Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter

Asbjørn Djupdal

16. juni 2003

Sammendrag

En CompactPCI datamaskin med et NallaTech BenERA FPGA-kort har blitt kjøpt inn til bruk innen forskning på evolusjonær maskinvare. Denne datamaskinen er i stand til å rekonfigurere og kjøre en vilkårlig krets på en FPGA, og kan kommunisere med FPGA-en kjapt over en PCI-buss. Øvre hastighet for kommunikasjon med moduler på FPGA-en begrenses av CompactPCI-bussen til 132MB pr. sekund.

Denne hovedoppgaven går ut på konstruksjon av et system basert rundt BenERA FPGA-kortet som skal benyttes til kjøring av sblokk-baserte eksperimenter, med fokus på development. Prosjektet baserer seg på tidligere forskning innen sblokker og development ved NTNU.

Det har blitt utviklet et system der sblokkmatriser kan lastes ned til en FPGA og kjøres der. Kretsen har også spesialkonstruert maskinvare som kan kjøre development på sblokkmatrisen. Sblokkmatrisen kan på et hvilket som helst tidspunkt undersøkes av programvare på CompactPCI-maskinen ved tilbakelesning av data over PCI-bussen. All styring skjer ved hjelp av programvare og kan automatiseres.

Kretsen er implementert som en samlebåndsbasert koprosessor. Koprosessoren er i stand til å prosessere to sblokker pr. sykel både ved kjøring av developmentsteg og ved konfigurering av sblokkmatrise. Ved kjøring av sblokkmatrise vil alle sblokker oppdateres pr. sykel. Ved tilbakelesning av data fra sblokkmatrise behandles 8 sblokker pr. sykel.

Ved syntese ble det oppnådd en klokkefrekvens på 80MHz.

Forord

Dette er en hovedoppgave utført ved Norges teknisk-naturvitenskapelige universitet, Institutt for datateknikk og informasjonsvitenskap, datamaskingruppen.

Hovedoppgaven som teller 10 vekttall er en fortsettelse av et 5-vekttalls prosjekt utført høsten 2002, og er avslutningen på det 5-årige sivilingeniørstudiet i datateknikk.

Takk til veileder Gunnar Tufte og faglærer Snorre Aunet for mange gode råd underveis i arbeidet. Takk til Svein Arne Aase for hjelp til utforming og kjøring av testeksperiment.

Asbjørn Djupdal
16. juni 2003

Innhold

Figurer

Tabeller

Kapittel 1

Introduksjon

Formålet med denne hovedoppgaven er å konstruere et system basert på rekonfigurerbar maskinvare som egner seg for kjøring av eksperimenter innen evolusjonær maskinvare. Systemet spesiallages for eksperimenter nyttige for deler av NTNUs forskning innen dette fagfeltet.

Hensikten med å gjøre dette i maskinvare og ikke i en simulator er at hastigheten vil kunne øke betydelig, noe som gjør kjøring av større eksperimenter mulig.

Rapporten er organisert slik:

- Kapittel 2 gir et kort overblikk over nødvendig teori innen evolusjonær maskinvare
- Kapittel 3 beskriver maskinvaren og utviklingsverktøyene som benyttes
- Kapittel 4 gir en funksjonell oversikt over hvordan systemet oppfører seg og hvordan det kan benyttes
- Kapittel 5 beskriver hvordan systemet har blitt implementert
- Kapittel 6 oppsummerer synteseresultater og demonstrerer systemet ved å kjøre noen tester.
- Kapittel 7 diskuterer resultater og beskriver gode og dårlige sider ved det som har blitt konstruert. Mulige forbedringer av systemet nevnes

Rapporten er skrevet på norsk. Det er likevel tilfeller der engelske ord og uttrykk har blitt brukt. En del engelske faguttrykk har ikke blitt oversatt da forfatteren ikke kjenner til gode norske ord. All VHDL-kode er skrevet på engelsk, noe som medfører en del engelske navn i denne rapporten der det har vært ønske om likhet med navnsetting i VHDL-koden.

Kapittel 2

Evolusjonær maskinvare

Evolusjonær maskinvare (*Evolvable Hardware* – EHW) tar for seg maskinvare-utvikling ved hjelp av evolusjonære teknikker, det vil si teknikker inspirert av naturens måte å utvikle organismer på.

Evolusjon – både kunstig og naturlig – er en måte å utvikle arter på der målet er artens (og indirekte også individers) dyktighet til å overleve. Kunstig evolusjon oppnås ofte med en *genetisk algoritme* (GA) som prøver å etterligne en del naturlige mekanismer. Evnen til å løse et spesifikt problem definerer overlevelsesdyktighet og bestemmer hvilke individer som skal få formere seg og danne en ny generasjon. Det er altså individer som løser problemet best som vil få sitt «genmateriale» overført videre til senere generasjoner.

Et individs genmateriale kalles for individets *genotype*. Genotypen er altså en beskrivelse av hvordan et individ skal konstrueres og kan sammenlignes med naturens DNA. Det ferdige individet kalles *fenotypen* og overgangen fra genotype til fenotype kalles genotype–fenotype-omforming.

