Institutt for fysikk og teknologi

Implementasjon av interrupt-styrd DMA-overføring på HLT-RORC

Masteroppgåve

av

Olav Torheim

September 2005
Universitet i Bergen

Samandrag

På CERN i Geneve vert det bygd ein ny partikkelakselerator kalla LHC. Ein av dei fire hovuddetektorane i LHC vert kalla for ALICE. I ALICE-detektoren møtest akselererte blyion i høgenergetiske kollisjonar. Med desse kollisjonane vonar ein å kunna skapa eit kvark-gluon-plasma. Fyremålet med ALICE-eksperimentet er å kunna detektera at eit slikt plasma har vorte dana.

ALICE er oppbygd av ei rad ulike subdetektorar der Time Projection Chamber, TPC, er den viktigaste. TPC er utstyrd med sin eigen utlesingselektronikk som syter for at mælingane vert digitaliserte og ordna i fiberoptiske datastraumar.

Datainnsamlingssystemet åt TPC er bygd opp med standard Linux-baserte datamaskinar og PCI-innstikkskort kalla RORC. RORC-korti har tilkoplingar for eit fiberoptisk grensesnitt. RORC kan soleis nyttast til å taka i mot data fiberoptisk frå detektoren og flytja mottekne data inn i eit computer memory.

For å redusera den innkomande datastraumen er ALICE-eksperimentet utstyrt med eit trigger-system. Trigger-systemet opererar på ulike nivå. På lægste nivå er trigger-systemet knytt til snøgge detektorar som triggar utlesingi av data frå TPC. På eit høgare nivå skal det implementerast ein høgnivåtrigger, HLT, som kan køyra sanntidsanalyse av den utlesne datastraumen og plukka ut interessante kollisjonsdata.

HLT er sett i hop av ei klynge med konvensjonelle datamaskinar. HLT implementerar sine eigne RORC-kort, HLT-RORC, til å taka i mot data frå utlesingselektronikken.

Fyremålet med denne oppgåva har vore å konstruera og testa eit grensesnitt som gjev HLT-RORC tilgang på alle dei generelle funksjonane som PCI-protokollen tilbyd. Dette grensesnittet er implementert for 66MHz arbeidsfrekvens og 64 bits ordbreidd og det er i stand til å initiera både DMA-styrd og interrupt-styrd IO. Til å letta implementasjonen har eg nytta kommersielle IP-kjernar for PCI. Både maskinvare, firmware og programvare har vorte utvikla, inkludert device-drivarar for Linux og eit testmiljø. All firmware har vorte verifisert både med simulering i programmet ModelSim og testing i tilgjengeleg maskinvare.

På slutten av arbeidet med masteroppgåva vart eg involvert i eit satellittprosjekt hjå Seksjon for romfysikk. I samband med dette implementerte og verifiserte eg eit SPARC-basert mikroprosessorsystem på den same maskinvara som eg hadde nytta til utvikling av HLT-RORC. Dette arbeidet er nærare dokumentert i eit eige kapittel.

Under arbeidet med masteroppgåva har eg dessutan hjelpt studentar ved Høgskulen i Bergen med utvikling og testing av firmware-modular for RCU-systemet. Desse modulane hadde til fyremål å dekoda meldingar frå TTCrx-brikka sin serialb-interface og er nærare dokumentert i 'Digital Module Requirement Specifiction' [1].

At nokre av kapitli er skrivne på engelsk og andre på norsk heng i hop med at dei engelskspråklege kapitli skulle lesast av folk ved Universitetet i Heidelberg i Tyskland.

Fyreord

Denne oppgåva har eg arbeidd med på Seksjon for instrumentering og elektronikk ved Institutt for fysikk og teknologi, Universitet i Bergen, frå januar 2005 til september 2005.

Eg vil retta ein stor takk til rettleidaren min, Kjetil Ullaland, for støtte og rettleiding i arbeidsbolken. Vidare vil eg takka Ketil Røed, Johan Alme og Werner Olsen for mange diskusjonar og gode råd i samband med dei praktiske sidone ved oppgåva. Bjørn Pommeresche, Per Heradstveit, Arne Solberg og Geir Frode Sørensen skal ha takk for praktisk hjelp og rettleiding med utlegg, montering og testing av mynsterkort. På same måten vil eg takka Artur Szostak, Torsten Alt og Timm Steinbeck for at dei kom med framlegg til forbetringar på det arbeidet eg hadde gjort med utvikling av maskinvare, firmware og programvare for HLT-RORC. Heinz Tilsner skal ha takk for at han gav meg kjeldekoden for HLT-RORC. Og Dieter Røhrich skal ha takk for at han las gjennom oppgåva mi og kom med framlegg om korleis ho skulle disponerast. Til slutt vil eg takka Liv Skartveit ved Odontologisk Fakultet for at dei gav meg tilgang til røntgenmaskinen deira.

Bergen, september 2005

Olav Torheim

Innhald

Kapittel 1

Datainnsamling for høgenergifysikk

1.1 Introduksjon til datainnsamling

1.1.1 Det generelle målesystemet

Fyremålet med eit datainnsamlingssystem er å samla inn numeriske data som svarar til fysiske variablar i den verkelege verdi. Komponentane i eit datainnsamlingssystem inkluderar høvelege sensorar, omformarar og prosesseringselement som innanfor ei endeleg måleuvisse er i stand til å konvertera målevariablane til elektriske signal, digitalisera dei elektriske signali og leggja dei inn i ein datamaskin der dei kan prosesserast.

Eit generelt målesystem kan til vanleg kløyvast opp i fire separate stykke [2]:

- Eit sensorelement som er i direkte kontakt med den prosessen det skal målast på. Døme på dette kan vera ein strekklapp der resistans korresponderar med mekanisk strekk.
- Eit omformarelement som konverterar responsen frå sensoren om til eit signal som er meir høveleg for prosessering, som regel eit spenningssignal eller eit straumsignal. Døme på dette kan vera ei målebru.
- Eit signalprosesseringselement som konverterar signalet frå omformarelementet til ei form som er meir høveleg for presentasjon. Døme på dette kan vera ei kjede med analogt filter, ein AD-omsetjar, eit digitalt filter og ein digital krins som køyrer Fast Fourier Transform på sampla data.
- Eit datapresentasjonselement som presenterar dei mælte verdiane for ein observatør.

1.1.2 PC-basert datainnsamling

For sensorane og omformarane har ein ikkje noko val, desse lyt alltid setjast i hop med analoge komponentar. Signalprosessering og datapresentasjon er derimot døme på operasjonar som godt kan implementerast i digitale datamaskinar, og som vert det i aukande mon.

Den personlege datamaskinen, PC, har heilt sidan han vart introdusert på byrjingi av 80-talet vore basert på opne standardar som alle stend frie til å implementera. Tilhøvet mellom pris og yteevna har støtt vorte betre samstundes som funksjonaliteten har vorte utvida. Eit utal typar programvarebibliotek og ekspansjonskort gjev PC'en høg fleksibilitet, noko som er med på å gjera PC-basert datainnsamling fyremålstenleg. I eit PC-basert datainnsamlingssystem kan ein anten nytta måleinstrument som kommuniserar med PC'en via kabel eller ein kan nytta eigne datainnsamlingskort som er plasserte i PC'en sin ekspansjonsbuss (sjå til dømes [3]).

For instrument som kommuniserar via kabel har tradisjonelle IO-standardar som GPIB og RS232-C vore vanlege. Dette er standardar som vart etablerte alt på slutten av 60-talet og som har ei svært avgrensa bandbreidd. Nye instrument nyttar gjerne meir moderne standardar som USB og FireWire.

Fram til midten av 90-talet var ISA-bussen den mest dominerande ekspansjonsbussen for Intel-baserte datamaskinar. Med introduksjonen av Pentium-prosessoren og operativsystem med støtte for automatisk konfigurering (Plug And Play, PnP) har det vore ein overgang frå tradisjonelle ekspansjonsbussar til sokalla lokalbussar der PCI-standarden har vore dominerande. Nemningi lokalbuss vert støtt mindre nytta og kjem av at denne busstypen i utgangspunktet var ei utviding av CPU-bussen: same breidd på data- og adressebussane og til å byrja med tilnærma same bandbreidd. Det var fyrst og fremst avgrensingar i bandbreidd som tvinga fram snøggare bussar. Til dømes var ISA-bussen avgrensa til 16-bits dataoverføring og ein arbeidsfrekvens på 8,33 MHz (sjå til dømes [4]).

1.1.3 PCI-standarden

PCI er ein open standard som vert halden ved lag av PCI Special Interest Group, PCI-SIG. Arbeidet med PCI-standarden byrja i 1990 hjå Intel. PCI 1.0, som fyrst og fremst var ein spesifikasjon på komponentnivå, kom ut i 1992. I 1993 kom PCI 2.0. PCI 2.0 var spesifisert for 5V IO-spenning, 33MHz arbeidsfrekvens og 32-bits ordbreidd. Seinare revisjonar av PCI har lagt til nye eigenskapar og ytingsforbetringar, mellom anna 3.3V IO, 66 MHz arbeidsfrekvens og 64-bits ordbreidd [5].

Omframt den konvensjonelle PCI-standarden er mellom anna desse PCI-variantane tilgjengelege:

PCI-X vert vedlikehalde av PCI-SIG. Det er ei utviding av PCI-standarden som fyrst og fremst er meint for servermaskinar. PCI-X 1.0 kan køyra på 133 MHz arbeidsfrekvens. PCI-X 2.0 har dessutan støtte for 266 MHz og 533 MHz arbeidsfrekvens [6].

CompactPCI vert vedlikehalde av PCI Industrial Computer Manufacturers Group og er meint for industrielle applikasjonar. CompactPCI definerar PCI-modular med standard Eurocard-dimensjonar som samsvarar med dei mekaniske standardane åt IEEE 1101.1. Dei elektriske spesifikasjonane er dei same som for konvensjonell PCI [7].

PXI er eit subsett av CompactPCI som vert vedlikehalde av PXI Systems Alliance. PXI står for PCI Extensions for Instrumentation og er utforma serskilt for industrielle instrumenterings- og automasjonssystem. Attåt dei vanlege PCI-funksjonane inneheld PXI mellom anna ein separat lokalbuss, ein triggerbuss og ei 10MHz referanseklokke. Programvarebiblioteket VISA, som vert

nytta i ei rad ulike kontroll- og instrumenteringssystem, har definert eigne grensesnitt for kontakt med PXI-devices [8].

Instrumenteringsbussar som PXI kan vera nyttige i miljø der ein har ulike PCI-kort med ulike funksjonar (aktuatorar, sensorar og so bortetter) som treng sanntids kommunikasjon og synkronisering seg i mellom. I datasystem der devicane opererer sjølvstendig har det mindre fyre seg å investera i denne standarden.

Medan parallellbussar som ISA og PCI har vorte omtala som 1. og 2. generasjons, vert seriebussar basert på differensiell lågspenningssignalisering, LVDS, rekna som 3. generasjons IO-standard. Parallellbussar har synt seg å ha naturlege avgrensingar når det kjem til skalering. Difor reknar dei fleste reknar med at ein overgang til LVDS-baserte serieprotokollar kjem til å taka over for dagsens parallellbussar. **PCI Express** vart tidlegare kalla 3GIO. Det er ein serieprotokoll som er basert på LVDS, men som er programvarekompatibel med konvensjonell PCI (sjå til dømes [9]).

1.1.4 HLT-RORC - datainnsamling for høgenergifysikk

Fyremålet med masteroppgåva var å konstruera eit datainnsamlingskort for høgenergifysikk, nærare kartlagt HLT-systemet åt ALICE-eksperimentet på CERN. Som det er synt i underkapittel 1.2 må dette datainnsamlingskortet, kalla HLT-RORC, vera i stand til å taka unna ei gjennomsnittleg datarate frå detektoren sin frontend-elektronikk på 70MByte/sek med peak-datarater på opp til 200 MByte/sek[1].

Med HLT-prosjektet er det dessutan eit mål å nytta konvensjonelt datautstyr so langt som det rekk. Den konvensjonelle PCI-standarden kjem soleis til å vera fyrstevalet, so sant han oppfyller kravi til datarate.

For å hindra at PCI-grensesnittet vert ein flaskehals, er det for det fyrste eit minstekrav at ein klarar å taka unna dei gjennomsnittlege datastraumane som kjem frå utlesingselektronikken. Klarar ein ikkje dette, er ikkje PCI-grensesnittet brukande.

Ein PCI-device[2] med 32-bits databuss og 33MHz arbeidsfrekvens kan ved maksimal bussutnytting oppnå ei datarate i underkant av 32*33MHz = 1056MBit/sek = 132 MByte/sek. Her er det ein fyresetnad at PCI-devicen er åleine på bussen og at dataoverføringi gjeng føre seg i langvarande burst-transaksjonar med minimal overhead.

Jamvel med desse vilkåri til stades er bandbreiddi for liti til å klara dei største peak-datastraumane på 200 MByte/sek utan bufring. Eit høveleg stort slakkbuffer kunne vore implementert på PCI-devicen med fyremålet å redusera kravet til datarate. Alt med dagsens HLT-RORC-arkitektur vert mottekne data mellomlagra i ein 256x64 bits FIFO (sjå kapittel 2). På same måten vert data som kjem frå frontend-elektronikken bufra i ein 16x32 bits FIFO fyre dei vert sende vidare over DDL-linken. Skal ein seia noko meir om korleis desse bufferi bør dimensjonerast, lyt ein gjera ein statistisk analyse av heile datastraumen. Di djupare buffer, di fleire eventfragment kan lagrast og di nærare den gjennomsnittlege datastraumen kan ein rimeleg sikkert leggja kravi til datarate.

Ein PCI-device som køyrer på 66 MHz med 64 bits databuss skulle derimot vera meir enn

[1] Talet 70MByte/sek finn ein om ein deler 15 GB/sek frå TPC-detektoren på 216 stk DDL-links.

[2] Innanfor dataterminologien er device ei vanleg nemning for ei maskinvareeining.

nok til å klara 200 MByte/sek. Eit grovt overslag gjev oss 66MHz*64bitar/8 = 528 MByte/sek som er meir enn det doble av minstekravet. I praksis er det mindre med di det er uråd å utnytta bussen so effektivt.

For det andre so er det fyremunleg om ein kan ha meir enn éin PCI-device på same mottakarmaskin og soleis spara maskinvareressursar. Om to eller fleire PCI-devices kan fungera fullnøyande på same PCI-segment er eit spørsmål om bussutnytting (utilisasjon). Om me tek utgangspunkt i ein PCI-device med 64-bits databuss og 66 MHz arbeidsfrekvens, kan me gjera fylgjande grove overslag:

datastraum / bandbreidd = 200 MByte/528 = 0,38.

Jamvel ved maksimal utlesingsrate frå detektoren kunne ein tenkja seg at PCI-bussen låg i ro meir enn halve tidi. Med fornuftig samordning kunne det soleis vera råd for to devices å utnytta det same PCI-segmentet. Som det vert diskutert i kapittel 3 lyt det gjerast ei veging mellom responstid og bandbreidd. Di lengre bursttransaksjonar det er høve til å køyra på bussen, di meir data lyt mellomlagrast i slakkbufferi slik at PCI-devicane fær tid til å venta på einannan.

CPU-avlasting og synkronisering

Dersom PC'en som hyser datainnsamlingskortet skal vera med og prosessera mottekne data, bør det leggjast til rettes for at CPU'en slepp å nytta verdfull tid på andre arbeidsoppgåvor som til dømes utlesing frå datainnsamlingskortet. Det er difor eit mål at datainnsamlingskortet sjølv skal vera i stand til å taka kontroll med PCI-bussen og skriva til datamaskinen sitt main memory. Dersom datainnsamlingskortet treng å samvirke med CPU'en, bør dette gjerast på ein slik måte at CPU'en ikkje treng å overvaka kva datainnsamlingskortet held på med. Til desse oppgåvene kan me nytta DMA og interrupts. Ein DMA-kontroller innebygd på datainnsamlingskortet gjev høve til å overføra data på PCI-bussen utan hjelp frå prosessoren medan interrupt-mekanismane gjev datainnsamlingskortet høve til å avbryta prosessoren asynkront (sjå til dømes[10]).

1.2 ALICE-eksperimentet på LHC

Dette underkapitlet tek fyre seg den grunnleggjande funksjonaliteten åt ALICE-eksperimentet sitt datainnsamlingssystem. Det meste av informasjonen er henta frå ALICE Technical Design Report [11], ALICE Physics Performance Report [12], ALICE Technical Design Report of the Time Projection Chamber [13], VHDL-implementation of the Cluster Finder algorithm for use in ALICE [14] og Pattern Recognition and Data Compression for the ALICE High Level Trigger [15].

1.2.1 Introduksjon til CERN og LHC

CERN, den europeiske organisasjonen for kjerneforsking, er heimsens største forskingslaboratorium for kjerne- og partikkelfysikk. Her møtest fysikarar for å utforska kva materien er sett i hop av og kva for krefter som held honom i hop.

CERN eksisterar fyrst og fremst for å gjeva forskarane tilgang på partikkelakseleratorar med tilhøyrande detektorar. Ved å akselerera partiklar og få deim til å kollidera, kan dei ulike detektorane nyttast til å identifisera resultatet av kollisjonane. Hovudfyremålet er å utforska dei grunnleggjande byggjesteinane åt materien og finna ut korleis desse vekselverkar med einannan. Den ekstreme energien som er tilgjengeleg i desse kollisjonane kan føra til at ein oppdagar nye partiklar og nye materietilstand.

Det neste store prosjektet på CERN vert kalla Large Hadron Collider, LHC, og er venta å koma i gang i 2007. LHC er ein partikkelakselerator som er i stand til å akselerera partiklar opp til høgare energiar enn det som tidlegare har vorte observert. Energiar i TeV-valdet[3] vil verta nådde med frontkollisjonar av proton og tunge ion. Dei tunge partiklane vert sende i motsette retningar og synkroniserte til å møtast og kollidera i høvelege detektorar. Sentrale kollisjonar midt i detektoren er meint å gjeva dei beste målingane.

1.2.2 Finst det nye materietilstand ved høge energiar?

Innanfor atomkjernane er det proton og nøytron. Desse elementærpartiklane er bygde opp av endå mindre partiklar - kvarkar. Ulike slag kvarkar finst - kalla opp, ned, topp, botn, charm og strange. Proton er alltid sette i hop av to opp-kvarkar og ein ned-kvark medan nøytron har to ned-kvarkar og ein opp-kvark. Kvarkane er botne saman innanfor nukleoni av den sterke kjernekrafi. Den sterke kjernekrafti vert formidla av kraftutvekslingspartiklar kalla gluon. Desse kreftene er so sterke at ingen einskildkvark nokon gong har vorte observert (sjå til dømes [16]).

Ved svært høge temperaturar er det venta at nukleoni kjem til å løysa seg opp og forma eit plasma av kvarkar og gluon - det sokalla kvark-gluon-plasmaet, QGP. Forskarane trur at Universet slik me kjenner det i dag oppstod med Den store smellen for om lag 15 milliardar år sidan. I det fyrste mikrosekundet etter at Universet vart til var energitettleiken so høg at ein reknar med at all materien i Universet må ha vore i dette tilstandet. Desse vilkåri ynskjer ein å rekonstruera i LHC-eksperimentet [17].

[3] Tera-elektronvolt

Når akselererte blyion møtest i høgenergetiske kollisjonar, er det venta at dei kjem til å dana eit kvark-gluon-plasma. Hadroni[4] løyser seg då opp og skaper ei sky av frie kvarkar. Dersom kvark-gluon-plasmaet vert dana, kjem det til å framstå som eit Big Bang i miniatyr. Plasmaet kjem til å ekspandera og verta omforma til hadroniske partiklar etter berre 10^{-23} s. Om ein skal observera om dette plasmaet verkeleg har vorte skapt, lyt ein observera dei partiklane som vert sende ut. Ein kollisjon som produserar QGP sender ut eit anna 'fingeravtrykk' av partiklar enn det ein kollisjon som ikkje produserar QGP hadde gjort. Talet på partiklar som vert sendt ut frå desse kollisjonane er enormt, og detektoren må vera i stand til å detektera partiklar med både høge og låge energiar. Detektoren må kunna skilja partikkelbanane frå einannan og kunna identifisera so mange partiklar som det trengst for å kunna stadfesta at eit kvark-gluon-plasma har vorte dana [14].

1.2.3 ALICE-introduksjon

ALICE er namnet på ein av dei fire detektorane som har vorte utvikla for LHC-prosjektet. Når akselererte blyion møtest i høgenergetiske kollisjonar, er det venta at dei kjem til å dana eit kvark-gluon-plasma. ALICE har til fyremål å påvisa denne plasmadaningi med di detektoren leitar etter spor frå partiklane som kjem frå kollisjonspunktet.

Til liks med andre detektorsystem for høgenergifysikk er ALICE sett i hop av ulike subdetektorar [16]:

- Inner Tracking System (ITS). ITS kan mellom anna spora og identifisera partiklar med låg rørslemengd.
- Time Projection Chamber (TPC). TPC er hovuddetektoren for partikkelsporing og dimed den viktigaste detektoren i ALICE-eksperimentet.
- Partikkelidentifikasjonsdetektorane Transition Radiation Detector (TRD) og Ring Imaging Cherenkov (RICH).
- Photon Spectrometer (PHOS). PHOS er eit høgoppløyseleg elektromagnetisk kalorimeter.
- Forward Muon Spectrometer
- Forward-detektorane Zero Degree Calorimeter (ZDC), Photon Multiplicity Detector (PMD), Forward Multiplicity Detector (FMD), V0 counters og T0 counters. Dette er etter måten små detektorsystem som triggar på globale kollisjonskarakteristikkar som til dømes kollisjonstidspunkt og impact parameter[5].

Ein mekanisme kalla Central Trigger Processor samlar data frå ulike trigger-detektorar og genererar serskilde triggermeldingar som vert nytta til å koordinera utlesingi frå dei hine subdetektorane. Ein stor solenoidmagnet omsluttar alle dei nemnde subdetektorane. Som me skal sjå seinare i dette kapitlet er denne magneten turvande for å kunna måla rørslemengdi åt dei detekterte partiklane.

[4] Hadron er partiklar laga av kvarkar og gluon

[5] Impact parameter eit eit mål for kor sentrale kollisjonane er

1.2.4 Time Projection Chamber

Subdetektoren TPC er sett i hop av to tynnor, ei indre og ei ytre. Den indre radiusen på kammeret er tilnærma lik 85 cm, den ytre er på tilnærma 250 cm. Lengdi på kammeret er tilnærma 500 cm. Volumet mellom dei to tynnone er fylt med 90% Ne og 10% CO_2. Ein sentral elektrode, plassert midt i kammeret, vert nytta til å setja opp eit uniformt elektrisk felt i kvar halvdel av TPC. Ein stor elektromagnet (solenoidmagneten) syter for at det vert sett opp eit moderat magnetisk felt på 0,5 Tesla parallellt med det elektriske feltet.

Endeflatone åt TPC har eit rutenett av ladningsforsterkarar, filter og AD-omsetjarar. Kvar endeflata er kløyvd opp i 18 sektorar der kvar sektor er oppkløyvd i 6 patches. Patchane er vidare kløyvde opp i rows og pads. Totalt er det 159 rows og 570.132 pads.

Når tunge partiklar kolliderar i den indre tynna, vert det dana ei rad nye partiklar. Dei ladde partiklane skal etter høgrehandsregelen verta avbøygde i det magnetiske feltet. Di større rørslemengd partiklane har, di mindre avbøygning.

Dei ladde partiklane som vert dana i kollisjonen kjem i neste omgang til å ionisera gassen i TPC med det resultatet at frie elektron vert førde mot det uniforme elektriske feltet mot endeflatone åt TPC. På endeflatone kan posisjonane deira detekterast i nettverket av rows og pads. Soleis fær me ei todimensjonal posisjonering av punkt i romet. Det tredje koordinatet er gjeve av drifttidi åt elektroni.

Dei primære elektroni er ikkje i stand til å indusera eit sterkt nok til signal til at det kan skiljast frå støyen i ladningsforsterkarane. Detekteringi av dei drivande elektroni vert difor gjord med Multi-Wire Proportional Chambers, MWPC. Konkret vil det seia at utlesingskammeri er sette i hop av eit gitter med anodeplan over eit katode-padplan, eit katodegitter og eit gating-gitter. Ei negativ spenning er påført katodeleidningane og katodepaddane. Kring anodeleidningane vert det soleis dana eit sterkt elektrisk felt, proporsjonalt med 1/r. Når dei drivande primærelektroni gjeng inn i regionen attom gating-gitteret, held dei fram med å driva langs feltlinone mot den næraste anodeleidningi. I det dei kjem inn i det sterke elektriske feltet nær anodeleidningi, vert elektroni akselererte slik at dei produserar eit skred av ladde partiklar (avalanche ionization). Dei positive ioni som vert frigjorde i prosessen induserer ladning på katodepaddane.

Funksjonen åt gating-gitteret er å opna og stengja forsterkingsregionen for drivande elektron. Når ein trigger (sjå neste seksjon) er vorten signalisert, vert leidningane i gating-gitteret lagde på same potensial slik at primærelektroni frå driftregionen fær driva inn i forsterkingsregionen. Når det ikkje er noko gyldig triggervilkår til stades, er gating-gitteret forspent med eit bipolart felt. Det bipolare feltet hindrar elektroni frå driftregionen i å driva inn i forsterkingsregionen samstundes som det hindrar positive ion frå tidlegare kollisjonar i å driva attende til driftregionen.

1.2.5 Trigger-systemet

Sannsynet for at to akselererte partiklar skal kollidera med einannan er ikkje serleg stort. Under eksperimentet vil det difor verta utsendt tette pakkar av partiklar, kalla bunches. Når to partikkelpakkar møtest, vert det kalla for ein bunch cross. Om ei sovori kryssing resulterar i ein kollisjon, vert det kalla for eit event.

Rata for bunch crossing kjem under eksperimentet til å liggja på 40MHz, men i realiteten er

dei fleste av desse bunchane tome slik at den gjennomsnittlege interaksjonsrata for tungionekollisjonar kjem til å liggja på 8KHz. Kor som er, TPC-kammeret og den tilhøyrande utlesingselektronikken er einast dimensjonert for å kunna lesa ut reine, sentrale kollisjonar med ei rate på 200Hz. Målet åt trigger-systemet er difor å berre lesa ut interessante events. Snøgge detektorar med låg responstid vert nytta til å avgjera om ein skal starta utlesing frå dei seine detektorane. Central Trigger Processor, CTP, samlar data frå trigger-detektorane og tek trigger-avgjerder på alle nivå. Dei fire triggernivåi som CTP nyttar seg av vert kalla Level 0 (L0), Level 1 (L1), Level 2 Reject (L2r) og Level 2 Accept (L2a). Ei klokka kalla LHC-klokka som køyrer på 40 MHz vert synkronisert med utsendingi av bunches og distribuert til alle subdetektorane.

Dei ulike subdetektorane tek i mot trigger-signal frå Central Trigger Processor via ein Local Trigger Unit. Local Trigger Unit nyttar fiberoptiske kablar til å distribuera LHC-klokka åt subdetektorane. To kommunikasjonskanalar, kanal A og kanal B, vert tidsmultipleksa ut på det same mediet. Kanal A er eksklusivt dedikert til å kringkasta L1a-meldingar. I kanal B vert ulike L1- og L2-triggermeldingar (og ymse andre slag meldingar) modulerte inn i LHC-klokkesignalet. På subdetektorsida er det ein eigen integrert krins, kalla TTCrx, som rekonstruerar LHC-klokka og demodulerar triggermeldingane [18]. Takk vere LHC-klokka kan alle detekterte events knytast til ei spesifikk bunch crossing. Den globale eventidentifikasjonen er sett i hop av ei bunch crossing og eit orbit-nummer [19]. Orbit-nummeret fortel kor mange gonger bunchane har passert gjennom akseleratoren.

Dei snøggaste detektorane er knytte til L0-triggeren med responstid på 1,2 mikrosekund. Deretter fylgjer L1-triggeren med responstid på 5,5 mikrosekund. Det er L1-triggeren som startar utlesingi frå TPC. TPC er støtt sensitiv til events, difor er ei av oppgåvone åt CTP å syta for sokalla past-future-protection. For å ha eit gyldig event må ingen andre events henda seinare enn 100 mikrosekund fyre eventet (past protection) eller tidlegare enn 100 mikrosekund etter eventet (future protection).

Om det er meir enn 100 mikrosekund sidan fyrre event, fær utlesingi frå TPC lov til å starta. Om det kjem nye events under utlesingi, vil L2r verta trigga og utlesingi avbroti. Eit L2a indikerar at utlesne data frå TPC er gyldige.

1.2.6 Frontend-elektronikken åt TPC

Front End Card

Frontend-elektronikken åt detektoren er ansvarleg for å lesa ut ladningi som er indusert på katodepaddane. Frontend-kortet, FEC, inneheld ei komplett utlesingskjede med ladningsforsterking, pulsforming, digitalisering, prosessering og bufring av TPC-signali. Kvart FEC-kort inneheld all den analoge og digitale elektronikken som trengst til å lesa ut 128 pad-signal.

For kvar utlesingskanal vert den induserte ladningi forsterka og integrert av ein ladningssensitiv forsterkar. Det integrerte signalet vert deretter sendt gjennom ein semi-gaussisk pulsformar av andre orden. Båe desse analoge funksjonane er implementerte i ein integrert PreAmplifier/Shaper (PASA). Det analoge utgangssignalet vert deretter sampla av ein 10-bits ADC med 10 MHz samplefrekvens. Til slutt vert dei digitaliserte signali filtrerte, prosesserte, og lagra i eit minne, reiduge til utlesing. Alt dette vert gjort i ein eigen integrert krins med namnet ALTRO. ALTRO

syter mellom anna for baselinekorreksjon, tredje ordens digital filtrering og zero-suppression (nullundertrykkjing) av sampla data. Det siste vil seia at verdiar under eit visst terskelverde vert undertrykte. Etter zero-suppression syter ALTRO-krinsen for at data vert formaterte inn i ord på 40 bitar i samsvar med ei dataliste. Denne datalista vert lagra i eit sokalla multi-event-buffer på 1024x40 bitar [20].

Teknikken som ALTRO-krinsen nyttar med zero-suppression fører til at det vert meir data å lesa ut frå ALTRO-brikker som har detektert viktige kollisjonsdata enn det er frå ALTRO-brikker som hovudsakleg har detektert støysignal. Den gjennomsnittlege datastraumen frå TPC vert soleis redusert.

Når eit L1-triggersignal er motteke, vert data lagra i multi-event-bufferet der dei ventar på ein L2-trigger (reject eller accept). Ved L2a vert lagra data frosne i minnet. Eit L2r fører derimot til at dei kan overskrivast ved neste L1-trigger. Det maksimale talet på samples pr event er 1000. I praksis fører zero-suppression til at talet vert mindre. Eventbufferet åt ALTRO-krinsen skal soleis vera i stand til å lagra 4 eventfragment før det er fullt.

Både PASA-brikka og ALTRO-brikka implementerar 16 utlesingskanalar kvar. For 128 kanalar gjev dette 8 stk PASA og 8 stk ALTRO for kvart frontend-kort.

Readout Control Unit

Readout Control Unit, RCU, er utforma for å kunna implementera kontroll-, utlesing- og overvakingsfunksjonar for alle frontend-kort som er baserte på ALTRO-brikka. RCU-kortet syter for at data frå ulike FEC-kort vert samla i ein DDL-datastraum (sjå neste underseksjon). Det er RCU-kortet som distribuerar trigger-signal og klokkesignal til FEC og det tener som bindelekk mellom kontrollsystemet åt detektoren (Detector Control System, DCS) og FEC. For kvar av dei 36 TPC-sektorane er frontend-elektronikken sett i hop av 121 stk FEC, 6 stk RCU og 6 stk DDL. For heile TPC-kammeret gjev dette 216 stk DDL-links.

RCU nyttar ein eigen ALTRO-buss til å lesa ut data frå ALTRO-krinsane på frontend-korti. ALTRO-bussen er sett i hop av 40 kombinerte data- og adresselinor og 7 kontrollinor. Utlesingi frå multi-event-bufferi vert synkronisert av ei 40 MHz utlesingsklokka. Dei utlesne ordi vert formaterte om til 32-bits ord og bufra i ein separat FIFO fyre dei vert sende vidare som 32-bits data på DDL-linken.

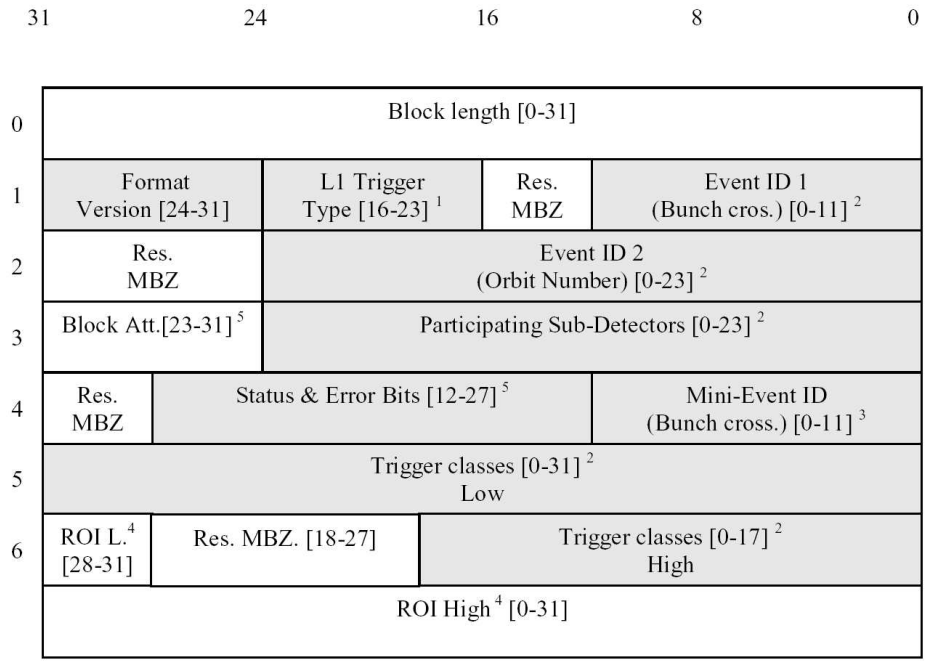
1.2.7 DDL og DAQ

Detector Data Link, DDL, er standarden for optisk dataoverføring mellom frontend-elektronikken åt detektoren og datainnsamlingssystemet DAQ. DDL er sett i hop av tre maskinvareeiningar: Source Interface Unit (DDL-SIU), Destination Interface Unit (DDL-DIU) og den optiske fiberen. DDL-SIU er plugga inn i frontend-elektronikken åt dei fleste detektorane. DDL-DIU er på DAQ-sida montert på mottakarkortet Readout Receiver Card, RORC.

Takk vere teknikken med zero-suppression av sampla data kan lengdi på eventfragmenti variera frå ALTRO-krins til ALTRO-krins. Me må like fullt taka høgd for at alle eventbufferi kan verta fulle, noko som maksimalt skulle gjeva ein peak-datastraum frå ALTRO-krinsane på 40bits x 40MHz = 1600 MBits/sek = 200 MByte/sek [21]. DDL-linken har teke høgd for dette med di

han er utforma til å klara ei peak-datarate på 264 MByte/sek og ei vedvarande datarate på 240 MByte/sekund [22].

So snart det har kome ei L2a-melding frå triggersystemet, vert data sendt med DDL frå detektoren til teljeromi via 216 optiske fiberkablar, éin for kvart segment. Datablokkene som vert overførde frå utlesingselektronikken lyt identifiserast og formaterast slik at dei kan verta prosesserte mest mogleg effektivt av mottakarnodane. Med DDL-dataformatet har ein sams header som inneheld det minimale talet på obligatoriske data vorte definert. Denne sams headeren vert lagd til rådatamengdi fyre ho vert send over DDL [19]. Figur 1.1 syner dei ulike blokkene som høyrer til i DDL-headeren.



Figur 1.1: DDL-headeren [19].

Data Acquisition, DAQ, er ansvarleg for å samla data frå alle subdetektorane og samla subevents til fulle events før dei vert sende til masselageret. Arkitekturen åt systemet er basert på vanlege PC-ar knytte i hop i eit nettverk, mest sannsynleg TCP over Gigabit Ethernet. Ved eit L2a vert data frå alle dei ulike detektorane overført via DDL til Local Data Concentrators, LDC. Samstundes vert ein kopi av DDL-datastraumen overførd til High Level Trigger, HLT, parallellt med DAQ.

DAQ Readout Receiver Card, D-RORC, er eit PCI-innstikkskort som LDC-maskinane nyttar til å knyta seg til DDL-datastraumen. D-RORC-teknologien er basert på FPGA. RORC har innebygd ein eigen DMA-kontroller som syter for at dei ulike event-fragmenti kan flytjast frå DDL beinveges inn i eit PC-memory med minst mogleg CPU-assistanse. Dataformateringi som vart introdusert med DDL gjev D-RORC høve til å skilja eventfragmenti i minnet, utan å avbryta

prosessoren åt LDC-maskinen. D-RORC inneheld 2 DDL-grensesnitt. For dei detektorane som vert analyserte av HLT, syter eitt av DDL-grensesnitti for å overføra ein kopi av datastraumen til ein av mottakarnodane i HLT-nettverket.

I LDC-maskinane vert eventfragment frå ulike RORC-kort sette i hop til sub-events. Etter at sub-events er sette i hop i LDC, vert dei overførde via eit event-builder-nettverk til ein Global Data Collector (GDC) der heile eventet kan setjast i hop. Dei ferdigbygde eventi vert til slutt sende til eit permanent lagringsmedium for arkivering og seinare offline-analyse.

1.2.8 High Level Trigger

Med sentrale Pb-Pb-kollisjonar er den planlagde TPC-utlesingsrata på 200Hz. Dette svarar til ei gjennomsnittleg datarate på 15GB/sek. Data kjem til å verta lagra på band for etterfylgjande offline-analyse, men den maksimale datarata til bandstasjonen er på 1,2GB/sek. Komprimering i ein orden av 10:1 er soleis turvande. Konvensjonelle tapsfrie kompresjonsteknikkar som RLE, Huffman og LZW oppnår berre 2:1. Med online tracking (partikkelsporing) og event-rekonstruksjon oppnår ein mykje betre kompresjon utan å missa viktig informasjon.

I teljeromi skal mottekne data gå gjennom ein sanntids dataanalyse før dei vert godtekne eller avviste av ein High Level Trigger, HLT. Ein hovudkomponent i HLT-systemet er evna til å køyra sanntids mynsterattkjenning på rådata frå detektoren. HLT-systemet er utforma til å utnytta informasjonen frå alle subdetektorane, inkludert TPC og dei snøgge detektorane. HLT-systemet er oppbygt som eit distribuert system med klynger av linux-baserte datamaskinar. Dersom ein ser på verkemåten åt HLT, kan ein slå fast at han byggjer på tre grunnleggjande funksjonar:

- **Trigger:** Aksepter eller avvis events basert på detaljert online-analyse av fysiske variablar.
- **Select:** Vel ut relevante delar av eventet eller regionar av interesse, til dømes utlesing av jet-regionar eller bortfiltrering av tracks (partikkelspor) med låg rørslemengd. Ved å analysera trackinginformasjonen eller utnytta informasjon frå andre detektorar som til dømes TRD kan slike regionar verta definerte.
- **Compress:** Reduser datamengdi som trengst til å koda event-informasjonen so mykje som råd utan å missa informasjon som gjeng på fysiske variablar.

Etter at eit event fyrst har vorte trigga og interessante regionar er valde ut, vert det komprimert og sendt til event-builder-nettverket for permanent lagring. Medan triggerane på L0, L1 og L2 er knytte til serskilde detektorar, er HLT einast knytt til analysering av den utlesne datastraumen.

Sporfinning

Det er ulike teknikkar som kan nyttast til sporfinning (tracking). Me skil mellom sekvensielle og iterative metodar:

I det sekvensielle tilfellet vert mynsterattkjenningi gjennomførd i to sekvensielle steg. I det fyrste steget vert klynger av ladde partiklar rekna om til punkt i romet. I det andre steget vert lista over punkt i romet samla i ulike track-segment.

