):

```
// Spesifikasjon av stakkstorleik
_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x400;

/* Define all the memory regions in the system */
// Storleiken på minneområdet er her 16kB som sett opp i BSB.

MEMORY {ilmb_cntlr : ORIGIN = 0x00000000, LENGTH = 0x3fff }

/* Specify the default entry point to the program */

// Dette er starten av oppstartsprogrammet. Adressa til _start skal
// plasserast i adresse 0 i minnet.

ENTRY(_start)

/* Define the sections, and where they are mapped in memory */
// Eit eksempel på korleis seksjonane vanlegvis er sette opp.
SECTIONS {
  .text : {
    _ftext = .;
    *(.text)
    *(.text.*)
    *(.gnu.linkonce.t*)
    _etext = .;
  } > ilmb_cntlr

  .rodata : {
    _frodata = .;
    *(.rodata)
    *(.gnu.linkonce.r*)
    _erodata = .;
  } > ilmb_cntlr

 .....

  .data : {
    . = ALIGN(4);
    _fdata = .;
    *(.data)
    *(.gnu.linkonce.d*)
    _edata = .;
  } > ilmb_cntlr

 .....

\pagebreak
```

```
  .bss : {
    . = ALIGN(4);
    PROVIDE (__bss_start = .);
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    PROVIDE (__bss_end = .);
  } > ilmb_cntlr

  .bss_stack : {
     . = ALIGN(8);
     _heap = .;
     _heap_start = _heap;
     . += _STACK_SIZE;
     . = ALIGN(8);
     _stack = .;
     __stack = _stack;
  } > ilmb_cntlr
}
```

C.4 Nedlasting og køyring av program

Ved å trykkja på nedlastingsknappen heilt til høgre i figur C.2 kan ein lasta ned programmet til FPGA-kretsen. Bitfila som blir lasta ned, inneheld to delar. Første del vil konfigurera maskinvaren i FPGA-en slik at den blir til ein mikrokontroller. Andre del av bitfila inneheld programmet som skal køyrast av denne mjuke mikrokontrolleren. Dette programmet, som er i form av maskinkode, blir plassert i BRAM-modulen, dvs. minnemodulen til mjukkontrolleren.
Etter nedlasting vil mjukkontrolleren automatisk starta køyringa av programmet.

Ein kan også etter at ein har laga ei køyrbar fil, *executable.elf*, utføra avlusing av programmet. Dette kan gjerast vhja. verktøyet Xilinx Microprocessor Debugger (XMD). Det finst her mange funksjonar til bruk ved avlusing som avbrekk ("breakpoint"), stegvis ("single-step") køyring samt lesing og skriving av variablar, minne og register.

Når ein er nøgd med programmet, kan ein viss ynskjeleg leggja programmet inn i Flash-minnet på Spartan3-kortet.
Vedlegg D viser korleis dette kan gjerast.

Vedlegg D

Om programmering av Flash-minnet på Spartan-kortet

D.1 Innleiing

Det vil i mange prosjekt vera ynskjeleg at systemet kjem opp av seg sjølv etter påslag av kraftforsyninga. Ved bruk av mjukprosessor i FPGA er ein då avhengig av at alle konfigurasjonsdata inkludert programvare kan liggja i eit permanent minne på kortet og lastast inn automatisk ved oppstart. Utan bruk av eit slikt minne må systemet koblast til programpakken SDK i verktøyet EDK og programmerast frå dette kvar gong, noko som er tungvint.

På kortet Spartan-3 Starter Board har ein til disposisjon eit serielt 2Mbit Flash-minne av typen XCF02S, og dette kan lett programmerast vhja. Xilinx sin programpakke Impact.
Ein framgangsmåte for dette er vist i det fylgjande.

D.2 Framgangsmåte

D.2.1 Generering av konfigurasjonsfil

1. Start opp programmet Impact som ligg under
   Xilinx ISE Design Suite 10.1 − > Accessories
   Det kjem då opp eit vindauge der ein første gong vel å laga eit nytt prosjekt. Finn på eit passande namn og legg prosjektfila i rotkatalogen til EDK-prosjektet ditt, dvs. i den katalogen som *.xmp*-fila ligg.

2. Ein vel nå å laga til ei PROM-fil og med PROM-filnamnet *Flash* (f.eks.). Formatet skal vera *MCS* og denne fila skal også plasserast i rotkatalogen.

3. Ein må nå velja seriell PROM, kretsfamilie *xcf* samt type *xcf02s* og henta denne inn vhja. *Add*-knappen.
   Gå så vidare og trykk så *Finish*.

4. Det kjem så opp eit vindu med namn *Add file*. Her går ein inn på underkatalogen *SDK_projects* $->$ *Implementation* og vel bitfila *download_sdk.bit*. Svar så nei til å leggja inn fleire filer.
   Ein skal nå sjå fila liggja under Spartan-kretsen i figuren.

5. Høgreklikk så i figurvinduet og vel der *Generate file*. Systemet vil då generera fila *Flash.mcs* som ein spesifiserte tidlegare.

D.2.2 Programmering

For at ein skal kunne gjera bruk av Flash-minnet, må koblingsbrua **JP1**, sjå brukarmanualen, stå i posisjon *Default* eller *Flash read*. I fyrstnemnde posisjon er Flash-kretsen bare aktiv under overføring av konfigurasjon til FPGA etter oppstart eller etter eit trykk på *PROG*-knappen. I sistnemnde posisjon er kretsen alltid aktiv og kan nyttast av programmet under køyring.
Xilinx sine applikasjonsnotat 482 og 694 seier meir om korleis prosjektet ditt må setjast opp for å få til dette. Det vil i mange system f.eks. vera ynskjeleg at systemet tek vare på nyinnsamla data eller data som blir endra frå gong til gong systemet er oppe og køyrer.

1. Ein skal nå dobbelklikka på *Boundary Scan* oppe til venstre og så høgreklikka i det vindauget som dukkar opp. Vel så *Initialize chain*.
   Ein skal nå først velja sjølve *.bit*-fila som er spesifisert tidlegare og så fila *Flash.mcs*.

2. I vindauget som nå kjem opp, vel *Device 2*, som er sjølve Flash-kretsen. Trykk *Apply* og så *OK*. Då skal det koma opp ein ny figur som viser kretsane og dei relevante filene.

3. Eit nytt høgreklikk i same vindauget og val av *Program* startar overføringa til Flash-kretsen.

4. Slå nå av og på kraftforsyninga til systemet eller trykk på *PROG*-knappen for å verfisera at ein er i mål.