En GA vil starte med et antall tilfeldige genotyper – en populasjon. Et sett med individer lages og testes ut for å sjekke hvor dyktige de er til å løse et gitt problem. Hvert individ gis en poengsum som angir hvor gode de er. Deretter konstrueres en ny generasjon av individer ved å krysse gode individer fra forrige generasjon og ved mutasjon av eksisterende individer. De dårligste individene forkastes. Resultatet er forhåpentligvis en populasjon som er bedre egnet til å løse problemet enn forrige populasjon. Gjentatte runder der nye generasjoner skapes vil øke artens og enkeltindividenes evne til å løse problemet.

2.1 Hensikt

Det er flere mål bak forskning innen EHW. En fordel ved å benytte EHW kan være å utvikle smarte tekniske løsninger som en menneskelig utvikler neppe ville tenkt på. Evolusjonære metoder er ikke begrenset av de mentale modellene en menneskelig utvikler benytter for å forenkle og forstå virkeligheten med, og kan derfor komme opp med uventede løsninger som er bedre enn de eksisterende [21].

Et annet mål med EHW kan være å utvikle store kretser. Størrelsen på VLSI-brikker øker raskere enn kompleksitetsnivået en utviklingsgruppe klarer å holde seg på. Dagens VLSI-brikker er allerede så store at det å utvikle design

på denne størrelsen er et stort metodemessig problem. Gapet mellom hvor store brikker vi klarer å produsere og hvor store design vi klarer å utvikle vil trolig øke [19]. EHW kan muligens være en løsning på dette problemet.

Feiltolerante og adaptive kretser er en tredje ting evolusjonære metoder kan hjelpe til med [10, 13, 17]. Det kan se ut til at evolusjon er i stand til å utvikle kretser som er ekstra tolerante for feil. Dette vil si at kretsen fortsetter å fungere tilfredsstillende selv etter at deler av teknologigrunnlaget feiler. Dersom man tillater evolusjonen å fortsette også etter at kretsen er i operativ tilstand, vil den kunne tilpasse seg et endrende miljø (eller nye funksjonelle krav) og kunne fungere brukbart der konvensjonelle systemer ville bryte sammen. Slike ting vil være passende i situasjoner hvor systemet skal befinne seg i et tøft miljø der utskiftning av deler ikke er mulig, for eksempel i en satellitt.

Et mål bak evolusjonær maskinvare kan være å benytte evolusjonære metoder til å oppnå komplekse kretser, det vil si utvikle kretser med en oppførsel som er så kompleks at tradisjonelle utviklingsmetoder ikke kan benyttes [7, 21]. Det å få til kompleks oppførsel i en krets basert på enkle elementer er et av områdene det forskes på ved NTNU. Naturen har klart å utvikle organismer med kompleks oppførsel, noe som kan tyde på at de samme prosessene kan benyttes til å utvikle store og komplekse kretser.

2.2 Teknologier for evolusjonær maskinvare

Metoder innen evolusjonær maskinvare kan grovt deles inn i *ekstrinsikk* og *intrinsikk* evolusjonær maskinvare [3]. *Ekstrinsikk* vil si at evolusjon foregår utenfor kretsen som skal benyttes til sluttresultatet. Dette vil som oftest si at kretsen evolveres ved hjelp av en simulator og bare realisert i maskinvare etter at det en ferdig. *Intrinsikk* vil si at evolusjon skjer på den teknologien som skal brukes til slutt. Ekstrinsikk evolusjon har fordelen ved å skje i en kontrollert simulator. Fordelen ved intrinsikk evolusjon er at de evolusjonære metodene kan benytte alle kretsens fysiske egenskaper, ikke bare de som er modellert i simulatoren. Det er også gode muligheter for at intrinsikk evolusjon vil gå raskere da simulering av kretsens egenskaper ikke er nødvendig.

Metoder innen intrinsikk evolusjonær maskinvare forutsetter en rekonfigurerbar maskinvareplattform, da svært mange konfigurasjoner må prøves ut i maskinvare før den endelige er funnet.

2.2.1 FPGA

Den ledende teknologien for rekonfigurerbar maskinvare i dag er *Field Programmable Gate Array* (FPGA). Virkemåten til denne forklares mer i detalj i kapittel 3.1.

Dersom man ønsker å benytte hele FPGA-en som basis for evolusjon, det vil si la evolusjonen komme opp med en fullstendig FPGA-konfigurasjon, vil man støte på flere vanskeligheter. FPGA-en er utviklet med hensyn på tradisjonell maskinvareutvikling, og har derfor egenskaper som ikke passer så godt for evolusjonære metoder. Den har veldig gode rutingsmuligheter, noe som medfører at store mengder konfigurasjonsdata går med til å sette opp ruting. All ruting må konfigureres selv om bare en liten del av den benyttes, og resultatet blir en stor fenotype – den inneholder mye irrelevant konfigurasjonsdata i forhold

til hva som faktisk benyttes av ressurser. Dersom man har en 1–1-omforming fra genotype til fenotype, f.eks der genotypen er FPGA-ens konfigurasjonsdata direkte, vil også genotypen bli stor med de problemene det medfører [8]. Dette er nærmere forklart i kapittel 2.4.