Dataflyten vert då slik:

- **Cluster Finder** Clusterfinderen vil avgjera kva for ladningsverdiar med tilhøyrande pads som utgjer ei klynge av ladde partiklar. Deretter reknar han seg fram til gravitasjonssenteret åt klynga og presenterer dette senteret som eit punkt i romet. Slik kan clusterfinderen oppnå presisjon på 'sub-pixel-nivå'.

- **Track Finder** Trackfinderen leitar etter punkt som høyrer til same partikkelbanen og samlar desse punkti i eigne subsett.

- **Track Fitter** Trackfitteren gjeng ut frå at punkti i dei ulike subsetti fylgjer spiralbanar og nyttar desse vilkåri til å henta ut dei tilhøyrande korgparametri.

- **Track Merger** Trackmergeren slær i hop ulike track segment som høyrer til same partikkelbanen.

Den sekvensielle tilnærmingi med utrekning av gravitasjonssenter har openberre avgrensingar når det kjem til overlappande ladningsdistribusjonar. Fyremålet med iterativ sporfinning er å få tilgang på sporinformasjon før ein byrjar leitar etter clusters.

Med denne framgangsmåten er mynsterattkjenningi sett i hop av to hovudkomponentar: Track candidate finding og cluster fitting. I det fyrste steget gjeng mottekne rådata gjennom ein Hough-transformasjon for å finna ei lista over sporkandidatar. Hough-transformasjon er ein standard framgangsmåte innanfor biletanalyse som gjev høve til attkjenning av globale mynster. Hough-transformasjon gjer punkt i biletromet om til korger i eit parametrisk rom. Med Hough-transformasjon kan mynsterattkjenning i biletromet reduserast til detektering av lokale maksimum i det parametriske romet. Etter at Hough-transformasjon og peak finding er fullført, gjeng dei aktuelle sporsegmenti vidare til ein Cluster Fitter som rekonstruerar klyngene sine massesenter langs partikkelbanen ved å måta dei tilhøyrande ladningsdistribusjonane til ei parametrisert form. Til slutt vert dei tilordna klyngene måta etter ein spiralbane med det fyremålet å få eit best mogleg estimat på sporparametri.

Dataflyten vert då slik:

- **Hough-transform** Hough-transformasjonen finn globale korger som kan parametriserast slik at dei høver med alle dei lokali punkti i biletet.

- **Peak Finder** Peakfinderen leitar i det Hough-transformerte romet etter lokale maksimum som korresponderar til moglege tracks.

- **Cluster Fitter** Clusterfitteren gjeng gjennom lista over sporkandidatar og vel ut dei klyngekoordinati som best kan måtast etter lista. Clusterfitteren gjeng ut frå at klyngene kan skildrast med todimensjonale gaussiske funksjonar.

- **Track Fitter** Trackfitteren køyrer korgtilmåting på dei utrekna punkti i romet slik at han kan henta ut dei tilhøyrande parametri.

HLT-arkitekturen

HLT har eigne grensesnitt mot ei rad ulike subsystem, men berre to grensesnitt i datavegen. På datainngangen tek HLT imot umodifiserte rådata via DDL-DIU. På datautgangen vert DDL-SIU nytta til å senda trigger-avgjerder og prosesserte data til DAQ-systemet på same måten som for ein annan sub-detektor.

Mottakarnodane åt HLT-systemet vert kalla HLT Front End Processors, FEP. Ein FEP er ein standard datamaskin som hyser PCI-instikkskortet HLT-RORC. Det er HLT-RORC som inneheld DDL-grensesnittet åt HLT. HLT-RORC har innebygd ein eigen DMA-kontroller som syter for at dei ulike event-fragmenti kan flytjast frå DDL beinveges inn i eit PC-memory med minst mogleg CPU-assistanse. Kvar FEP inneheld eitt eller fleire HLT-RORC-kort. Datamaskinen sitt main memory fungerer som eventbuffer. FEP og HLT-RORC utgjer det fyrste nivået i HLT-hierarkiet.

Det er ein høg grad av lokalitet i dei data som vert mottekne av dei ulike FEP-nodane. Soleis er det fullt mogleg å rekna ut lokale variablar, td punkt i romet, utan å ha nokon kjennskap til data som er mottekne andre stader. Clusterfinding og Hough-transformasjon er operasjonar som må utførast før andre slag operasjonar som track finding og track fitting kan gjennomførast. Desse operasjonane har høg grad av parallellitet og er dimed ideelle å køyra i ein eigen coprosessor på HLT-RORC slik at ferdigprosesserte data kan førast rett inn i minnet åt FEP. På HLT-RORC skal det difor implementerast ein eigen coprosessor som tek seg av desse operasjonane og syter for at mottekne rådata er omgjorde til fysiske variablar før dei vert lesne inn i eventbufferet åt FEP.

HLT-maskinklynga

Etter at HLT-RORC har flutt prosesserte data inn i FEP sitt main memory, vil FEP flytja mottekne data til ein node på neste nivå i HLT-hierarkiet. På dette nivået kjem prosesseringi truleg til å inkludera track finding eller peak finding på sektornivå. Kvart nivå er sett i hop av so mange nodar som trengst til å gjera den turvande prosesseringi. Slik vil utgangsdata produserte på kvart nivå verta vidaresende til neste nivå med nodar til dess det endelege nivået i hierarkiet er nådd. På toppen av hierarkiet finn ein eit globalt nivå der alle turvande data er samla og det endelege eventet rekonstruert, dvs at spor frå dei ulike sektorane er slegne i hop og tilmåta, og ei endeleg trigger-avgjerd vert teki på grunnlag av utveljingsalgoritmar. Trigger-avgjerdi og korresponderande data vert deretter sende til DAQ-systemet for utlesing og lagring. Grensesnittet mellom DAQ og HLT er sett i hop av DDL-links mellom eit sett av HLT-nodar og DAQ-maskinar.

HLT-prosesskommunikasjon

For å utveksla data mellom dei ulike programvarekomponentane som køyrer på HLT-nodane har det vorte utvikla ein eigen infrastruktur for prosesskommunikasjon. Dette råmeverket definerer dataprodusentar - Publisher - og datakonsumentar - Subscriber. Prosessar som konsumerar data lyt instansiera eigne Subscriber-objekt medan prosessar som produserar data lyt instansiera eigne Publisher-objekt. Subscriber-objektet kan då registrera seg hjå eit Publisher-objekt for å taka i mot eventdata med kvart som desse vert reiduge. For skuld effektiviteten vert ikkje data overført mellom prosessane. I staden utvekslar prosessane strukturar med peikarar til data. På denne måten vert data lagra i minnet lengst mogleg, utan uturvande kopiering.

1.3 Om dette arbeidet

Fyremålet med denne oppgåva har vore å konstruera og testa eit grensesnitt som gjev HLT-RORC tilgang på alle dei generelle funksjonane som PCI-protokollen tilbyd. Dette grensesnittet er implementert for 66MHz arbeidsfrekvens og 64 bits ordbreidd og det er i stand til å initiera både DMA-styrd og interrupt-styrd IO. Til å letta implementasjonen har eg nytta kommersielle IP-kjernar for PCI. Både maskinvare, firmware og programvare har vorte utvikla, inkludert device-drivarar for Linux og eit testmiljø. All firmware har vorte verifisert med simulering i programmet ModelSim og testing i tilgjengeleg maskinvare.

Masteroppgåva er vorti oppkløyvd i desse bolkane:

Kapittel 2 tek fyre seg arkitekturen åt HLT-RORC og gjeng stutt gjennom det som har vorte gjort for å utvida og forbetra denne. Kapitlet er oppkløyvt i tre underbolkar:

Underkapittel 2.1 dryfter dei funksjonelle kravi til HLT-RORC og kva for nye funksjonar som må leggjast til den opphavlege arkitekturen.

Underkapittel 2.2 gjev eit stutt oversyn over dei ulike modulane som høyrer med i HLT-RORC-designet.

Underkapittel 2.3 tek fyre seg arbeidet som vart gjort med utvikling av ein maskinvare-prototyp for HLT-RORC med PCI-kantkontakt og Xilinx Virtex-4 (eit PCI-testkort).

Kapittel 3 fokuserar på PCI-spesifikasjonen. Kapitlet er oppkløyvt i tre underbolkar:

Underkapittel 3.1 tek fyre seg hovuddragi åt PCI-standarden.

Underkapittel 3.2 tek fyre seg interrupts og interrupt acknowledge slik det vert gjennomført på PCI-bussen.

Underkapittel 3.3 tek fyre seg korleis PCI-standarden kan implementerast med dei FPGA-baserte IP-kjernane Altera PCI MegaCore og Xilinx LogiCORE PCI[6].

I **kapittel 4** ser me nærare på dei ulike grensesnitti som vart implementerte på HLT-RORC. Kapitlet er oppkløyvt i fire underbolkar:

Underkapittel 4.1 tek fyre seg designendringar som vart gjorde for å integrera interrupt-handtering på HLT-RORC.

Underkapittel 4.2 tek fyre seg uventa hendingar som ikkje vart handterte i den opphavlege utgåva av HLT-RORC.

Underkapittel 4.3 tek fyre seg designendringar som vart gjorde for å retta opp i dei vandemåli som vart påviste i underkapittel 4.2. Desse endringane kom på plass før arbeidet med å migrera over på Xilinx-teknologi tok til.

Underkapittel 4.4 tek fyre seg arbeidet som vart gjort med å flytja HLT-RORC-designet over på Xilinx-teknologi.

Kapittel 5 tek fyre seg utvikling av drivarar og brukarapplikasjonar i Linux generelt og for HLT-RORC spesielt. Kapitlet er oppkløyvt i to underbolkar:

[6]Ein IP-kjerne er ein kommersielt tilgjengeleg og syntetiserande maskinvarespesifikasjon

Underkapittel 5.1 tek fyre seg måten Linux-systemet er utforma til å handtera lågnivå maskinvarehandtering og interrupt-handtering[7].

Underkapittel 5.2 tek fyre seg arbeidet som vart gjort med å konstruera ein eigen drivar og ein API for HLT-RORC.

Kapittel 6 tek fyre seg eit sideprosjekt som eg hadde på slutten av masteroppgåve der eg integrerte eit SPARC-basert mikroprosessorsystem på PCI-testkortet.

I **kapittel 7** dryfter eg alternative standardar som PCI-X og PCI Express. I slutten av kapitlet freistar eg draga nokre konklusjonar frå arbeidet med HLT-RORC.

Omframt dei vanlege kapitli er fylgjande tillegg tekne med:

Tillegg A inneheld ei laboratorieøving der studentane fær høve til å simulera, syntetisera og implementera HLT-RORC på ein Altera-basert FPGA. Til slutt fær dei høve til å køyra gjennom eit lite testscenario der HLT-RORC genererer eigne data og overfører desse til eit mottakarminne.

Tillegg B tek fyre seg bussanalysatoren Vanguard. Denne analysatoren hadde eg mykje nytte av under arbeidet med testing og debugging av HLT-RORC. Han vert ogso nytta i laboratorieøvingi i tillegg A.

Tillegg C gjer ei jamføring av transaksjonsgangen på dei to IP-kjernane Altera PCI MegaCore og Xilinx LogiCORE PCI.

I **tillegg D** ser me nærare på det arbeidet som vart gjort med montering og funksjonstesting av PCI-testkortet. I dette tillegget diskuterer ein mellom anna ymse designfeil og produksjonsfeil som kom for dagen etter at mynsterkortet var produsert og FPGA-krinsen montert.

Tillegg E inneheld teikningar med skjema og layout for PCI-testkortet slik det vart etter at designfeili som vert kommenterte i tillegg D er vortne retta opp.

Tillegg F inneheld ei lista over dei viktigaste filene som eg har arbeidd med.

Tillegg G inneheld ei alfabetisk ordna liste over alle styttingar som har vorte nytta i dette arbeidet.

[7] Lågnivå er det nivået der programvara har direkte tilgang til register, minne og IO

Kapittel 2

HLT-RORC

Dei to fyrste seksjonane i dette kapitlet tek i grove trekk fyre seg arkitekturen åt HLT-RORC og det arbeidet som har vorte gjort for å utvida funksjonaliteten og gjera arkitekturen meir robust. Den tredje seksjonen tek fyre seg arbeidet som har vorte gjort med å utvikla ein maskinvareprototyp for HLT-RORC basert på Xilinx Virtex-4.

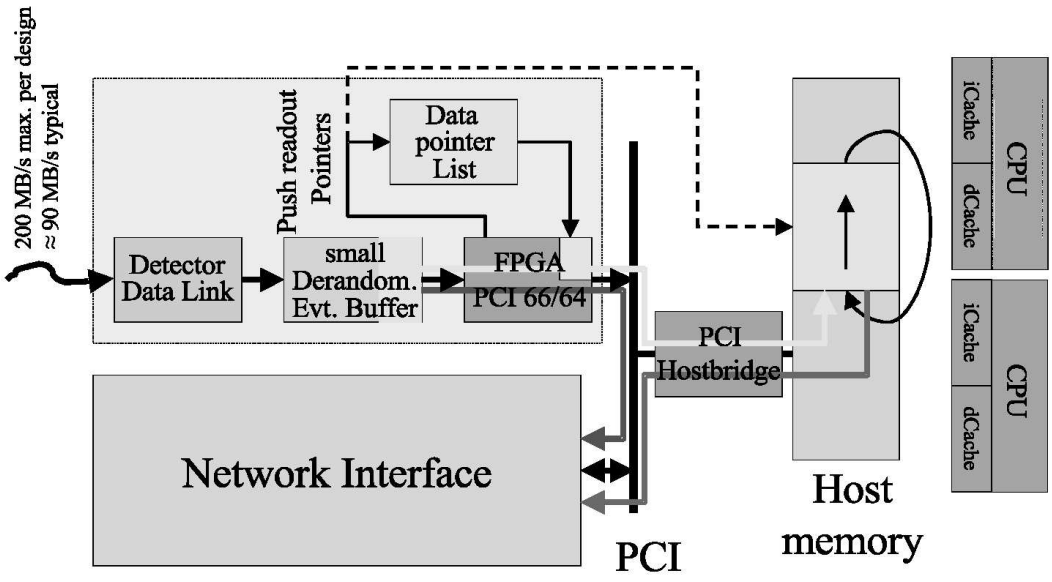
2.1 Funksjonelle krav

2.1.1 Opphavleg verkemåte

HLT-RORC er ei stytting for High Level Trigger Readout Receiver Card. HLT-RORC er utlesingskortet som High Level Trigger (HLT) nyttar til å lesa data frå frontend-elektronikken åt TPC-detektoren. HLT-RORC er utstyrt med eit grensesnitt for PCI og eit mottakargrensesnitt for DDL-protokollen. DDL-protokollen er basert på fiberoptisk dataoverføring og nyttar eit sendargrensesnitt kalla SIU og eit mottakargrensesnitt kalla DIU [23]. På denne måten kan HLT-RORC nyttast til å taka i mot data frå frontend-elektronikken sin SIU-interface. Ein DMA-kontroller på HLT-RORC syter dinæst for at informasjonen vert vidaresend til host-computeren[1] via PCI-bussen. Sjå figur 2.1.

Dataoverføringi fungerar på den måten at data mottekne frå DDL vert mata inn i ein 64 bitar breid og 256 bitar djup FIFO på HLT-RORC. Når datamengdi i FIFO'en overstig ei fastsett grensa, kjem DMA-kontrolleren på HLT-RORC til å initiera vidaresending av data på PCI-bussen. Soleis syter DMA-kontrolleren for at mottekne data vert overførde til fastsette minnelokasjonar på mottakarsida.

[1] Frontend-prosessoren, FEP



Figur 2.1: Oversyn over arkitekturen åt HLT-frontendprosessoren [11].

2.1.2 Utvidingar og forbetringar

Interrupt-handtering

HLT-RORC har eigne register som inneheld informasjon om baseadresse og storleik på mottakarbufferet (eventbufferet). Dette bufferet er sett opp som eit ringbuffer, og HLT-RORC må difor implementera ein mekanisme som syter for at HLT-RORC ikkje overskriv data før dei er lesne av host-computeren. I det opphavlege HLT-RORC-designet vart denne mekanisme implementert ved hjelp av peikarar. Lokalt på HLT-RORC sin DMA-kontroller finst det register som inneheld informasjon om basesegment og offset som HLT-RORC skal skriva til. Host-computeren har på si sida eit eige lesepeikarregister som fortel kor langt host-computeren er komen med å lesa data ut or ringbufferet. Dette lesepeikarregisteret er tilgjengeleg for HLT-RORC via PCI-bussen, og med det opphavlege designet var HLT-RORC utforma slik at HLT-RORC sjølv laut lesa ut lesepeikarverdien frå ein minnelokasjon på mottakarsida. Når HLT-RORC hadde sendt so mykje data at skrivepeikarverdien var åt å gå forbi lesepeikarverdien, skulle HLT-RORC initiera ein ny transaksjon på PCI-bussen der HLT-RORC las frå lesepeikarregistret. Dersom lesepeikaren ikkje hadde vorte oppdatert, heldt HLT-RORC fram med å lesa frå lesepeikarregistret heilt til lesepeikarregistret hadde fått ein verdi som gjorde det mogleg å halda fram å skriva til hostbufferet. Denne teknikken vert til vanleg kalla polling eller busy waiting.

Busy waiting er ein ressursøydande metode som skaper uturvande aktivitet på PCI-bussen. Det var difor ynskjeleg å få HLT-RORC til å nytta interrupts når kortet ville oppdatera lesepeikaren. Interrupts ville gjera det mogleg å berre beda om ny lesepeikar éin gong. Deretter kunne HLT-RORC venta til dess mottakarsystemet fann det for godt å gje kortet ein ny lesepeikarverdi.

På same måten trongst det ein ny mekanisme for å melda frå til prosessoren når HLT-RORC var ferdig med å overføra eventdata. I det opphavlege designet var det slik at HLT-RORC skreiv til minnet eit sokalla rapporteringsord med informasjon om lengdi på eventet for kvar gong eit heilt event hadde vorte overført. Her var det eventuelle drivarprogram som vart nøydde til å driva med polling i rapporteringslokasjonane for å finna ut om det hadde kome nye data eller ikkje. Det me burde ha var ein interrupt-mekanisme som kunne avbryta prosessoren kvar gong HLT-RORC var ferdig med å overføra eit rapporteringsord.

Pr 1. januar 2005 var det ikkje gjort noko for å byggja inn interrupt-handtering på HLT-RORC. Derimot hadde Artur Szostak gjort ein del eksperimentering med interrupt-signalisering og interrupt-handtering generelt. På maskinvaresida hadde Szostak konstruert ein IRQ-modul som kunne signalisera interrupts på PCI-bussen. På programvaresida hadde han konstruert ein loadable kernel module som kunne taka i mot interruptet og schedule ein signal handler i user space til å fylgja opp fyrespurnaden (sjå underkapittel 5.1 for meir informasjon om programvareutvikling i Linux).

Attåt interrupt-handtering hadde Szostak nytta seg av ein eigendefinert kjernemodul kalla irqsignal og ein anna kjernemodul kalla kalla PSI (PCI and Shared Memory Interface) til å kommunisera med HLT-RORC.

Eg kunne difor slå fast at mykje av arbeidet med å få HLT-RORC til å kommunisera med mottakarsystemet og til å takla interrupts var gjort. Det som stod att var å få sett det heile i hop til eit system. Fyrste del av arbeidet mitt vart difor å integrera arbeidet åt Szostak i lag med det opphavlege designet som var gjort på HLT-RORC. Dei konkrete designendringane som vart gjorde for å kunna implementera interrupt-styrd dataoverføring er detaljert omtala i underkapittel 4.1.

På programvaresida arbeidde eg med å konstruera eit demonstrasjonsprogram som kunne setja opp HLT-RORC til å generera hendingar og initiera eigne transaksjonar. Programmet skulle dessutan vera i stand til å svara på interrupt-fyrespurnadene frå HLT-RORC.

Utvida feilhandtering

Under arbeidet med å integrera interrupt-handtering hadde eg HLT-RORC ståande på ein 32-bits PCI-buss. Det vart difor i fyrste omgang gjort ein del debugging på HLT-RORC for å få designet til å køyra feilfritt på 32-bits databuss. Mellom anna vart designet forbetra til å handtera latency timer expirations og retries på PCI-bussen. Desse endringane er nærare omtala i underkapittel 4.3.

Overføring frå Altera-basert til Xilinx-basert teknologi

Den opphavlege utgåva av HLT-RORC var implementert på ein Altera-basert FPGA, nærare kartlagt APEX20KE400. Denne brikka var det meiningi å fasa ut or HLT og skifta ut med Xilinx Virtex-4. Saman med rettleidaren valde eg difor å frysa arbeidet med HLT-RORC på APEX20KE400. I staden ville eg arbeida vidare med å flytja HLT-RORC over på ny teknologi. Reint konkret innebar dette at me laut konstruera eit testkort som hadde kantkontakt for PCI og Xilinx Virtex-4 påmontert. Me laut dessutan syta for at PCI-kjernen Altera PCI MegaCore og

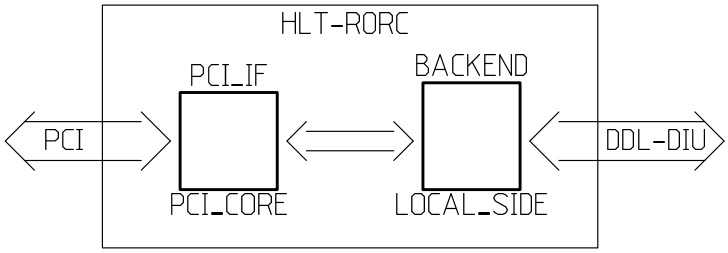
andre teknologispesifikke funksjonar vart bytte ut med tilsvarande løysingar frå Xilinx. Når det kom til PCI-grensesnitt, fall valet på Xilinx LogiCORE PCI v3.0.

Som synt i underkapittel 4.4 var det uturvande å gjera store endringar i funksjonaliteten eller arkitekturen for å få HLT-RORC til å fungera i Xilinx-teknologi. Krav til timing førde til at designet laut partisjonerast opp i to ulike klokkedomene og sume signal laut registrerast for å hindra at dei skapte vandemål med setuptider og holdtider i det dei kryssa grensone mellom klokkedomeni.

2.2 Firmware-oversyn

2.2.1 top_module.vhd

Figur 2.2 syner blokkskjemaet for firmware-modulen[2] top_module. Modulen top_module kapslar inn heile det hierarkiske VHDL-designet åt HLT-RORC. Dei to hovudmodulane i top_module er PCI-kjernen frå Altera (pci_core) og den lokale arkitekturen (local_side) som i den eine enden kommuniserar med PCI-kjernen og i hin enden tek i mot data frå DIU.



Figur 2.2: Blokkskjema for top_module.vhd.

2.2.2 local_side.vhd

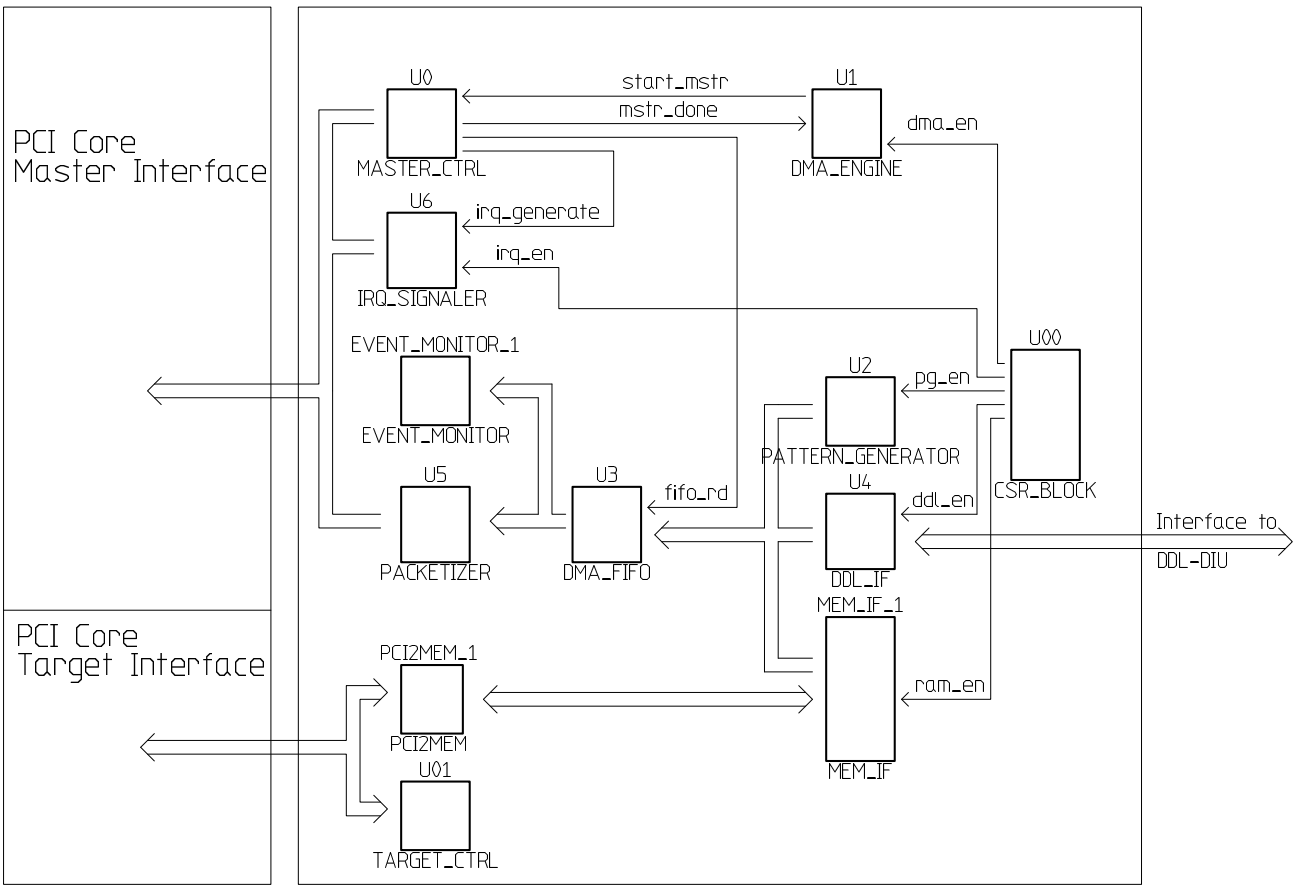
Figur 2.3 syner eit funksjonelt blokkskjema for modulen local_side. Modulen local_side inneheld desse undermodulane:

- **Master_Ctrl** styrer initiator-transaksjonane åt HLT-RORC. Desse gjeng i hovudsak ut på å senda mottekne eventdata og statusinformasjon til host-computeren.
- **Target_Ctrl** styrer target-transaksjonane åt HLT. Desse gjeng i hovudsak ut på å lesa og skriva til ulike register i HLT-RORC sitt lokale konfigurasjonsgrensesnitt. Dette konfigurasjonsgrensesnittet er ikkje synt i blokkskjemaet på figur 2.2, men det er implementert

[2]For programmerande krinsar skil ein mellom firmware og hardware. Firmware er den maskinvarespesifikasjonen som vert mappa på eksisterande hardware når krinsen vert programmert.

i alle dei ulike modulane på HLT-RORC som inneheld register som må vera tilgjengelege frå PCI-bussen.

- **CSR_Block** innheld register med kontroll- og statusinformasjon for HLT-RORC. Bitane i det globale kontroll- og statusregisteret i CSR vert mellom anna nytta til å enable DMA-overføringar, interrupt-handtering og mynstergenerering.
- **DMA_Engine** signaliserar til **Master_Ctrl** kva tid HLT-RORC skal initiera dataoverføringar på PCI-bussen.
- **IRQ_Signaler** vert nytta til å generera interrupts på PCI-bussen.
- **Pattern_Generator** vert nytta til å generera eventdata internt på brikka. Modulen er einast meint til testing og debugging.
- **DDL_IF** er grensesnitt mot DDL.
- **Mem_IF** er eit grensesnitt som kan nyttast til å lesa ut simulerte eventdata frå eit eksternt minne på HLT-RORC-kortet. Det er einast meint til testing og debugging.
- **DMA_FIFO** inneheld ein FIFO som vert fyld opp med mottekne data frå DDL_IF, MEM_IF eller Pattern_Generator. Når talet på lagra words i FIFOen overstig ei fastsett grense, gjev DMA_Engine signal om at Master_Ctrl lyt starta dataoverføring.
- **Packetizer** syter for at mottekne events vert kløyvde opp i pakkar med words som kan overførast på PCI-bussen i tur og orden.
- **PCI2MEM** er grensenitt mellom PCI-kjernen og det eksterne minnet på HLT-RORC-kortet.
- **Event_Monitor** held orden på kor mange events som er vortne overførde på PCI-bussen.

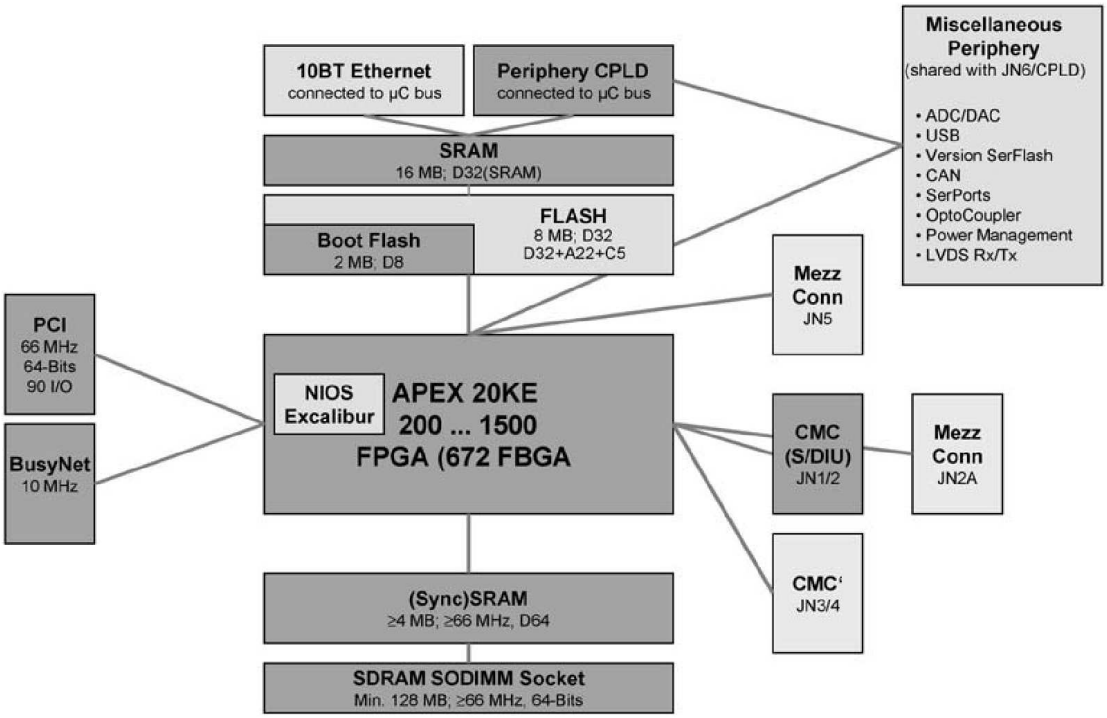


Figur 2.3: Blokkskjema for local_side.vhd.

2.3 Maskinvare-prototyping - PCI-kort med Xilinx Virtex-4

Dette underkapitlet gjeng stutt gjennom utformingi av eit PCI-testkort med Xilinx Virtex-4. Testkortet vart utvikla i samband med overflyttingi av HLT-RORC frå Altera-teknologi til Xilinx-teknologi.

Som nemnt tidlegare i kapitlet var den fyrste utgåva av HLT-RORC basert på FPGA-brikka APEX20KE400, komponentnummer APEX20KE400EFC-1X. Til denne krinsen hadde det vorte utvikla ein egen maskinvare-prototyp. Prototypen var realisert som eit 64-bits, 66 MHz PCI-kort. Eit funksjonelt blokkdiagram for prototypen er synt i figur 2.4 [24].

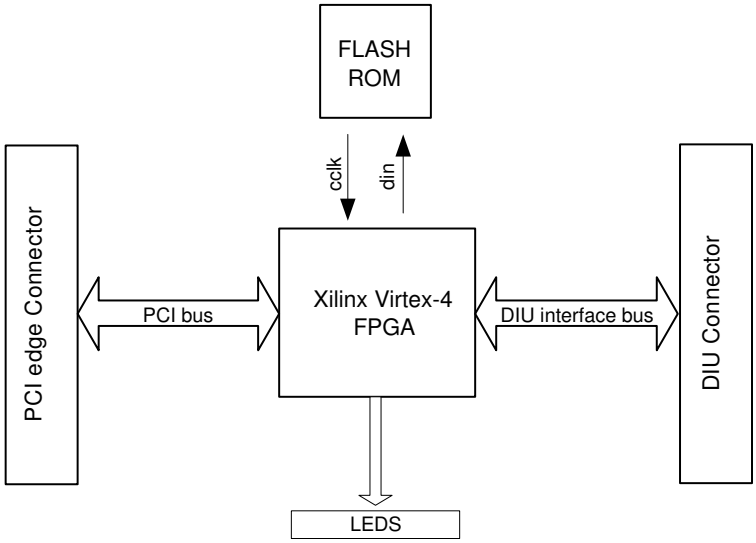


Figur 2.4: Funksjonelt blokkdiagram for ein Altera-basert maskinvareprototyp.

Då eg byrja på arbeidet med HLT-RORC i januar 2005, var det klårt at APEX20KE400 skulle bytast ut med Xilinx Virtex-4, men nokor ny maskinvare for å driva utvikling på Xilinx-plattform var enno ikkje tilgjengeleg. Truleg kom ho heller ikkje til å verta det på mange månader. Soleis vart eg nøydd til å utvikla mi eigi maskinvare. Eit PCI-testkort med Xilinx Virtex-4 og vart difor konstruert. PCI-testkortet inneheld det minimale talet på komponentar som trongst for å oppfylla dei funksjonelle kravi åt HLT-RORC og for å kunna driva testing og debugging av maskinvara på ein fullnøyande måte.

2.3.1 Funksjonalitet

Figur 2.5 syner eit funksjonelt blokkdiagram for PCI-testkortet.



Figur 2.5: Funksjonelt blokkdiagram PCI-testkortet.

Den viktigaste komponenten i testkortet er ein Xilinx Virtex-4 FPGA, komponentnummer XC4VLX25-FF668. FPGA-brikka har tilkoplingar til ein PCI-kantkontakt og ein DIU-konnektor. Soleis fær kortet tilgang på all den funksjonaliteten det treng til å implementera HLT-RORC. FPGA-krinsen kan programmerast anten beinveges gjennom JTAG-kjeden eller få programmet sitt lasta inn serielt frå to flashromar ved oppstart. Nokre ljosdiodar er lagde til for skuld debugging og overvakingsfyremål.

2.3.2 Xilinx Virtex-4

Partisjonering

IO-cellone åt Xilinx Virtex-4 er kløyvde opp i 11 ulike blokker, kalla bankar. Bank 0 inneheld konfigurasjonsgrensesnittet åt Virtex-4, inkludert JTAG. Dei 10 hine bankane disponerar me fritt [25]. Fyremålet med å kløyva krinsen opp i bankar er å gjeva høve til å samla logikk som høyrer naturleg i lag i eigne blokker på brikka, noko som vil gjeva betre yteevna.

Konfigurering

Virtex-4 er ein SRAM-basert FPGA (sjå til dømes [26]) som vert konfigurert på den måten at ein bitstraum med konfigurasjonsdata vert lasta inn i eit internt konfigurasjonsminne. Bitstraumen vert lasta inn i minnet gjennom serskilde konfigurasjonspinnar. Desse konfigurasjonspinnane tener som grensesnitt for ulike konfigurasjonsmodi:

- Master-serial og Slave-serial implementerar kvar sine seriegrensesnitt for konfigurering.

- Master-SelectMAP og Slave-SelectMAP implementerar kvar sine parallgrensesnitt for konfigurering.

Krinskonstruktøren vel sjølv konfigurasjonsgrensesnitt med di han hardwirar tre pinnar kalla M0, M1 og M2. Attåt dei fire ulike konfigurasjonsmodusane kan bitstraumen alltid lastast inn gjennom JTAG-grensesnittet [27].

Pinnar

Xilinx Virtex-4 har ulike slag pinnar. Nokre av pinnane er læste til serskilde funksjonar, til dømes power, jord og programmering. Andre kan konfigurerast fritt. Dei ulike bankane har eigne spenningsforsyningar til dei IO-cellone dei disponerar. For kvar bank gjeld det difor at me kan nytta eigne spenningsnivå til IO-pinnane.

Med unnatak for power-pinnane kan alle pinnane i bank 1 til 10 konfigurerast som vanlege IO-pinnar. Attåt vanleg IO har kvar av desse pinnane dedikerte omframtfunksjonar. Pinnane i bank 3 og 4 kan til dømes nyttast som globale klokkeinngangar. I andre bankar kan ein finna spenningsreferansepinnar og pinnar med serskilt låg kapasitans. Alle dei ulike bankane har dessutan sine eigne sett med lågkapasitans 'klokkekapable' pinnar som kan nyttast som regionale klokkor innanfor dei næraste klokkeregionane på brikka. Med unnatak for lågkapasitanspinnane er IO-pinnane ordna i par som kan nyttast til differensiell lågspenningssignalisering.

2.3.3 Design-prosessen

PCI-testkortet vart utvikla ved hjelp av Mentor-verktypakken VeriBest. VeriBest inneheld mellom anna programmet Design Capture til skjemateikning og programmet Expedition PCB til utlegg av mynsterkort.

Nye komponentar kan konstruerast frå programmet Library Manager. Library Manager har eigne redigeringsverkty for ei rad ulike einingar, som til dømes skjemasymbol, borehol, pads og fotavtrykk. For å få lagt ein ny komponent inn i databasen, lyt det definerast fotavtrykk til implementering på layout-nivå og symbol til innsetjing på skjemanivå. Deretter lyt det definerast ein eigen 'Part' som knyter i hop fotavtrykk og symbol. Alle dei ulike einingane vert lagra i eit sams sentralt bibliotek.

VeriBest nyttar ein sentralisert design-database (kalla Common Database, CDB) til å halda orden på dei ulike elementi som høyrer med i designprosessen. So lenge dei ulike verktyi ser den same databasen, kan ny informasjon annoterast att og fram mellom verktyi. Designendringar som vert gjorde på layoutnivå kan til dømes annoterast attende til skjemateikningi, og omvendt. Dette gjev ein svært fleksibel designprosess.

Designhierarki på skjemanivå

PCI-testkortet vart utforma hierarkisk med ulike blokker for avkopling, DIU-interfacing, PCI-interfacing, JTAG-interfacing og spenningsregulering. Sidan det ikkje var so mykje kontakt mellom desse ulike blokkene, heldt det å definera jord, power, klokke og eit par andre signal som

globale. Det vart vidare teikna eitt symbol for kvar komponent som var med. Unnataket var FPGA-krinsen som fekk eitt symbol for kvar bank.

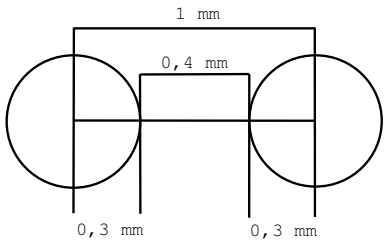
Plassering og ruting

Pakkar som deler tidskritiske signal bør plasserast so nær einannan som råd er. Det er viktig å orientera pakkane slik at det vert minst mogleg leidningskryssing. I Expedition PCB vert leidningar som enno ikkje er ruta teikna med 'luftlinor' og desse 'luftlinone' gjev oss ofte ein god indikasjon på kva som er fornuftig orientering. Dersom det er høve til å byta om pinnar på pakkane, kan ogso 'swapping'[3] av pinnar vera ein god strategi for å minimera leidningskryssing.

I vårt tilfelle er PCI-kjernen som skal implementerast på FPGA-brikka læst fast til serskilde pinnar i bank 6 og bank 10. Dette har vorte gjort for å oppfylla dei strenge tidskravi som gjeld for PCI-standarden. Problemet med denne løysingi er pinnane kjem so tett at me lyt opp med via mellom kvar einaste pinne i dei to bankane det gjeld. Dette fører til at me lyt nytta fleire lag enn det me elles trong å gjera for å kunna fullføra rutingi av alle pinnane.