Et annet problem er at de fleste FPGA-er vil kunne konfigureres på en slik måte at de kan ødelegges [28]. To utganger kan for eksempel kobles sammen. Normalt er ikke dette noe problem fordi synteseverktøyene vil garantere korrekte konfigurasjoner. Men for evolusjonære teknikker vil dette være et problem. Evolusjonen kan komme opp med en løsning som ødelegger FPGA-en.

Det vil ikke være nok å ha en rekonfigurerbar maskinvare. Hastigheten maskinvaren kan konfigureres med er også svært sentral, da antallet rekonfigureringer i løpet av et eksperiment kan være svært stort. For å øke hastigheten er det nyttig å benytte *delvis rekonfigurering*, det vil si at deler av maskinvaren kan konfigureres uten at resten av maskinvaren berøres. Ved å bare konfigurere de delene av maskinvaren som er forandret siden forrige konfigurasjon, kan mengden konfigurasjonsdata minke betraktelig og hastigheten øker tilsvarende. Flere FPGA-er tilbyr delvis rekonfigurasjon, blant annet VirtexE-serien fra Xilinx som benyttes i dette prosjektet.

Det høye antallet rekonfigureringer må støttes av maskinvaren. Maskinvare som baserer seg på å lagre konfigurasjonsdata i et medium med begrenset antall rekonfigureringer (EEPROM, Flash) vil være dårlig egnet for dette formålet [21]. Alle FPGA-er fra Xilinx benytter SRAM til lagring av konfigurasjonsdata og har derfor i praksis ingen øvre grense på antall rekonfigureringer.

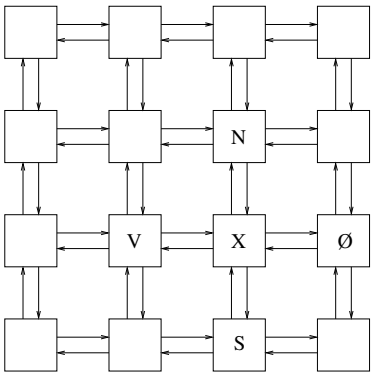
Én måte å løse mange av problemene nevnt ovenfor på, er å benytte FPGA-en til å implementere en maskinvareplattform som er bedre egnet til evolusjon. Man kan implementere en slags evolusjonsvennlig, virtuell FPGA inne i den virkelige FPGA-en, og utstyre denne med de nødvendige konfigurasjons- og utlesningsmuligheter. På denne måten vil man være mye mindre avhengig av hvilke egenskaper som tilbys av den valgte FPGA-en.

2.3 En virtuell EHW-FPGA

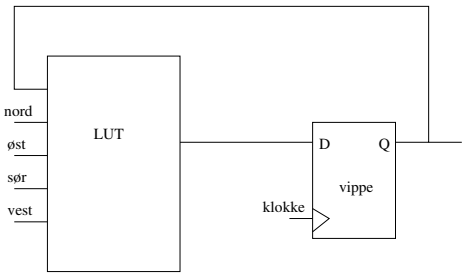
Som et forslag til en EHW-vennlig FPGA ble *sblokken* introdusert [5]. En sblokk er en enkel rekonfigurerbar logisk komponent. Den kan beskrives som en forenklet versjon av en FPGA-CLB (se FPGA-beskrivelse i kapittel 3.1). Sblokk-FPGA-en består av et rutenett av sblokker med svært enkel ruting mellom disse.

Hovedpoenget med den sblokkbaserte FPGA-en er at den er EHW-vennlig. Det er flere grunner til dette: All ruting er fastlagt på forhånd – rutingen kan ikke konfigureres, noe som medfører at mengden med konfigurasjonsdata vil minke betraktelig. Det medfører også at ulovlige konfigurasjoner der utgang kobles til utgang med kortslutning som resultat, ikke vil være mulig. En sblokk er også mindre kompleks enn de fleste vanlige CLB-er, noe som medfører mindre konfigurasjonsdata på grunn av det.

Sblokkbegrepet ble videreført i [6] som del i en virtuell EHW-FPGA. En masseprodusert EHW-vennlig FPGA vil neppe komme i nær fremtid, så sblokk-FPGA-en ble gjort om til en virtuell FPGA som enkelt kan implementeres i en ekte, og kommersielt tilgjengelig FPGA.



Figur 2.1: Utsnitt av sblokkmatrise. Hvert kvadrat representerer en sblokk. Alle utganger fra en bestemt sblokk representerer én og samme utgang.



Figur 2.2: Sblokk

2.3.1 Beskrivelse

En sblokkbasert FPGA består av et rutenett med sblokker. Disse er koblet opp som vist i figur 2.1, og kalles en *sblokkmatrise*. Hver sblokk kommuniserer kun med nabo-sblokkene i nord-, sør-, øst- og vest-retning. Naboskapet er vist i figuren; Sblokken «X» kommuniserer med nabosblokkene merket «N», «S», «Ø» og «V».

Denne konfigurasjonen kan benyttes som en uniform cellular automaton (CA) [24] med et von Neumann naboskap [2]. Sblokkmatrisen kan også være en ikkeuniform CA, det vil si at hver sblokk kan oppføre seg forskjellig fra hverandre. En slik konfigurasjon er bevist å være Turing-komplett, noe som vil si at den kan benyttes som en generell regnemaskin [20].

Oppbygningen av hver enkelt sblokk vises i figur 2.2. Sblokken har fire innganger og én utgang. Sblokken består av to hovedkomponenter: En feminngangs oppslagstabell (LUT) og en vippe. Utgangen på vippen er sblokkens utgangssignal. Inngangene til oppslagstabellen består av sblokkens inngangssignaler (vippeutgangene til nabosblokkene) samt sin egen vippeutgang.

Oppførselen til en sblokk bestemmes av LUT-en. En feminngangs LUT kan implementere en hvilken som helst feminngangs logisk funksjon. En feminngangs LUT fungerer ved at den har en tabell på $2^5 = 32$ elementer. Den binærkodede verdien som de fem inngangene representerer benyttes som en indeks inn i tabellen, og utgangen til LUT-en settes lik verdien som ligger på tabellplassen

som indeksen peker på.

Kjøring av sblokkmatrisen vil si å tillate alle vippene i sblokkmatrisen å oppdatere tilstand. En sykel der sblokkmatrisen kjøres kalles et *tilstandssteg*.

LUT-innholdet til en sblokk bestemmer sblokkens *type*. En gitt sblokk kan altså være én av 2^{32} forskjellige typer. Sblokkens *tilstand* er det samme som verdien av vippen.

En uniform CA kan settes opp ved å sette alle sblokker i sblokkmatrisen til samme type. Dersom det finnes flere forskjellige sblokktyper vil sblokkmatrisen være en ikkeuniform CA.

Evolusjon for sblokkmatriser kan skje både ekstrinsikk og intrinsikk. Sblokkmatrisen synkrone natur gjør at den ikke vil bli påvirket av ytre faktorer som temperatur og interferens. Funksjonelt vil det derfor være ekvivalent om evolusjonen gjøres ekstrinsikk eller intrinsikk. Hastigheten vil derimot være mye høyere i en maskinvareimplementasjon (intrinsikk).

2.4 Development

Evolusjonær maskinvare er fremdeles på et tidlig forskningsstadium. Evolusjonære teknikker har vist seg å skalere dårlig med størrelsen på kretsen som skal utvikles. Tradisjonelt har det vært benyttet en 1–1-omforming fra genotype til fenotype. Dette betyr at genotypen må inneholde all informasjon om den ferdige kretsen (ruting og logikk) og størrelsen øker lineært med størrelsen på kretsen som skal utvikles. De genetiske algoritmene som skal utvikle genotypen går mot å bli et tilfeldig søk når genotypen blir større, trolig på grunn av for dårlig representasjon (måten genotypen beskriver fenotypen på). Dette gjør at ressursene som skal til for å utvikle en krets øker dramatisk med størrelsen på kretsen.

En løsning på dette problemet er å minske mengden konfigurasjonsdata som kan benyttes av den genetiske algoritmen, det vil si minske størrelsen på fenotypen. Man kan for eksempel begrense hvilke rutingsmuligheter den genetiske algoritmen skal få lov til å benytte. Denne løsningen har bare begrenset nytte når man skal utvikle komplekse kretser fordi man uansett vil nå et punkt der fenotypen blir for stor. Løsningen skalerer ikke.

En annen løsning er å minske størrelsen på genotypen uten at størrelsen på fenotypen minker, med andre ord gå bort fra en 1–1-omforming. En måte å oppnå dette på kan være å benytte genetiske algoritmer til å utvikle et sett med regler som beskriver hvordan en krets skal bygges i stedet for å utvikle en beskrivelse av hvordan den ferdige kretsen skal være. Slike byggeregler kan på en svært kompakt måte beskrive svært store kretser. En del av kompleksiteten flyttes dermed over i omformingsprosessen fra genotype til fenotype, som nå vil være en fase der kretsen bygges opp etter reglene funnet av den genetiske algoritmen. Denne teknikken kalles *development*. [7, 8, 11]

2.4.1 Biologisk development

Bruk av development for utvikling av kretser henter mye av sin terminologi og metode fra biologien.

Biologisk development begynner med et befruktet egg. Egget inneholder DNA, som er en plan for hvordan organismen skal utvikle seg frem til et ferdig

individ. Utviklingen fra et befruktet egg og til et ferdig individ kan forenklet deles inn i flere (avhengige og delvis overlappende) faser:

- *Pattern formation*; Celler organiseres fysisk i regioner som senere vil tilsvare kroppsdeler i det ferdige individet.
- *Morphogenesis*; Celler forandrer form og grupperes sammen i klaser.
- *Differentiation*; Celler forandrer struktur og funksjon
- *Growth*; Den nesten ferdig formede organismen øker i størrelse ved hjelp av cellevekst og gjentatte celledelinger. Apoptosis – celledød – kan hjelpe til med å skille ut mindre kroppsdeler fra større.