Leidarbreidder ytterlag

Di større dimensjonar me kan få på minimum width og spacing, di billegare vil det verta å produsera kortet. I vårt tilfelle er det rutingi under FPGA-pakken, som er ein BGA med 668 pinnar, som vil avgjera kva som vert dei minste dimensjonane.



Figur 2.6: Dimensjonering for minimum width og spacing. Sirklane illustrerar loddeballar.

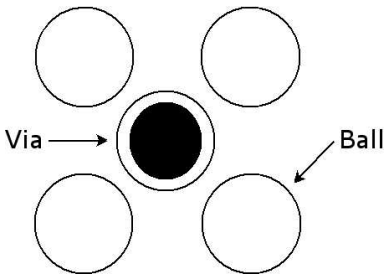
BGA-pakken har pin-to-pin-fråstand[4] på 1 mm. Diameteren på ballane er 0,6mm. Edge-to-edge-fråstandet[5] vert då 0,4 mm. Sjå figur 2.6.
Me vil no finna ut kor store dimensjonar me kan ha på leidningar som gjeng inn mellom loddeballane. Dersom me set lik verdi for width og spacing, kan me dela edge-to-egde-fråstandet mellom loddeballane på tre. Dette gjev oss 0,4 mm / 3 = 0,13mm width/spacing. Minimum width og minimum spacing kjem soleis på 0,13 mm.

[3] Byting av pinnar mellom pads på same pakken.
[4] Pin-to-pin-fråstandet er lengdi det er millom sentri åt dei næraste loddeballane.
[5] Edge-to-edge-fråstandet er lengdi mellom kantane åt dei næraste loddeballane.

Dimensjonering av viahol

For å klara å fullføra rutingi lyt me koma oss opp med via mellom loddeballane åt brikka. Dersom viaholi vert plasserte midt mellom loddeballane, kjem kvart viahol til å verta omgjeve med fire loddeballar, ein i kvart hyrne. Sjå figur 2.7.

Om me tenkjer oss at tre loddeballar danar ein rettvinkla trekant, kan me nytta Pythagorassetningi til å finna det diagonale fråstandet mellom sentri åt loddeballane. Her fær me to katetar med lengder lik pin-to-pin-fråstandet på 1 mm, og dette gjev oss ein hypotenus lik $\sqrt{2}$ mm. Det diagonale pin-to-pin-fråstandet mellom loddeballane vert soleis lik $\sqrt{2}$ mm. Dreg me i frå at kvar loddeball har radius på 0,3 mm, vert det diagonale edge-to-edge-fråstandet lik ($\sqrt{2}$ mm - 0,3*2 mm) = 0,81 mm. Fråstandet mellom to loddeballar lyt difor reknast likt 0,81 mm.



Figur 2.7: Viahol mellom loddeballar.

Me har frå før av definert minimum clearance lik 0,13 mm. Dersom me skal setja inn ein viapad mellom to loddeballar, lyt me soleis setja av 0,13 mm isolasjonsavstand på kvar side. Den maksimale diameteren på viapaden vert soleis 0,81 mm -0,13*2 mm = 0,55 mm. Elprint hadde minstekrav til annular[6] på 0,30 mm. Dette gjev oss ein boreholdiameter på (0,55 mm - 0,30 mm) = 0,25 mm. Sjå figur 2.8.

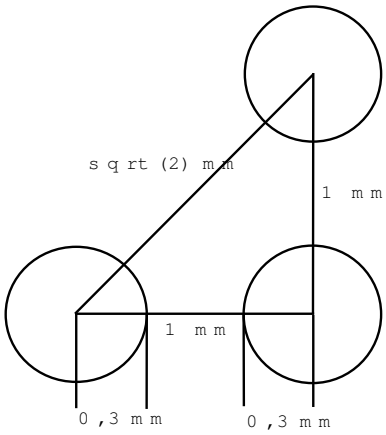
Leidarbreidder på innerlag

På innerlagi treng me ikkje å taka omsyn til paddane. Her er det fråstandet mellom viaholi som avgjer width og spacing. Om me gjeng ut frå at me lyt gå opp med via mellom kvar pad, vert fråstandet mellom viaholi det same som fråstandet mellom loddeballane, dvs eit fråstand via-to-via[7] på 1 mm. Sidan viapaddane har diameter 0,55 mm, kjem edge-to-egde-fråstandet mellom viapaddane til å koma på 1 mm - 0,55 mm = 0,45 mm. Dersom me skal gå mellom viapaddane med leidningar, gjev dette oss minimum width og clearance på 0,45 mm / 3 = 0,15 mm. Minimum width og spacing på innerlagi kan soleis justerast opp til 0,15 mm.

Det siste reknestykket var diverre ikkje teke omsyn til då testkortet vart produsert, men for eit åttelagskort som vårt hadde det gjeva ein monaleg stor reduksjon i pris om innerlagi kunne leggjast ut med mindre presisjon.

[6] Diameter for viapad minus diameter for viahol.

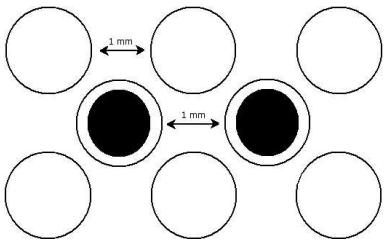
[7] Via-to-via-fråstandet er fråstandet mellom sentri åt dei næraste viaholi.



Figur 2.8: Dimensjonering for minimum viahol.

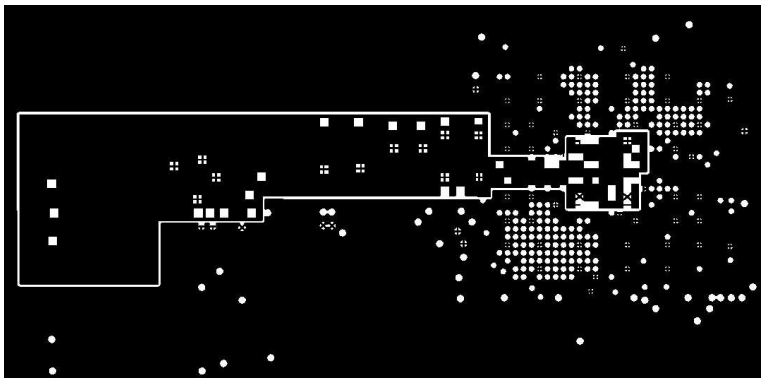
Nettklassar og rule areas
Dei ulike netti kan kløyvast opp i nettklassar der kvar nettklasse har sine eigne reglar for width og spacing. Ein kan dessutan teikna opp ulike rule areas i layouten der nettklassane har eit eige sett med reglar for kvart rule area. Powernett og klokke er døme på nett som har eigne krav til width og spacing og som det difor høver godt å leggja i ein eigen klasse.

Plan
Det lyt alltid nyttast jordplan og powerplan når ein har med høgfrekvenskrinsar å gjera. Powerplan og jordplan gjev god skjerming og låg impedans for straumane. I vårt tilfelle har me ein FPGA som treng tre ulike forsyningsspenningar, VCCINT (til kjernen), VCCO (til IO) og VCCAUX (til m.a. serie-interface). Det er VCCINT og VCCO som er dei mest kritiske. Me vel difor å dedikera eit eige powerlag for VCCO og VCCINT. VCCAUX vert ruta for hand med tjukke powerleidningar. I powerlaget teiknar me eigne shapes for VCCINT slik at me kjem til på FPGA-krinsen med powertilkoplingar midt i FPGA-krinsen. Resten av powerlaget vert overlate til VCCO. Expedition PCB har sin eigen planprosessor som kan teikna opp jordplan og powerplan i samsvar med desse retningslinone etter at rutingi er fullført. Sjå figur 2.10.



Figur 2.9: Fråstand mellom viahol.

Jordplan og powerplan bør plasserast nærast mogleg einannan. Di nærare plani kjem, di min-



Figur 2.10: Plane shapes i powerplanet. VCCINT vert omgjeve av VCCO.

dre vert straumsløyfone og di mindre induktans vil me få i jord- og powerplan. Multilagskort vert laga på den måten at ein legg ut tolagskort som ein laminerar i hop med isolerande lag i mellom (sjå til dømes [28]). Dei to plani bør difor plasserast slik at dei kjem nærast mogleg einannan.

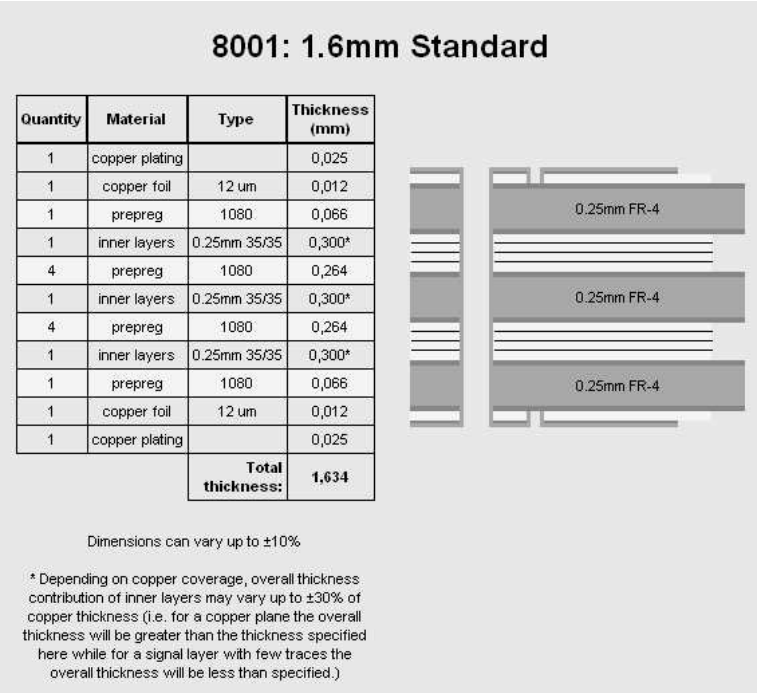
Kortet me skal laga er meint å vera av vanleg FR-4-materiale. Sidan det er eit PCI-instikkskort, lyt ein nytta ei standardtjukt på 1,6 mm for å koma ned i PCI-sokkelen. Frå programmet Macaos (sjå seksjonen om produksjonsfiler) kan me lesa ut at dette gjev oss tolagskort med tjukter på 0,3 mm. Av desse 0,3 millimetrane utgjer FR-4-substratet 0,25 mm, men med variasjonar heilt opp til 30 prosent av tjukti på koparlagi. Samstundes har lamineringi, som vert kalla prepreg, ei gjennomsnittleg tjukt på 0,264 millimeter. Soleis er det praktisk tala det same om jord- og powerplani vert skilde med substrat eller laminat. Sjå figur 2.11. Eg valde til slutt å nytta innerlag 4 og innerlag 5 som er skilde med laminering.

Avkoplingar

Det er ogso viktig å syta for rikesleg med avkoplingskondensatorar åt alle krinsar som svitsjar med snøgge stigetider. Desse må plasserast so nært som råd til dei pakkane det gjeld. Avkoplingskondensatorane har to funksjonar. Det fyrste er at dei vert nytta til å svelja unna transientar i forsyningsspenningi slik at desse ikkje uroar dei hine krinsane. Det andre er at avkoplingane fungerar som energilager. Om ein krins svitsjar med snøgge stigetider, treng han snøgg tilførsle av ladningar. Induktansar i tilførselslinone kan då føra til spenningsfall i forsyningsspenningi. Her vil avkoplingskondensatoren hjelpa med å avlasta straumforsyningi.

Ein verkeleg kondensator er ikkje berre kapasitiv. Ein seriespole og ein seriemotstand vil i praksis verta hekta på den ideelle kondensatoren. Dette vil gjeva kondensatoren ein resonansfrekvens der impedansen ligg på eit minimum, medan han i andre frekvensvald vil ha høg impedans. Det trengst difor ulike kapasitansverde dersom me skal få ein jamnt låg impedansdistribusjon over heile det aktuelle frekvensvaldet. Ein brukande tommelfingerregel er å nytta éin eller fleire kondensatorar for kvar dekade av eit gjeve kapasitansintervall. Ein relativt flat impedansdistribusjon kan verta oppnådd når talet på kondensatorar vert dobla for kvar dekade som kapasitansverdien vert redusert [29].

I praksis fører ofte usymmetrisk plassering av kondensatorar og tilleidningar til at kon-



8001: 1.6mm Standard

Quantity	Material	Type	Thickness (mm)
1	copper plating		0,025
1	copper foil	12 um	0,012
1	prepreg	1080	0,066
1	inner layers	0.25mm 35/35	0,300*
4	prepreg	1080	0,264
1	inner layers	0.25mm 35/35	0,300*
4	prepreg	1080	0,264
1	inner layers	0.25mm 35/35	0,300*
1	prepreg	1080	0,066
1	copper foil	12 um	0,012
1	copper plating		0,025
		Total thickness:	1,634

Dimensions can vary up to ±10%

* Depending on copper coverage, overall thickness contribution of inner layers may vary up to ±30% of copper thickness (i.e. for a copper plane the overall thickness will be greater than the thickness specified here while for a signal layer with few traces the overall thickness will be less than specified.)

Figur 2.11: Skjermbilete frå Macaos.

densatorar med nominelt lik kapasitans likevel vil gjeva ulike resonansfrekvensar når dei vert plasserte i nettverket. Soleis har det ogso vore vanleg å nytta like kapasitansverdiar til avkopling. Kondensatorverdiar på 0,1 uF og 0,01 uF har vore greide tommelfingerreglar. Sidan parallellkopling av spolar gjev redusert induktans og parallellkopling av kondensatorar gjev auka kapasitans, kan me ogso her slå fast at di fleire kondensatorar me parallellkoplar, di betre vert impedanskarakteristikkane (sjå til dømes [30]).

Autorouting

Expedition PCB er utstyrd med ein eigen autorouter. Rutingi kan i ein viss mon styrast ved å justera på ulike parameter som til dømes viakostnad og lengdeavgrensingar, men ho er i mange tilfelle ikkje god nok. Det er serskilt klokke og power som ofte må rutast for hand fyre me kan sleppa autorouteren til på dei hine netti. Powerpinnane på regulatorane bør til dømes ha flest mogleg via opp til powerplanet slik at impedansen mellom regulatorutgangen og powerplanet vert minst mogleg. På PCI-kantkontakten gjeld det å syta for at alle jord- og powertilkoplingar gjeng beint ut frå kontakten og inn på ein via med tilkopling til eit plan. Her har autorouteren ein tendens til å stengja for andre leidningar som skal ut frå konnektoren.

Avkoplingkondensatorane bør ogso rutast for hand. Her er det viktig at avkoplingskondensatorane har tilkopling til viahol i kvar ende slik at ein sikrar seg stuttast mogleg veg frå pad til powerplan og jordplan. Di lengre tilleidningar, di større induktans i serie med kondensatoren, og det vil me ikkje ha.

Linor som fører høgfrekvente signal bør ikkje brytast tvert med 90 graders vinkel med di dette kan føra til refleksjonar. Dei bør heller rundast av med til dømes 45 graders vinkel. Autoruteren har ein eigen opsjon for dette. Linor med snøgge stigetider i høve til linjelengdi bør dessutan terminerast for å sleppa unna transmisjonslineeffektar.

Etter at me er ferdige å ruta dei kritiske netti for hand, kan me sleppa til autoruteren. Her kan det vera lurt å velja ut nett som høyrer naturleg i lag og autoruta desse for seg. Berre det å autoruta ulike nett i rekkjefylgja gjev betre ruting med di det då er lettare for routeren å 'halda tunga rett i munnen'.

Design Rule Check

Etter at me er ferdige med plassering, ruting og planprosessering, bør me køyra ein Design Rules Check (DRC). Expedition PCB har ein funksjon kalla Batch DRC som etter tur køyrer ei rad ulike sjekkar for å sjå etter uheldige strukturar og brot på dei reglane me sjølve har sett opp, til dømes kravi til width og spacing.

Produksjonsfiler

Når designet er ferdig lagt ut, må det sendast til produksjon i eit format som produsenten er i stand til å lesa. Produsenten må ha informasjon om kvart lag og kor store hol som må borast.

Dei ulike lagi kan definerast i det sokalla Gerberformatet som gjev oss vektoriserte bilete av kvart lag. Gerberformatet er eit subsett av formati EIA RS-274D eller RS-274X. Holdiametrar kan definerast i ei NC-drillfil som svarar til EIA RS-274D-standarden.

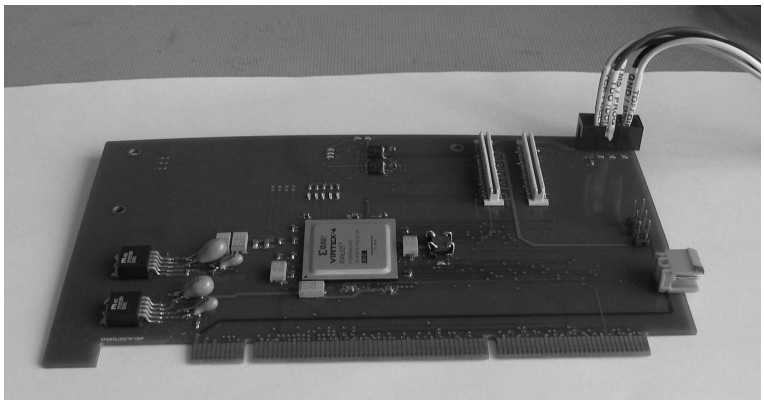
Hjå utleggsfirmaet Elprint vil dei helst ha utlegget tilsendt i det sokalla .mos-formatet. Desse filene kan genererast i programmet Macaos som fritt kan hentast ned frå Elprint sine heimesidor. Macaos-programmet er i stand til å importera dei fleste typane utleggsfiler som finst på marknaden, mellom desse finn me Gerber og NC Drill. I Macaos-programmet fær ein ogso høve til å definera val av substratmateriale, laminering, gullfingrar og so bortetter. På denne måten gjev .mos-formatet oss ein fullstendig mynsterkortspesifikasjon som kan sendast bort og produserast.

2.3.4 Programmering

Det ferdige krinskortet er utstyrt både med ein Virtex-krins og med to flashromar. Flashromane er av typen XCF04S og inneheld 4 megabyte kvar, i alt 8 megabyte. Fyremunen med å nytta to flashromar på 4 MB framfor éin flashrom på 8 MB var at sistnemnde trong si eigi forsyningsspenning på 1,8V medan dei mindre flashromane kunne klara seg med 3,3V frå powerplanet [31]. Både Virtex-krinsen og flashromane kan programmerast gjennom JTAG-kjeda. Omframt denne funksjonaliteten er Virtex-krinsen konfigurert til å lasta inn styreprogrammet frå dei to flashromane i det han startar opp. Kommunikasjonen med flashromane gjeng fyre seg over eit konfigurasjonsgrensesnitt av typen Master-serial.

I sekundi etter at forsyningsspenningi vert slegi på, ligg det totale straumforbruket åt krinskortet på om lag 0,6 A. Når Virtex-krinsen har lasta inn styreprogrammet frå flashromen, gjeng straumforbruket endå meir opp. Om ein derimot freistar programmera Virtex-krinsen direkte via

JTAG-grensesnittet, kan det henda at straumforbruket gjeng ned. Eit lite kombinatorisk dummy-design utan klokke kan faktisk halvera straumforbruket. Dette tyder på at ymse slag internt klokka konfigurasjonslogikk vert slege av etter at krinsen er vorten programmert gjennom JTAG-grensesnittet. Ein kan ogso tenkja seg at det designet som vert lasta inn frå PROM-filene inneheld ekstra logikk som held eigne porsjonar på brikka i aktivitet.



Figur 2.12: Fotografi av det ferdig monterte krinskortet.

Kapittel 3

PCI-protokollen - spesifikasjon og implementasjon

3.1 PCI-standarden

Dette underkapitlet gjev ein stutt introduksjon til PCI-standarden. Informasjonen er hovudsakleg henta frå den offisielle PCI-spesifikasjonen [5].

3.1.1 Eigenskapar og fyremuner

PCI Local Bus vart spesifisert for å etablera ein industristandard lokalbuss med høg yteevna, låge kostnader og høg fleksibilitet. Siste revisjon av standarden er PCI Rev. 3.0. Den siste revisjonen av standarden er spesifisert både for 32-bits og 64-bits databuss, for 33MHz og 66MHz arbeidsfrekvens og for 5V og 3,3V IO-spenning. PCI-bussen er ein typisk bakplanbuss der dei ulike PCI-devicane vert sette ned i eigne slots som er laga til på bakplanet åt datamaskinen.

Kontrollsignali på PCI-bussen er aktivt låge, noko som vert markert med at namni på kontrollsignali vert avslutta med 'N' eller '#'[1]. Når bussen er i ro, skal alle signal liggja høgt.

Ein PCI-transaksjon vert alltid initiert av ein master og må rettast mot eit target. Det er fullt høve til å implementera både master-funksjonalitet og target-funksjonalitet på den same PCI-devicen. Ein sentral buss-arbiter koordinerar busstilgangen slik at det ikkje vert konflikt mellom to eller fleire initiatorar. Kva slags prioritetsalgoritme som skal implementerast i buss-arbiteren stend krinskonstruktøren fri til å velja sjølv.

Kvar PCI-slot har sitt eige par med REQN- og GNTN-linor som gjeng inn til buss-arbiteren. Initiatorane bed om kontroll med bussen med di dei slær på[2] REQN-signalet. Arbiteren nyttar signalet GNTN til å signalisera at masteren fær lov å taka over bussen. Arbitrering er 'gøymt' for PCI-bussen på den måten at mekanismen ikkje konsumerar klokkeperiodar på bussen. Den

[1] # vert nytta i den offisielle spesifikasjonen og er soleis mest korrekt, men 'N' vert ofte nytta, mellom anna av Altera. Eg har sjølv valt å nytta 'N'.

[2] Eg har valt å nytta den norske ordleidingi 'slå på' for den engelske ordleidingi 'assert'. Tilsvarande skriv eg konsekvent 'slå av' der det på engelsk heiter 'deassert'.

ventande initiatoren kan taka i mot GNTN-signal parallellt med andre busstransaksjonar og soleis overtaka bussen med det same desse transaksjonane er avslutta [32].

3.1.2 Adresseområde

Både CPU'en og PCI-devicane må ha tilgang til minnelokasjonar som er delte mellom dei. Dette minnet vert nytta til å kontrollera PCI-devicane og til å overføra informasjon mellom dei. Som eit minimum inneheld det delte minnet til vanleg kontroll- og statusregister for devicen. Desse registri kan nyttast til å styra devicen og lesa ut statusinformasjon.

PCI tilbyd tre slag address space; PCI IO, PCI Memory og PCI Configuration Space([33]).

Configuration Address Space

PCI-spesifikasjonen legg til rettes for hundre prosent programvaredrivi initialisering og konfigurering gjennom eit separat konfigurasjonsvald, kalla Configuration Address Space. PCI-devices lyt setja av 256 bytes med ulike konfigurasjonsregister åt dette valdet. Ei oversikt over registri i dette valdet, kalla Configuration header, finst i figur 3.1. Kvar PCI-device har sin eigen IDSEL-pinne (Initialization Device Select) som må vera slegen på for at ein skal kunna få tilgang til Configuration Address Space.

Dei viktigaste registri i Configuration Address Space er:

Vendor ID Register
Eit 16-bits read-only register som som identifiserar device-produsenten.

Device ID Register
Eit 16-bits read-only register som identifiserar devicen.

Command Register
Eit 16-bits read-write register som kontrollerar kva høve devicen har til å utføra eller svara på busstransaksjonar.

Status Register
Eit 16-bits register som gjev statusinformasjon om hendingar på bussen. Bitane i registret kan clearast, men ikkje setjast.

Base Address Registers
Ein PCI-device kan ha opp til seks baseadresseregister, i stuttform kalla for BAR. Kvart baseadresseregister (BARn) har identisk utforming. Bit 0 for kvar BAR er read only, og vert nytta til å indikera om det reserverte adressevaldet er memory eller I/O. BARs som mappar til memory space må hardwira[3] bit 0 til 0, og BARs som mappar til I/O må leggja bit 0 til 1. I ein Memory-

[3]Som nemningi indikerer er ein bit som er hardwired fysisk kopla til forsyningsspenning eller jord og på den måten tilordna ein statisk verdi.

BAR indikerar bit 1 og 2 korleis adresseområdet skal memory-mappast (ein kan til dømes implementera ein 64-bits BAR ved å kombinera BAR0 og BAR1). Bit 3 fortel for ein memory-bar om minnet kan prefetchast eller ikkje [34].
Når PCI-devicen vert sett opp, lyt den opphavlege informasjonen i BARn lesast ut og deretter overskrivast med baseadressa for det minneområdet som adresseområdet skal mappast inn i.

Memory Address Space

Memory address space skal ved oppstart mappast inn i main memory. System som treng memory spaces større enn 4GB må ha 64-bit adressering.

I/O Address Space

Denne typen address space har vore med i PCI-spesifikasjonen for å gje attoverkompatibilitet med den gamle x86-arkitekturen. PCI-SIG tilrår alle å nytta memory space i staden for I/O space. Ein device som nyttar memory space framfor I/O space kan nyttast i system som ikkje har støtte for I/O space.

3.1.3 Busstransaksjonar

Adressedekodingi på PCI-bussen er distribuert, dvs at ho vert gjord på kvar device. Sentralt trengst det soleis ikkje nokon logikk for dekoding eller device-utveljing utover det eine IDSEL-signalet som trengst til konfigurasjonstransaksjonar.

PCI har støtte for både single cycle transactions og burst transactions. Medan ein single cycle transaction berre har éin datafase pr transaksjon, gjev burst transactions høve til å lesa eller skriva skurdar av data i gongen. Som eit døme på signalgangen ved ein PCI-transaksjon kjem eg til å taka fyre meg ein 64-bits burst read transaction og ein 64-bits burst write transaction. Timing-diagrammi i denne seksjonen syner sambandet mellom dei viktigaste signali som er involverte i ein 64-bits transaksjon.

Burst read transaction

1. Adressefasen byrjar når PCI-masteren slær på FRAMEN og REQ64N samstundes som han driv AD og CBEN. AD lyt i adressefasen innehalda adressa masteren vil skriva til. CBEN fortel kva slags type transaksjon det er masteren vil utføra (i vårt tilfelle lesing som svarar til CBEN=6).

2. Turnaround-periodane[4] byrjar i det masteren tristatar[5] AD medan han ventar på at target skal svara. CBEN må til og med siste datafasen drivast med byte enables. Byte enables

[4]Turnaround-periodar vert nytta til å hindra at det vert konflikt når ein device sluttar å driva eit signal og ein annan device byrjar å driva det.

[5]Tristating vil seia at ein kan slå av ein utgang med di ein styrer kjeldeimpedansen åt utgangen. Når utgangen skal driva eit signal, lyt kjeldeimpedansen vera låg. Når utgangen skal slåast av, lyt kjeldeimpedansen vera stor. Tristate-logikk vert til vanleg implementert med passtransistorar.

Table 3–12. PCI Bus Configuration Registers

Address	Byte			
	3	2	1	0
0x00	Device ID		Vendor ID	
0x04	Status Register		Command Register	
0x08	Class Code			Revision ID
0x0C	BIST	Header Type	Latency Timer	Cache Line Size
0x10	Base Address Register 0			
0x14	Base Address Register 1			
0x18	Base Address Register 2			
0x1C	Base Address Register 3			
0x20	Base Address Register 4			
0x24	Base Address Register 5			
0x28	Card Bus CIS Pointer			
0x2C	Subsystem ID		Subsystem Vendor ID	
0x30	Expansion ROM Base Address Register			
0x34	Reserved			Capabilities Pointer
0x38	Reserved			
0x3C	Maximum Latency	Minimum Grant	Interrupt Pin	Interrupt Line

Figur 3.1: Studde konfigurasjonsregister i PCI Configuration Address Space [35].

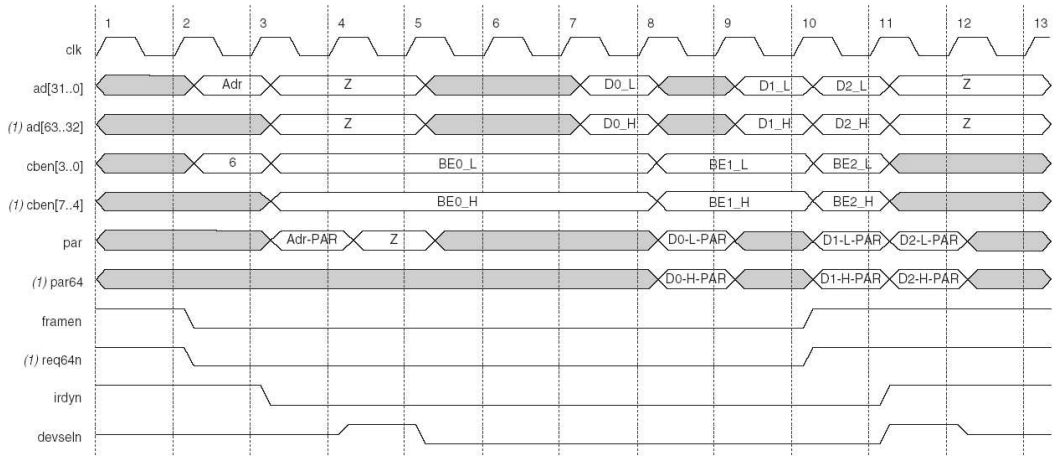
er til liks med andre kontrollsignal aktivt låge og dei fortel target-devicen kva for bytes på AD-bussen som må drivast med gyldige data. Samstundes som AD vert tristata, vert IRDYN slege på for å signalisera at PCI-masteren er reidug til å taka i mot data.

3. Om adressa som vart lagd ut på AD svarar til eitt av baseadresseregistri, svarar den korresponderande target-devicen med å slå på drivarane for signali AD[63..0], DEVSELN, TRDYN og STOPN. Drivarane for PAR og PAR64 vert slegne på i den påfylgjande klokkeperioden.

4. Target-devicen slær på DEVSELN og ACK64N for å indikera andsynes master-devicen at han godtek transaksjonen.

5. To eller fleire datafasar fylgjer. Under datafasen lyt target-devicen slå på TRDYN-signalet for å informera PCI-masteren om at AD vert drive med gyldige data. I siste datafasen lyt PCI-masteren slå av FRAMEN for å signalisera at transaksjonen skal avsluttast.

6. PCI-masteren driv signali DEVSELN, ACK64N, TRDYN og STOPN høgt i éin klokkeperiode fyre dei vert tristata.



Figur 3.2: 64 bit burst read transaction [35].

Burst write transaction

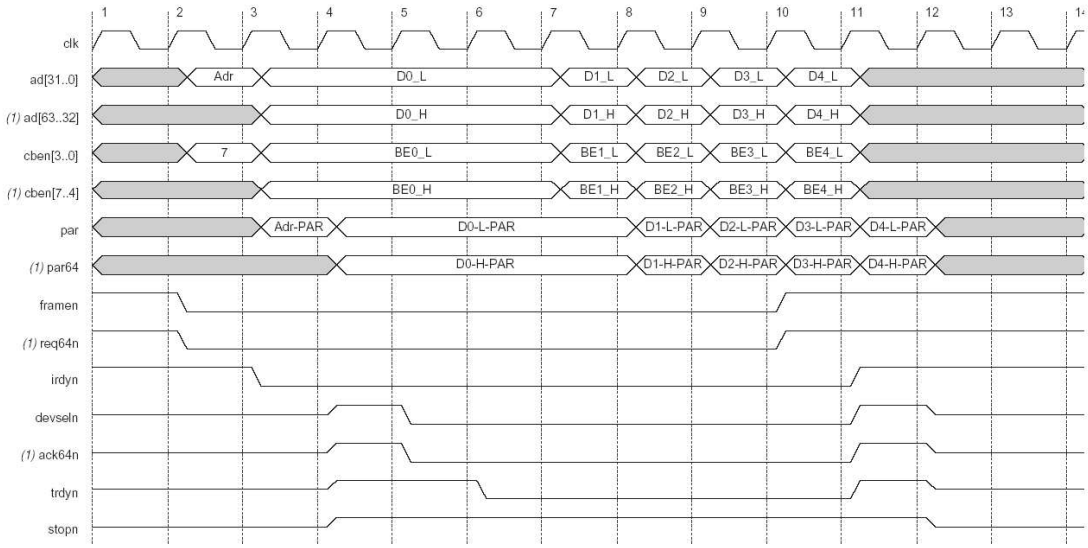
1. Adressefasen byrjar når PCI-masteren slær på FRAMEN og REQ64N og driv AD[63..0] og CBEN[3..0].

2. PCI-masteren held fram med å driva data og byte enables til AD[63..0] og CBEN[3...0] samstundes som han slær på IRDYN for å informera target-devicen om at det er gyldige data på bussen. Om adressa for transaksjonen svarar til ein BAR, lyt target-devicen svara med å slå på drivarane for AD[63..0], DEVSELN, ACK64N, TRDYN og STOPN. Drivarane for PAR og PAR64 vert slegne på i den påfylgjande klokkeperioden.

3. Target-devicen slær på DEVSELN og ACK64N for å indikera andsynes master-devicen at han godtek 64-bit-transaksjonen.

4. To eller fleire datafasar fylgjer. Under datafasen lyt target-devicen slå på TRDYN-signalet for å informera PCI-masteren om at AD vert drive med gyldige data. I siste datafasen lyt PCI-masteren slå av FRAMEN for å signalisera at transaksjonen skal avsluttast.

5. PCI-masteren driv signali DEVSELN, ACK64N, TRDYN og STOP høgt i éin klokkeperiode fyre dei vert tristata.



Figur 3.3: 64 bit burst write transaction [35].

3.1.4 Unormale avbrot

Master Abort

Når initiatoren har slege på FRAMEN for å starta ein ny transaksjon, kan det gå maksimalt 6 klokkeperiodar før target-devicen må ha svara med å slå på DEVSELN. Om ikkje må initiatoren gå ut frå at target-devicen ikkje er i stand til å svara eller at adressa var ugyldig og soleis avbryta transaksjonen.

Target Abort

Dersom target-devicen finn ut at han straks lyt avbryta transaksjonen og samstundes vil melda frå til initiatoren at han ikkje vil at transaksjonen skal takast opp att, kan han signalisera target abort. For å signalisera eit target abort, lyt han slå på STOPN samstundes med at han slær av TRDYN og DEVSELN.

Retry

Dersom ein target-device ikkje er reidug til å taka i mot data, kan han signalisera eit retry. Dette gjer han ved å slå på signalet STOPN, samstundes som han let vera å slå på TRDYN. Etter PCI-spesifikasjonen er initiatoren då nøydd til å freista taka opp att transaksjonen seinare.

Disconnect with data

Dersom ein target-device ynskjer å avbryta ein burst transfer eller kan henda ikkje er i stand til å taka i mot burst transfers i det heile, kan han signalisera eit disconnect with data. Dette gjer han ved å slå på signali STOPN og TRDYN på likt. Target-devicen signaliserar då at han ikkje ynskjer å taka i mot fleire data. Klokkeperioden der STOPN er slegen på skal etter PCI-spesifikasjonen vera den siste klokkeperioden då target-devicen tek i mot data.

Disconnect without data

Dersom ein target-device ynskjer å avbryta ein burst transfer, kan han signalisera eit disconnect without data. Dette gjer han ved å slå på signalet STOPN og slå av signalet TRDYN. Burst-transaksjonen lyt då stoggast av initiatoren. Når signalet TRDYN er slege av, treng ikkje target-devicen å lesa data frå PCI-bussen.

Latency timer expiration

Alle PCI-devices har sitt eige Latency Timer Register plassert i Configuration Address Space. Dette registeret inneheld informasjon om kor mange klokkeperiodar devicen kan halda på bussen. Dersom transaksjonen ikkje er avslutta fyre latency-timeren har talt ned til null, vert transaksjonen avbroten.

Wait states

Dersom ein target-device ikkje er reidug til å fullføra ein transaksjon med det same, kan han leggja inn sokalla wait states. Dette vil seia at target-devicen let vera å slå på TRDYN i éin eller fleire datafasar. Initiatoren er då nøydd til å driva bussen med same gyldige data heilt til target-devicen slær på TRDYN og transaksjonen kan halda fram. Ein target-device som etter å ha dekoda transaksjonen fyrst legg inn wait states og deretter handterar resten av transaksjonen utan fleire wait states, vert sagd å ha ‘inital latency’.

Ogso initiatoren kan leggja inn wait states. Dette gjer han med å slå av IRDYN-signalet. Dette er mindre vanleg sidan initiatoren som regel er reidug til å senda data når han bed om å få utføra ein transaksjon.

3.1.5 Bandbreidd og responstid

I PCI-system lyt det alltid vera ei veging mellom responstid og bandbreidd. Høg bandbreidd kan ein oppnå om ein gjev devices løyve til å nytta langvarande burst-transaksjonar. Snøgg responstid

kan ein oppnå om ein reduserar den maksimale bursttransaksjonslengdi.

Busskommandoane som vert nytta, lengdene på bursttransaksjonane, master og target latency og verdiane som master og target nyttar for latency-timeren er dei primære parametri som kan nyttast til å gjeva bussen dei rette eigenskapane.

3.2 Interrupt-handtering på PCI-bussen

Dette underkapitlet tek fyre seg interrupt-handtering på PCI-bussen. Informasjonen er hovudsakleg henta frå den offisielle PCI-spesifikasjonen [5].

3.2.1 Interrupt-signalisering

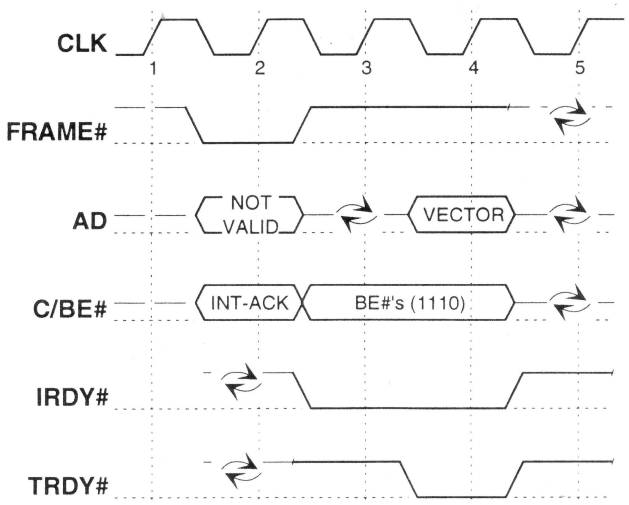
Interrupts på PCI er valfrie og definerte som nivåsensitive. Fysisk skal dei etter spesifikasjonen vera av typen open drain med di PCI gjev ulike devices høve til å dela den same interrupt-lina (shared interrupt). Interrupt-linone er soleis aktivt låge. Ein PCI-device kan ha opp til fire ulike interrupt-pinnar, INTAN, INTBN, INTCN og INTDN. Styringi av desse pinnane er asynkrone i høve til CLK. Ein 'Single function device' kjem til vanleg einast til å nytta seg av INTAN. Det er Interrupt Pin Register, lokalisert i Configuration Address Space, som definerar kva for linor devicen nyttar til interrupts. For kvar PCI-slot vert dei fire interrupt-pinnane ruta til interrupt-kontrolleren. Pinne A frå PCI-slot 4 kan til dømes verta ruta til pinne 6 på interrupt-kontrolleren, pinne B på PCI-slot 4 til pinne 7 og so bortetter.

Korleis interrupt-pinnane vert ruta til prosessoren er heilt systemavhengig. På Intel-baserte PC'ar er det BIOS'en som set opp devicen under oppstart. I system utan BIOS er det Linux-kjernen som køyrer setup-koden. Setup-koden må soleis ha kjennskap til rutingstopologien åt interrupt-kontrolleren. Setup-koden skriv det tilsvarande pinnenummeret åt interrupt-kontrolleren inn i devicen sitt Interrupt Line Register (lokalisert i Configuration Address Space). Når device-driveren køyrer, kan han lesa ut det tilordna IRQ-nummeret frå Interrupt Line Register og nytta dette nummeret til å be linux-kjernen om å få kontrollen med dette interruptet [36].

3.2.2 Interrupt acknowledge cycle

PCI-standarden har støtte for ein valfri Interrupt Acknowledge Cycle som synt på figur 3.4. Denne figuren illustrerar ein x86 Interrupt Acknowledge Cycle på PCI der ein einskild byte enable er slegen på og han er berre synt fram som eit døme (Til vanleg er det Byte Enable som fortel kva for bytes som er involverte i transaksjonen).

Under adressefasen inneheld ikkje AD[31::00] nokor gyldig adressa, men han må drivast med stabile data. PAR er gyldig, og pariteten kan kontrollerast. Ein transaksjon av typen Interrupt Acknowledge har ingen adresseringsmekanisme og er implisitt rett i mot interrupt-kontrolleren i systemet. På same måten som med vanlege transaksjonar, må kontrolleren svara med å slå på DEVSELN. Deretter lyt han returnera vektoren i same klokkeperioden som han slær på TRDYN.



Figur 3.4: Interrupt acknowledge cycle [5].

3.3 IP-kjernar for PCI

Dette underkapitlet gjev ein stutt gjennomgang av IP-kjernane Altera PCI MegaCore v2.1 og Xilinx LogiCORE PCI v3.0. Informasjonen er hovudsakleg henta frå PCI MegaCore Function User Guide [34] og LogiCORE PCI v3.0 User Guide [37].

Ombruk av design har vorte stadig meir vanleg innanfor maskinvareutvikling, på same måten som det lenge har vore innanfor programvareutvikling. Syntetiserande maskinvarespesifikasjonar er i dag tilgjengelege for ei rad ulike funksjonar. Desse funksjonane må som regel kjøpast eller lisensierast, og vert til vanleg kalla for Intellectual Property, nedstytta til IP.

I motsetnad til full-custom ASICs eller cellebaserte ASICs er FPGA-basert firmware læst til ein serskild arkitektur. Dei strukturane som vert realiserte kan ofte verta noko heilt anna enn dei ein freistar implisera i VHDL-koden, kort og godt av di sistnemnde strukturar ikkje er tilgjengelege på brikka. Det krinskonstruktøren vil ha er strukturar som er mest mogleg optimale med omsyn til snøggleik og arealforbruk. FPGA er soleis eit omkverve der bruk av ferdige funksjonar har mykje fyre seg. I staden for å konstruera generelle kodestrukturar der ein har liten kontroll med resultatet, kan ein nytta funksjonar som er ferdig optimaliserte og verifiserte av ein produsent som kjenner teknologien.

3.3.1 Altera PCI MegaCore

Arkitektur

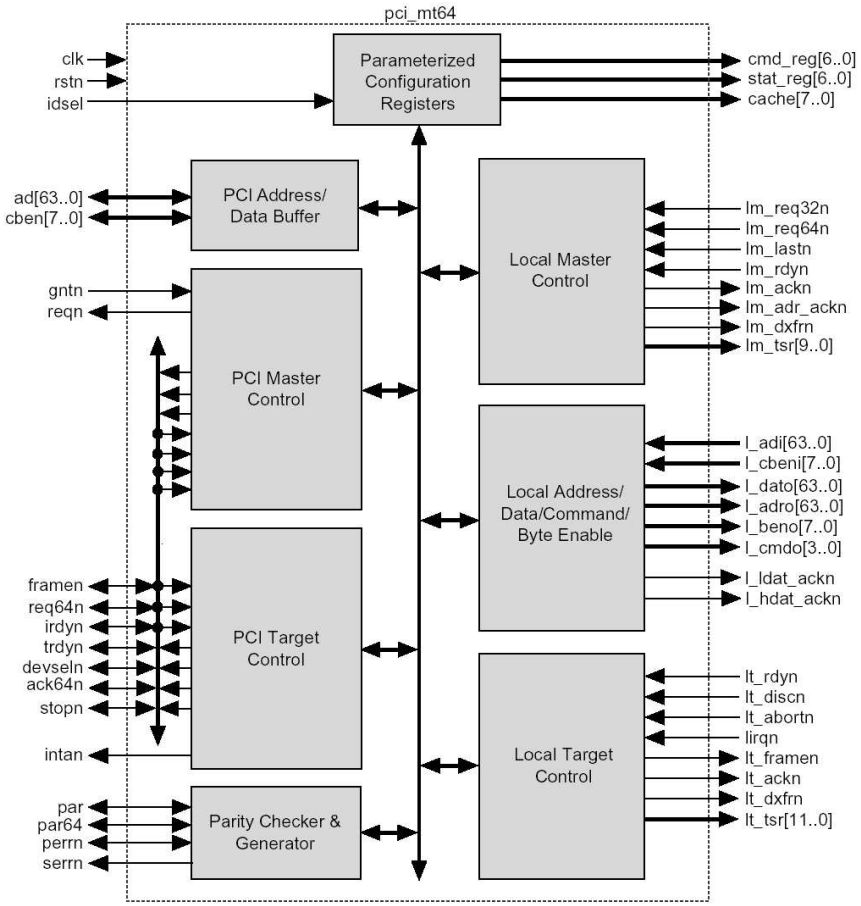
Altera PCI MegaCore er ein syntetitiserande IP-modul som tilbyd eit einfelt grensesnitt til Altera-baserte FPGA-krinsar som skal nytta seg av PCI-protokollen. Takk vere MegaCore-kjernen kan

Altera-utviklaren retta hovudfokus mot dei funksjonelle kravi åt applikasjonen framfor å nytta verdfull tid på testing og verifisering mot alle dei ulike logiske og elektriske kravi som PCI-standarden set.

MegaCore-funksjonen finst i fire ulike utgåvor, pci_mt64, pci_t64, pci_mt32 og pci_t32. Eg vil i dette kapittelet taka fyre meg funksjonen pci_mt64. Sistnemnde funksjon inneheld både ein initiator-modul og ein target-modul og kan køyra både 64-bits og 32-bits transaksjonar.

Figur 3.5 syner eit funksjonelt blokkdiagram for pci_mt64-funksjonen. Funksjonen har ein PCI-arkitektur som kan koplast til PCI-bussen og ein lokal arkitektur som kan knytast til ein eigendefinert brukarapplikasjon (som i vårt tilfelle er HLT-RORC). Signali med prefikset lm_ høyrer til det lokale grensesnittet åt initiator-modulen medan signali med prefikset lt_ høyrer til det lokale grensesnittet åt target-modulen. Signali med prefikset l_ er lokale signal som vert nytta av båe modulane.

I tillegg C er det gjort ein gjennomgang av korleis Altera PCI MegaCore implementerar dei viktigaste transaksjonane som vert nytta på HLT-RORC.



Figur 3.5: Funksjonelt blokkdiagram for mt64 [35].

Studde Altera-devices

Dei fyrste utgåvone av pci_mt64 hadde studnad for APEX, ACEX, Flex, Excalibur, Stratix og Cyclone [34]. Seinare utgåvor av PCI MegaCore har einast studnad for Stratix, Stratix II, Cyclone og Cyclone II [35]. Dei nyaste utgåvone av PCI MegaCore kunne soleis ikkje nyttast til maskinvareutvikling på HLT-RORC, med di dette systemet skulle implementerast på ein APEX20KE400. Til dette fyremålet laut ein stø seg på eit av dei gamle biblioteki, som til dømes versjon 2.1.1 som eg valde å nytta.

3.3.2 Xilinx LogiCORE PCI

Arkitektur

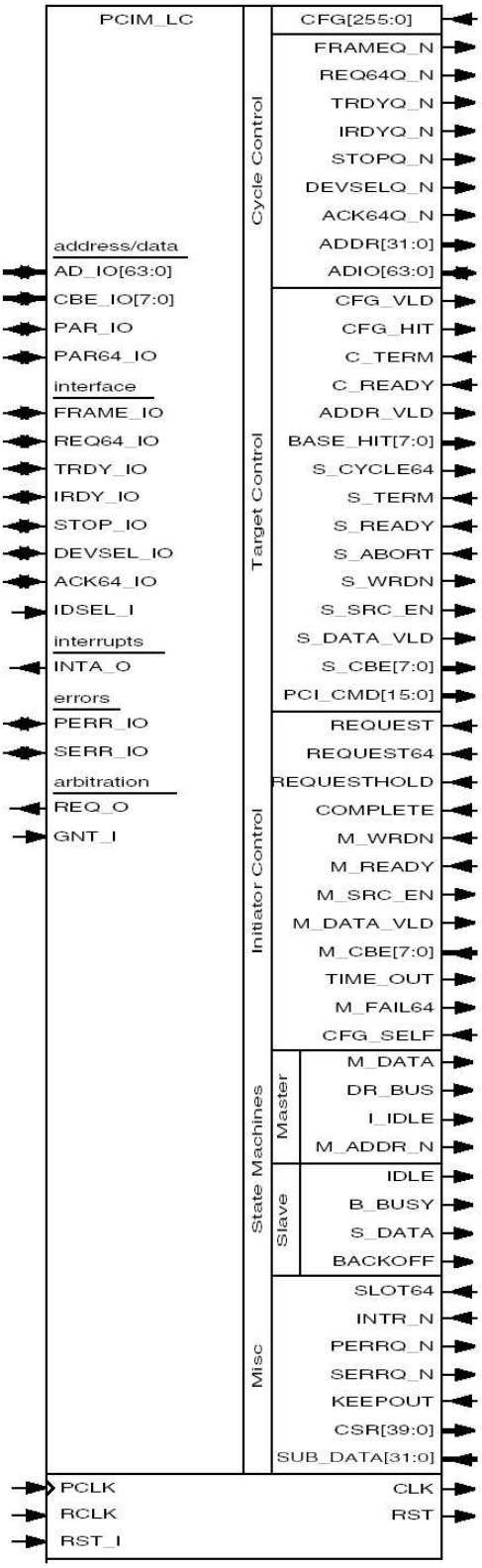
LogiCORE PCI er ein syntetitiserande IP-modul som tilbyd eit einfelt grensesnitt til Xilinx-krinsar som vil nytta seg av PCI-protokollen. LogiCORE PCI kan køyra både target-transaksjonar og initiator-transaksjonar og har studnad for både 32-bits og 64-bits transaksjonar.

Figur 3.6 syner eit funksjonelt blokkdiagram for LogiCORE PCI. Funksjonen har ein PCI-arkitektur som kan koplast til PCI-bussen og ein lokal arkitektur som kan knytast til den eigendefinerte logikken (som i vårt tilfelle vil vera HLT-RORC). Signali med prefikset M_ høyrer til det lokale master-grensesnittet medan signali med prefikset S_ høyrer til det lokale target-grensesnittet. Signal med andre prefiks er sams for båe funksjonane.

I tillegg C er det gjort ein gjennomgang av korleis Xilinx LogiCORE PCI implementerar dei viktigaste transaksjonane som vert nytta på HLT-RORC.

Studde Xilinx-devices

LogiCORE PCI har studnad for Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-4, Spartan-II, Spartan-II E og Spartan-3.



Figur 3.6: Funksjonelt blokkdiagram for LogiCORE PCI [37].

Kapittel 4

HLT-RORC - Implementation of interfaces

This chapter describes the changes that were done to the different interfaces of the HLT-RORC firmware.

4.1 Integration of interrupt handling on the HLT-RORC

4.1.1 Integrating IRQ Signaler firmware module from Artur Szostak

Artur Szostak had been doing some research on interrupt handling on the PCI bus. During this research he had created an IRQ_signaler module in VHDL [38]. The original module contained lots of unnecessary functionality, but it was possible to strip it down and modify it to suit my purposes. I also had to standardize the module with the same configuration interfaces that were used in the other modules of the HLT-RORC.

4.1.2 Modified IRQ Signaler module

The downstripped and modified IRQ_signaler module contains two registers that can be accessed from the the PCI bus:

- The pulse_length register specifies the maximum duration (in PCI clock cycles) that the interrupt line will be asserted. This register can both be read and written to from the PCI bus.
- The irq_status register contains status bits that will be set equal to one when the IRQ_signaler has generated an interrupt. This register is necessary to make the host computer handle shared interrupts. The status bits cannot be set from the PCI bus, but can only be cleared when read (automatic clearing saves time spent in the interrupt routine on the host computer).

The IRQ_signaler module is requested to generate an interrupt when the input signal irq_generate is asserted. The finite state machine of the IRQ_signaler will then go from an IDLE state to a

PULSE state, asserting the lirqn[1] signal until the interrupt status register has been read (by the interrupt service routine on the host computer)[2]. If the interrupt request is not serviced within a finite amount of clock cycles (specified by the pulse_length register), the IRQ_signaler module gives up, deasserting the lirqn signal.

4.1.3 Implementation of interrupt handling in the master_ctrl module

The original HLT-RORC design had to update its read pointer when the write pointer was 128 bits behind the read pointer (host buffer full condition). As long as this condition was true, the HLT-RORC would be polling the read pointer address on the host memory for new data. The purpose of the interrupt mechanism is the get rid of the polling. Instead of polling, the read pointer updating will be taken care of through interrupt signaling. When the HLT-RORC needs to update its read pointer, it requests the IRQ_signaler to generate an interrupt. After generating the interrupt, the master_ctrl unit will be waiting in the irq_wait state until the host buffer full condition is no longer true.

4.1.4 Implementation of interrupt handling on the host computer

Artur Szostak had created some test programs, using the PSI[3] kernel module and his self-defined irqsignal module to set up interrupt handlers and to read and write data to the local registers of the HLT-RORC.

When using hardware devices in a shared interrupt environment, you need to have some sort of mechanism for finding out which device is interrupting. First I tried to modify the interrupt service routine inside the irqsignal module to read the interrupt status register from the HLT-RORC to check if the HLT-RORC had actually interrupted. If one or more of the status bit were set, the interrupt service routine scheduled another routine (called the dispatcher routine), to service the interrupt. The interrupt dispatcher routine would in the next turn use POSIX signals to invoke a signal handler running in user space. The signal handler was expected to service the device.

After modifying the interrupt service routine inside the irqsignal module, I also had to write a separate user space program that was setting up the device and instantiating a signal handler that could be invoked from the irqsignal kernel module. Parts of this program were cut and paste from Szostak's own sample programs.

After discussing with Artur Szostak, we decided to revert the changes made to the irqsignal module. Instead of using the irqsignal module to pick up the correct interrupt line and reading the status register, the user program itself would use PSI to pick up the irq line from the Interrupt

[1] lirqn is the local interrupt line of the pci_mt64 core.

[2] In the original IRQ_Signaler module created by Szostak the interrupt line was asserted until the interrupt had been acknowledged by the bus controller. Since the interrupt acknowledge cycle is only optional, I had to replace this mechanism with something else for implementing a fully compliant device.

[3] The PSI module was used for mapping registers of the PCI Address space into ordinary data structures, thus making it possible for user space programs to access the PCI bus directly.

Line Register (located in the Configuration Address Space of the PCI Device) and the signal handler would perform the reading from the interrupt status register itself.

The new approuch would of course lead to more activity on the CPU, reducing the performance as the signal handler gets fired each time a device sharing the same irq number is interrupting. But as long as our software development is at the level of prototyping, this is not so important. The more important is that we still have a fully generic irqsignal module, giving full flexibility to do all the device specific transactions from the user program. Later on, there has to be a transfer of all the driver functionality onto one single kernel module, skipping both the user space program, the PSI module and the irqsignal module.

4.1.5 Pulse length considerations

Hardware devices interfaced by an ordinary driver will normally continue asserting the IRQ line until the interrupt is serviced (i.e. when the status register of the device is read and cleared). In our prototype driver, the interrupt service routine does nothing more than signalling a user space process to handle the interrupt. There is no guaranteed time for when this user space process is given priority to service the interrupt. This means that our hardware device will continue to assert the interrupt line for an unpredicted amount of time. And as long as the interrupt line is kept low, all other interrupt service routines sharing the same interrupt line will be kept busy. Keeping the IRQ line low for too long time can also hang the bus.

Using a programmable timeout, specified in amounts of bus cycles, will solve the problem. For both the prototype driver and the final driver, a timeout is needed when the driver is not responding. Playing around with different pulse widths revealed that 8 bus cycles should be enough for the operating system to make it certain that all the interrupt requests were detected. Pulse widths greater than this number were only generating more CPU activity caused by interrupt service routines for hard drives and other devices sharing the same interrupt number (which means that other processes will have to wait for them to complete).

With the prototype driver, it would be useful to keep the pulse width small. For a final driver, running in kernel mode and clearing the interrupt register from the service routine, this issue will no longer matter. Here we only need a final timeout for not hanging the bus when no driver is accessible.

For the prototype driver, small pulse widths means that the interrupt signal will be deasserted before the interrupt status register is read. The device will have no guarantee that the interrupt was actually detected by the operating system. It was therefore discussed if a second interrupting mechanism should be implemented, regenerating an interrupt if the former interrupt was not serviced within a finite amount of time. This kind of mechanism would be useful with the protype driver, ensuring no interrupts were undetected by the operating system. But in a later development cycle, this feature would be completely rudimentary as the interrupt register would be cleared immediately from the service routine. Therefore it was not implemented. Instead we sought some kind of compromise for the prototype driver, making sure the pulses are long enough to be detected, but not so long that the interrupt routines keep the CPU busy, slowing other processes down.

4.1.6 Interrupt scenario with the new design:

HLT-RORC:

1. The master_ctrl module of HLT-RORC detects that the host buffer full condition is true. Next it enters the irq_req state.

2. After entering the irq_req state, the master_ctrl module asserts irq_generate for one clock cycle. Next it enters the irq_wait state.

3. After receiving the generate_irq signal, the IRQ_signaler enters the PULSE state to signal an interrupt. The interrupt signal is deasserted when the interrupt status register is cleared. The status register will automatically be cleared after reading. If the status register is not read, the interrupt signal will be deasserted after a finite amount of bus cycles specified by a pulse length register on the IRQ_Signaler module.

HOST COMPUTER:

4. After receicing the interrupt, the interrupt service routine schedules a signal handler located in user space to handle the interrupt.

5. The signal handler reads the interrupt status register to check if the HLT-RORC had requested the interrupt. If true, it increments its read pointer and writes the new read pointer value into the read pointer register of the HLT-RORC.

HLT-RORC:

6. After receiving the updated read pointer value, the host buffer full condition will no longer be true. The master_ctrl unit will return to idle state where new transactions can be requested.

4.1.7 Adding more interrupts

With the the described design changes, we got rid of the polling used in the former version of HLT-RORC. But we would also need some kind of mechanism for notifying the software on the host computer that a new event had been inserted into the event buffer, making sure the necessary processing of received event data was initiated. The solution was to implement a second interrupt that was generated each time a whole event had been transferred to the event buffer. As soon as the master_ctrl module has completed transferring the report word, it would request the IRQ_signaler to generate an interrupt. To make the software able to distinguish between the two types of interrupts (host buffer full or end of event), we decided to use to use the second bit in the interrupt status register to indicate an end of event (The implementation of this second interrupt was actually done by Artur Szostak when he was residing in Bergen in february 2005).

4.1.8 Added or modified firmware and software:

IRQ_signaler.vhd - added
master_ctrl.vhd - modified
local_side.vhd - modified
generate_irq.c - added user program running through the described scenario using PSI and irqsignal

4.2 Bugs and unhandled exceptions in the HLT-RORC

This section contains information on bugs and unhandled exceptions in the HLT-RORC, documented from VHDL simulation and real-time hardware evaluation. The errors were detected on the APEX20KE400 implementation and corrected before the design was transferred onto Xilinx technology.

4.2.1 Experimental setup

The starting point of my work with HLT-RORC was a VHDL design created by Heinz Tilsner. The newest files of this design were dated back to 28.07.2004. The HLT-RORC design was first simulated, then synthesized and implemented on an APEX20KE400.

As already mentioned in chapter 2, HLT-RORC was originally using the Altera PCI MegaCore for interfacing the PCI bus. During implementation, I discovered that Altera had abandoned support for the APEX20KE from the PCI MegaCore. I therefore had to use the old libraries from pci_compiler-v2.1.0 to compile the design. After this minor correction, compilation succeeded without errors.

After transferring the compiled design to the FPGA, I could use the Analyzer and Exerciser functions of the Vanguard Logic Analyzer to monitor and stimulate the HLT-RORC through the PCI interface, making sure the behavior of the HLT-RORC was in accordance with the functional VHDL simulation (as described in 'Testing in hardware and re-evaluation in ModelSim'). For the hardware testing I used the 8 MB Target Memory of the Vanguard Logic Analyzer as host computer memory. This memory is capable of running zero wait state burst transactions after initial latency.

4.2.2 Functional verification in ModelSim

From the mstr_tranx.vhd module, the HLT-RORC registers are initially given this configuration:

```
-- Writing to the configuration registers:
cfg_wr(x"10000004",command_reg,"0000"); --cfg write to command register
cfg_wr(x"10000010",bar0,"0000"); --cfg write to bar0 of Altera PCI MegaCore
cfg_wr(x"10000014",bar1,"0000"); --cfg write to bar1 of Altera PCI MegaCore
cfg_wr(x"10000018",bar2,"0000"); --cfg write to bar1 of Altera PCI MegaCore
cfg_wr(x"20000010",trgt_tranx_bar0,"0000"); --cfg write to Target Transactor

-- Writing to the HLT-RORC local memory:
mem_wr_32(bar0+x"00", x"20000000", 1);  -- DMA base address
mem_wr_32(bar0+x"04", x"00000400", 1);  -- DMA buffer length
mem_wr_32(bar0+x"08", x"20000000", 1);  -- report buffer
mem_wr_32(bar0+x"2c", x"20000000", 1);  -- rd_ptr_addr;
mem_wr_32(bar0+x"30", x"20000000", 1);  -- rd_ptr;
mem_wr_32(bar0+x"18", x"0000000c", 1);  -- enable PG and DMA transfers
```

```
mem_wr_32(bar0+x"20", x"00000019", 1);  -- event length in words (4 bytes)
mem_wr_32(bar0+x"24", x"00000001", 1);  -- reset internal pattern generator
mem_wr_32(bar0+x"24", x"00000000", 1);  -- un-resetting pattern generator
mem_wr_32(bar0+x"28", x"affed00f", 1);  -- generate an event
```

The 'affed00f' instruction tells the HLT-RORC to start generating event data. The lenght of the generated events has been set to 0x19 = 25 words of data following the DDL header that has a length of 8 words. The DDL data stream is ended by a status word, giving a total number of 8 + 25 + 1 = 34 words.

When the HLT-RORC is ready to transfer the generated data stream, the master_ctrl module will be initiating a new transaction. As data is being transferred in 64 bit burst transactions where each transaction has 16 data phases, the last words of the ending burst transaction has to be zero padded.

The zero padding, implemented by the packetizer module, needs some more explanation. The finite state machine of the packetizer module includes three different states: normal, trans_status and zero. As long as the packetizer is receiving datawords (in state 'normal') from the DMA FIFO, it keeps forwarding these words into the PCI core which places the data on the PCI bus. After transferring the status word, the packetizer enters the state zero, feeding the bus interface with zeroes for the rest of the transaction.

In 32-bit mode, these burst transfers will have lengths of 32 data cycles. What can be observed on the bus should therefore be two burst transactions, the first one including the DDL header and 24 words of event data and the second one including the 25th word of the event, the status word and 15 more data cycles of zero padded words.

4.2.3 Testing in hardware and re-evaluating with Modelsim

Latency timer expiration followed by dead lock

When stimulating and observing the HLT-RORC through the Vanguard Bus Analyzer, I notized that the 32-bit burst transactions were ended after 25 data cycles. I traced the problem back to an abnormal transaction termination due to a latency timer expiration. The latency timer register value had been set by the operating system at startup.

Now I observed that the same error happened when simulating the design with the latency timer register set to 32 and modifying the testbench target memory to include initial latency cycles:

```
cfg_wr(x"1000000C", x"00001000","0000"); -- cfg write to latency timer reg.
```

As already mentioned, the Target Memory of the Vanguard Logic Analyzer was used as host computer memory. This memory had an initial latency of 7 bus cycles. As the BIOS had given the latency timer of the HLT-RORC a default value of 32, the transaction would be terminated after (32 - 7) = 25 data cycles. The same errors would appear when running in 64-bit mode if the value of the latency timer register was less than or equal to 16.

A simple configuration write to the latency timer register (located in the configuration space of the PCI interface), for example setting the latency timer register to 64, was enough for making this error never happen. The configuration write could either be done manually through the Exerciser interface of the Vanguard Logic Analyzer or by writing and running a simple C-program.

As we could observe, the original version of HLT-RORC had no mechanism for handling the abnormal transaction terminations occuring when the latency timer expired. Other kinds of disconnects were handled, but not this one.

With the given design flaws, the HLT-RORC would enter some kind of deadlock after the latency timer expiration had occured. First, the master_ctrl module made the PCI core reclaim the bus and issued another transaction. If it succeeded, it continued sending data. But this follow-up transaction was done without incrementing the DMA address, so instead of continuing writing into the next offset of the host memory, it started all over again at the base DMA address, overwriting already written data.

The zero padding, controlled by the packetizer module, introduces even more problems. After receiving the status word, the packetizer module enters the zero state, padding the following data words with zeroes. But we are still left with the abnormal terminations (generated by the latency timer expirations), and this time, when the PCI interface disconnects and the status word has been sent, the master_ctrl will not reclaim the bus for sending more zeroes. This means that the zero padding, that should include 15 64-bit words with zeroes (given an event length of 0x19), ends after reaching the first abnormal termination.

Instead of reclaiming the bus for sending the last zeroes, the master_ctrl module now enters the report_req state, asking the PCI core to request the bus for making a 32 bit write transaction. But this request is made one clock cycle before the former burst transaction has properly completed. Therefore, the PCI core will not respond to the request. Now the master keeps waiting forever for the PCI core to respond and nothing more happens.

4.2.4 Other errors detected from hardware testing and re-simulation

No retry handling implemented

No mechanism was implemented for resending data when a master write transaction had been terminated with a retry.

Skipping qwords when resuming aborted 32-bit transfers

The reason for this problem can be traced back to the master_ctrl module. Each time a data transfer is made on the local side of the PCI core, a new read request is made to the FIFO. But if the current qword transaction is being abnormally terminated, the current qword will be replaced by the next qword from the FIFO before it has been transferred.

4.3 HLT-RORC design changes for implementing exception handling

This section contains information on exception handling mechanisms that were added to the design before it was transferred onto Xilinx technology.

The original HLT-RORC design contained its own state machine for handling abnormal transaction terminations. For the moment this state machine was only taking care of disconnect with data and disconnect without data. Other abnormal transaction terminations like retry and latency timer expiration were not handled. Fortunately the existing structures were general enough that they allowed other kinds of functionality to be added without too much work.

4.3.1 Simulation setup

To correct the problems detected from the hardware testing, I had to modify the testbench to make it simulate the errors. To make the PCI core terminate the transaction after 25 data cycles, I first set the latency timer to 32. Next I modified trgt_tranx.vhd by inserting initial latency cycles. With these modifications, the PCI core terminated the transactions after 32 PCI clock cycles, only including 25 data cycles.

Retries were enabled by asserting the signal trgt_tranx_retry in the mstr_tranx.vhd module.

4.3.2 Design changes

Address updating after abnormal terminations due to latency timer expiration:

After a latency timer expiration, the HLT-RORC reclaims the bus to finish the aborted burst transaction. In 32-bit transfer mode, the 64-bit data words are being cut up in two 32-bit data words when being transferred on the PCI bus. Instead of making logic for all kinds of special cases (as for example terminating with an odd length of transferred words, cutting the last 64-bit word in half), the last sent data word will be fetched from a data buffer (the data_store buffers) and resent before the program continues sending new data from the FIFO. This approach strongly simplifies the error handling logic (as there always will be a tradeoff between logic complexity and handling all kinds of special cases).

Retry handling in data transfers and report transfers

When a data transfer is ended with a retry, the HLT-RORC reclaims the bus and resends the retried data words in the next burst transfer. The resent words are stored in the data_store buffers. There is a finite limit of 8 retries before the HLT-RORC handles the retries as a fatal error and ends the DMA transfers.

When a report transaction is ended with a retry, the HLT-RORC reclaims the bus and resends the retried data words.

4.3.3 Added or modified design modules:

master_ctrl.vhd - modified
local_side.vhd - modified
packetizer.vhd - modified

4.4 Migration of HLT-RORC from APEX20KE400 to Xilinx Virtex-4 LX25

This section describes the design changes that were done to the HLT-RORC during the transfer from APEX20KE400 to Xilinx Virtex-4 LX25.

4.4.1 Overview

Most of the HLT-RORC design was fully generic, with two exceptions. The first and most important one was the PCI interface. The second one was the DMA FIFO. Both modules were so-called megafunctions, created with the Altera MegaWizard Plug-In Manager. To fullfill the 66MHz requirements of the PCI Core, we also had to partition the design into two different clock domains.

4.4.2 PCI Interface

It was not necessary to add any new states in the master_ctrl and target_ctrl modules. Most of the implementation could be reduced to simple remapping of control and data signals. Table 4.1 shows the corresponding signals in the two cores.

While the old MegaCore-function had one l_adi-input for writing addresses and data on the local side and another l_dato-output for reading data, LogiCORE PCI has only one bidirectional ADIO-bus for both reading and writing. This means that our VHDL implementation has to use tristate logic when driving the ADIO signals.

Unlike the Altera PCI MegaCORE, the Xilinx LogiCORE had its own clock output and reset output that could be used for the application logic. The output reset signal was active high, so it had to be inverted before it was connected to the active low reset inputs of the HLT-RORC application. As would be described later in this chapter, the clock output of the PCI Core could in practice only be used for parts of the HLT-RORC design.

4.4.3 Clocking

Providing a reference clock for the delay buffers

Virtex-4 implementations of LogiCORE PCI make use of the IDELAY input delay buffer primitives. An IDELAY input delay buffer is a calibrated and adjustable delay line. This delay mechanism is said to provide superior performance over the legacy input delay buffers. Designs that use IDELAY primitives also require the use of the IDELAYCTRL primitive. The function of the IDELAYCTRL primitive is to calibrate the IDELAY delay lines. To perform this calibration, the IDELAYCTRL primitive requires a 200 MHz input clock. The design and wrapper files for using LogiCORE PCI with reference clocks contain IDELAY instances, IDELAYCTRL instances, and an additional input, RCLK, for a 200 MHz reference clock from an I/O pin. This reference clock is distributed to all applicable IDELAYCTRL primitives using a global clock buffer [39].

It is important to note that there is some flexibility in the origin, generation, and use of this 200 MHz reference clock. The provided design and wrapper files represent a trivial case that can be modified to suit specific design requirements. For designs that do not have a 200 MHz reference clock, it may be possible to generate a 200 MHz reference clock internally using a Digital Clock Manager (DCM) and another clock. The other clock may be available internally or externally, but must be fixed frequency. The fixed frequency requirement of the source clock precludes the use of the PCI bus clock, unless the design is used in an embedded/closed system where the PCI bus clock is known to be a fixed frequency [39].

To fullfill these requirements with the HLT-RORC design, an external on-board crystal oscillator was used as input for the DCM. The oscillator was running at 40 MHz, giving a multiplication factor of 5 for the DCM frequency synthesizer.

To connect the internally generated clock to the RCLK input of the PCI core, the pin assignments of the RCLK pin had to be removed from the constraint file. We also had to remove the assigned input buffer (IBUF) and the global clock buffer (BUFG) from the wrapper file, thus connecting the RCLK input directly to the global clock buffer driven by the DCM.

Using Digital Clock Managers for frequency synthesis and phase shifts

The Virtex-4 Digital Clock Managers (DCM) provide a wide range of clock management features [40]:

- **Clock Deskew**. The DCM contains a delay-locked loop (DLL) to completely eliminate clock distribution delays, by deskewing the DCM's output clocks with respect to the input clock.

- **Frequency Synthesis**. Separate outputs provide a doubled frequency (CLK2X and CLK2X180). Another output, CLKDV, provides a frequency that is a specified fraction of the input frequency. Two other outputs, CLKFX and CLKFX180, provide an output frequency derived from the input clock by simultaneous frequency division and multiplication.

- **Phase Shifting**. The DCM allows coarse and fine-grained phase shifting. The coarse phase shifting uses the 90°, 180°, and 270° phases of the CLK0 output to make CLK90, CLK180 and CLK270 clock outputs. The 180° phase of CLK2X and CLKFX provide the respective CLK2X180 and CLKFX180 clock outputs. There are also four modes of fine-grained phase-shifting; fixed, variable-positive, variable-center, and direct modes. Fine-grained phase shifting allows all DCM output clocks to be phase-shifted with respect to CLKIN while maintaining the relationship between the coarse phase outputs.

- **Dynamic Reconfiguration**. There is a bus connection to the DCM to change DCM attributes without reconfiguring the rest of the device. These features can be used to adjust the operating frequencies during run-time.

In the HLT-RORC design, two DCM modules were instantiated. In the first DCM, where the onboard crystal oscillator was serving as input signal, the CLKFX output was used with a

multiplication factor of 5, giving a 200 MHz reference clock for the PCI core. In the second DCM, where the PCI bus clock was serving as input signal, the CLK180 output was used for coarse-grained phase shifting of the global clock, thus removing problems with setup and hold times for signals crossing different clock domains (see the next part of this subsection).

Clock regions and clock domain partitioning

Virtex-4 devices are divided into clock regions. Regional clock signals enter at the center of a given region, and span the region of entry in addition to the region above and the region below. The reach of a regional clock is physically limited to three clock regions. Some Virtex-4 implementations of LogiCORE PCI, including all supported 66 MHz implementations, have to be clocked by a regional clock buffer (using the BUFR primitive) instead of a global clock buffer (inferred by the BUFG primitive) [39].

For designs using regional clocking, the PCI interface and those portions of the user application clocked from the PCI bus clock must completely fit inside the three clock regions accessible to the regional clock signal. This restriction limits the number of FPGA resources that may be synchronous with the PCI bus clock. Access to additional logic is available by crossing to another clock domain.

The HLT-RORC design is too big to fit into three clock regions. Partitioning the design into different clocking domains is therefore necessary. Modules that have to be synchronous with the local side of the LogiCORE PCI module must be clocked by the regional clock buffer. For the other modules we can use a global clock. In the HLT-RORC design, master_ctrl, target_ctrl, dma_fifo and packetizer are the only modules that are directly in contact with the PCI Core. These modules are therefore clocked by the regional clock. For the other modules we are still using the global clock buffer.

Both the regional clock input and the global clock input are being driven by the external PCI bus clock. The Place and Route Report shows that the regional clock is being delayed by 1 ns while the global clock has been delayed by more than 3 ns. Clock skew between the two clock buffers leads to setup and hold problems for signals crossing the clock domains.

Two signals with the same frequency, but different phase, are said to be mesochronous. A mesochronous signal should be safe to sample if it is delayed by a constant amount to fall outside aperture between setup and hold times (see for example [41]). Running a full synchronization (always necessary when dealing with asynchronous signals) could therefore be considered an overkill. Instead we decided to delay the global clock, introducing a phase shift of 180 degrees. The phase shift could be implemented using a DCM. Simulating the design with a back-annotated netlist and timing information showed that phase shifting the global clock with 180 degrees gave the output signals enough time to settle before the next clock arrived.

NB! A few additional fixes had to be done to get rid of all the setup/hold problems. Some combinatorial signals entering the DMA FIFO from the global clock domain had to be registered before entering the regionally clocked FIFO.

4.4.4 Other issues

The HLT-RORC architecture was never designed to handle target burst transfers. In the Altera-based HLT-RORC implementation, the target_ctrl module would assert the lt_discn signal to immediately issue a disconnect if a burst transfer was detected. In the Xilinx-based implementation, we would simply keep the S_TERM signal asserted for always ending target transactions with a disconnect with data, thus ensuring all data transfers being single-phase.

4.4.5 DMA FIFO

Xilinx ISE had its own CoreGen & Architecture Wizard. Using a FIFO generator (Fifo Generator v2.1) included in the wizard, it was possible to generate a new FIFO with exactly the same width and depth and an almost identical interface as the Altera-generated FIFO.

4.4.6 Interrupts

LogiCORE PCI has support for one interrupt pin. All that had to be done to move the interrupt handling from MegaCore to LogiCORE was to map the local interrupt line to the INTR_N signal.

4.4.7 Added or modified design modules:

pcim_top_r.vhd - replacement of top_module.vhd
local_side.vhd - modified
master_ctrl.vhd - modified
target_ctrl.vhd - modified
dma_fifo.vhd - modified

Tabell 4.1: Corresponding signals in the different PCI interfaces

MegaCore PCI	**LogiCORE PCI**	**Comment**
l_adi	ADIO	ADIO driver has to be tristated when not used.
l_dato	ADIO	
l_adro	ADIO ADDR	Address is clocked into ADDR one cycle after it appears on ADIO.
l_cmdo	S_CBE S_WRDN	S_CBE was never used in our new implementation. Instead we used S_WRDN to indicate the direction of the target transactions.
lirqn	INTR_N	
lt_abortn	S_ABORT	Changed from active low to active high.
lt_discn	S_TERM S_READY	Combination of the two active high signals determines type of disconnect.
lt_rdyn	S_READY	Not fully equivalent. S_READY has to be asserted during the entire transaction.
lt_framen	FRAMEQ_N	FRAMEQ_N was never used in our new implementation.
lt_ackn	S_DATA_VLD S_DATA	On Target Write: Captured data on S_DATA_VLD. On Target Read: Drive data to ADIO on S_DATA.
lt_tsr[5...0]	BASE_HIT	
lt_tsr[9]	S_SRC_EN	
lt_tsr[10]	S_DATA_VLD	
l_cbeni	M_CBE	
lm_req32n	REQUEST	Changed from active low to active high.
lm_req64n	REQUEST64	Changed from active low to active high.
lm_lastn	COMPLETE	Changed from active low to active high.
lm_rdyn	M_READY	Not fully equivalent. M_READY has to be asserted during the entire transaction or wait states will be inserted.
lm_adr_ackn	M_ADDR_N	
lm_ackn	M_DATA_VLD	Changed from active low to active high.
lm_dxfrn	M_SRC_EN	Not fully equivalent. lm_dxfrn is asserted when a local transfer is successfull while the purpose of M_SRC_EN is to enable loading of the next valid data word in a burst transaction.
lm_tsr[4]	TIME_OUT	
lm_tsr[5]	CSR[36] AND NOT M_DATA_VLD	Must be registered locally when M_ADDR_N is asserted.
lm_tsr[6]	CSR[36] AND M_DATA_VLD	Must be registered locally when M_ADDR_N is asserted.
lm_tsr[7]	CSR[37]	Must be registered locally when M_ADDR_N is asserted.
stat_reg	N/A	Not implemented yet.

Kapittel 5

Programvare for HLT-RORC

5.1 Hardware-interfacing og interrupt-handtering i Linux-kjernen

Dette underkapitlet tek fyre seg korleis Linux handterar maskinvareeiningar som minne og interruptlinor. Diskusjonen er i hovudsak basert på lærebøkene 'Understand the Linux Kernel' av Daniel P. Bovet og Marco Cesati [42], 'Linux Device Drivers' av Jonathan Corbet og Alessandro Rubini [43] og 'Red Hat Linux: The Complete Reference' av Richard Peterson [44]. Dei hine referansane som vert nemnde er artiklar som eg har funne på internett.

5.1.1 Minnehandtering i moderne prosessorar

Ein 32-bits datamaskinarkitektur kan referera opp til 4 GB med fysisk minne (2^{32}). Prosessorar som har ein Memory Management Unit, MMU, kan nytta seg av sokalla virtuelt minne. Virtuelt minne gjev programmi høve til å nytta meir minne enn det som er fysisk tilgjengeleg.

Ein MMU nyttar seg av sokalla sidetabellar der virtuelle adresser, innbolka i sokalla pages, vert mappa til fysiske adresser. Dei fysiske adressene er partisjonerte opp i sokalla page frames. Dersom MMU-krinsen ikkje finn nokor korresponderande page frame i sidetabellen, vert det generert ein sokalla 'page fault'. Ei lite nytte side vert då skufla ut or det fysiske minnet og lagd over på disk. Denne sida kan no bytast ut med den sida som svara til den virtuelle adressa [45].

Når operativsystemet skifter kontekst frå éin prosess til ein annan, vert det sett opp eit nytt sett med sidetabellar for denne prosessen. På denne måten kan kvar prosess få tilgang til sitt eige lineære adressevald på 4 GB. Sidan adressene som vert refererte frå prosessen er virtuelle, treng dei ikkje referera til dei same fysiske adressene. Til dømes kan to prosessar som syner til adresse 0x08329 peika til to ulike lokasjonar i det fysiske minnet.

5.1.2 User Mode og Kernel Mode

80x86-prosessoren kan køyra i fire ulike modi. Linux nyttar berre to av desse og kallar dei for Kernel Mode og User Mode (sjå til dømes [46]). Prosessoren har implementert eigne funksjonar til å byta frå User Mode til Kernel Mode og attende.