2.4.2 Lindenmayersystem

Lindenmayersystem (L-system) er en matematisk formalisering som blir brukt i studier av biologisk development. L-systemet ble opprinnelig beskrevet av Aristid Lindenmayer for å modellere planters vekst [12].

Et L-system består av et alfabet, et sett med prioriterte regler, og et aksiom som er en startstreng med tegn fra alfabetet. Reglene består av to deler; en betingelse og et resultat. Både betingelsen og resultatet er strenger av det gitte alfabetet.

Når systemet kjører, vil reglene benyttes til å endre på startstrengen. Når en del av startstrengen er lik betingelsesstrengen i en regel, vil denne delen byttes ut med resultatstrengen. Ved gjentatt bruk av reglene på det som opprinnelig var startstrengen, vil det utvikle seg en helt ny streng som etterhvert vil svare til det ferdige individet.

Grovt sett kan reglene deles inn i disse typene:

- *Change*-regler der resultatstrengen er like lang som betingelsesstrengen. Slike regler vil føre til en endring i type, men ikke i størrelse.
- *Growth*-regler der resultatstrengen er større enn betingelsesstrengen. Slike regler vil føre til at organismen vokser i størrelse.
- *Celledød*-regler der resultatstrengen er mindre enn betingelsesstrengen. Slike regler fører til at organismen minker i størrelse.

Eksempel

Gitt:

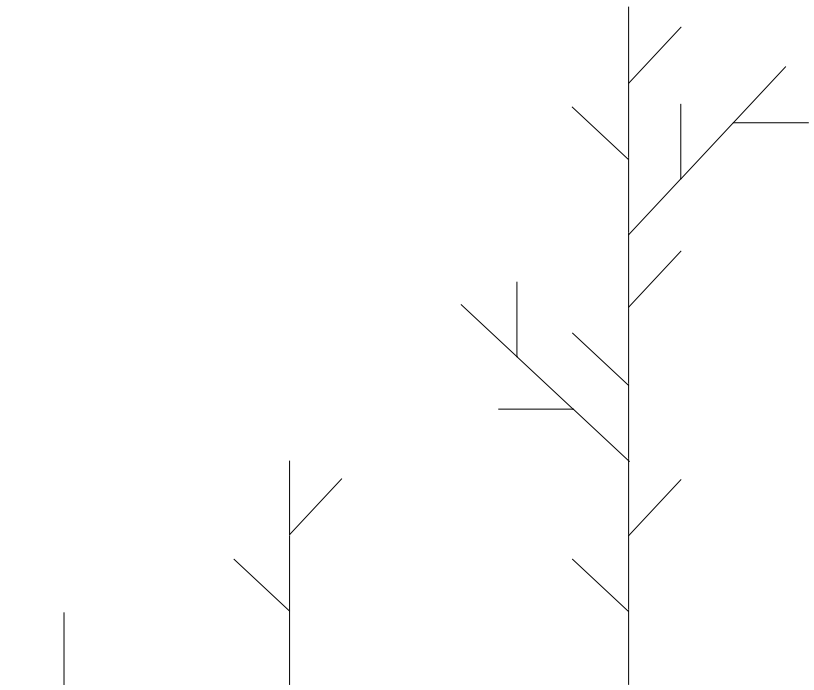
- *Alfabet*: (F, +, -, [,])
  Dette alfabetet kan tolkes på denne måten:
  - *F*; Tegn en linje en gitt avstand frem i gjeldende retning
  - *+*; Endre gjeldende retning et gitt antall grader til høyre
  - -; Endre gjeldende retning et gitt antall grader til venstre
  - *[*; Legg gjeldende posisjon og retning på en stakk
  - *]*; Hent gjeldende posisjon og retning fra stakken

- *Aksiom*: (F)
  Begynner med en rett strek
- *Regel*: (F → F[-F]F[+F]F)
  En enkelt *Growth*-regel. *F* er betingelsen, og *F[-F]F[+F]F* er resultatet. Dette vil si at alle forekomster av F vil byttes ut med F[-F]F[+F]F for hver gjennomkjøring.

Dette eksempelet vil utvikle seg på denne måten:

1. F
2. F[-F]F[+F]F
3. F[-F]F[+F]F[-F[-F]F[+F]F]F[-F]F[+F]F[+F[-F]F[+F]F]F[-F]F[+F]F

Dette er vist grafisk i figur 2.3.



(1) Aksiom (2) Etter 1. iterasjon (3) Etter 2. iterasjon

Figur 2.3: 3 stadier i utvikling av L-system

2.4.3 Development på sblokkmatriser

Development på sblokkmatriser kan gjøres på flere måter. En metode basert på L-system har blitt forsøkt [8]. Alfabetet er (0, 1, 2) og tolkes som konfigurasjonsdata for en sblokkmatrise. Verdien 2 er *don't care* og kan benyttes i betingelsen

til en regel. En genetisk algoritme finner et aksiom og et sett med regler. Developmentprosessen vil utvide aksiomet til det er stort nok til å konfigurere en hel sblokkmatrise.