Eit brukarprogram køyrer til vanleg i User Mode og byter til Kernel Mode dersom det skal han tilgang til tenester frå Linux-kjernen. Når kjernen har fylgt opp fyrespurnaden frå programmet, returnerar programmet til User Mode. I Linux skil me soleis mellom kernel space og user space. Kernel space er eit lineært adressevald dedikert til Linux-kjernen medan user space er eit lineært adressevald nytta av brukarprosessane.

For ein 32-bits prosessor femner som nemnt det virtuelle adressevaldet om 4GB med minne. I Linux er 3GB av desse sette av til user space. Det virtuelle adressevaldet frå 3GB til 4GB høyrer til kernel space. Dei fyrste 896 mb med kernel space svarar beinveges til dei fyrste 896 mb med fysisk minne, dei er berre skilde med ein offset. Dette er likt for alle prosessar (sjå til dømes [47]).

I motsetnad til kernel space svarar ikkje user space beinveges til fysiske minnelokasjonar. Prosessane som køyrer i user space må nytta seg av systemkall til kjernen for å få tilgang til maskinvara. I/O til og frå devices er typiske operasjonar som må gjerast i kernel space. Sjølve ser ikkje prosessane noko anna enn sitt eige virtuelle adressevald. Kjernen fungerar derimot som eitt stort program der alle rutinane ser det same lineære adresseområdet [48].

Drivarar som skal ha direkte tilgang til maskinvara må difor kompilerast inn i kjernen. Alternativt kan ein kompilera kjernen med støtte for dynamisk lasting av modular. Kjernemodular er kodebitar som kan verta dynamisk linka inn i kjernen etter at systemet har starta opp. Dei kan unlinkast frå kjernen når det ikkje lenger er bruk for dei. Dei fleste kjernemodulane er device drivers eller pseudo-device drivers som t.d. filsystem (sjå til dømes [49]).

5.1.3 Device-filer

Som nemnt arbeider device-drivarane i kernel space der dei har direkte tilgang til maskinvara. For å gjera tenestone som devicen tilbyd tilgjengelege for vanlege prosessar i user space, kan kjernen nytta seg av serskilde device-filer, lokaliserte i katalogen /dev. Desse filene vert nytta som bindelekk mellom prosessane i user space og device-drivarane i kernel space.

Ei device-fil må vera av typen b, c, p eller u. Bokstaven b indikerar ein block device, c ein character device, u ein unbuffered character device og p ein FIFO device. Device-filene vert tilordna eit major-nummer og eit minor-nummer. Alle devices som er kontrollerte av den same device-driveren vert gjevne det same major-nummeret. Devices med same major-nummer kan skiljast frå einannan med di dei har kvart sitt minor-nummer [44]. Vanlege brukarprosessar kan nytta funksjonane open og ioctl til å opna device-filer og manipulera device-parameter. Desse filoperasjonane vert i den andre enden mappa til funksjonar i device-drivaren [50].

Systemkallet ioctl() vert kalla med tre parameter: ein peikar til den rette device-fili, eit ioctl-nummer og ein parameter som er av typen long. Med den sistnemnde parameteren kan ein nytta typekonverting til å overføra kva som helst, til dømes ein peikar til ein mykje større datastruktur.

Ioctl-nummeret vert nytta til å koda ymse slag informasjon, mellom anna kva slags type parameter som vert overført. Dette nummeret vert til vanleg skipa med eit makrokall i ei headerfil. Denne headerfili bør verta inkludert både i programmet som nyttar systemkallet og i kjernemodulen som implementerar det (sjå til dømes [51]).

Systemkallet mmap() kan nyttast til å mappa vanlege filer beinveges inn i adressevaldet åt brukarprosessen [45]. Dersom ein nyttar mmap() på ei device-fil, må ein syta for at drivaren som

kontrollerar device-fili har implementert dette systemkallet. Dersom drivaren vil gje brukarprosessen tilgang til fysisk minne, kan han returnera funksjonen remap_page_range med den fysiske minneadressa som inngangsparameter. I kjernen kan virtuelle adresser setjast om til fysiske adresser med funksjonen virt_to_phys().

5.1.4 Minneallokering i kernelspace

I kjernen kan ein nytta seg av funksjonen kmalloc() for å verta tiletla spesifiserte porsjonar med kernel memory. Funksjonen kmalloc returnerar ein peikar med virtuell baseadresse til eit fysisk samanhangande minnevald.

5.1.5 Memory-mapping i kernelspace

Fysiske minnelokasjonar som ligg over 896 mb høyrer til i det ein kallar high memory [52]. Dette fysiske minnet svarar ikkje beinveges til virtuelt minne i kernel space. Skal kjernen ha tilgang til desse fysiske minnelokasjonane, lyt dei fyrst mappast inn på ein ledig plass i sidetabellen, dvs i ein av dei siste 128 mb med kernel memory.

Periferieiningar som er memory-mappa, til dømes PCI-devices, vert som regel lagde i high memory, typisk 0xf0000000 og høgare. Desse einingane kan difor ikkje aksesserast direkte frå kjernen slik som vanleg main memory, men må fyrst leggjast inn i sidetabellen. Dette kan gjerast med funksjonen ioremap(). Funksjonen ioremap nyttar den fysiske bussadressa som inngangsparameter.

I linux-kjernen kan PCI-devices leitast fram med funksjonen pci_find_device der Device ID og Vendor ID er inngangsparamater. Denne funksjonen returnerar ein datastruktur som mellom anna inneheld bussadressene til dei ulike baseadresseregistri åt PCI-devicen. Etter at desse adressene er flutte inn i kernelspace med funksjonen ioremap(), kan dei aksesserast med funksjonane readl() og writel(). Desse funksjonane les og skriv 32-bits data i tilviste minnelokasjonar [53].

5.1.6 Synkronisering i kjernen

Den vanlegaste måten å synkronisera ulike operasjonar på, er ved bruk av semaforar [53]. I linux-kjernen vert semaforar implementerte med strukturen semaphore. Denne strukturen kan verta initialisert som ein teljande semafor eller som ein mutex-semafor[1].

5.1.7 Interrupt-handtering i kjernen

Ein modul som køyrer i kernel mode kan nytta funksjonen request_irq() til å knyta ein av funksjonane sine til eit serskilt interrupt. Denne funksjonen vil då verta køyrd kvar gong dette interruptet vert signalisert [54]. Når eit interrupt vert signalisert, fører det til at prosessoren må avbryta dei

[1] Mutex stend for 'mutual exclusion'. Dei vert nytta til å syta for at berre éin prosess eller tråd om gongen kan gå inn i ein kritisk region.

instruksjonane han held på med for å kunna køyra interrupt-rutina. Det er difor rekna som god programmeringsskikk å halda talet på instruksjonar utførde i ei interrupt-rutina nede på eit minimum.

PCI-standarden gjev høve til å dela interrupts. Interrupt-rutinor som skal handtera PCI-devices må difor taka høgd for at interruptet kan koma frå ein annan device. Ein greid måte å handtera dette på er å implementera eit statusregister i PCI-devicen. Interrupt-rutina vil då få høve til å lesa dette registeret for å finna ut om devicen verkeleg har sendt interruptet og eventuelt kvifor. PCI Local Bus Specification (Revision 3.0) har definert bit 6 i configuration header sitt statusregister som interrupt status bit (sjå til dømes [35]). Denne biten er sett so lenge interrupts er enabla og INTAN-signalet er slege på. Diverre er denne statusbiten berre vorten implementert i nyare utgåver av PCI Megacore, som t.d. 3.2.0, og desse har ikkje støtte for APEX. Og som me kan sjå i underkapittel 4.4 trengst det ofte meir enn éin bit for å koda kva slags type interrupt som var signalisert.

Ein fornuftig strategi for ei interrupt-rutine kan vera at interrupt-rutina fyrst les frå statusregistret åt devicen. Om statusbitane ikkje er sette, vil rutina returnera med det same. Dersom statusbitane er sette, kan rutina gå vidare og schedule ein vanleg funksjon til å handtera fyrespurnadene frå devicen seinare, dvs so snart funksjonen fær prioritet til å køyra. Denne funksjonen kan om turvande signalisera til ein prosess i user space at han må taka seg av fyrespurnaden.

5.1.8 Signalhandtering i userspace

Signal er notifikasjonar sende til ein prosess for å informera honom om ulike hendingar (sjå til dømes [55]). Mottakarprosessen må implementera ein eigen signal handler for å taka i mot signalet. Denne signalhandleren avbryt straks det prosessen held på med slik at han kan handtera signalet. Kvart signal er representert med eit heiltalsnummer. Funksjonen sigaction vert nytta til å knyta ein funksjon til eit serskilt signal.

Ein modul som køyrer i kernel mode kan nytta funksjonen send_sig_info() til å senda signal åt ein serskild prosess.

5.2 A simple driver and API solution

This section describes a simple driver and an API for interfacing the HLT-RORC.

5.2.1 Introduction

During the prototyping of software for the HLT-RORC, all hardware interfacing was done through the kernel modules PSI and irqsignal. These modules made it possible to access and respond to the HLT-RORC directly from the user application. Now it was time to implement one single driver where all hardware acesses were confined withing the kernel. The user application would only be interfering through specific system calls and these system calls would be encapsulated inside simple and well-defined functions made available by an API. This solution would have the benefit of both improved performance and hiding all unnecessary details from the application programmer.

During the prototyping I had been using an external target memory as event buffer. The external memory was memory mapped into one of the BARs of the Vanguard Logic Analyzer (see appendix B). With the final driver, I had to allocate ordinary main memory on the host computer for the DMA event buffer. The main memory is accessible from the PCI bus through the PCI host brigde (see for example [56]).

5.2.2 Practical use of the Publisher/Subscriber philosophy

The HLT interprocess communication is based on a Publisher/Subscriber framework. Processes exchanging event data communicate through well-defined objects called Publisher and Subscriber.

A process responsible for delivering event data has to implement a Publisher object. A process that is consuming event data has to use a Subscriber object to register itself with a Publisher object. The publishers will notify the subscribers as soon as new event data has arrived. To avoid unnecessary copying, only pointers to data structures are passed between the objects.

In ordinary inter-process communication, these mechanisms could be implemented with shared memories and counting semaphores synchronizing data producers and data consumers.

On our level of development, it is the driver, running in kernel mode, and the user application, running in user mode, that are in need of synchronization and sharing of data structures. On this level of operation the driver and the user processes can communicate through device files where specific system calls are used to pass information between the user process and the kernel routines.
As explained in subsection 5.1, the ioctl() system call can be used to exchange messages between the kernel and the user process. In the same way we can use the mmap() system call for mapping portions of kernel space into user space. And when it comes to synchronization issues, semaphores can be initialized and manipulated inside the kernel routines invoked by those already mentioned system calls.

As already mentioned, the Publisher/Subscriber philosophy is aimed at keeping copying of data structures at a minimum. Instead of copying we have to make the memory locations con-

taining event data available for both consumers and producers. A physically contiguos event buffer can be allocated in kernel space by the kmalloc() function. Then the driver can use the remap_page_range() function to map the event buffer into user space when implementing the mmap() system call.

To save the application programmer from dealing with all the different kinds of system calls, a simple API was constructed, hiding all unnecessary details. With this API, interfacing the HLT-RORC is reduced to using these five different functions:

rorcdriver_status_t rorcdriverRegister(unsigned long event_buffer_length, unsigned long report_buffer_length, unsigned long irq_pulse_length, unsigned long event_length, unsigned long timeout_in_millisecs, int enable_dma_transfers, int enable_irq_transfers, int enable_pg_transfers)
This function is used for registering the user application with the driver and setting up HLT-RORC for interrupt-based DMA transfers.

int * rorcdriverOpenEventBuffer(int length)
This function is used for receiving a pointer to the event buffer.

int rorcdriverGetReportWord(void)
This function is used for receiving event information when available from the driver.

void rorcdriverProcessingCompleted(void)
This function is used for reporting back to the driver that the application has finished processing an event.

rorcdriver_status_t rorcdriverUnregister(void)
This function is used to end the DMA transfers and unregister the user application from the driver.

The rorcdriverOpenEventBuffer() function is using mmap() for mapping the event buffer into user space. The other functions are using ioctl() to communicate with the driver module, each using their own unique ioctl number. The driver routine implementing the ioctl() call uses the ioctl number to distinguish between the different calling functions, invoking one separate function for each of the ioctl numbers.

5.2.3 Loading and unloading of the kernel module

When the rorcdriver module is initialized, it starts by looking for the HLT-RORC device. A data structure with device information is returned by the function pci_find_device(). Next it uses the function ioremap() for mapping the registers of the HLT-RORC into kernel space.

When the driver module is unloaded from the kernel, the HLT-RORC registers will be unmapped from kernel space. This is done by calling the iounmap() function.

5.2.4 Registering the user application with the driver

Before the user application can start requesting for data, the HLT-RORC device has to be initialized with the appropriate settings. An interrupt service routine also has to be set up for handling requests from the device. The application can use rorcdriverRegister() with the appropriate parameter settings to instruct the driver how to set up the device. The rorcdriverRegister() function uses the ioctl() system call for passing a data structure with initialization information to the kernel. After the ioctl number is checked, a dedicated routine is invoked by the driver to take care of the request.

To make the HLT-RORC ready for operation, the dedicated kernel routine has to allocate memory for the DMA transfers and write the appropriate instructions to the registers of HLT-RORC. The DMA buffer is allocated inside kernel space using kmalloc(), giving physically contigous memory available for the DMA engine. Since kmalloc() is returning the virtual base address of the allocated memory region, the driver routine has to use virt_to_bus() for translating the virtual base address into a PCI bus address before writing the base address into the appropriate register of HLT-RORC. To deal with interrupt requests from HLT-RORC, an interrupt routine is being set up using request_irq(). The interrupt number to listen for can be picked up from the data structure that was returned by pci_find_device().

But when it comes to processing of event data, the user application does not only need a buffer to read from. It also needs a buffer where it can write back processed data. Since the HLT processing is distributed across a network of different nodes, the processed event data may become DMA-readout by the network interface. Events that are processed by the user application should therefore be placed in physically contiguos memory locations.

NB! The former is however a matter of network architecture. With low overhead protocols for System Area Networks, it is possible for the network interface to DMA-readout the data stream directly. With conventional architectures like Gigabit Ethernet, the event information first has to be split up and packed into network packets of a given size where chunks of data are preceeded by a packet header and followed by a CRC checksum. The strategy pursued by HLT is to rely on Gigabit Ethernet as an already available baseline solution [57]. As long as Gigabit Ethernet continues to be the preferred choice of network architecture, it will not give any performance benefit to keep processed data streams in physically contiguos regions before they are placed in network packets.

Instead of allocating two different memory regions, I have kept it simple, allocating the event buffer twice as big as it had to be. Splitting the buffer into two equally sized segments, the HLT-RORC can use the first half of the buffer for placing DMA-transferred event data, leaving the second half for the user application to store processed event data.

5.2.5 Opening the event buffer from the user application

When the user application is going to access the event buffer, it can use rorcdriverOpenEvent-Buffer() to map the event buffer into user space. This function is using the mmap() system call for doing the memory mapping. The driver routine implementing mmap() is in the next turn using remap_page_range() to map physical memory into user space. The physical address of the event

buffer is obtained by calling virt_to_phys().

After the user application has registered with rorcdriverRegister() and opened the event buffer with rorcdriverOpenEventBuffer(), it is ready to process arriving events.

5.2.6 Passing read and write information between the application and the driver

The user process is using rorcdriverGetReportWord() to receive notifications from the driver about incoming events. This function is returning an integer with information about the size of the event.

Since the driver cannot return such information before an event actually has been DMA-transferred by the HLT-RORC, some kind of synchronization mechanism has to be implemented. This mechanism is based on a counting semaphore. When the driver routine handling the request is run, it counts the semaphore down. If no one else has incremented the semaphore before, the routine will be blocked, waiting for another routine to wake it up.

Now comes the interrupt routine:

As soon as an interrupt is signaled on the PCI bus, the interrupt service routine is invoked. Since the PCI standard supports sharing of interrupts, it first has to check the interrupt status register of the HLT-RORC. If the register value is zero, it returns immediately. If not, it schedules another function to handle the interrupt (time spent in the interrupt service routine has to be kept on a minimum).

HLT-RORC can generate two types of interrupts. The first kind of interrupt is generated when the DMA engine has finished transferring a whole event. The second kind of interrupt is generated when HLT-RORC has to receive a new read pointer from the driver. The status register of the HLT-RORC is used to distinguish between the two kinds of interrupts. The least significant bit indicates that the event buffer is full. The second least significant bit indicates the end of an event.

If the interrupt handling function detects a hit on the second least significant bit[2], it will increment the counting semaphore. Any blocked kernel routine invoked by rorcdriverGetReportWord() will now be allowed to continue and return the event information to the user application.

Since the event buffer is a ring buffer, we also needs some kind of mechanism to avoid that formerly written data is being overwritten by the DMA engine before it is processed by the user application and moved somewhere else. As already mentioned in former chapters, HLT-RORC uses a read pointer and a write pointer to synchronize reading and writing to the event buffer. The write pointer is generated internally on the HLT-RORC while the read pointer has to be obtained from the host computer by generating an interrupt.

If the function handling the interrupt requests detects that the least significant bit in the interrupt status register is set[3], it has to write back an updated read pointer value to the HLT-RORC. If the user application hasn't been asking for more events since the last time the read pointer was transferred, the driver has to wait for the read pointer to become incremented. This time

[2]In practice this is done by checking if the register value is 2 or 3

[3]Checking if the register value is 1 or 3

we simply use a mutex (mutual exclusion semaphore) to make sure the read pointer is updated before it is transferred. If a read pointer request is detected, we will decrement the mutex. If no one else has has incremented it, the routine will be blocked, waiting for the user process to report that more events have been processed. This report message can be delivered by the user process by calling rorcdriverProcessingCompleted() each time it has completed processing an event. The kernel routine invoked by rorcdriverProcessingCompleted() will automatically increment the read pointer and release the mutex, allowing the interrupt handling function to move on and write back an updated read pointer to the HLT-RORC.

5.2.7 Unregistering the user application and ending the DMA transfers

When the user process wants to end the DMA transfers, it has to unregister from the driver using rorcdriverUnregister(). The driver will now turn off the interrupt routine by calling free_irq().

5.2.8 Setting a final timeout when HLT-RORC is not responding

As long as everything is working correctly, the above mentioned functionality should be enough. But there might also be situations where the user application is calling rorcdriverGetReportWord() without receiving any more data from HLT-RORC. In that case there has to be some kind of mechanism to wake up the sleeping process and return an error message.

The semaphore operation down_interruptible() has the ability to return immediately if the running process is receiving a signal. A signal can be sent to a process by calling the function kill_proc(). If we want to put a final timeout on our semaphore, the easiest way of doing it is to invoke a timer before getting locked by the semaphore. Timers can be invoked by instantiating a structure called timer_list. This structure contains fields for declaring when the timer is supposed to expire and which function to call when the timer expires. If this function issues a call to to kill_proc() using the process ID of the sleeping process as input parameter, the sleeping process will be interrupted and the down_interruptible call returns (-1).

And this is exactly how it has been done in my simple API. When the user application is registering itself with the driver, it has to specify a timeout value. The value is given in milliseconds and passed with one of the input parameters of rorcdriverRegister(). When rorcdriverGetReportWord() is called, the semaphore operation has to return within the specified amount of time or else it will be interrupted by the timer sending the SIGALRM signal. If the kernel routine detects that the semaphore operation has been interrupted, it will return (-1) instead of a valid report word. This gives the user application the ability to detect that the data request was unsuccessfull.

Since the SIGALRM signal is sent each time the HLT-RORC is not delivering requested event data, the user process has to implement a signal handler for SIGALRM. The only purpose of this handler is to make the process ignore the signal instead of getting terminated. For this reason I have put the signal handler inside the API, completely hidden from the application programmer. The handler is automatically being initialized when rorcdriverRegister() is called.

5.2.9 Further improvements - replacing kmalloc() with bigphysarea

One major weakness of the original implementation was the use of kmalloc() for allocating kernel memory. With kmalloc(), the maximum allocated memory size is 128 kilobytes. Since the received event data has to be stored on the Front-End-Processors until the necessary distributed processing has completed, a much larger buffer is needed [58]. Therefore the kmalloc() function had to be replaced by some other memory allocation mechanism.

An approach that can be used to make large, contiguous memory regions available to drivers is to apply the bigphysarea kernel patch. This unofficial patch has been floating around the net for years. It is so renowned and useful that some distributions apply it to the kernel images they install by default. The patch basically allocates memory at boot time and makes it available to device drivers at runtime. You'll need to pass a command-line option to the kernel to specify the amount of memory that must be reserved at boot time [59].

The big physical area is mainly managed by two functions. The first one, bigphysarea_alloc_pages(), allocates a specified number of pages. To free an allocated area, one can simply call the second function, bigphysarea_free_pages(caddr_t base), with 'base' set to the value returned by bigphysarea_alloc_pages(). The bigphysarea approach had already been used successfully together with the PSI module at the University of Heidelberg.

To make the new driver version 'backwards-compatible' with the original, unpatched kernel, I decided to support both kmalloc() and bigphysarea in the latest edition. The compiler directive WITHBIGPHYS is used at compile-time to decide which function shall be instantiated. An example of how it has been done is shown below:

```
#ifdef WITHBIGPHYS
        mem_map_unreserve(page);
        bigphysarea_free_pages((caddr_t)event_buffer);
        bigphysarea_free_pages((caddr_t)report_buffer);
#else
        mem_map_unreserve(page);
        kfree(event_buffer);
        kfree(report_buffer);
#endif
```

5.2.10 Files

Table 5.1 lists the new source files implementing the driver, the API and a sample user application. The files can be found in their respective subdirectories located in directory RORCDRIVER in the CD-ROM shipped with this thesis.

Tabell 5.1: List of source files for the HLT-RORC software

File Name	**Directory**	**Comment**
rorcdriver.h	include	Header file for the API.
rorcdriver_mod.h	src	Header for for the kernel module.
rorcdriver_mod.c	src/driver	Source code for the kernel module.
rorcdriver_api.c	src/library	Source code for the API.
generate_irq.c	example	A sample application processing incoming event data from HLT-RORC.

Kapittel 6

Implementasjon av eit mikroprosessorsystem på Xilinx Virtex-4

Dette kapitlet tek fyre seg implementasjonen av eit SPARC-basert mikroprosessorsystem på PCI-testkortet med Xilinx Virtex-4. Grunnen til at eg byrja på dette arbeidet var at eg vart involvert i eit eit satellittprosjekt hjå romfysikkseksjonen, kalla SIR-2. Her skulle det mellom anna byggjast eit FPGA-basert datasystem der all kontrollogikken vart styrd av ein mikroprosessorkjerne. Som synt seinare i kapitlet fekk eg med dette prosjektet demonstrert at det let seg gjera å køyra eit heilt ordinært mikroprosessorsystem på den maskinvara som var meint for HLT-RORC. Ogso dette systemet var i stand til å kommunisera med vertmaskinen via PCI-bussen.

6.1 Programvare og maskinvare i innebygde system

Innebygde system, eller embedded systems som dei oftast vert kalla, er ein disiplin som krev kjennskap til både maskinvare og programvare. I slike system vert sume funksjonar køyrde på ein konvensjonell mikroprosessor og andre i dedikert maskinvare. Kva for funksjonar som skal liggja i programvare og kva som skal liggja i maskinvare vert ofte fastlagt undervegs i utviklingsprosessen. Som regel vert det fyrst laga ein prototyp i programvare som oppfyller alle dei funksjonelle kravi. Dersom prototypen ikkje oppfyller ytingskravi, kan ein byrja å flytja funksjonar over i dedikert maskinvare. I det opphavlege programmet kan desse funksjonane bytast ut med lese- og skriveoperasjonar til register i maskinvaremodulane. Har mikroprosessorsystemet støtte for memory-mappa IO, kan desse registeraksessane implementerast med lesing og skriving til vanlege peikarstrukturar i programvara (sjå til dømes [60]).

6.2 SPARC-prosessoren

SPARC, som stend for Scalable Processor Architecture, er eit ope sett med tekniske spesifikasjonar som alle har høve til å lisensiera og nytta til utvikling av RISC-baserte mikroprosessorar. SPARC vart opphavleg utvikla hjå Sun Microsystems. Sidan 1989 er SPARC-spesifikasjonen vorten varveitsla av den sjølvstendige organisasjonen SPARC International.

SPARC-spesifikasjonen har opp gjennom åri vore gjennom ulike revisjonar der SPARC V9 er den siste. Alle dei tidlegare versjonane er framleis tilgjengelege. SPARC-instruksjonssettet er standardisert og publisert som IEEE 1754-1994 [61].

6.3 Biblioteket GRLIB

Det svenske firmaet Gaisler Research tilbyd syntetiserande prosessorkjernar tufta på arkitekturen SPARC V8. Kjernane er tilgjengelege som open kjeldekode og fylgjer med biblioteket GRLIB. Omframt SPARC-prosessoren LEON3 gjev GRLIB tilgang på maskinvarekjernar for AHB/APB-kontroll (sjå fyrstkomande underseksjon), 32-bits PC133 SDRAM-kontroll, 32-bits PCI-bru med DMA, 10/100 Mbits Ethernet-grensesnitt, PROM- og SRAM-kontroll, CAN-kontroll, TAP-kontroll, UART med FIFO, modulære tidtakarar, interrupt-kontroll og ein 32-bits IO-port. Minnegeneratorar og padgeneratorar er tilgjengelege for Virage, Xilinx, UMC, Atmel, Altera og Actel [62].

GRLIB-biblioteket er tilgjengeleg under lisensen GNU GPL og kan lastast ned frå heimesidone åt Gaisler [63]. Saman med GRLIB fylgjer det ei rad ulike 'reference designs' som syner GRLIB-funksjonane i praktisk bruk og som kan tilmåtast brukaren sine eigne applikasjonar.

6.3.1 AHB og APB

For å gjera samspelet mellom dei ulike funksjonelle einingane i GRLIB-biblioteket mest mogleg einfelt, er alle dei ulike kjernane tilgjengelege via ein av bussprotokollane AMBA 2.0 AHB eller AMBA 2.0 APB. AMBA er ein open og de facto busstandard for samvirke mellom ulike funksjonsblokker på integrerte krinsar [64].

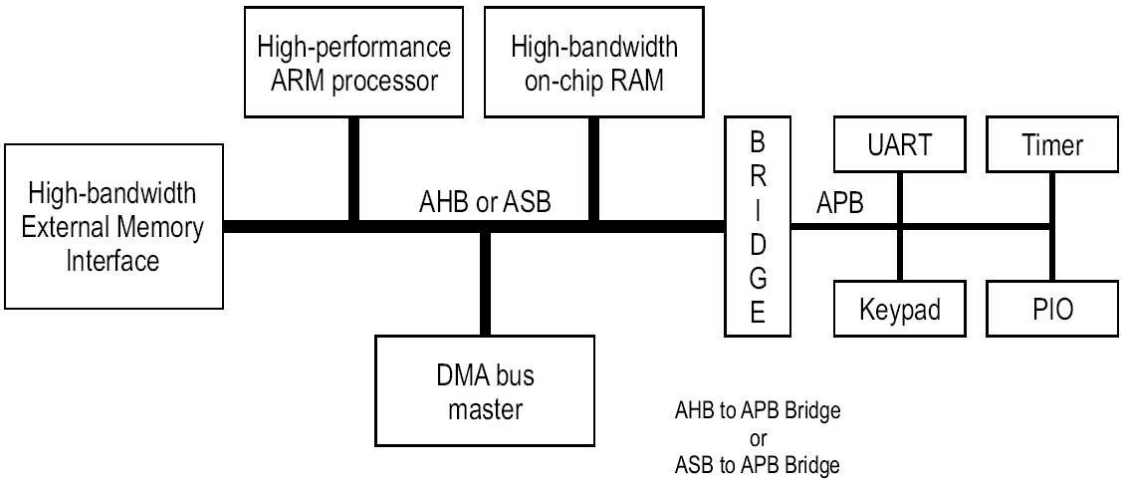
Systembussen AMBA AHB knyter i hop prosessorar med andre snøgge einingar som til dømes DMA-kontrollerar. Til liks med PCI-protokollen nyttar AHB eit arbitreringssystem med støtte for multiple bussmasterar og valfrie prioritetsalgoritmar [65].

Periferibussen AMBA APB er ein meir einfeld protokoll som er meint for generelle periferieiningar som tidtakarar, IO-portar, UART'ar og so bortetter. Funksjonelle einingar som nyttar APB-grensesnitt kan knytast til AHB-bussen via ei AHB/APB-bru. Figur 6.1 syner eit blokkdiagram for eit tenkt AHB/APB-basert mikroprosessorsystem.

6.3.2 SPARC-prosessoren LEON3

LEON3 er ein 32-bits prosessor tufta på arkitekturen åt SPARC V8. LEON er vorten sertifisert av SPARC International som V8-kompatibel. Prosessoren er implementert med ein 7-trinns pipeline med separate bussar for instruksjonscache og datacache (dvs Harvard-arkitektur). LEON3 kan nyttast i synkrone multiprosessorkonfigurasjonar (SMP) og inneheld maskinvarestøtte for cache-koherens og interrupt-styring.

Den grunnleggjande prosessorkjernen (pipeline, cache controllers og AHB-interface) konsumerar om lag 20.000 gates og kan implementerast både i FPGA-krinsar og i ASICs. I ein typisk 0,13 mikrometer standardcelleteknologi kan ein nå arbeidsfrekvensar på heile 400 MHz [66].



Figur 6.1: Døme på eit AMBA AHB/APB-basert mikroprosessorsystem [65].

Omframt den grunnleggjande prosessorkjernen har LEON3 valfri maskinvarestøtte for V8-instruksjonane MUL, DIV, MAC og ein IEEE-754-kompatibel FPU. Ein valfri MMU-komponent gjev høve til å implementera virtuelt minne dersom ein ynskjer det.

LEON3 nyttar AHB-bussen som memory-buss. Alle funksjonelle einingar som er tilgjengelege via AHB-bussen kan soleis memory-mappast. Til dømes kan ein IO-port med APB-grensesnitt gjerast AHB-tilgjengeleg via ei AHB/APB-bru og aksesserast som memory-mappa IO (eit døme på dette er synt seinare i kapitlet).

Til liks med dei hine einingane i GRLIB-biblioteket kan LEON3-prosessoren parametriserast gjennom VHDL-generics[1]. Det er soleis mogleg å instansiera ulike prosessorkjernar i det same designet med kvar sin konfigurasjon. Det fylgjer med i GRLIB eit referansedesign der LEON3 er med i ein multiprosessorkonfigurasjon med eit stort utval periferieiningar. Dette referansedesignet vert kalla LEON3MP og det gjev høve til å instansiera heile fire samvirkande prosessorkjernar i det same systemet.

6.3.3 Periferieiningar

Eg kjem i det fylgjande til å gå gjennom nokre få av dei hine funksjonelle einingane som er tilgjengelege i GRLIB-biblioteket. Grunnen til at nett desse einingane er valde ut framfor andre er at dette var einingar eg sjølv laut instansiera i LEON3MP-designet for å få implementert og testa ut LEON3MP på det PCI-testkortet eg hadde laga og som er nærare skildra i underkapittel 2.3. For alle dei omtala funksjonane gjeld det at ein kan nytta VHDL-generics under instansieringi til å styra kva slags minneadresser på AHB- eller APB-bussen som dei skal allokerast til:

[1]Det ytre grensesnittet åt ein VHDL-komponent kan innehalda både portar og generics. Generics gjev generell informasjon om korleis komponenten skal setjast i hop. I motsetnad til portar svarar dei ikkje direkte til fysiske strukturar.

AHBRAM implementerar ein 32-bits vid RAM med eit AHB-slavegrensesnitt. Minnestorleiken kan konfigurerast med VHDL-generics. Minste storleik er 1 kb. Største storleik skil seg på target-teknologien.

AHBROM implementerar ein 32-bits vid ROM med eit AHB-slavegrensesnitt. For å laga ein AHBROM lyt ein fyrst kompilera programmet sitt til ei binærfil i ELF-formatet. Denne binærfili kan nyttast som innputt til eit program som lagar ein VHDL-modell av ein AHBROM.

APBCTRL implementerar ei AHB/APB-bru. APB-brui er ein APB-bussmaster som stør opp til 16 APB-slaveeiningar.

APBUART er meint for seriekommunikasjon. Han implementerar ein UART med APB-slavegrensesnitt. UART'en har støtte for data-frames med 8 databits, ein valfri paritetsbit og ein stoppbit. Til å generera bitrate har kvar UART ein programmerande 12-bits klokkedelar. Flytkontroll er støtta med RTSN/CTSN-handshakesignal. To konfigurerande FIFO'ar vert nytta til å bufra mellom APB-bussen og UART'en.

GRGPIO implementerar ein skalerande IO-port med interrupt-støtte og APB-slavegrensesnitt. Portbreiddi kan justerast frå 2 til 32 bitar ved hjelp av VHDL-generics.

6.3.4 DSU3 - Debug Support Unit

LEON3-prosessoren gjev høve til å køyra i ein eigen debug-modus der prosessoren vert kontrollert gjennom eit serskilt debug-grensesnitt. Debug Support Unit, DSU, vert nytta til å kontrollera prosessoren når han er i debug-modus. DSU inneheld eit AHB-slavegrensesnitt som kan aksesserast av andre AHB-bussmasterar. Ein ekstern vertsmaskin kan soleis styra DSU gjennom mange ulike modular som til dømes ein UART eller ein PCI-kjerne med AHB-grensesnitt. GRLIB-modulane AHBUART og PCITARGET er døme på slike komponentar.

6.4 Programvareutvikling for LEON3

6.4.1 LEON Bare-C Cross-compiler

Sidan LEON3 er kompatibel med SPARC V8, kan kompilatorar og mikrokjernar for SPARC V8 nyttast med LEON3. For å letta programvareutviklingi tilbyd Gaisler Research kompilatoren Bare-C Cross-compiler, BCC. Kompilatoren gjev høve til å køyra sekvensielle program med interrupt-støtte. Kompilatoren har støtte både for harde og emulerte flyttalsoperasjonar. BCC er basert på GCC og biblioteket Newlib [67].

Newlib er eit POSIX-kompatibelt C-bibliotek med full støtte for alle matematiske funksjonar. Newlib har diverre ingen stønad for filer eller IO-relaterte funksjonar, med unnatak for stdin/stdout. Desse funksjonane er mappa til UART nr 1.

Dersom ein ynskjer å skriva til ein IO-port, er den greidaste måten å gjera dette på skriva beinveges til dei aktuelle registri ved hjelp av memory-mapping. Dersom me let vera å nytta virtuelt minne, kan me peika direkte til dei AHB-adressene som registri åt IO-portane er allokerte til.

Nedanfor er det synt eit døme på korleis dette kan gjerast i eit C-program:

```
volatile int *pio = (int *)0x80000b00;
/* (IO er memory-mappa til AHB-adresse 0x80000b00) */

/*
 * pio[0] = din
 * pio[1] = dout
 * pio[2] = dir
 * pio[3] = imask
 */

pio[3] = 0;
pio[2] = 0;
pio[1] = 0;

pio[2] = 0x000000FF; // 8 bits ordbreidd på utgangssignalet

pio[1] = 0x000000AA; // "AA" skal koma ut på porten
```

6.4.2 Organisering i RAM og ROM

Binærfilene som vert genererte av BCC kjem ut i det sokalla ELF-formatet og har tre hovudsegment - kodesegment (text-segment), datasegment og bss-segment.

Når eit program skal kompilerast og linkast, lyt det avgjerast om det skal køyra i ROM eller RAM. BCC gjev høve til å laga komprimerte boot-proms som startar i ROM og lastar applikasjonen til byrjingi av RAM, initierar ulike register og startar applikasjonen. Det er ogso mogleg å kompilera program slik at dei køyrer i ROM, men har data og stack i RAM.

Byggjing av applikasjonar vert til vanleg gjort i desse stegi:

- Kompiler og eventuelt link programmet med GCC
- Lag promfil med programmet mkprom
- Lag AHBROM av promfili (dersom bootpromen skal plasserast internt på brikka)

6.4.3 GRMON

Programmet GRMON er utvikla for å kunna kommunisera med LEON3-prosessoren sin DSU. Kommunikasjonen kan anten gå fyre seg serielt via AHBUART-modulen eller over PCI-bussen med PCITARGET-modulen. GRMON kan nyttast til å lesa og skriva til register i prosessoren og minne på AHB-bussen. På denne måten kan GRMON nyttast til å lasta programvare inn i

RAM og køyra denne programvara på prosessoren. Under køyringi er det høve til å leggja inn breakpoints[2] eller gjera single-stepping[3] [68].

6.5 Praktisk implementering på Virtex-4

Målet mitt var å få eit SPARC-basert mikroprosessorsystem til å køyra på testkortet mitt. For å få testa at systemet faktisk fungerte, ville eg skriva eit ljosmynster ut på diodane og senda ei melding med 'Hello word!' ut på UART'en. Her var planen å kopla meg til serieporten på PC'en via ein RS232-nivåomformar og nytta Hyperterminal-programmet i Windows til å lyda etter livsteikn frå mikroprosessorsystemet på testkortet.

6.5.1 Konfigurering

Den greidaste framgangsmåten for å få sett i hop eit ferdig system var å taka utgangspunkt i referansedesignet LEON3MP. Det opphavlege LEON3MP-designet var konfigurert til å nytta eksterne ROM- og RAM-brikker. Desse brikkene var tilgjengelege frå AHB-bussen via eigne kontrollermodular som høyrde til GRLIB-biblioteket. Sidan eg ikkje hadde nokon ekstern ROM eller RAM tilgjengeleg på kortet mitt, laut eg kvitta meg med desse kontrollerane og heller instansiera ein intern ROM til å leggja styreprogrammet mitt i. På same måten ville eg instansiera ein intern RAM som kunne tena til main memory. Eg ynskte ogso å instansiera ein GRGPIO-komponent slik at eg kunne nytta ljosdiodane på testkortet til å demonstrera at programmet mitt var 'i live'. Dessutan so hadde eg tenkt å instansiera ein PCITARGET-modul og ein AHBUART slik at eg fekk høve til å overvaka kva som gjekk fyre seg internt i systemet.

Dei ulike styreparametri, som til dømes talet på prosessorar og kva slags periferieiningar som skulle vera med, kunne setjast i konfigurasjonsfili Config.vhd. Eg valde å nytta 1 stk LEON3-prosessor[4], 1 stk APBUART, 1 stk AHBUART, 1 stk PCITARGET, 1 stk AHBROM, 1 stk AHBRAM, og 1 stk GRGPIO. AHBROM og AHBRAM var direkte tilknytte AHB-bussen medan APBUART og GRGPIO var tilknytte omveges med ei AHB/APB-bru.

Når det galdt instansiering av minne og instansiering av pads, so var ein nøydd til å velja i konfigurasjonsfili mellom serskilde teknologiar. Det næraste eg kom Virtex-4 var Virtex-2.

6.5.2 Syntetisering, plassering, ruting og programmering

I referansedesignet LEON3MP fanst det opplegg til ferdige scripts for generering av implementasjonsfiler. Dersom ein gjekk i kommandolina og skreiv 'make scripts', vart det sett opp scripts til å køyra implementasjon gjennom ulike slag syntheseverkty. Mellom desse var Xilinx ISE. Før

[2]Breakpoints er spesifiserte adresser i instruksjonsminnet der programeksekveringi vert stogga til dess det kjem klårsignal frå brukaren til å gå vidare

[3]Single-stepping vil seia å gå stegvis gjennom programkøyringi, instruksjon for instruksjon. Kvart nytt steg lyt initierast av brukaren

[4]BCC hadde ikkje har støtte for multiprosessorkøyring, difor var det heller ingen vits i å instansiera meir enn éin prosessor. Eg sette difor talet på prosessorar lik 1.

eg gjekk vidare med implementasjonen, laut eg gå inn og endra sume av scripti for hand, mellom anna laut eg endra target-teknologi frå Virtex-2 til Virtex4 i .xst-filene[5] og endra på pinnetilordningane i .UCF-filene[6]. Heldt ein fram med å skriva 'make ise', vart det køyrd syntese, place and route og generering av programmeringsfil. Eg hadde som nemnt spesifisert Virtex-2 som target-teknologi i konfigurasjonsfili Config.vhd, men her synte det seg at syntheseverktyet var intelligent nok til å byta ut dei Virtex-2-komponentane som var instansierte i GRLIB-biblioteket med tilsvarande Virtex-4-komponentar. Om ikkje so hadde det truleg vore ei smal sak å gå inn i GRLIB-koden og gjera desse endringane sjølv.

6.5.3 Testing

Etter at bitfili var overført til Virtex-4-krinsen på testkortet, kunne eg med eigne auga sjå at annankvar diode lyste opp. I Hyperterminal observerte eg at meldingi 'HELLO!' spratt fram i terminalvindauga. Frå bussanalysatoren Vanguard kunne eg dessutan sjå at konfigurasjonsheaderen åt PCI-grensesnittet dukka opp. Og om eg freista lesa ut data AHB-minnet via PCI-bussen, såg eg at eg fekk ut den same informasjonen som eg hadde sett frå simulering skulle liggja der. Eg kunne soleis konstatera at implementasjonen av mikroprosessorsystemet på testkortet hadde vore vellukka.

6.5.4 Programmering med GRMON

Som nemnt tidlegare i kapitlet lyt programvara som køyrer på prosessoren leggjast i ein ROM og syntetiserast inn i resten av VHDL-designet. Dette vil i praksis seia at heile VHDL-designet lyt syntetiserast på nyo kvar gong eg vil gjera endringar i programmet mitt. Under programvareutviklingsprosessen hadde det soleis vore ein stor fyremun om ein kunne lasta programmet direkte inn i ein RAM og starta programkøyringi derifrå. Programmet GRMON gjev som nemnt høve til nett dette. Etter at programmet GRMON har starta opp og kome i kontakt med DSU-modulen (anten via PCI-bussen eller via RS232-porten), kan ein nytta kommandoen 'load' til å leggja binærfiler inn i LEON3 sitt main memory og deretter kommandoen 'run' for å få prosessoren til å byrja køyra programmet. Sidan desse programmi skal køyra i RAM og ikkje i ROM, lyt dei linkast på vanleg måte med GCC og ikkje med ROM-byggjaren mkprom. Her er det dessutan ein fyresetnad at RAM har vorte dimensjonert stort nok til å få plass åt eit heilt program.

[5] Syntheseskriptfiler

[6] UCF stend for User Constraint File. I desse filene kan ein leggja inn retningslinor for plassering og ruting av designet.

Kapittel 7

Avslutning

Dette kapitlet diskuterar eventuelle framtidige endringar med overgang til meir moderne bussarkitekturar. Til slutt freistar eg å gjera ei oppsummering av det arbeidet som har vorte gjort og kva som er lærdomane av dette.

7.1 Framtidige forbetringar

7.1.1 Kva kan gjerast for å auka overføringsrata?

Slik HLT-RORC-arkitekturen er i dag, kjem det inn data frå berre éin DDL-link pr HLT-RORC-kort. Her kunne ein alternativt tenkja seg at innkomande data vart bufra slik at det vart høve til å implementera 2 eller fleire DDL-links pr HLT-RORC-kort. På same måten kunne ein ha tenkt seg at fleire enn eitt RORC-kort kunne plasserast i det same PCI-segmentet, noko som hadde spart maskinvareressursar.

Alt dette må gjerast på fyresetnaden at datarata på PCI-bussen vert høg nok til å klara dette.

7.1.2 Bussarkitekturar og Moores lov

Under utviklingi av ALICE-eksperimentet har ein freista taka høgd for Moores lov [11][1]. Soleis gjeng ein ut frå at mange av dei maskinvareløysingane som pr i dag er for dårlege til å oppfylla dei strenge kravi til yteevna, kjem til å verta brukande når dei vert implementerte med ny og betre prosessteknologi.

Dei som har fylgt voksteren åt datateknologien det seinste tiåret, har kunna observera at det har vorte eit støtt større gap mellom bandbreiddi på PCI-bussen og dei dataratene som prosessoren sjølv er i stand til å klara. For prosessorar og internminne har utviklingi med lægre forsyningsspenning og høgare klokkefrekvens halde fram, medan ho har stade i ro på IO-sida. Jamvel i vanlege desktop-PC'ar har Gigabit Ethernet og støtt snøggare grafikkort gjort PCI til ein flaskehals, på same måten som ekspansjonsbussen ISA ein gong var det. Grafikkort er døme

[1] Moores lov er den empiriske observasjonen at talet på transistorar som ein kan få inn på ein integrert krins har dobla seg kvar attande månad.

på ekspansjonskort som alt for lenge sidan har vore nøydde til å omgå PCI-standarden med sin eigen AGP-standard.

7.1.3 PCI-X

Ynskjer ein endå høgare datarater enn det ein kan få med konvensjonell PCI, kan PCI-X vera eit alternativ. PCI-X er attoverkompatibelt med PCI, men med utvida eigenskapar som skal utnytta bussen betre:

Bussutnytting

PCI-X introduserar konseptet med transaksjonssekvensar, der kvar logisk overføring kan kløyvast opp i fleire transaksjonar.

Forseinka lesetransaksjonar kan bytast ut med sokalla split-transaksjonar. I staden for å signalisera retry eller leggja inn wait states kan target-devicen signalisera ein sokalla split response. Deretter kan han sjølv initiera ein eller fleire split completion transactions for å fullføra den opphavlege fyrespurnaden frå initiatoren. I mellomtidi kan andre devices nytta bussen. Lesetransaksjonar kan på denne måten gjerast om til ein serie av target-initierte skrivetransaksjonar.

PCI-X er utforma for langvarande bursttransaksjonar. Standard block size er 128 bytes. Maksimal block size er 4096 bytes.

Alle typar wait states utanom target initial wait states er avskipa.

Timing

Logikken er vorten forbetra med omsyn til timing på den måten at alle data på bussen gjeng beinveges frå register til register. Medan transaksjonskommandoane i vanleg PCI vert dekoda beinveges av mottakarlogikken, vert dei i PCI-X klokka rett inn i eit register og dekoda i neste klokkeperiode.

PCI-X 2.0 spesifiserar 4 ulike speed grades: PCI-X 66, PCI-X 133, PCI-X 266, og PCI-X 533 [6]. PCI-X 66 og PCI-X 133 er dei same som i PCI-X 1.0. Ein sideeffekt med dei auka frekvensane som PCI-X tilbyd er at kravi til kapasitiv last har vorte strengare, noko som i praksis krev stuttare linelengder og færre nodar pr line. Dette fører til at PCI-bussen lyt kløyvast opp i fleire segment. I kvart av desse segmenti lyt ein full PCI-buss rutast ut frå kontrolleren til kvar slot. Soleis har ein PCI-X-buss som køyrer på 66 MHz støtte for fire slots medan ein PCI-X-buss som køyrer på 100 MHz har støtte for berre to. Ved høgare frekvensar (PCI-X 133, PCI-X 266, and PCI-X 533) er det berre støtte for éin slot pr kontroller (sjå til dømes [69]). Fyremunene med PCI-X lyt difor vurderast opp mot dei ekstra kostnadene det er med denne implementasjonen.

LogiCORE PCI-X v5.0

Xilinx har sin eigen IP-kjerne som kan nyttast til å implementera PCI-X 66 eller PCI-X 133. LogiCORE PCI-X v5.0 kan implementera både master-funksjonalitet og target-funksjonalitet [70].

7.1.4 PCI Express

Frå parallell til seriell og differensiell signalering

Når det kjem til den parallellbuss-filosofien som PCI og PCI-X byggjer på, har han sine naturlege avgrensingar. Signal som kjem parallelt må synkroniserast av ei klokke og dette gjer timing vanskeleg ved høge klokkerater. I industrielle miljø er dessutan eit poeng at parallellbussar er meir vare for krysskopling, noko som historisk har gjort parallellbussar lite aktuelle som feltbussar (sjå til dømes [71]).

Å utvida ordbreiddi på bussen er heller ikkje uproblematisk. Det er ikkje so lett å auka talet på pinnar på eit innstikkskort med dagsens FR4-teknologi. Eit aukande tal på tilleidningar fører til støtt strengare reglar for ruting på PCB-nivå for å klara klokkesynkroniseringsproblemi (sjå til dømes [9]).

Det trengst difor ein ny standard som er kostnadseffektiv og lett å implementera på PCB-nivå. PCI-X er snøggare, men mindre kostnadseffektiv enn konvensjonell PCI. Dei grunnleggjande vandemåli med ein parallellbuss kan einast løysast med ei fullstendig omforming av arkitekturen, basert på ny teknologi. Dette har ført til at PCI Express har sett dagsens ljos:

PCI Express vert av mange rekna som tredje generasjons IO. Seriell og differensiell signalering gjer det mogleg å skalera overføringsrata heilt opp mot 12GHz-valdet som er grensa for kva eit koparmedium kan klara. Differensielle signal har dessutan fyremunen med at dei gjer det lettare å undertrykkja 'common mode'-støy(sjå til dømes [72]).

PCI Express er basert på ein seriearkitektur. Punkt-til-punkt-samband[2], differensiell signalisering og innebygd klokka fjernar mange av dei avgrensingane som er med parallellbussar. Den innebygde klokka reduserar talet på pinnar (det trengst ingen separate kontrollinor eller klokkelinor) og gjer synkronisering lettare.

PCI Express og OSI-modellen

Arkitekturen åt PCI Express fylgjer OSI-modellen (sjå til dømes [73]) og er soleis sett i hop av eit programvarelag, eit transaksjonslag, eit datalinklag og eit fysisk lag. Takk vere den lagdelte strukturen er PCI Express vorten programvarekompatibel med PCI 2.2. PCI Express nyttar den same konfigurasjonsmodellen og kommunikasjonsmodellen som PCI og PCI-X. PCI Express har støtte for tilsvarande transaksjonar som memory read/write, IO read/write og configuration read/write. Modellane for memory, IO og configuration address space er dei same som for PCI og PCI-X. Sidan adresseringsmodellane er dei same, kan eksisterande drivarar køyra i eit system basert på PCI Express utan modifikasjonar av noko slag.

Omframt dei tre adressevaldi for memory, IO og configuration har PCI Express støtte for eit fjerde adressevald kalla messages space. Message space vert nytta til å støtta alle dei gamle sidebandsignali frå PCI-spesifikasjonen som til dømes interrupts, resets og so bortetter. Andres 'special cycles' som Interrupt Acknowledge vert tekne hand om på same måten. Ein kan velja å sjå på meldingane som 'virtuelle leidningar' sidan fyremålet deira er å verta kvitt dei mange sidebandsignali som vert nytta i konvensjonell PCI (sjå til dømes [74]).

[2]Eit nett med berre to kommuniserande nodar

Programvarelaget
Programvarelagi genererar lese- og skrivefyrespurnader som vert transporterte til transaksjonslaget.

Transaksjonslaget
Transaksjonslaget tek i mot lese- og skrivefyrespurnader frå programvarelagi og lagar request-pakkar som vert vidaresende til link-laget. Alle requests vert implementerte som split-transaksjonar og sume request-pakkar krev difor ein response-pakke.

Datalink-laget
Link-laget legg til sekvensnummer og CRC til desse pakkane for å skapa ein pålitande overføringsmekanisme.

Det fysiske laget
Det grunnleggjande fysiske laget er sett i hop av ein dobbel simplex kanal implementert som eit sendarpar og eit mottakarpar. Sendarparet og mottakarparet er samla kalla for ein lane. Den opphavlege bandbreiddi på 2.5 GB/sek gjev ei nominell bandbreidd på om lag 250 MB/sek i kvar retning pr lane. So snart overhead er teke med i reknestykket, kan ein seia at om lag 200 MB/sek er brukande til dataoverføring. Denne rata er mykje høgare enn det ein kunne ha oppnådd med ein gamaldags 33MHz og 32 bits PCI-buss. Og ulikt PCI, der bandbreiddi var delt mellom devices, er dette bandbreidd som vert tiletla kvar device.

Variabel bandbreidd
Med PCI Express kan kvar link skalerast opp i bandbreidd ved å leggja til fleire lanes. Spesifikasjonen definerar breidder på x1, x2, x4, x8, x16 og x32 lanes. Bandbreiddi åt ein PCI Express link kan skalerast opp lineært ved å leggja til fleire lanes. Innkomande datapakkar kan delast opp i bytes som vert sende med kvar sin lane. Kvar byte vert då overførd i samsvar med eit 8b/10b-kodeskjema (sjå til dømes [75]). Det er dette kodeskjemaet som gjer det mogleg å henta klokka ut or signalet.

Xilinx PCI Express Endpoint
Xilinx har utvikla sin eigen IP-kjerne for PCI Express, kalla PCI Express Endpoint [76]. Kjernen har støtte for éin eller fire lanes. Den lokale sida av Endpoint-kjernen implementerar eit vanleg parallellbussgrensesnitt, mykje likt den same einfelde filosofien som vert nytta i Xilinx LogiCORE PCI (eller Altera PCI MegaCore for den saks skuld). På dette nivået er det soleis ingen nye konsept (seriegrensesnitt, split-transaksjonar og so bortetter) som må innarbeidast i brukarapplikasjonen.

Konsekvensar for HLT-RORC
Dersom HLT-RORC skal nytta seg av PCI Express, lyt det i so fall utformast ny maskinvare som høver med dei mekaniske spesifikasjonane åt PCI Express. Det må dessutan leggjast inn ein ny IP-kjerne som gjer det mogleg for HLT-RORC å interface PCI-grensesnittet med vanlege

parallelle signal. På programvaresida treng ein derimot ikkje å gjera nokon endringar. Med overgangen frå Altera-teknologi til Xilinx-teknologi har me alt sett at det ikkje er so mykje som skal til for å byta ut ein bussinterface med ein annan.

7.2 Konklusjon

Eg har i dette arbeidet utvikla både firmware, programvare og maskinvare for eit høgenergifysikk-datainnsamlingskort. Dette datainnsamlingskortet er i stand til å initiera eigne transaksjonar på PCI-bussen og det kan nytta interrupt-signalisering til å kommunisera med CPU'en på ein mest mogleg effektiv måte. På same måten er drivaren og brukarapplikasjonane utforma for å kunna utnytta CPU-tidi optimalt. Dei ulike modulane har vorte testa og verifiserte med den maskinvara som har vore tilgjengeleg.

I arbeidet har me sett at eit datainnsamlingskort for høgenergifysikk set heilt eigne krav både til maskinvare og programvare. Skal ein få fullnøyande informasjon om dei høgenergetiske kollisjonane, lyt ein taka hand om ovstore datamengder. Dette krev nøye gjennomtenkt utforming på alle nivå:

Systemet må kunna skalerast
Same kor effektivt me utformar systemet vårt, er datastraumane mykje større enn det ein einskild datamaskin kan klara. Systemet vårt må difor ikkje berre vera so effektivt at me kan tyna mest mogleg or av konvensjonell maskinvare og programvare, det lyt dessutan vera skalerande slik at me kan leggja til so mange nodar me treng for å fullføra den turvande innsamlingi og prosesseringi av kollisjonsdata frå detektoren. Sidan dei ulike eventfragmenti ikkje kan analyserast isolert, lyt drivarar og brukarapplikasjonar ha strategiar for distribuert prosessering av eventdata.

Systemet må kunna henta ut fysiske variablar og filtrera bort uinteressante hendingar i sanntid
Som me har sett trengst det lågnivå-triggermekanismar som styrer kva tid systemet skal byrja å sampla data. Desse avgjerdene vert tekne før data kjem so langt som til HLT-RORC sin DIU-interface. På eit høgare nivå lyt data gå gjennom sanntidsprosessering før informasjonen vert lagra på disk. Fyrste del av denne prosesseringa kan gjerast på sjølve HLT-RORC-kortet med di denne prosesseringa ikkje skil seg på kva data som er mottekne av andre HLT-RORC-kort. På neste nivå lyt ein nytta distribuerte algoritmar der dei ulike nodane samarbeider om å prosessera heile events.

Systemet må kunna implementera ei mest mogleg effektiv arbeidsdeling mellom CPU og dedikert maskinvare
HLT-RORC-korti er som nemnt i tidlegare kapittel plasserte i eigne nodar kalla frontend-prosessorar. Omframt dataprosessering som cluster finding og Hough-transform er det HLT-RORC si oppgåve å få lagt mottekne data i eit eige event memory slik at det vert tilgjengeleg for vidare prosessering hjå frontend-prosessoren. Sidan CPU'en åt frontend-prosessoren skal nyttast til å prosessera innkomne data, kan me ikkje hefta denne prosessoren meir enn det som er høgst turvande. Med innebygd DMA-kontroller på HLT-RORC kan datainnsamlingskortet flytta data sjølv, utan hjelp frå prosessoren.

Eit minimum av kommunikasjon mellom CPU'en og datainnsamlingskortet er påkravt. Til dømes må prosessoren få melding om at nye eventdata er vortne overførde til frontend-prosessoren sitt main memory. Her kan datainnsamlingskortet nytta asynkrone interrupt-signal til å melda frå

til prosessoren. På denne måten slepp prosessoren å nytta tid på å kontrollera om datainnsamlingskortet har levert nye data.

Prosessar i programvare lyt synkroniserast slik at dei ikkje kastar bort CPU-tid på venting og kopiering
Medan DMA og interrupts er høvelege mekanismar til å driva datautveksling og synkronisering på maskinvarenivå, er shared memories og semaforar tilsvarande mekanismar til å driva datautveksling og synkronisering på programvarenivå. Shared memories gjev ulike prosessar høve til å sjå dei same minnelokasjonane. I staden for å nytta ressursar på uturvande kopiering, kan prosessane utveksla peikarar til minnelokasjonar. Og i staden for at prosessane skal kasta bort CPU-tid på å venta på data, syter semaforane for at ventande prosessar gjev frå seg CPU'en, til dess at dei produserande prosessorane har gjeve dei noko å tyggja på.

Systemet må ha ein strategi for jording, avkopling og klokking
Som synt i kapittel 1 ynskte me for skuld bandbreiddi å nytta 66MHz PCI-buss til datainnsamlingskortet. Den snøgge arbeidsfrekvensen sette eigne krav til maskinvara, både på krinskortnivå og på FPGA-nivå. På krinskortnivå laut det utarbeidast ein fornuftig strategi for powerplan, jordplan og avkopling. På FPGA-nivå laut me partisjonera opp i ulike klokkedomene for at PCI-kjernen skulle møta kravet til arbeidsfrekvens.

Systemet må vera robust utforma
Dei resultati ein fær med simulering er ikkje betre enn den testbenken ein har sett i hop på fyrehand. I den verkelege verdi kan systemet koma ut for ei rad ulike situasjonar som ein ikkje hadde rekna med. Då gjeld det at den opphavlege arkitekturen er utforma so generelt som råd slik at nye mekanismar kan leggjast til utan store endringar i verkemåten. I underkapittel 4.2 har eg synt at det opphavlege RORC-designet vanta handtering for unormale avbrot som retry og latency timer expiration. Til lukka fanst det mekanismar for feilhandtering på HLT-RORC som var ålmenne nok til at desse nye funksjonane kunne leggjast inn utan serleg omgjering av den grunnleggjande strukturen.

Ombruk av maskinvarekjernar lettar implementasjonen
Overflyttingi av HLT-RORC frå Altera-teknologi til Xilinx-teknologi synte at dei fleste kommersielle IP-kjernar tilbyd greide grensesnitt som er lette å kopla seg til. Designendringane som laut til for å byta frå PCI MegaCore til LogiCORE PCI var minimale. Truleg vann me mykje på å kunna instansiera ferdige IP-kjernar framfor å konstruera dei sjølve, ikkje minst når det kom til den tidi me sparde på å sleppa å verifisera designet vårt mot alle dei ulike logiske og elektriske kravi som PCI-standarden set. Soleis kan ein konstatera at bruken av ferdige IP-kjernar gjev konstruktøren høve til å konsentrera seg om funksjonaliteten åt applikasjonen framfor å nytta ressursar på det Sisyfos-arbeidet det elles hadde vore å kontrollera det ferdige produktet mot alle slags ulike standardar kvar gong det skal utformast eit nytt VHDL-design.

Tillegg A

Laboratory exercise with APEX20KE400

This exercise is a walkthrough on building and running the HLT-RORC firmware and software on the APEX 20KE400 FPGA.

A.1 Simulation in ModelSim

1. Extract HLT-RORC.ZIP into your working directory.
2. Set environment variable PROJECT_PATH equal to /my_working_directory/HLT-RORC/modelsim.
3. Start ModelSim, then open project file HLT-prosjektfil.mpf in directory HLT-RORC/modelsim.
4. Map the simulation libraries. You can do this by typing from the command line (in the following order):

```
vlib work
vmap altera_pci /pakke/mgc/altera/vhdl/altera_pci
vmap lpm /pakke/mgc/altera/vhdl/220model
vmap altera_mf /pakke/mgc/altera/vhdl/altera_mf
```

If the compiled simulation libraries have become obsolete, you will have to compile the libraries yourself. This can be done by typing (in the following order):

```
vcom -work altera_mf altera\_mf_components.vhd
vcom -work altera_mf altera_mf.vhd
vcom -work lpm 220pack.vhd
vcom -work lpm 220model.vhd
```

1. The mstr_tranx.vhd module contains the master transactions controlling the simulation scenario. To run the simulation with interrupt transfers, set the variable int_or_not equal to 1. To run the simulation with DMA transfers only, set the variable to zero.

2. Now you should compile your testbench with the design into your library work. The testbench could then be loaded by typing vsim -t ps work.testbench from the ModelSim command line (not specifying the resolution in ps will result in a a fatal error). To run the simulation with 32 bit PCI bus, you could simply type *force -freeze /testbench/ack64n 1* before running simulation.

3. Run the simulation by typing run [time in ns].

(You can get rid of all the warnings if you choose Simulate -> Simulation Options -> Assertions -> Ignore assertions for warning from the menu line)

A.2 Synthesizing, technology mapping and programming

1. Open Precision RTL Synthesis. Choose File -> New Project. Set the working directory to whatever you want to.

2. Add all the required VHDL files to you project. Make sure local_side.vhd is the last added file (The last added file will automatically be set as top of design).

3. Choose Setup Design to set the following options: APEX 20KE as technology, EP20K400EFC672 as device and -2X for the speed grade. The design frequency shold be 40 MHz.

4. Click Compile, followed by Synthesize.

5. Close Precision RTL Synthesis. Make sure to anser "Yes"when asked about saving the current implementation.

6. A subdirectory called local_side_impl_1 has now been created in your working directory. Copy the content of local_side_impl_1 to /your_working_directory/HLT-RORC/LS/run_1. Answer yes when questioned to overwrite existing data.

7. Open top_module.qpf from Quartus. Convert the project file to the new Quartus file format if asked to do so.

8. If necessary, choose Tools -> SignalTap Logic Analyzer and update the SignalTap file to the new file format.

9. Choose Assigments -> Settings -> User Library and add the correct library path (/your_working_directory/HLT-RORC/pci_compiler-v2.1.1/lib). The newer versions of the pci core cannot be used, since the APEX20KE has been abandoned from further development.

10. Choose Processing -> Start Compilation to compile the design (this will take some time).

11. Choose Tools -> Programmer. Now choose JTAG as MODE. To make your programming file come through the scan chain, you have to add the device EPM7256B. The EPM device has to be the first on the list.

12. From the file menu you can now choose Create/update -> Create JAM, SVF or ISC file.
13. SSH to pc-rcu.ift.uib.no. Enter the directory where the .jam file has been saved (should be /your_working_directory/HLT-RORC/Quartus)
14. type jamplayer -p378 -aCONFIGURE top_module.jam.
15. Reboot your system. The HLT-RORC should now be ready and running. On pc-rcu.ift.uib.no, BAR0 will probably be mapped to FB400000. This can be checked by typing /sbin/lspci -v from the command line.

A.3 Verifying the implementation with the Vanguard Logic Analyzer

1. Start the Analyzer (BusView.exe)
2. Choose File -> New -> Analyzer Setup
3. Add this event: PCI0 in transfer mode with address FD0xxxxx.
4. Add this trigger:

```
Samling in TRANSFER mode
Store (ALL)
If (PCI0) then
        Trigger at 50% of trace
Endif
```

5. Cut and paste this text into an empty text file and save the file as testbench.scr:

```
f FB400000 FB400000 FD000000
f FB400004 FB400004 00000400
f FB400008 FB400008 FD000000
f FB40002C FB40002C FD000000
f FB400030 FB400030 FD000000
f FB400018 FB400018 00000011
f FB400020 FB400020 00000019
f FB400024 FB400024 00000001
f FB400024 FB400024 00000000
f FB400028 FB400028 AFFED00F
```

NB! This script assumes that the base address for BAR0 in the HLT-RORC is FB400000. It also assumes that the PCI base address for the Vanguard local target memory is FD000000.

6. Choose Exerciser -> Script -> Load and enter testbench.scr.

7. Choose Exerciser -> Target and enable the local target memory. Check that the Memory Window Address is consistent with the memory addresses in the script file.

8. Enter F9 to start sampling on the PCI bus. Then enter F10 to run the exerciser script. The exerciser script now runs the same scenario as in the VHDL testbench. If the HLT-RORC design is working, the analyzer should be triggered by the HLT-RORC writing bursts of data words to the local memory.

9. Enter Alt-F9 to halt the analyser after the analyzer has been been triggered. Then choose Exerciser -> Local -> Display to examine the data written to local target memory by HLT-RORC.

A.4 Verifying the implementation with the software

NB! Step 1-2 can only be done by a user with root permission. The first steps should therefore be taken care of by the supervisor, so the students doing this exercise can jump directly to point 4.

To run the software, a kernel module called psi has to be inserted into the kernel. The PSI tar.gz file can be downloaded from http://www.kip.uni-heidelberg.de/ti/HLT/software/software.html. If you download PSI version 0.8.2 from august 2003, you will have to patch psi_mod.c to make it compile on kernel version 2.4.20-30.9 or newer. A patchfile called psi_mod.patch is located in the HLT-RORC directory. Before compiling, remember to change the TOPDIR variable in the Makefile and remove -DWITHBIGPHYS if bigphys is not compiled into the kernel (and it probably isn't). Now you can install the module by entering directory psi-0.8.2/src/driver.linux and typing (in the following sequence):

```
make clean
make module
make install
make dev (or typing manually mknod /dev/psi c 100 0)
chmod 666 /dev/psi (to give normal users permission to access the device file)
```

The module can now be installed by typing insmod psi.o
(You will need root permissions to do this)
To compile the PSI API, go to directory psi-0.8.2/src/libpsi.linux and type make.

1. Enter directory HLT-RORC/src. If necessary, you will have to edit Makefile and Rules.make before compiling. Then run make all followed by make install and mknod /dev/irqsignal c 120 1. Now you can insert the irqsignal module by typing insmod irqsignal major=120.

2. Enter directory HLT-RORC/example. Compile the included files by typing make generate_irq. Remember to change the PSIDIR variable in the Makefile before compiling.

3. Run the compiled program generate_irq. The program will set up the HLT-RORC card to start interrupt-driven DMA-transfers. A signal handler located in the generate_irq program will respond to the interrupts. The program can be terminated by pressing Ctrl-C.

4. To observe the transactions with the Vanguard Analyzer, use the same events and trigger conditions that were shown in the previous example.

Tillegg B

Bussanalysatoren Vanguard

Dette kapitlet tek fyre seg bussanalysatoren Vanguard. Informasjonen er henta frå Vanguard User Guide [77].

B.1 Oversyn

Vanguard er ein logikkanalysator for PCI. Med Vanguard fylgjer det ei samling av verkty som kan nyttast til utvikling, debugging og testing av programvare og maskinvare som nyttar seg av PCI-bussen. Alle verktyi vert kontrollerte gjennom eit Windows-basert program kalla BusView. Logikkanalysatoren har eigen Ethernet-interface som gjer det mogleg å etablera kommunikasjon mellom logikkanalysatoren og BusView på TCP/IP-nivå.

Verktyi som fylgjer med logikkanalysatoren er:

1. **State Analyzer** (Sampling og analyse av signali på bussen)
2. **Protocol Checker** (Detekterar brot på protokollstandarden)
3. **Statistics Function** (Statistisk representasjon av aktiviteten på bussen)
4. **Exerciser** (Let brukaren få køyra eigne transaksjonar på bussen)
5. **Compliance Verification** (Køyrer ulike sjekkliste-scenario)

B.1.1 Maskinvare og programvare

Vanguard er sett i hop av eit dotterkort, kalla State Analyzer Module, og eit berarkort. Dotterkortet kann nyttast i ulike berarkort, som til dømes PCI-kort eller VME-kort.

BusView er eit Windows-kompatibelt program som kan kontrollera alle Vanguard-verktyi. BusView kan verta installert på det same systemet som Vanguard er installert, eller det kan verta installert på eit anna system. Sambandet mellom Vanguard og BusView kan anten vera direkte (via USB-kabel eller Ethernet-kabel) eller over eit nettverk (TCP/IP).

B.2 State Analyzer

State Analyzer er ansvarleg for sampling og lagring av trafikken på PCI-bussen. Det kan setjast opp sokalla 'trigger conditions' som styrer når analysatoren skal slutta å sampla. På same måten kan det setjast opp 'store conditions' som fortel analysatoren kva for sampla data som skal lagrast.

B.2.1 Sampling

So snart analysatoren har starta, byrjar han å samanlikna sampla data med brukardefinerte 'trigger conditions' og 'store conditions'. Lagra samples vert plasserte i eit 'trace buffer' med sirkulært minne. So snart bufferet er fullt, held lagringi fram frå byrjinga att. Dei lagra samplingane kan overførast frå trace-bufferet til BusView. So kan BusView syna fram samplingane i eit vindauga kalla 'Current Trace'.

Analysatoren kan sampla i tre ulike modi, TRANSFER, STANDARD eller CLOCK.

I Standard Mode vert det sampla frå bussen for kvar klokkeperiode. Standard Mode kan soleis gje oss detaljert informasjon frå kvar einskild transaksjon.

I Transfer Mode fær analysatoren høve til å trigga på hendingar definerte på eit høgare nivå enn det er høve til i Standard Mode. Transfer Mode gjev til dømes høve til å spesifisera kva for data som vert overførde på ein bussyklus eller om overføringi vart abroti. Dette er det ikkje høve til når ein samplar i standard mode.

B.2.2 Trigger Conditions

Triggers vert definerte i eit vindauga som BusView kallar 'TraceSetup'. Det er Trigger og Trigger Position som definerar når samplingi skal stogga. Analysatoren kan stillast inn til å trigga på einskildhendingar eller kombinasjonar av hendingar. Brukaren kan definera store og komplekse trigger conditions ved hjelp av vanlege programmeringsstrukturar som IF, THEN og ELSE i kombinasjon med teljarar og logisk operatorar.

B.3 Exerciser

Exerciseren gjev brukaren høve til å generera eigne PCI-transaksjonar med kjende karakteristikkar.

B.3.1 Scripting

Ein innebygd skripttolk gjev brukaren høve til å køyra eigne skript. Skriptet inneheld gjerne eit scenario som på førehand er prøvd i simulering. Brukt i lag med analysatoren kan exerciseren soleis nyttast til å testa ein PCI-device med det same testbenkscenarioet som vart nytta til å verifisera designet i simulering.

B.3.2 DMA-overføringar

Exerciseren har høve til å vera target eller master og køyra store eller små DMA-overføringar.

B.3.3 Local Target Memory

Exerciseren gjev full tilgang til eit lokalt I/O-vald på 256 bytes og eit lokalt memory på 8 MB. Data kan verta skrive til og lese frå som single cycle eller som zero wait state burst cycles (men med initial latency).

B.3.4 Interrupts

Exerciseren kan generera alle dei fire ulike typane interrupts (A, B, C eller D).

Tillegg C

PCI-transaksjonar med MegaCore og LogiCORE

Master burst transactions og master/target single cycle transactions er dei viktigaste PCI-transaksjonane som HLT-RORC nyttar seg av. I dette tillegget jamfører me måten IP-kjernane Altera PCI MegaCore og Xilinx LogiCORE PCI har implementert desse transaksjonane. Me ser dessutan nærare på korleis ymse slags unormale avbrot vert signaliserte av dei to kjernane. Informasjonen er hovudsakleg henta frå PCI MegaCore Function User Guide[34] (Altera) og LogiCORE PCI v3.0 User Guide[37] (Xilinx).

C.1 Busstransaksjonar med Altera PCI MegaCore

Timing-diagrammi i denne seksjonen syner sambandet mellom dei viktigaste signali som er involverte i 64-bits transaksjonar med pci_mt64. Eg kjem her til å taka fyre meg nokre av dei viktigaste busstransaksjonane som vert gjorde på HLT-RORC og gjera greida for det som hender i kvar klokkeperiode.

NB! På HLT-RORC er det berre initiator-transaksjonane som er 64-bits. Target-transaksjonane er ikkje 64-bits, men 32-bits. For target-transaksjonane vert scenarioet soleis likt med det som vert skildra nedanfor, men med desse unnataki:

- I tredje adressefasen let PCI-masteren vera å slå på REQ64N
- MegaCore-funksjonen slær ikkje på ACK64N
- Den lokale sida vert informert om at transaksjonen er 32-bits med di lt_tsr[7] ikkje vert slege på når lt_framen vert slege på.

C.1.1 Single cycle memory read target transaction

I denne lesetransaksjonen fungerar pci_mt64 som target. Sjå figur C.1.

1. PCI-bussen ligg i ro.

2. Initiatoren legg adresse og kommando ut på PCI-bussen (adressefasen).

3. PCI MegaCore latchar inn adresse og kommando, og dekodar adressa for å sjekka om ho hamnar innan rekkevidd for ein av BAR'ane. I tredje klokkeperioden slær masteren av framen og req64n og slær på irdyn for å indikera at det berre er éin datafase att. Samstundes slær initiatoren på irdyn-signalet og tristatar ad-signalet.

4. Om PCI MegaCore detekterar eit adressetreff i tredje klokkeperioden, kjem han til å informera om dette på den lokale sida:

   - PCI MegaCore slær på lt_framen og bitane i lt_tsr[5...0] (som svarar til ei one-hot-dekoding av kva for ein BAR som er råka). PCI MegaCore driv transaksjonskommandoen på l_cmdo[3..0] og adressa på l_adro[31..0].
   - PCI MegaCore slær på drivarane for devseln, ack64n, trdyn og stopn.
   - lt_tsr[7] vert slegen på for å indikera at transaksjonen er 64-bits.
   - lt_tsr[8] vert slegen på for å indikera at PCI-sida av megakjernen er oppteki.

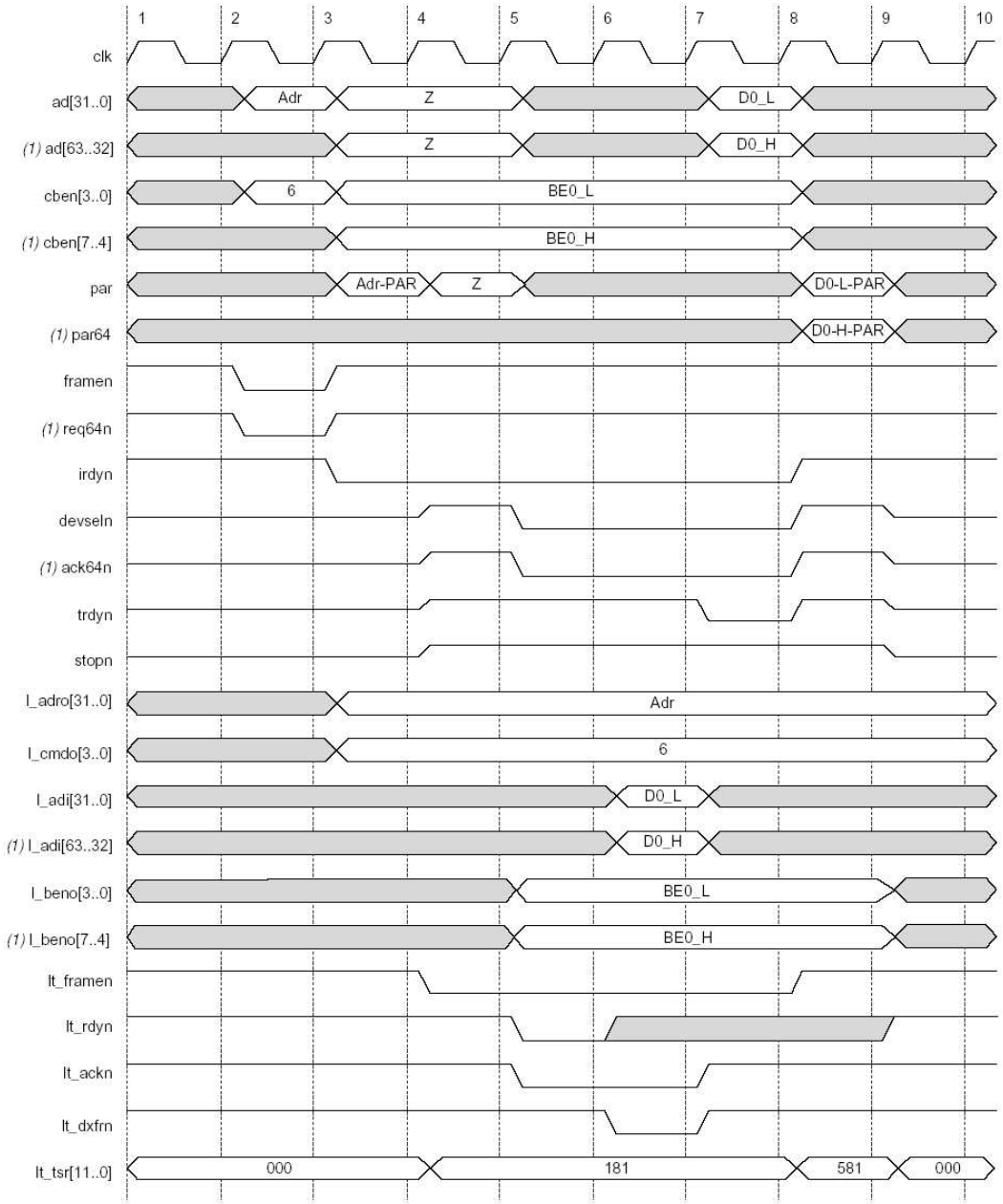
5. PCI MegaCore slær på devseln og ack64n for å krevja transaksjonen. Funksjonen driv ogso det lokale signalet lt_ackn for å indikera at han er reidug til å taka i mot data på l_adi-bussen. PCI MegaCore slær dessutan på utgangsdrivarane for AD-bussen for å sikra at han ikkje er tristata i lang tid medan han ventar på gyldige data. Jamvel om den lokale sida slær på lt_rdyn i femte klokkeperiode, kjem ikkje dataoverføringi til å henda før i sjette klokkeperiode.

6. lt_rdyn vart slege på i femte klokkeperiode som ein indikasjon på at gyldige data var tilgjengelege på l_adi-bussen. PCI MegaCore registrerar data inn i den interne pipelina på stigande flanke i sjuande klokkeperiode. Signalet lt_dxfrn indikerar om overføringi på den lokale sida var vellukka.

7. Den stigande flanken i sjuande klokkeperioden registrerar gyldige data frå l_adi og driv data på AD-bussen. Samstundes slær PCI MegaCore på trdyn-signalet for å indikera at det er gyldige signal på bussen.

8. PCI MegaCore slær av trdyn, devseln og ack64n for å avslutta transaksjonen. Desse signali vert drivne høge i éin klokkeperiode fyre drivarane vert slegne av. Den stigande flanken i åttande klokkeperiode signaliserar slutten på datafasen med di framen er slegen av og irdyn og trdyn er slegne på. I åttande klokkeperiode informerar PCI MegaCore den lokale sida at det ikkje trengst meir data med di han slær av lt_framen. lt_tsr[10] vert slege på for å indikera at dataoverføringi i fyrre klokkeperioden var vellukka.

9. PCI MegaCore informerar den lokale sida om at transaksjonen er fullført ved å slå av lt_tsr[11..0]-signali. Attåt dette syter han for at signali devseln, ack64n, trdyn og stopn vert tristata.



Figur C.1: Single Cycle Target Memory Read on Altera PCI MegaCore[35].

C.1.2 Single cycle memory write target transaction

I denne skrivetransaksjonen fungerar pci_mt64 som target. Sjå figur C.2.