Dette kan sies å være en kunnskapsfattig representasjon [8], da developmentprosessen ikke antar noe som helst om teknologien den utvikler for. En enkelt regel kan forandre deler av konfigurasjonsdata for flere forskjellige sblokker.

Siden developmentprosessen som blir brukt er deterministisk, vil det være slik at når genotypen er mindre enn fenotypen vil søkerommet man leter etter løsninger i også bli mindre. Dersom man benytter en kunnskapsfattig metode slik som beskrevet over, vil man begrense søkerommet til en liten og tilfeldig undermengde av alle mulige løsninger.

For å få større mulighet til å få en god løsning innen det begrensede søkerommet har det også blitt forsøkt med en representasjon som er inspirert av L-system, men som i tillegg er kunnskapsrik [7]. Det vil si at kunnskap om hva som skal utvikles er bygget inn i regler og metode. Metoden tar hensyn til at den jobber på en sblokkmatrise; den benytter seg av kunnskapen om hvordan sblokker er koblet sammen. På samme måte som at hver sblokk under kjøring bare er påvirket av seg selv og sine fire naboer, vil hver sblokk under development bare være påvirket av seg selv og sine fire naboer.

Metoden begrenser også hvilke typer (LUT-innhold) sblokkene skal ha lov til å ha. Hver sblokk kan i prinsippet ha én av 2^{32} forskjellige typer. Denne metoden tillater ikke bruk av alle sammen og begrenser seg til bruk av et lite antall (< 32) forhåndsbestemte typer totalt. Dette blir en begrensning i antall celletyper. Tanken er at man kjenner til hvilke celletyper som er nødvendige for å lage en gitt krets, og man luker derfor vekk det store antallet med unyttige celletyper som neppe vil føre til noen god løsning.

Metode

Developmentmetoden som benyttes i resten av denne rapporten tilsvarer metoden i [7]; en kunnskapsrik, L-system-inspirert developmentmetode.

Under kjøring av development, vil sblokkenes typer og tilstander endres slik det er spesifisert av et sett med prioriterte regler. Tanken er at gjentatt kjøring av development vil utvikle kretsen frem til det ferdige produktet.

På samme måte som for L-system, består developmentreglene av en betingelse og et resultat. Men i motsetning til L-system-reglene, vil betingelsen her være en spesifisering av hvordan naboskapet til en gitt sblokk skal se ut for at regelen skal slå til på denne sblokken. Resultatet vil beskrive hvordan denne ene sblokken skal endres dersom regelen slår til. Reglene kan enten bare ta hensyn til sblokkenes type eller de kan ta hensyn til både type og tilstand.

Developmentprosessen deles inn i *developmentsteg*. Under hvert developmentsteg vil alle sblokker testes opp mot alle regler. De stedene der en regel slår til, vil sblokkens type (og muligens tilstand) endres. Konseptuelt sett vil all endring skje samtidig etter dette developmentsteget og før neste developmentsteg. Dersom flere regler slår til på samme sblokk, vil regelen med høyest prioritet gjelde.

Før første kjøring av developmentsteg, vil sblokkenes typer og tilstander settes til en kjent starttilstand (aksiom).

Det finnes to typer regler; *Change* og *Growth*:

- Change-regler vil medføre at den aktuelle sblokken endrer type (og muligens tilstand).

- Growth-regler vil si at den aktuelle sblokken (som vil ha typen «tom») får satt sin verdi fra en av nabosblokkene (definert i regelen). Dette er altså en slags motsatt vekst; Den tar ikke utgangspunkt i en celle (sblokk) og vokser utover. I stedet tar den utgangspunkt i et tomt område ved siden av en sblokk og vokser til denne.

Eksempel

Her er et eksempel på development. Vi har tre regler, vist i tabell 2.1.

Prioritet	Type	Senter	Nord	Sør	Øst	Vest
1	Growth fra øst	Z	D	D	J	Z
2	Growth fra sør	Z	D	I	D	D
3	Change til J	I	Z	Z	Z	Z

Tabell 2.1: Developmentregler i eksempel. *D* angir «Don't care», og betyr at sblokkens type ikke har noe å si

Disse reglene har en gitt prioritet. Feltene *Senter*, *Nord*, *Sør*, *Øst* og *Vest* er betingelsen og sier hvilke typer de respektive sblokkene må ha for at regelen skal slå til (se naboskapsoversikten i figur 2.1).

For enkelthetsskyld tar disse reglene kun hensyn til sblokkenes type. Det finnes tre sblokktyper: *Z* som betyr «tom», *I* og *J*.

Z	Z	Z
Z	I	Z
Z	Z	Z

(a) Startverdi

Z	I	Z
Z	J	Z
Z	Z	Z

(b) Etter 1. developmentsteg

Z	I	Z
J	J	Z
Z	Z	Z

(c) Etter 2. developmentsteg

Figur 2.4: Eksempel på development

Figur 2.4(a) viser utgangspunktet før development starter. Når første developmentsteg kjører, vil to regler slå til; regel 2 og regel 3. Regel 2 medfører at det vokser ut en ny sblokk med type I. Regel 3 medfører at den opprinnelige sblokken med type I skifter til type J. Dette vises i figur 2.4(b). Legg merke til at reglene ikke vil påvirke resultatet av hverandre fordi skifte av verdi ikke skjer før etter at reglenes betingelser sjekkes.