1. PCI-bussen ligg i ro.
2. Initiatoren legg adresse og kommando ut på PCI-bussen (adressefasen).
3. PCI MegaCore latchar inn adresse og kommando, og dekodar adressa for å sjekka om ho fell innanfor ein BAR. I tredje klokkeperioden slær initiatoren av framen og req64n og slær på irdyn for å indikera at det berre er att éin datafase i transaksjonen. MegaCore-funksjonen nyttar tredje klokkeperioden til å dekoda adressa, og dersom adressa fell innanfor ein BAR, krev han transaksjonen.
4. Om PCI MegaCore detekterar eit adressetreff i tredje klokkeperioden, hender dette i fjerde klokkeperioden:
   - PCI MegaCore informerar den lokale sida om at han kjem til å krevja transaksjonen. Dette gjer han ved å slå på lt_framen og biten i lt_tsr[5..0] som svarar til BAR-treffet.
   - PCI MegaCore driv transaksjonskommandoen på l_cmdo[3..0] og adressa på l_adro[31..0].
   - PCI MegaCore slær på drivarane for devseln, ack64n, trdyn og stopn.
   - lt_tsr[7] vert slege på for å indikera at transaksjonen er 64-bits.
   - lt_tsr[8] vert slege på for å indikera at PCI MegaCore er oppteken med ein transaksjon.
5. PCI MegaCore slær på devseln for å krevja transaksjonen. Systemet som er på den lokale sida slær på lt_rdyn for å indikera at det er reidugt til å taka i mot data frå MegaCore-funksjonen i sjette klokkeperioden.
   For å gje den lokale sida høve til å signalisera eit retry, kjem ikkje MegaCore-funksjonen til å slå på trdyn i den fyrste datafasen med mindre den lokale sida slær på lt_rdyn. Om lt_rdyn ikkje er slege på i den femte klokkeperioden, kjem PCI MegaCore til å venta med å slå på trdyn.
6. PCI MegaCore slær på trdyn for å informera initiatoren at han er reidug til å taka i mot data.
7. Den stigande flanken i sjuande klokkeperioden registrerar gyldige data frå AD-bussen og driv data til l_dato-bussen, registrerar gyldige byte enables frå cben-bussen, og driv byte-enablane til l_beno-bussen. Samstundes slær PCI MegaCore på lt_ackn for å indikera at det er gyldige data på l_dato-bussen og gyldige byte enables på l_beno-bussen. Sidan lt_rdyn er slege på i sjette klokkeperioden og lt_ackn er slege på i sjuande klokkeperioden, kjem data til å verta overført i sjuande klokkeperioden. lt_dxfrn er slege på i sjuande klokkeperioden for å markera at data har vorte overført til den lokale sida. lt_tsr[10] er ogso slege på for å indikera ei vellukka dataoverføring på PCI-sida i den fyrre klokkeperioden. PCI MegaCore slær no av trdyn, devseln og ack64n for å avslutta transaksjonen. Saman med stopn vert desse signali drivne høgt i éin klokkeperiode før drivarane vert slegne av.

8. PCI MegaCore nullsteller alle lt_tsr[11..0]-signali av di PCI-sida har fullført transaksjonen. Han tristatar ogso kontrollsignali.

9. PCI MegaCore slær av lt_framen for å indikera andsynes den lokale sida at det ikkje er meir data i den interne pipelina.

C.1.3 Burst memory write master transaction

I denne skrivetransaksjonen fungerar pci_mt64 som master. Sjå figur C.3.

1. Den lokale sida slær på lm_req64n for å beda om ein 64-bits transaksjon.

2. PCI MegaCore legg reqn lågt for å be buss-arbiteren om å få kontroll med bussen. Samstundes slær han på lm_tsr[0] for å indikera andsynes den lokale sida at masteren bed om å få kontroll med bussen.

3. Buss-arbiteren slær på gntn for å gjeva PCI MegaCore kontroll med bussen. Jamvel om figur C.3 syner at gntn-signalet gjeng på med det same og at bussen ligg i ro i det gntn gjeng på, treng ikkje dette å henda i ein verkeleg transaksjon. Fyre PCI MegaCore held fram, lyt han venta til dess gntn-signalet gjeng på og PCI-bussen ligg i ro.

4.

5. PCI MegaCore slær på utgangsdrivarane, reidug til å taka til med adressefasen.

   PCI MegaCore slær ogso på lm_adr_ackn for å indikera andsynes den lokale sida at fyrespurnaden er godteken. I den same klokkeperioden lyt den lokale sida leggja PCI-addressa ut på l_adi[31..0] og PCI-kommandoen på l_cbeni[3..0].

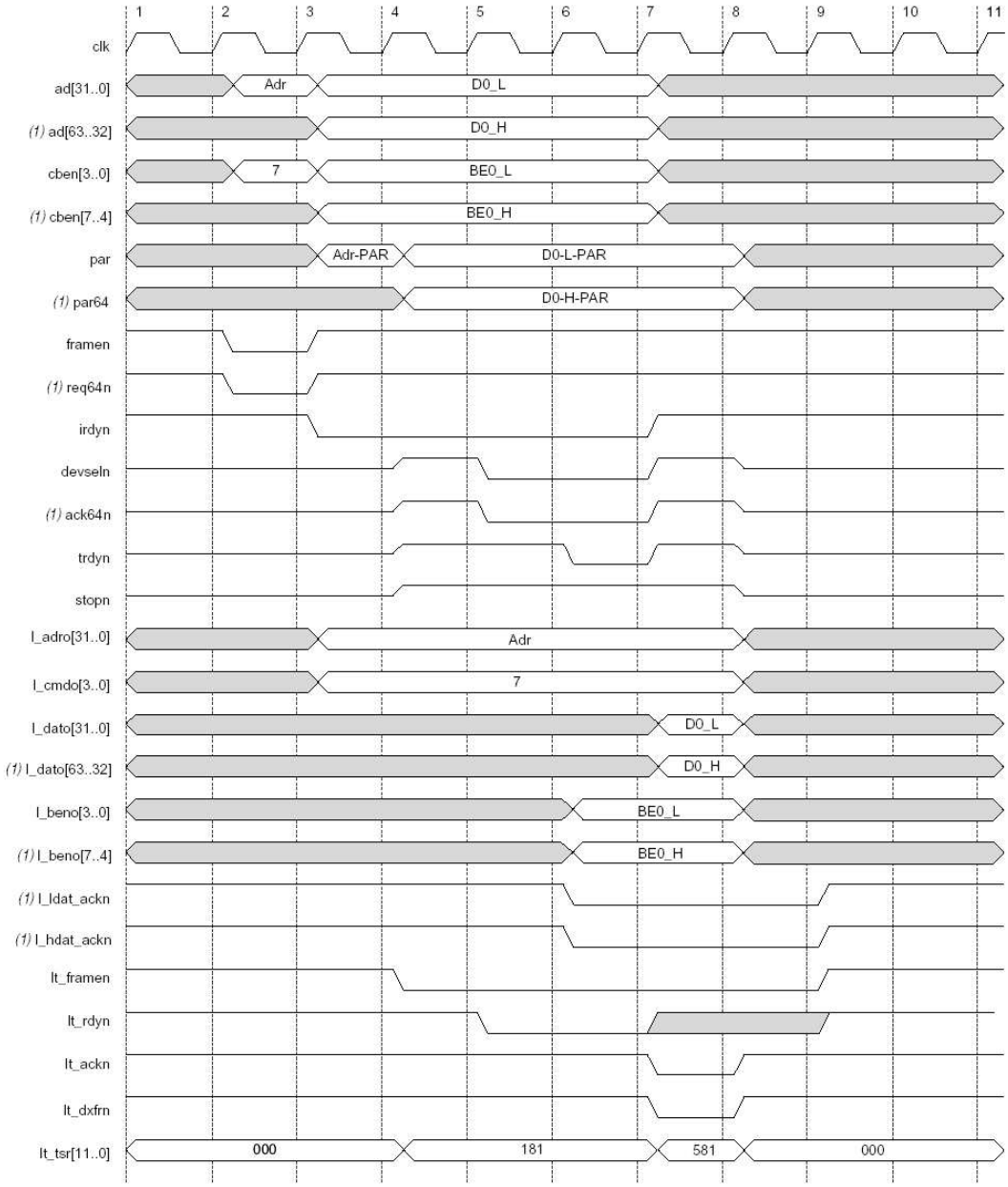
   Den lokale sida slær på lm_rdyn for å indikera at ho er reidug til å senda data.

   PCI MegaCore held fram med å halda reqn-signalet på heilt til slutten av adressefasen. Han slær ogso på lm_tsr[1] for å indikera andsynes den lokale sida at han har fenge kontroll med PCI-bussen.

6. PCI MegaCore startar skrivetransaksjonen ved å slå på framen og req64n.

   Samstundes lyt den lokale sida levera byte enables for transaksjonen på l_cbeni-bussen.

   PCI MegaCore slær på lm_ackn for å indikera andsynes den lokale sida at han er reidug til å overføra data. Sidan lm_rdyn vart slegen på i fyrre klokkeperioden og lm_ackn er på i den påfylgjande klokkeperioden, kjem PCI MegaCore til å slå på lm_dxfrn. Signali lm_dxfrn og l_hdat_ackn indikerar andsynes den lokale sida at PCI MegaCore har overført eit dataord frå l_adi-bussen.



Figur C.2: Single Cycle Target Memory Write on Altera PCI MegaCore[35].

PCI MegaCore slær på lm_tsr[2] for å indikera andsynes den lokale sida at PCI-bussen er i adressefasen.

7. Target-devicen krev transaksjonen ved å slå på devseln og ack64n (Om ack64n ikkje vert slegen på, held transaksjonen fram som ein 32-bits transaksjon). Target-devicen slær ogso på trdyn for å informera om at han er reidug til å taka i mot data.

   I denne klokkeperioden slær MegaCore-funksjonen ogso på lm_tsr[3] for å informera den lokale sida om at han er komen i datafasemodus. Funksjonen slær av lm_ackn av di den interne pipelina har gyldige data motteke frå den lokale sida i fyrre klokkeperiode, men ingen data overførde på PCI-sida. For å sikra at rette data er overførde på PCI-bussen, ventar PCI MegaCore med å slå på irdyn til dess target-devicen har slege på devseln.

8. PCI MegaCore slær på irdyn for å informera target-devicen om at funksjonen er reidug til å senda data. Sidan både irdyn og trdyn er på, vert det fyrste 64-bits dataordet overført på stigande klokkeflanke i niande klokkeperiode.

   PCI MegaCore slær på lm_tsr[9] for å indikera andsynes den lokale sida at target kan taka i mot 64-bits data. Funksjonen slær ogso på lm_ackn for å informera den lokale sida om at PCI-sida er reidug til å taka i mot data. Sidan lm_rdyn var slege på i fyrre klokkeperiode og lm_ackn gjeng på i den påfylgjande klokkeperioden, slær funksjonen på lm_dxfrn. Signali lm_dxfrn, l_ldat_ackn og l_hdat_ackn indikerar andsynes den lokale sida at PCI MegaCore har overført eit dataord frå l_adi-bussen.

9. Sidan irdyn og trdyn er på, vert det andre dataordet overført til PCI-sida på den stigande flanken i tiande klokkeperiode. PCI MegaCore slær på lm_ackn for å indikera andsynes den lokale sida at PCI-sida er reidug til å taka i mot data.

   Funksjonen slær på lm_tsr[8] i same klokkeperioden for å informera den lokale sida om at ein vellukka datafase er fullførd på PCI-bussen i fyrre klokkeperiode. Funksjonen slær ogso på lm_tsr[9] for å informera den lokale sida om at target har kravd 64-bitstransaksjonen.

10. Sidan irdyn og trdyn er på, vert det tredje dataordet overført til PCI-sida på stigande flanke i ellevte klokkeperiode.

    PCI MegaCore slær på lm_ackn for å informera den lokale sida om at PCI-sida er reidug til å taka i mot data. Sidan lm_rdyn var på i fyrre klokkeperiode og lm_ackn gjeng på i den påfylgjande klokkeperioden, slær funksjonen på lm_dxfrn.

    Den lokale sida signaliserar at transaksjonen skal avsluttast ved å slå på lm_lastn.

    PCI MegaCore slær på lm_tsr[8] i same klokkeperioden for å informera den lokale sida om at ein vellukka datafase vart fullførd i fyrre klokkeperioden.

11. Sidan lm_lastn vart slege på og ein datafase fullført i fyrre klokkeperioden, slær PCI MegaCore av framen og req64n for å signalisera den siste datafasen. Sidan irdyn og trdyn framleis er på, vert den siste datafasen fullført på stigande klokkeflanke i tolvte klokkeperiode.

    På den lokale sida slær funksjonen av lm_ackn og lm_dxfrn sidan den siste datafasen på den lokale sida vart fullført i fyrre klokkeperioden.

    Funksjonen held fram med å halda på lm_tsr[8] slik at han kan informera den lokale sida om at ei vellukka dataoverføring vart fullført i fyrre klokkeperioden.

12. PCI MegaCore slær av irdyn og tristatar framen og req64n. Target-devicen slær av devseln, ack64n og trdyn.

13. PCI MegaCore slær av lm_tsr[3] for å informera den lokale sida om at dataoverføringi er avslutta.

C.2 Busstransaksjonar med Xilinx LogiCORE PCI

Eg vil her taka fyre meg dei viktigaste busstransaksjonane som er implementerte på HLT-RORC:

C.2.1 Single cycle memory read target transaction (32 bit)

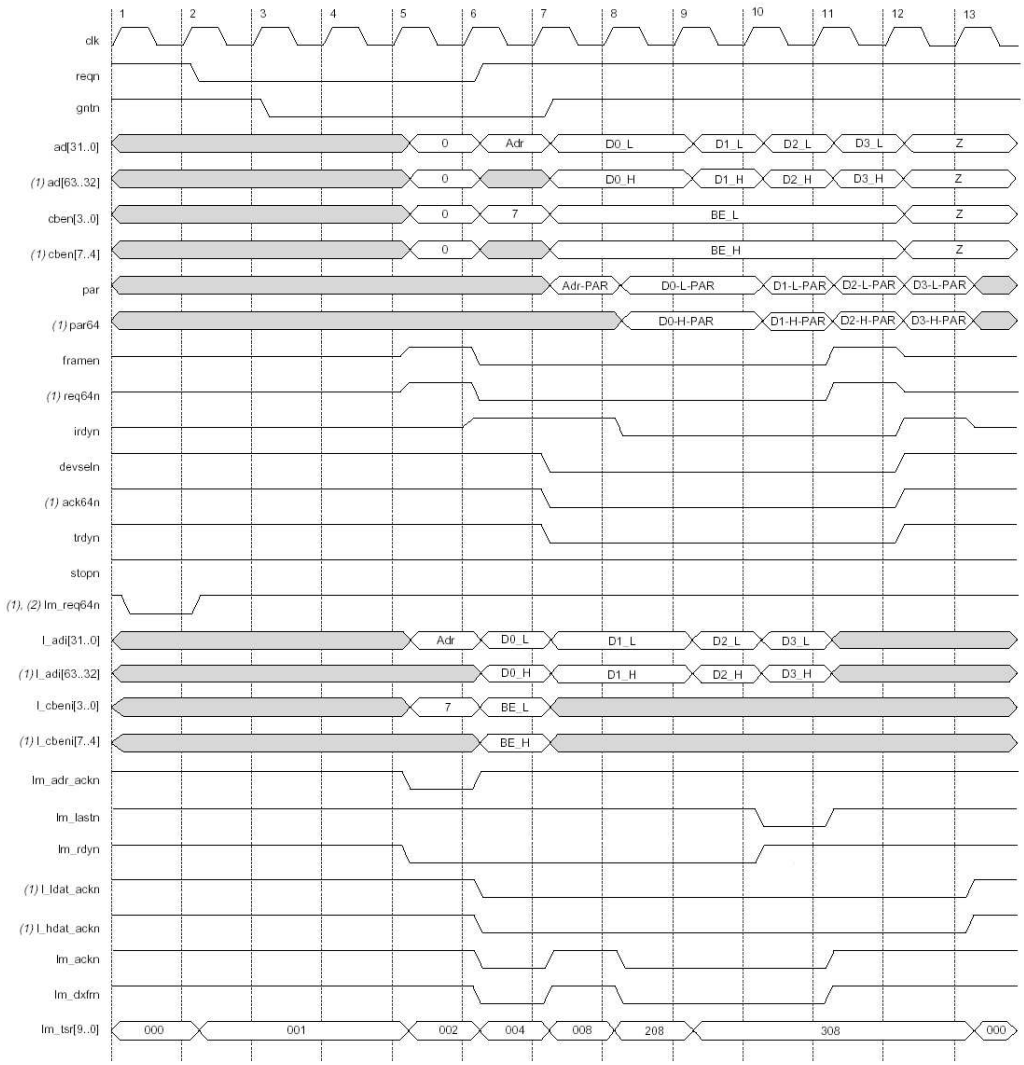
1. PCI-bussen ligg i ro.

2. Initiatoren legg adresse og kommando ut på PCI-bussen (adressefasen).

3. LogiCORE PCI latchar inn adresse og kommando, og dekodar adressa for å sjekka om ho hamnar innan rekkevidd for ein av BAR'ane. LogiCORE PCI driv transaksjonskommandoen på S_CBE og adressa på ADIO. I tredje klokkeperioden slær masteren av FRAMEN og slær på IRDYN for å indikera at det berre er éin datafase att. Samstundes tristatar han AD-signalet.

4. Om LogiCORE PCI detekterar eit adressetreff i tredje klokkeperioden, vil han slå på DEVSELN og informera den lokale sida:

    - LogiCORE PCI slær av IDLE samstundes som han slær på bitane i BASE_HIT (som svarar til ei one-hot-dekoding av kva for ein BAR som er råka). LogiCORE PCI driv adressesignalet frå fyrre klokkeperiode på ADDR. Signalet S_WRDN vil liggja lågt under heile transaksjonen for å indikera andsynes den lokale sida at det er tale om ein lesetransaksjon.
    - B_BUSY vert slege på for å indikera at PCI-sida av LogiCORE-funksjonen er oppteken med ein transaksjon.



Figur C.3: 64-bit Master Burst Memory Write on Altera PCI MegaCore[35].

5. LogiCORE PCI driv det lokale signalet S_DATA for å indikera at han er reidug til å taka i mot data. Den lokale sida svarar med å leggja ut data på ADIO.

6. LogiCORE PCI slær på TRDYN og driv AD med gyldige data.

7. LogiCORE PCI slær av TRDYN, DEVSELN og ACK64N for å avslutta transaksjonen. Desse signali vert drivne høge i éin klokkeperiode før drivarane vert slegne av.

8. LogiCORE PCI informerar den lokale sida om at transaksjonen er fullført ved å slå på S_DATA_VLD.

9. LogiCORE PCI slær på att IDLE og syter han for at signali DEVSELN, TRDYN og STOPN vert tristata.

C.2.2 Single cycle memory write target transaction (32 bit)

1. PCI-bussen ligg i ro.

2. Initiatoren slær på FRAMEN samstundes som han driv adresse og kommandoar til AD og CBE.

3. Initiatoren slær på IRDYN samstundes som han driv data og byte enables til AD og CBE. LogiCORE PCI driv adresse og kommandoar frå fyrre klokkeperiode til ADIO og S_CBE. Signalet S_WRDN vil liggja høgt under heile transaksjonen for å indikera andsynes den lokale sida at det er tale om ein skrivetransaksjon.

4. LogiCORE PCI dekodar transaksjonen og slær på DEVSELN. Samstundes driv han lågt det lokale IDLE-signalet og slær på biten i BASE_HIT som svarar til BAR-treffet. Adressa som transaksjonen retter seg mot vert klokka inn i ADDR. Signalet B_BUSY vert slege på for å informera den lokale sida om at LogiCORE-funksjonen er oppteken med ein transaksjon.

5. LogiCORE PCI slær på TRDYN for å signalisera at han er reidug til å taka i mot data. Samstundes driv han høgt det lokale signalet S_DATA for å indikera andsynes den lokale sida at han er komen inn i datafasen.

6. LogiCORE PCI driv bussignali høgt i éin klokkeperiode til før signali vert tristata. Samstundes vert gyldige data drivne til ADIO og signalet S_DATA_VLD vert drive høgt for å fortelja den lokale sida at ho lyt lesa av gyldige data.

C.2.3 Burst memory write master transaction (64 bit)

I denne skrivetransaksjonen fungerar LogiCORE PCI som initiator:

1. Den lokale sida slær på REQUEST64 for å beda om ein 64-bits transaksjon.

2. LogiCORE PCI legg REQN lågt for å be buss-arbiteren om å få kontroll med bussen.
3. Buss-arbiteren slær på GNTN for å gjeva PCI MegaCore kontroll med bussen.
4. LogiCORE PCI legg M_ADDR_N lågt for å signalisera andsynes den lokale sida at gyldig adresse og kommando lyt leggjast ut på ADIO og M_CBE.
5. LogiCORE PCI slær på FRAMEN samstundes som han driv gyldig adresse og kommando til AD og CBE. M_DATA gjeng på for å signalisera at initiatoren er i dataoverføringstilstandet. Den lokale sida lyt driva ADIO og M_CBE med data og byte enables. Sidan transaksjonen er ein burst-transaksjon, gjeng M_SRC_EN på for å signalisera at den lokale sida lyt leggja nye data ut på ADIO i neste klokkeperiode.
6. Target-devicen dekodar transaksjonen og slær på DEVSELN. LogiCORE PCI slær av M_SRC_EN av di han ikkje vil taka i mot fleire data før den fyrste datafasen er fullførd på PCI-bussen.
7. Buss-arbiteren slær av GNTN-signalet. Etter nokre klokkeperiodar svarar target-devicen med å slå på TRDYN.
8. LogiCORE PCI slær på M_DATA_VLD for å indikera andsynes den lokale sida at ein vellukka datafase vart fullførd i fyrre klokkeperioden. Samstundes slær han på att M_SRC_EN for å beda den lokale sida om å klokka inn nye data.
9. Etter nokre klokkeperiodar med burst-overføring slær den lokale sida på COMPLETE for å avslutta transaksjonen.
10. LogiCORE PCI slær av FRAMEN for å indikera at transaksjonen er komen til den siste datafasen. Samstundes vert M_SRC_EN slegen av.
11. LogiCORE PCI driv bussignali høgt i éin klokkeperiode til før signali vert tristata.

C.3 Unormale avbrot med Altera PCI MegaCore

C.3.1 Retry

I Altera PCI MegaCore vert signalet lm_tsr(5) nytta til å informera den lokale masteren om at transaksjonen har enda med eit retry. Det er då opp til den lokale masteren å ha implementert logikk som kan syta for at transaksjonen vert teken opp att.

C.3.2 Disconnect with data

I Altera PCI MegaCore vert det lokale signalet lm_tsr(7) nytta til å informera den lokale masteren om at transaksjonen har enda med eit disconnect with data.

Når Altera PCI MegaCore fungerar som target, kan den lokale sida signalisera eit disconnect ved å slå på signalet lt_discn.

C.3.3 Disconnect without data

I Altera PCI MegaCore vert det lokale signalet lm_tsr(6) nytta til å informera den lokale masteren om at transaksjonen har enda med eit disconnect without data.

C.3.4 Latency timer expiration

I Altera PCI MegaCore vert det lokale signalet lm_tsr(4) nytta til å informera den lokale masteren om at transaksjonen har enda med ein latency timer expiration.

C.4 Unormale avbrot med Xilinx LogiCORE PCI

C.4.1 Retry

Det lokale signalet CSR(36) inneheld det kombinatoriske signalet (TRDYQ_N * !STOPQ_N * !DEVSELQ_N) der TRDYQ_N, STOPQ_N og DEVSELQ_N er registrerte utgåvor av TRDYN, STOPN og DEVSELN. Dette signalet kan nyttast til å detektera retries når initiatoren er komen inn i den fyrste datafasen. I HLT-RORC har retry-deteksjon vorte implementert på den måten at eit lokalt retry-register lastar inn CSR(36) når M_DATA er slege på.

C.4.2 Disconnect with data

Det lokale signalet CSR(37) inneheld det kombinatoriske signalet (!TRDYQ_N * !STOPQ_N * !DEVSELQ_N). Dette signalet kan nyttast til å detektera disconnect with data når initiatoren er komen inn i datafasen. I HLT-RORC har deteksjon av disconnect with data vorte implementert på den måten at eit lokalt disc_wd-register lastar inn CSR(37) når M_DATA er slege på.

C.4.3 Disconnect without data

Det lokale signalet CSR(36) inneheld det kombinatoriske signalet (TRDYQ_N * !STOPQ_N * !DEVSELQ_N). Dette signalet kan nyttast til å detektera disconnect without data når initiatoren er komen inn i den andre datafasen eller seinare. I HLT-RORC har deteksjon av disconnect without data vorte implementert på den måten at eit lokalt disc_wod-register lastar inn CSR(36) når M_DATA er slege på. For å få skilt retry og disconnect without data frå einannan, kan eg nytta M_DATA_VLD til å røkja etter om den fyrre datafasen var vellukka. Dersom han var det, hadde me hatt eit disconnect without data og ikkje eit retry.

C.4.4 Latency timer expiration

Det lokale signalet TIME_OUT indikerar at den interne latency-timeren har gjenge ut og at brukarapplikasjonen har bruka opp den tilgjengelege tidi på bussen.

Tillegg D

PCI-testkortet - Montering, testing og korrigering

Dette tillegget tek fyre seg det arbeidet som vart gjort med å montera og testa PCI-testkortet. Tillegget diskuterar ogso dei ulike designfeili og produksjonsfeili som vart oppdaga og kva for endringar som vart gjorde for å korrigera desse.

D.1 Montering og funksjonstesting

Etter at åttelagskortet hadde vorte produsert hjå Elprint, vart det sendt vidare til Oslo. Hjå Universitet i Oslo fanst det utstyr som kunne nyttast til å lodda Virtex-brikka på mynsterkortet.

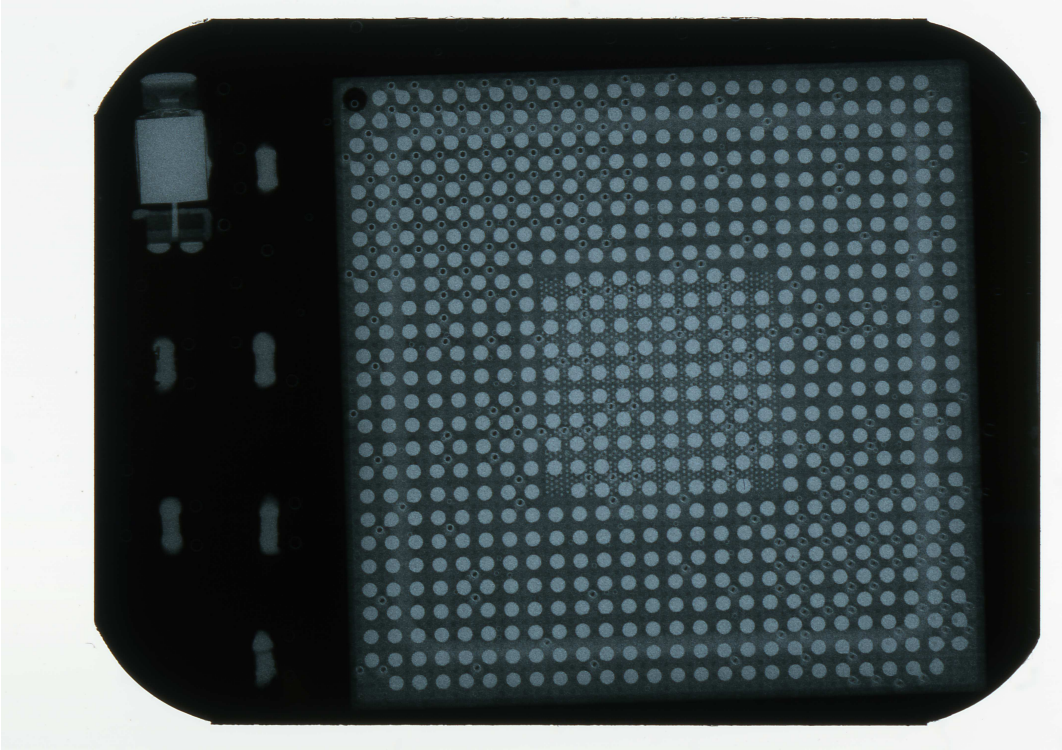
D.1.1 Produksjonsfeil

Etter at Virtex-krinsen var montert på mynsterkortet, synte det seg der at kortet hadde bøygt seg i omnen under omsmeltingi. Dei ulike materiali som substratet er sett i hop av har termiske utvidingskoeffisientar [78], og truleg var det ulik oppvarming i dei ulike lagi i substratet som hadde hadde gjeve ulik utviding og dimed gjort heile kortet vindskeivt. Ei mogleg årsak til denne samandragingi kan vera rang varmeprofil i omnen.

Med den krummingi som var på mynsterkortet kunne ein tydeleg sjå at FPGA-brikka gjekk opp frå underlaget i to av hyrni. Her var det likt til at det kunne vera brot i tilkoplingane.

Det fyrste me ville gjera var å leita etter moglege kortslutningar. Me tok difor med oss det ferdige krinskortet til Odontologisk Fakultet der det var teke røntgenfotografi av krinsen. Røntgenbileti synte at det ikkje var kortslutningar under brikka. Sjå figur D.1.

Derimot synte vidare funksjonstesting at det i to av hyrni på brikka ikkje var skikkeleg kontakt mellom loddeballane og koparbanane. Til lukka synte ein inspeksjon av koplingsskjemaet at desse tilkoplingane ikkje var so kritiske. Ei av tilkoplingane gjekk ut til ein ljosdiode, dei hine gjekk til jord.



Figur D.1: Røntgenbilete med 50 ms eksponeringstid. Dei småe ballane i midten avslører at brikka er ein flip-chip.

D.1.2 Designfeil

Ved montering av komponentar og etterfylgjande funksjonstesting kom tre ulike designfeil for dagen:

Mekanisk
Fotavtrykket for krystalloscillatoren synte seg å vera spegelvendt. Dette problemet var ikkje verre enn at oscillatoren vart montert på opp ned med ekstra tilleidningar og isolerande lag mellom oscillatoren og underlaget.

Elektrisk
JTAG-tilkoplingane vanta tilkoplingar for jord og referansespenning. Dette problemet vart løyst med strapping av dei turvande spenningane. Strapping vil seia at vanlege leidningar vert lodda på fotavtrykki for å gjeva den turvande tilkoplingi.

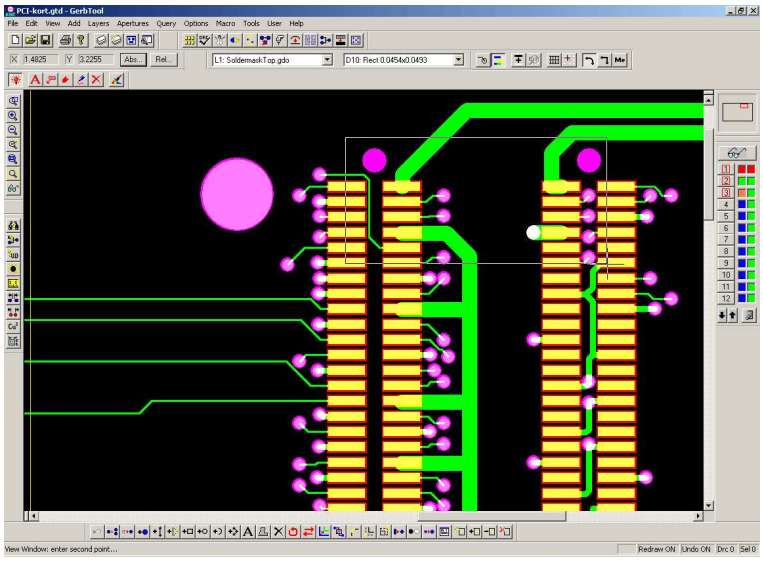
Funksjonelt
Då testkortet vart utforma, gjekk me ut frå at DIU- og SIU-konnektorane nytta dei same pinnetilordningane. Me tok difor utgangspunkt i SIU når me laga skjemasymbol for denne kon-

nektoren. Når me seinare fekk tak i datablad for DIU, såg me at me hadde teke i miss. DIU-grensesnittet vårt var soleis ikkje brukande.

D.1.3 Brot på normale design rules

Geir Frode Sørensen ved Universitet i Oslo gjorde oss elles merksame på fylgjande design rule violations:

Necking ved entring av pads



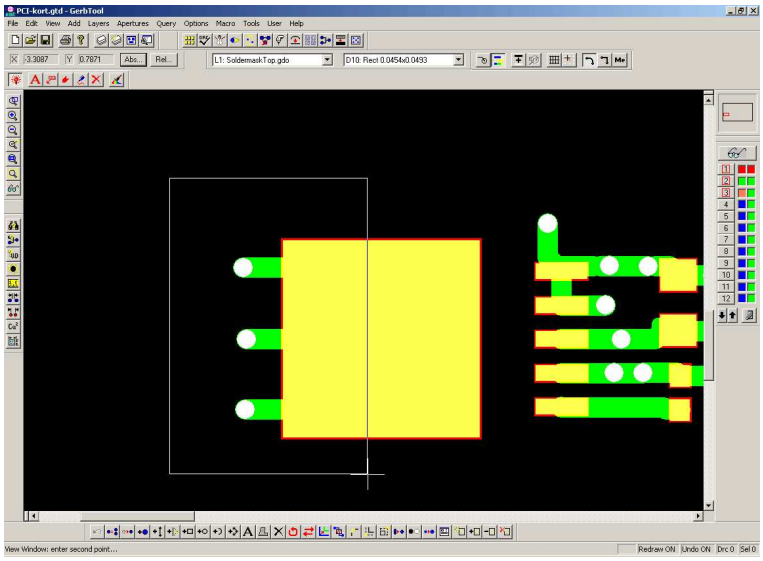
Figur D.2: Vantande necking ved entring av pads.

Sume av leidningsbanane var tjukkare enn dei paddane dei terminerte i. Dette galdt mellom anna ymse powerlinor som terminerte på konnektorpaddane åt DIU-kontakten. Her skulle den siste biten på leidningi ha vorte krympa ned til maksimalt same breidd som padden. Teknikken vert kalla for necking og reglane er til for å hindra at varmen vert ført bort med leidningi når kompenten skal loddast på fotavtrykket.

Det var dessutan brote med fyresetnadene for kva leid feite banar har lov til å gå inn på ein usymmetrisk pad på. Leidningane bør gå inn på padden i lengderetningi og ikkje på tvers. Å bryta med desse retningslinone er ikkje ulovleg, men vert av røynde fagfolk rekna som tvilsamt. Det kan til dømes verta problematisk under bylgjelodding som er svært var for varmebortleiding og i kva for retning det hender. Sjå figur D.2.

Termisk kopling til jordplan

Når pakketypar som krev kjøling vert ytemonterte so har ein to val:



Figur D.3: Varmepaddane vantar perforering med termisk via.

Det eine er å klampa ei kjøleribba oppå komponenten. Dette er problematisk av to grunnar - Det er manuelt arbeid og dinest er silisiumkjernen vend mot komponenten si metalliske underside.

Sovorne komponentar krev difor det andre valet - termisk kopling mot store plan internt i kortet, normalt powerplan. Sidan desse plani er interne so krev det at kjøleflata under komponenten vert perforert med viahol kopla ned til powerplanet. Dette hadde ikkje vorte gjort. Sjå figur D.3.

Loddemaske for BGA

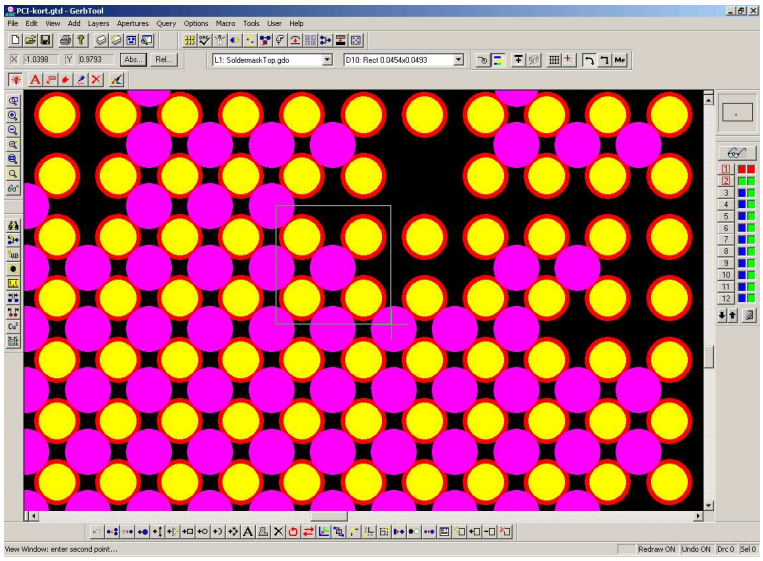
Det vanta loddestopplakk på viapaddane, og under Virtex-brikka hadde dette ført til overlappande loddemaskar. Soleis var det ein viss fåre for at tilkoplingar under brikka kunne ha kortslutta. Sjå figur D.4.

Når FPGA-brikka skal loddast, er det mogleg å setja henne rett ned på fotavtrykket og smelta om loddeballane. Denne teknikken vert kalla turrlodding. Den andre løysingi er å leggja på ein pastamaske og smørja våt loddepasta på fotavtrykket fyre ein set nedpå FPGA-brikka. Denne teknikken er best eigna til å sikra at tinnet flyt. Av same grunnen lyt me halda oss til turrlodding når loddemaskane overlappar. Noko anna hadde garantert ført til kortslutning.

D.2 Nytt testkort med korrigert utlegg

Etter at det fyrste testkortet var ferdig testa, vart det gjort ein ny iterasjon med korrigering av dei gamle skjemateikningane og det gamle utlegget. Det nye testkortet vart aldri produsert, men det nye utlegget ligg reidugt til produksjon.

Dei endringane som vart gjorde var:



Figur D.4: Overlappande loddemaskar.

- Tilkoplingar for jord og referansespenning vart sette på JTAG-headeren.
- Det spegelvende fotavtrykket åt krystalloscillatoren vart snudd.
- Loddemasken på viaholi vart fjerna slik at viaholi fær eit 'telt' av loddestopplakk over seg.
- Det vart laga nye skjemasymbol for DIU-konnektoren med korrekte pinnetilordningar.
- Tjukke leidningar vart krympa i endane slik at dei ikkje overlappa paddane dei gjekk inn på.
- Sume leidningar vart flutte slik at dei gjekk inn på paddane i lengderetningi og ikkje på tvers.

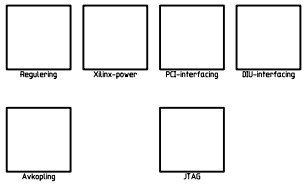
Teikningar av skjema og utlegg for det korrigerte testkortet er å finna i tillegg E.

Tillegg E

Skjema og utlegg for PCI-testkortet

Dette tillegget inneheld skjema for:

1. Avkoplingsnettverk
2. Tilkoplingar for DIU-interfacing
3. Tilkoplingar for programmering og klokke
4. PCI-interfacing
5. Regulatorar
6. Tilkoplingar for power og jord
7. Toppskjema for heile hierarkiet
8. Teikning av utlegg (jord- og powerplan er ikkje synte)



DWG. NO.	SH.	REV

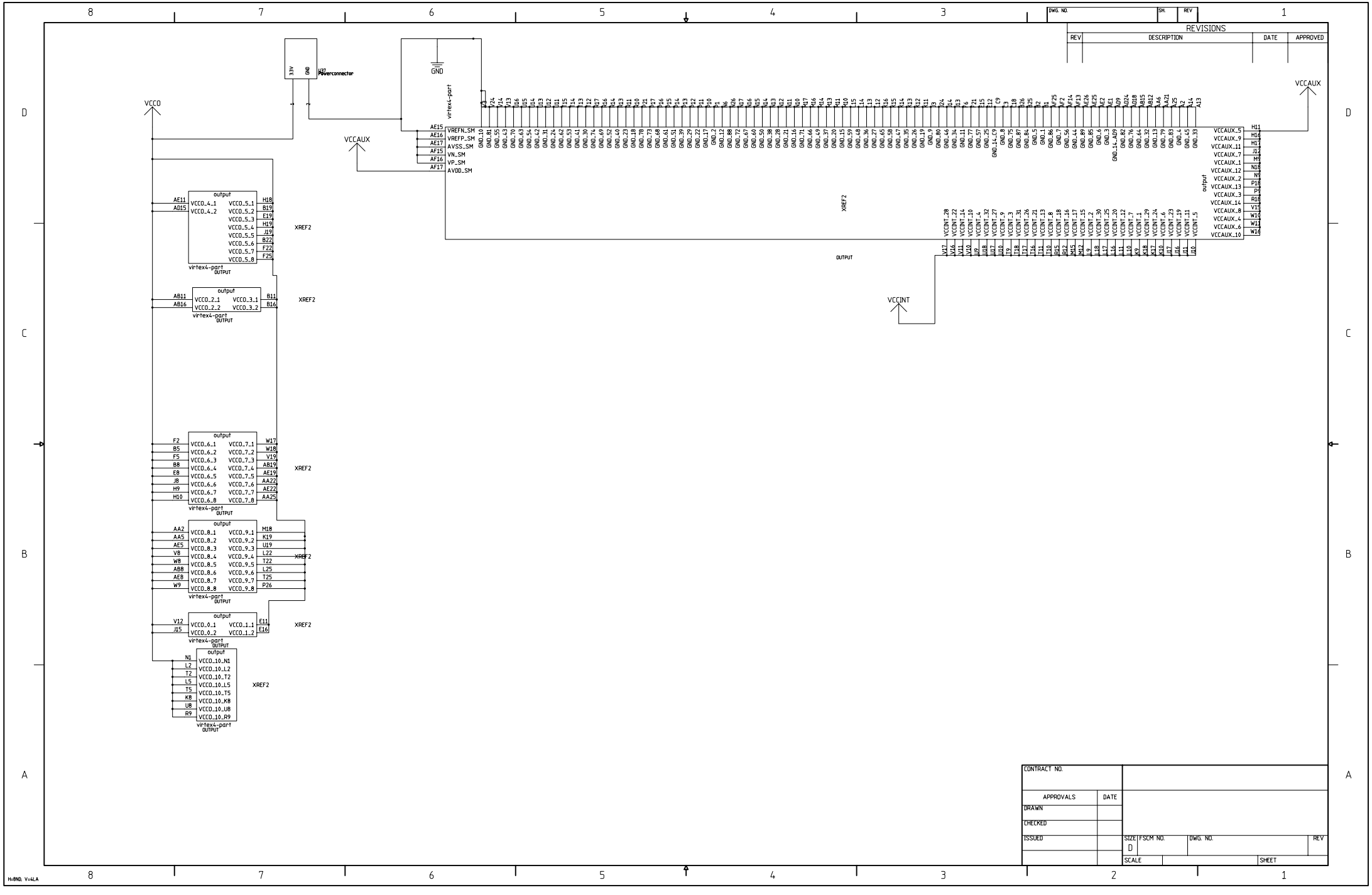
REVISIONS

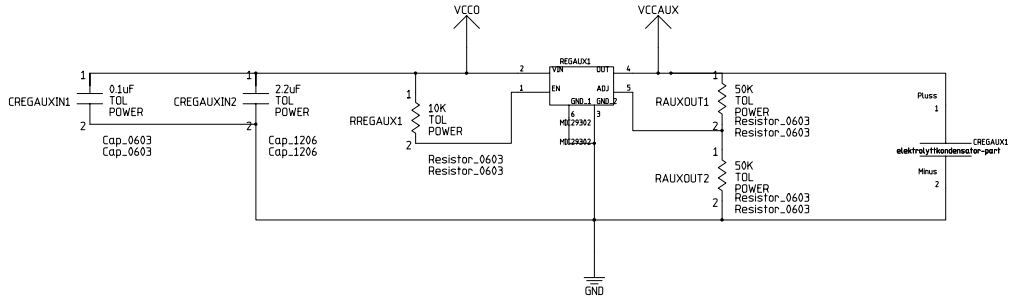
REV	DESCRIPTION	DATE	APPROVED

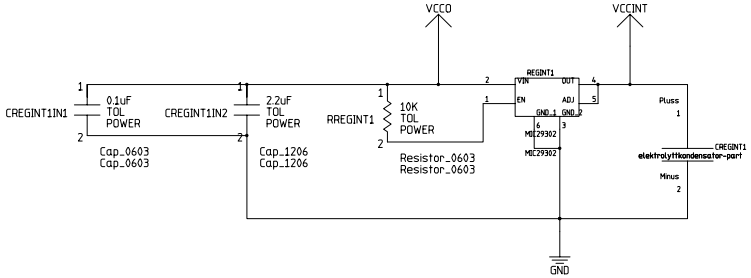
CONTRACT NO.	
APPROVALS	DATE
DRAWN	
CHECKED	
ISSUED	

SIZE	FSCM NO.	DWG. NO.	REV
D			
SCALE		SHEET	

H:BND, V:4LA


DWG. NO.
SH.
REV
REVISIONS
REV
DESCRIPTION
DATE
APPROVED
VCCO
VCCAUX
VCCINT
GND
3.3V
K37
Powerconnector
virtex4-part
output
XREF2
VREFN_SM
VREFP_SM
AVSS_SM
VN_SM
VP_SM
AVDD_SM
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
SIZE
FSCM NO.
DWG. NO.
REV
D
SCALE
SHEET





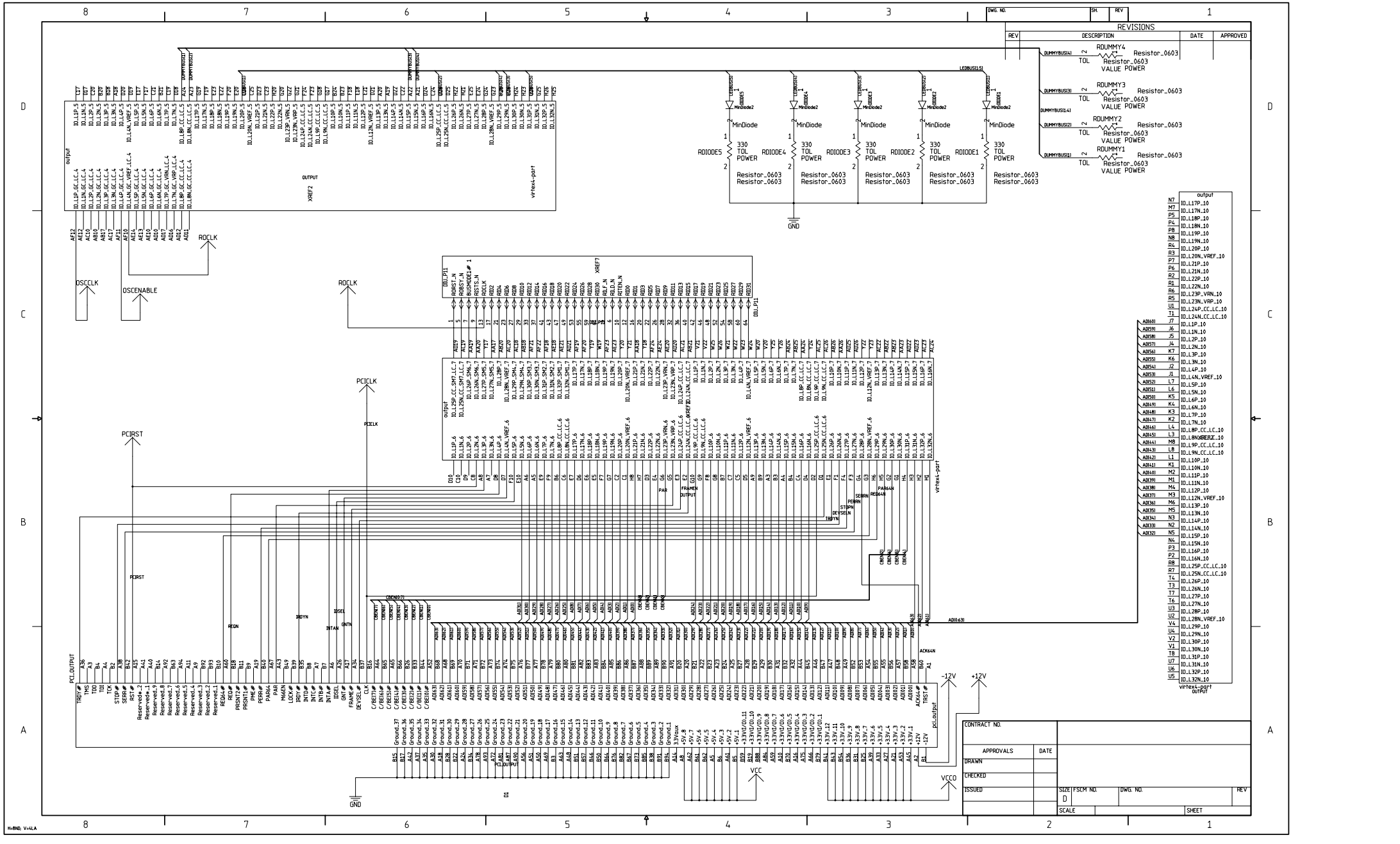
REVISIONS			
REV	DESCRIPTION	DATE	APPROVED

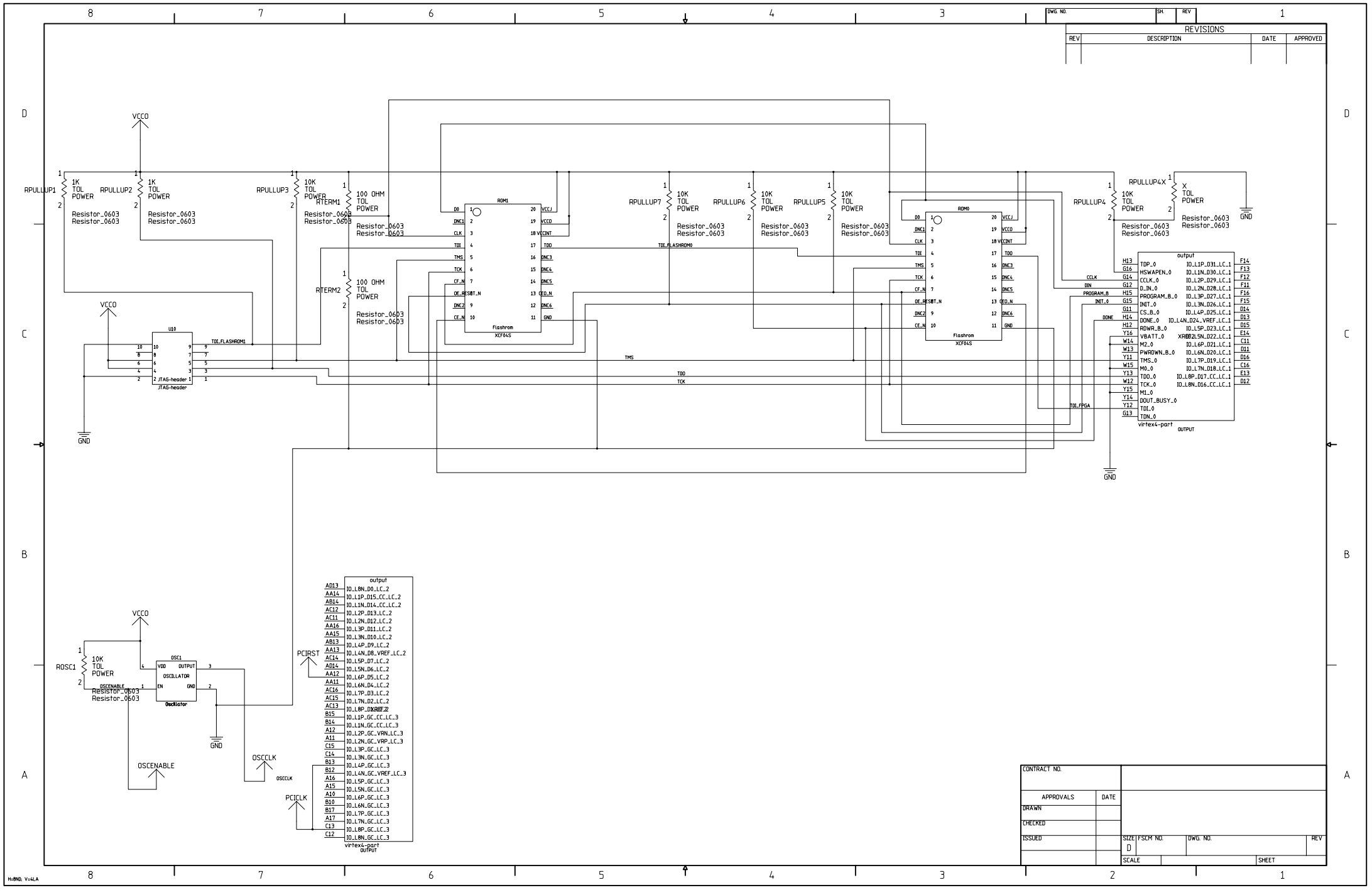
DWG. NO.	SH.	REV

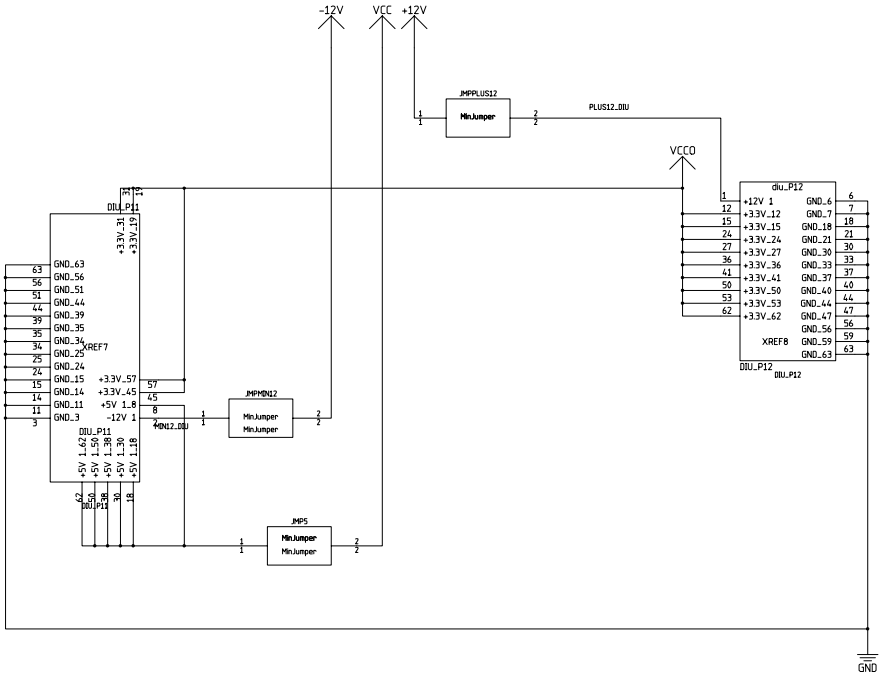
CONTRACT NO.	
APPROVALS	DATE
DRAWN	
CHECKED	
ISSUED	

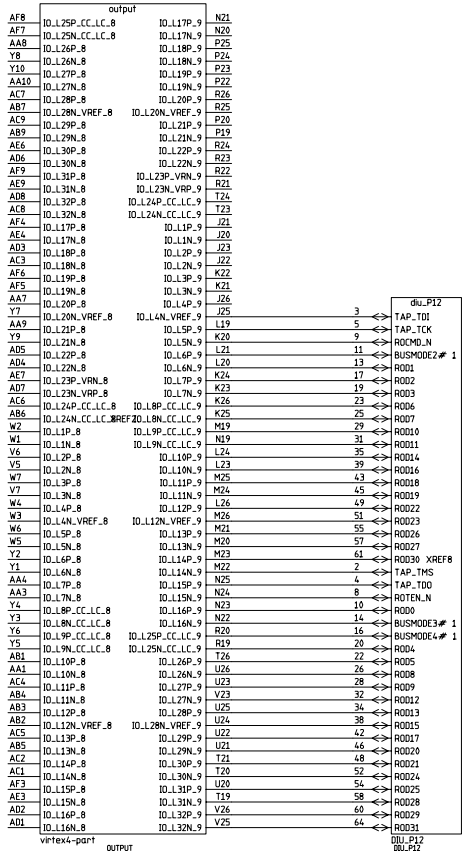
SIZE	FSCM NO.	DWG. NO.	REV
D			
SCALE		SHEET	

H:BND; V:4LA


REVISIONS
REV
DESCRIPTION
DATE
APPROVED
DWG. NO.
SH.
REV
RDUMMY4
RDUMMY3
RDUMMY2
RDUMMY1
Resistor_0603
VALUE POWER
TOL
RDIODE5
RDIODE4
RDIODE3
RDIODE2
RDIODE1
330
MinDiode
MinDiode2
GND
VCC
VCCO
-12V
+12V
ROCLK
OSCCLK
OSCENABLE
PCICLK
PCIRST
PCI_OUTPUT
virtex4-part
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
SIZE
FSCM NO.
DWG. NO.
REV
SCALE
SHEET


DWG. NO.
SH.
REV
REVISIONS
REV
DESCRIPTION
DATE
APPROVED
VCCO
RPULLUP1
RPULLUP2
RPULLUP3
RPULLUP4
RPULLUP4X
RPULLUP5
RPULLUP6
RPULLUP7
1K TOL POWER
10K TOL POWER
100 OHM TOL POWER
X TOL POWER
Resistor_0603
RTERM1
RTERM2
ROM1
ROM0
Flashrom
XCF04S
TDI_FLASHROM0
TDI_FLASHROM1
U10
JTAG-header
TMS
TDO
TCK
GND
CCLK
DIN
PROGRAM_B
INIT_0
DONE
TDI_FPGA
output
virtex4-part
OUTPUT
ROSC1
OSC1
OSCILLATOR
Oscillator
OSCENABLE
OSCCLK
PCIRST
PCICLK
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
SIZE
FSCM NO.
DWG. NO.
REV
SCALE
SHEET





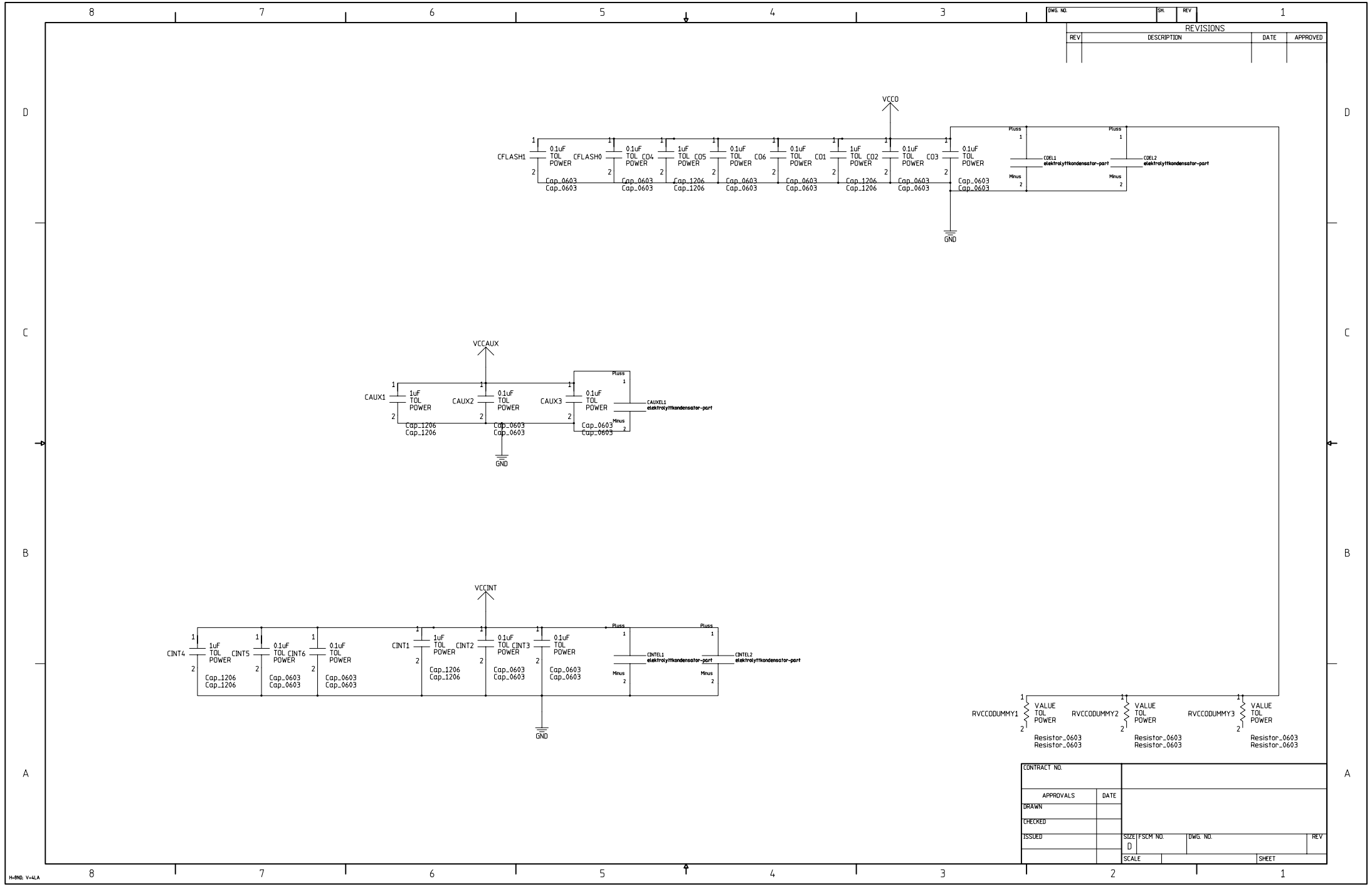
REVISIONS			
REV	DESCRIPTION	DATE	APPROVED

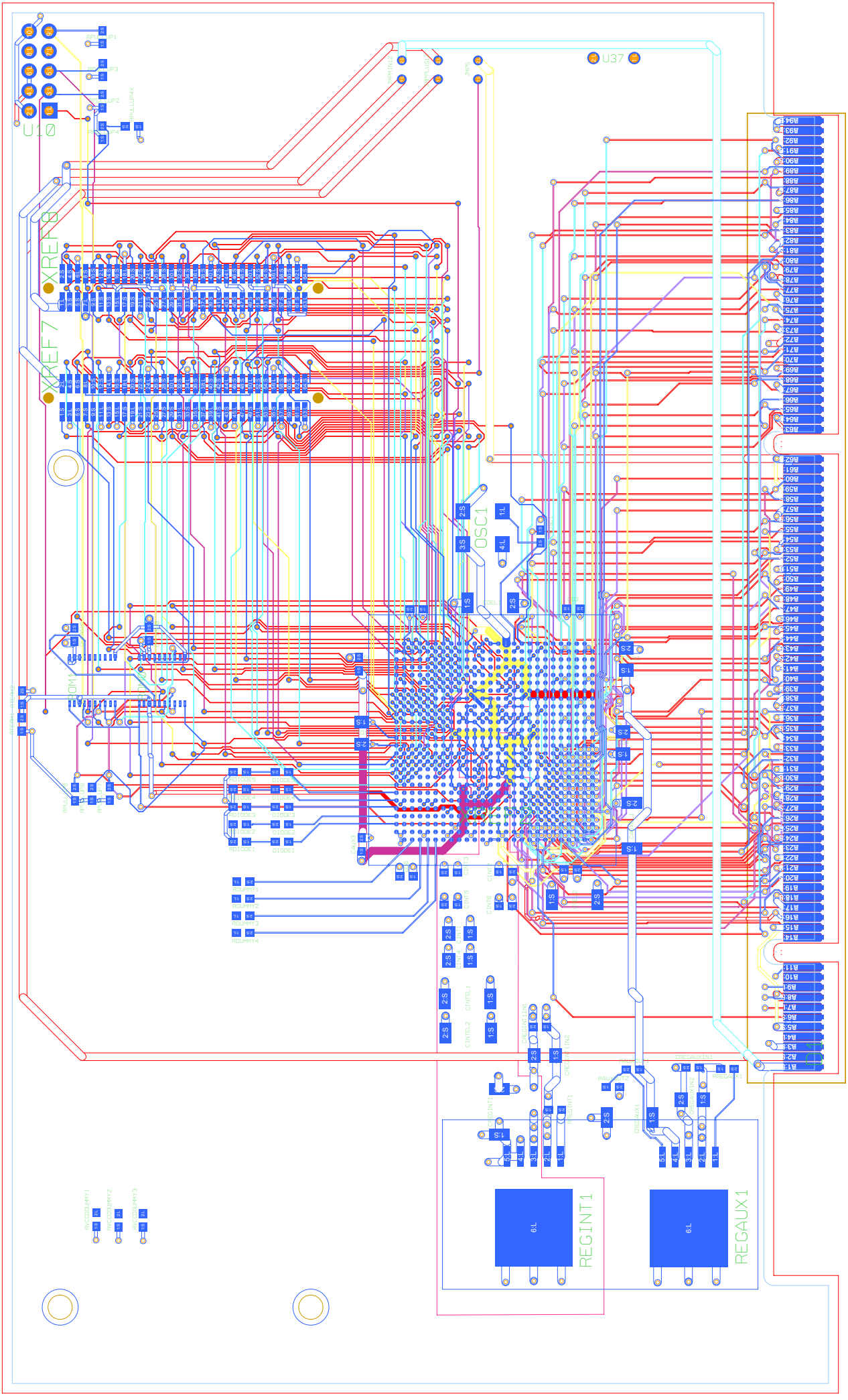
DWG. NO.	SH.	REV

CONTRACT NO.	
APPROVALS	DATE
DRAWN	
CHECKED	
ISSUED	

SIZE	FSCM NO.	DWG. NO.	REV
D			
SCALE		SHEET	

H:8ND; V:4LA


DWG. NO.
SH.
REV
REVISIONS
REV
DESCRIPTION
DATE
APPROVED
VCCO
CFLASH1
0.1uF
TOL
POWER
Cap_0603
CFLASH0
CO4
1uF
CO5
Cap_1206
CO6
CO1
CO2
CO3
Pluss
Minus
COEL1
COEL2
elektrolyttkondensator-part
GND
VCCAUX
CAUX1
CAUX2
CAUX3
CAUXEL1
VCCINT
CINT4
CINT5
CINT6
CINT1
CINT2
CINT3
CINTEL1
CINTEL2
RVCCODUMMY1
RVCCODUMMY2
RVCCODUMMY3
VALUE
Resistor_0603
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
SIZE
FSCM NO.
DWG. NO.
REV
D
SCALE
SHEET
H:BND; V:4LA


U10
U37
XREF0
XREF7
OSC1
REGINT1
REGAUX1

Tillegg F

Files

Table F.1 lists the different design files used for the HLT-RORC firmware and software. The files can be found on the CD-ROM that is shipped with this thesis:

- The VHDL files can be found in directory HLT-RORC-XILINX/vhdl/example/src.
- The prototype software files can be found in directory HLT-RORC-XILINX/example.
- The driver, API and application files can be found in their respective subdirectories located under directory RORCDRIVER (see subsection 5.2 for further details).
- The VHDL files, constraint files and synthesis script files for the LEON3 design can be found in directory LEON3.

See the respective README files of directories HLT-RORC-XILINX, RORCDRIVER/src and LEON3 for more information on how to build the different applications from the source code.

The schematic, layout and production files for the PCI card can be found in the following directories:

- The schematic and layout files for the first version of the PCI card can be found in directory PCI_CARD_V1
- The schematic and layout files for the second version of the PCI card can be found in directory PCI_CARD_V2
- The production file for the first version of the PCI card can be found in directory PRODUCTION_FILES_V1

Tabell F.1: Description of files

VHDL Files	**Description**
pcim_top_r.vhd	Top level design. Instantiates the LogiCORE PCI module, a LogiCORE configuration module, the DCM modules and the local side modules.
cfg_ping.vhd	LogiCORE configuration module. Configuration signals are attached to the PCI core through a 256-bit wide std_logic_vector.
local_side.vhd	Encloses all our self-defined logic.
HLT_RORC_definitions.vhd	Contains the address mappings of the different HLT-RORC registers.
master_ctrl.vhd	Controls the initiator transactions of HLT-RORC.
target_ctrl.vhd	Controls the target transactions of HLT-RORC.
csr_block.vhd	Contains a register with control and status information.
dma_engine.vhd	Signalizes the master_ctrl unit when to start transferring event data on the PCI bus.
IRQ_signaler.vhd	Used for generating interrupt signals on the PCI bus.
ddl_if.vhd	DDL interface used for receiving incoming events.
pattern_generator.vhd	Used for generating event data internally.
mem_if.vhd	Interface for reading event data from an external memory. The external memory can be filled with simulated event data through the pci2mem interface.
pci2mem.vhd	Interface between the PCI core and the external memory.
dma_fifo.vhd	FIFO for buffering externally received or internally generated event data.
packetizer.vhd	Splits event data into packets of 64-bit words before putting them on the PCI bus.
event_monitor.vhd	Keeps track of the number of transferred events.
Prototype Software Files	**Description**
generate_irq.c	A C program using PSI and irqsignal to run through a complete DMA transferring scenario.
Driver Files	**Description**
rorcdriver.h	Header file for the API.
rorcdriver_mod.h	Header for for the kernel module.
rorcdriver_mod.c	Source code for the kernel module.
rorcdriver_api.c	Source code for the API.
generate_irq.c	A sample application using the rorcdriver API for reading and processing incoming event data from HLT-RORC.
LEON3 Files	**Description**
leon3mp.vhd	Top module for the GRLIB-based microprocessor system.
config.vhd	Configuration module storing VHDL constants used by leon3mp.vhd.
ahbrom.vhd	ROM module containing example program to run on the LEON3.
leon3mp.ucf	Constraint file for place and route.
leon3mp.xst	Synthesis script for Virtex-4.
systest.c	Sample C program printing ‘HELLO!’ on the UART and shining on half of the LEDs.

Tillegg G

Styttingar

Dette appendikset inneheld ei lista over styttingar nytta i masteroppgåva.

AD	Analog / Digital
ADC	Analog / Digital Converter
AHB	Advanced High-Performance Bus
BCC	Bare-C Cross-compiler
ALICE	A Large Ion Collider Experiment
ALTRO	ALICE TPC Readout
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application Specific Integrated Circuit
BAR	Base Address Register
BGA	Ball Grid Array
BIOS	Basic Input Output System
CDB	Common Data Base
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSR	Control and Status Register
CTP	Central Trigger Processor
CTSN	Clear to send
DAQ	Data Acquisition
DCS	Detector Control System
DDL	Detector Data Link
DCM	Digital Clock Manager
DIU	Destination Interface Unit
DLL	Delay Locked Loop
DMA	Direct Memory Access
DSU	Debug Support Unit
ELF	Executable and Linkable Format
FEC	Front End Card
FEP	Front End Processor
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FMD	Forward Multiplicity Detector
GDC	Global Data Collector
GNU	GNU is not Unix (Unix-implementasjon)
GPL	General Public License
GRLIB	Gaisler Research Library
HLT	High Level Trigger
HLT-RORC	HLT Readout Receiver Card
IDSEL	Initialization Device Select
IO	Input / Output
IP	Intellectual Property
IRQ	Interrupt Request
D-RORC	DAQ Readout Receiver Card
IEEE	Institute of Electrical and Electronics Engineers
ISA	Industry Standard Architecture
ITS	Inner Tracking System
JTAG	Joint Test Action Group

IO	Input / Output
L0	Level 0 Trigger
L1	Level 1 Trigger
L2r	Level 2 Reject
L2a	Level 2 Accept
LDC	Local Data Concentrators
LEP	Large Electron Positron Collider
LHC	Large Hadron Collider
LTU	Local Trigger Unit
LVDS	Low Voltage Differential Signaling
LZW	Lempel-Ziv-Welch (datakomprimeringsalgoritme)
MB	Megabyte
MMU	Memory Management Unit
MWPC	Multi-wire Proportional Chambers
OSI	Open Systems Interconnect
PASA	Pulse Amplifier Shaper
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCI-SIG	PCI Special Interest Group
PCI-X	PCI Extensions
POSIX	Portable Operating System Interface
PROM	Programmable Read Only Memory
PSI	PCI and Shared Memory Interface
PXI	PCI Extensions for Instrumentation
QGP	Quark Gluon Plasma
RAM	Random Access Memory
ROI	Region Of Interest
ROM	Read Only Memory
PMD	Photon Multiplicity Detector
PnP	Plug And Play
QGP	Quark Gluon Plasma
RCU	Readout Control Unit
RICH	Ring Imaging Cherenkov (partikkelidentifikasjonsdetektor)
RISC	Reduced Instruction Set Computing
RLE	Run-length Encoding
ROI	Region Of Interest
RS232	Recommended Standard 232 (seriekommunikasjon)
RTSN	Request to send
SIU	Source Interface Unit
SMP	Synchronous Multiprocessing
SPARC	Scalable Processor Architecture
TCP	Transmission Control Protocol
TRD	Transition Radiation Detector
TPC	Time Projection Chamber
TRD	Transition Radiation Detector
UART	Universal Asynchronous Receiver/Transmitter
UCF	User Constraint File

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VME	VERSAModule Eurocard bus
ZDC	Zero Degree Calorimeter

Figurar

Litteratur

[1] Johan Alme. *Digital Module Requirement Specification, Revision 0.1*. 2005.

[2] John P. Bentley. *Principles of Measurement Systems*. Prentice-Hall, third edition, 1995. ISBN 0-582-23779-3. Side 3–4.

[3] Geir Anton Johansen. *Laboratoriekurs i instrumentering og prosessregulering*. 1997. Side 5–8.

[4] Geir Anton Johansen. *Laboratoriekurs i instrumentering og prosessregulering*. 1997. Side 68.

[5] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*. 1995.

[6] http://www.pcisig.com/news_room/faqs/faq_20/.

[7] http://www.picmg.org/compactpci.stm.

[8] http://www.pxisa.org.

[9] http://www.xilinx.com/systemio/pciexpress/pciex_basics.htm.

[10] Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, second edition, 2001. ISBN 0-596-00008-1. Kapittel 13.

[11] CERN. *ALICE Technical Design Report*. 2004. ISBN 92-9083-217-7.

[12] CERN. *ALICE Physics Performance Report*. 2003. ISBN 92-9083-210-X.

[13] CERN. *ALICE Technical Design Report of the Time Projection Chamber*. 2000. ISBN 92-9083-155-3.

[14] Gaute Grastveit. *VHDL-implementation of the Cluster Finder algorithm for use in ALICE*. 2003. Side 1–25.

[15] Anders Strand Vestbø. *Pattern Recognition and Data Compression for the ALICE High Level Trigger*. 2004. ISBN 82-497-0218-2. Side 29–91.

[16] Jørgen A. Lien. *The Readout Control Unit of the ALICE TPC*. 2004. ISBN 82-497-0246-8. Side 1–4.

[17] Ketil Røed. *Irradiation tests of of ALTERA SRAM-based FPGAs*. 2004.

[18] J. Christiansen, A. Marchioro, P. Moreira, T. Toifl. *TTCrx Reference Manual, v3.9*. 2004.

[19] R. Divià, P. Jovanovic, P. Vande Vyvre. *Data Format over the ALICE DDL*. 2002.

[20] CERN. *ALICE TPC Readout Chip User Manual, draft 0.2*. 2002.

[21] C. Engster, A. Junique, B. Mota, L. Musa, J. Alme, J. Lien, B. Pommeresche, M. Richter, K. Roed, D. Roehrich, K. Ullaland, T. Alt, R. Bramm, C. Gonzales Gutierrez, R. Campagnolo. *The ALICE TPC Readout Control Unit*. 2004.

[22] http://www.cerntech.kfkipark.hu/ddl.htm.

[23] CERN, EP Division, AID Group, RMKI, RFFO Division, Detector Building Group. *ALICE Detector Data Link Hardware Guide for the Front-end Designers, Revision 2.1*. 2003.

[24] CERN. *ALICE Technical Design Report*. 2004. ISBN 92-9083-217-7. Kapittel 14.7.

[25] Xilinx Inc. *Virtex-4 Packaging and Pinout Specification, v2.3*. 2005.

[26] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, first edition, 1997. ISBN 0-201-50022-1. Side 174.

[27] Xilinx Inc. *Virtex-4 Configuration Guide, v1.1*. 2004.

[28] http://eislab.gatech.edu/projects/proam/resources/pwb-fab-process/.

[29] Xilinx Inc. *Virtex 4 PCB Designers Guide, v1.1*. 2004.

[30] http://www.hottconsultants.com/techtips/decoupling.html.

[31] Xilinx Inc. *Platform Flash In-System Programmable Configuration PROMS Preliminary Product Specification, v2.6*. 2005.

[32] http://www.techfest.com/hardware/bus/pci.htm#2.0.

[33] http://www.tldp.org/LDP/tlk/dd/pci.html.

[34] Altera Corporation. *PCI MegaCore Function User Guide, v2.1*. 2002.

[35] Altera Corporation. *PCI MegaCore Function User Guide, v3.2*. 2004.

[36] http://www.tldp.org/LDP/tlk/dd/interrupts.html.

[37] Xilinx Inc. *LogiCORE PCI v3.0, User Guide*. 2004.

[38] Private conversation with Artur Szostak.

[39] Xilinx Inc. *LogiCORE PCI v3.0, Getting Started Guide*. 2004.

[40] Xilinx Inc. *Virtex-4 User Guide, v1.2*. 2005.

[41] Neil H. E. Weste, David Harris. *CMOS VLSI Design*. Addison-Wesley, third edition, 2005. ISBN 0-321-26977-2. Side 460–465.

[42] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0.

[43] Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, second edition, 2001. ISBN 0-596-00008-1.

[44] Richard Peterson. *Red Hat Linux: The Complete Reference*. First edition, 2000. ISBN 0-07-212537-3. Side 616–618.

[45] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0. Kapittel 1.6.

[46] http://en.wikipedia.org/wiki/User_mode.

[47] http://www.informit.com/articles/article.asp?p=389712&seqNum=3&rl=1.

[48] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0. Kapittel 8–9.

[49] http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=88.

[50] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0. Kapittel 13.2–13.3.

[51] http://www.tldp.org/LDP/lkmpg/2.6/html/c870.htm.

[52] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0. Kapittel 2.5.

[53] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0. Kapittel 13.1.

[54] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2002. ISBN 0-596-00213-0. Kapittel 4.6.

[55] http://users.actcom.co.il/ choo/lupg/tutorials/signals/signals-programming.html.

[56] http://docs.sun.com/app/docs/doc/816-4854/6mb1o3aib?a=view.

[57] CERN. *ALICE Technical Design Report*. 2004. ISBN 92-9083-217-7. Kapittel 13.1.3–13.1.5.

[58] CERN. *ALICE Technical Design Report*. 2004. ISBN 92-9083-217-7. Kapittel 13.1.1.

[59] Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, second edition, 2001. ISBN 0-596-00008-1. Kapittel 7.

[60] Frank Vahid, Tony Givargis. *Embedded System Design: An Unified Hardware/Software Introduction*. Wiley, first edition, 2002. ISBN 0-471-38678-2.

[61] http://www.sparc.org/faq.html.

[62] Jiri Gaisler, Edvin Catovic. *Gaisler Research IP Cores Manual, v1.0.1*. 2005.

[63] http://www.gaisler.com.

[64] http://www.arm.com/products/solutions/AMBAHomePage.html.

[65] ARM Limited. *AMBA Specification, Rev. 2.0*. 1999.

[66] Gaisler Research AB. *GRLIB Product Brief*. 2005.

[67] http://www.gaisler.com/doc/bcc.html.

[68] Gaisler Research AB. *GRMON Users Manual, Version 1.0.13*. 2005.

[69] http://zone.ni.com/devzone/conceptd.nsf/webmain/70B1D3E338E6F52386256E37006DFDB3.

[70] Xilinx Inc. *LogiCORE PCI-X v5.0 User Guide, v5.0.100*. 2005.

[71] Geir Anton Johansen. *Laboratoriekurs i instrumentering og prosessregulering*. 1997. Side 38–39.

[72] http://www.fairchildsemi.com/an/AN/AN-5017.pdf.

[73] William Stallings. *Local and Metropolitan Area Networks*. Prentice Hall, sixth edition, 1999. ISBN 0-13-012939-9. Side 65–70.

[74] Ajay V. Bhatt. *Creating a PCI Express(TM) Interconnect*. 2002.

[75] http://www.rhyshaden.com/encoding.htm.

[76] Xilinx Inc. *PCI Express Endpoint Cores Product Specification, v2.0.5*. 2005.

[77] VMETRO Inc. *Vanguard User Guide for BusView(R) 4*. 2004.

[78] James K. Hollomon. *Surface-mount technology for PC Boards*. Prompt Publications, first edition, 2005. ISBN 0-7906-1060-4. Side 165–169.