Det andre developmentsteget medfører at sblokken med type J vokser ut slik at det til slutt er to sblokker av denne typen. Dette vises i figur 2.4(c).

Nå kan developmentprosessen sies å være ferdig. Neste skritt vil da være å oversette sblokktypene Z, I og J til 32-bits LUT-verdier, slik at en kjørbar sblokkmatrise kan lages ut i fra denne konfigurasjonen.

2.5 Beslektede arbeider

Denne diplomoppgaven bygger videre på et prosjekt gjort høsten 2002 [4]. Resultatene fra dette prosjektet er oppsummert her.

En sblokkmatrise ble implementert for den samme maskinvaren som blir brukt i denne oppgaven. Formålet bak dette arbeidet var å få en oversikt over hva som var mulig å oppnå med den aktuelle maskinvaren i forbindelse med sblokkmatriser.

En sblokkmatrise ble implementert i maskinvare. Denne ble konfigurert ved å endre på FPGA-ens bitfil[1] direkte, før bitfilen ble lastet inn i FPGA-en. Nøyaktig spesifisering av posisjonen til de enkelte sblokkene på FPGA-en gjorde det mulig å regne ut posisjonen til disse i bitfilen. Tilbakelesning av data skjedde av logikk internt på FPGA-en og sendt over PCI-bussen.

Det ble konkludert med at det som var mest hensiktsmessig var å lage en sblokkmatrise som både kan konfigureres og undersøkes ved hjelp av egenkonstruert logikk på FPGA-en, og dermed unngå å endre bitfilen. Hele problematikken rundt delvis rekonfigurering og manipulering av den dårlig dokumenterte bitfilen vil dermed unngås. Når konfigurering av sblokkmatrisen skjer av egenkonstruert logikk internt på FPGA-en, blir det også mulig å legge developmentprosessen og eventuelt også GA i maskinvare.

[1] FPGA-ens konfigurasjonsfil, se kapittel 3.1 for mer informasjon om FPGA og bitfil

Kapittel 3

Utviklingsplattform

Dette kapittelet beskriver uviklingsplattformen som benyttes. Først beskrives FPGA-er generelt, deretter beskrives den konkrete maskinvaren som disponeres. Til slutt er det en oversikt over hvilke utviklingsverktøy som benyttes.

3.1 Field Programmable Gate Array

En *Field Programmable Gate Array* (FPGA) er en rekonfigurerbar logisk komponent. I praksis vil dette si at det er en komponent der man kan spesifisere logikk på portnivå og rutingen mellom portene gjentatte ganger etter at komponenten er produsert (også mens komponenten sitter i målsystemet).

I dette prosjektet er det utelukkende Xilinx VirtexE FPGA-er som benyttes, og all beskrivelse av FPGA-er vil derfor være rettet mot VirtexE-arkitekturen [26].

For å få til rekonfigurerbarhet, benyttes SRAM internt i FPGA-en til å holde konfigurasjonsdata. Ved oppstart (eller ved rekonfigurering) må FPGA-en få en bitstrøm (bitfil) med beskrivelse av hvordan FPGA-en skal konfigureres.

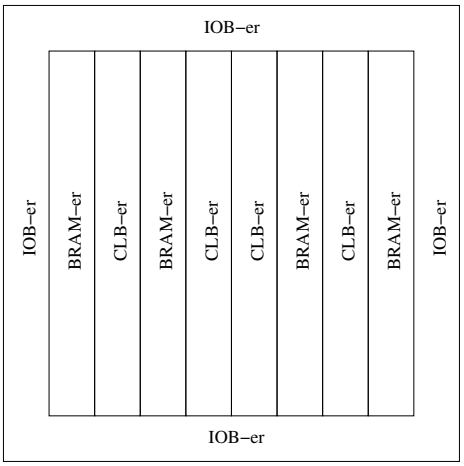
En VirtexE FPGA består hovedsaklig av tre typer rekonfigurerbare komponenter:

- Konfigurerbare logiske blokker (CLB)
- Inn- / ut-blokker (IOB)
- Block SelectRAM (BRAM)

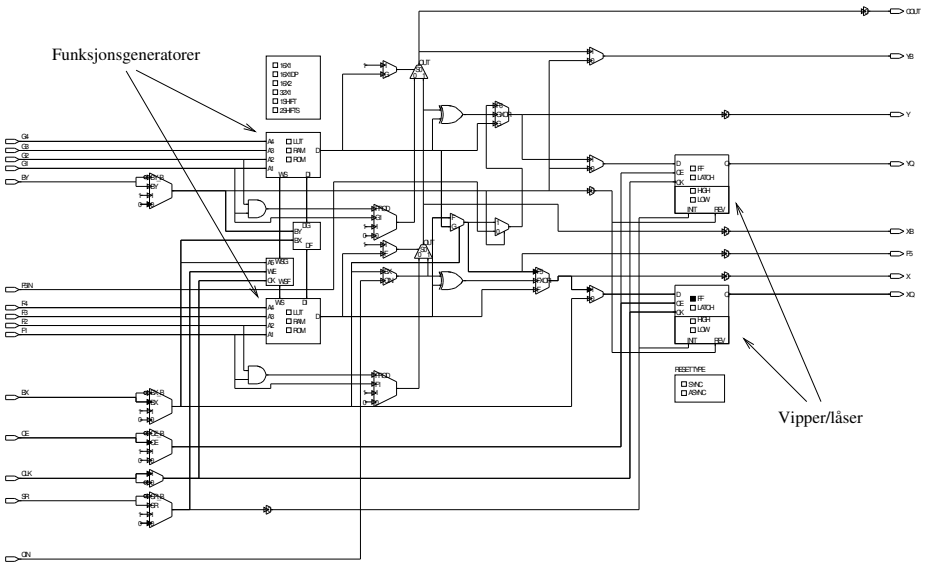
Mesteparten av FPGA-en består av CLB-er. Disse er koblet opp i et rutenett. Langs kanten av FPGA-en befinner IOB-ene seg, nær I/O-pinnene på brikken. BRAM-ene er fordelt jevnt utover FPGA-en. I tillegg kan ruting mellom komponenter konfigureres. Figur 3.1 viser en forenklet oversikt over hvordan de forskjellige komponentene er plassert fysisk inne i en FPGA.

3.1.1 Beskrivelse av FPGA-komponenter

CLB En VirtexE CLB består av to «slicer» og to tristatebuffer, knyttet ekstra tett sammen. En slice, som vist i figur 3.2, består av to fireinngangs funksjonsgeneratorer, to vipper/låser og en rekke multipleksere der noen kan settes opp



Figur 3.1: Oversikt over VirtexE-FPGA



Figur 3.2: En «slice» i en VirtexE FPGA

statisk ved konfigurering og andre kan styres av dynamiske kontrollsignaler. Funksjonsgeneratorene brukes til å implementere en hvilken som helst fireinngangs logisk funksjon, og kan konfigureres som oppslagstabeller (LUT), RAM, ROM eller som skiftregister.

IOB Det finnes én *IOB* for hver IO-pinne på FPGA-en. Hensikten med denne er å ta seg av tilkoblinger til eksternt utstyr. Den kan konfigureres til å støtte flere forskjellige IO-standarder.

BRAM I tillegg til å kunne benytte funksjonsgeneratorene i CLB-ene til RAM, finnes det i VirtexE-serien større blokker med dedikert RAM som kan benyttes fritt. Disse oppfører seg stort sett som vanlig to-ports synkron SRAM. Begge portene kan benyttes til lesing og skriving. Databussbredden kan settes til 1, 2, 4, 8 eller 16 bit. Det kan være forskjellig bussbredde på de to portene. Adressebussbredden avhenger av databussbredden, slik at det alltid adresseres et ord av valgt størrelse. Det er 4096 bits i hver enkelt BRAM.

3.1.2 Konfigurering

En VirtexE konfigureres ved å skrive en bitstrøm til dens konfigurasjonport (som oftest JTAG [9]) [27, 28]. Data grupperes i 32-bits ord, og skrives til JTAG-porten én bit av gangen. Bitstrømmen som sendes til konfigurasjonsporten er en samling med kommandoer og tilhørende data, normalt produsert av Xilinx-verktøyet *bitgen*.

VirtexE støtter delvis rekonfigurering, det vil si at deler av FPGA-en kan rekonfigureres uten at resten av FPGA-en blir berørt. Den minste konfigureringsenheten kalles en ramme og spenner over flere CLB-er.

3.1.3 Konstruksjon av FPGA-kretser

Konfigurasjonen til en FPGA kan konstrueres på «CLB-nivå», det vil si at man manuelt konfigurerer hver enkelt slice, og all rutingen mellom disse. Dette blir fort svært uoversiktlig, og gjøres aldri i praksis. I stedet konstrueres maskinvaren i et maskinvarebeskrivende språk (for eksempel VHDL eller Verilog), som syntetiseres til en gyldig FPGA-konfigurasjon av et synteseverktøy. Syntetiseringen foregår automatisk, uten større innblanding fra konstruktør.

3.2 CompactPCI datamaskin

Datamaskinen som benyttes er en CompactPCI datamaskin. Denne er satt sammen av et VMICPCI-7760 hovedkort, et VMIACC-0320 I/O-kort og et BenERA FPGA-kort. Disse er montert i et CompactPCI-kabinett med strømforsyning og kjølesystem. Tilsammen blir dette en maskin med disse egenskapene [22, 23]:

- To Pentium III-prosessorer på 1GHz
- Fem CPCI-kortplasser
- 512MB RAM

- To 10/100 ethernetkort
- 64MB CompactFlash
- SVGA grafikkort, basert på Chips & Technologies 69030
- ATA-tilkobling med 40GB harddisk og CD-ROM
- RS232, parallellport, USB, diskettstasjon

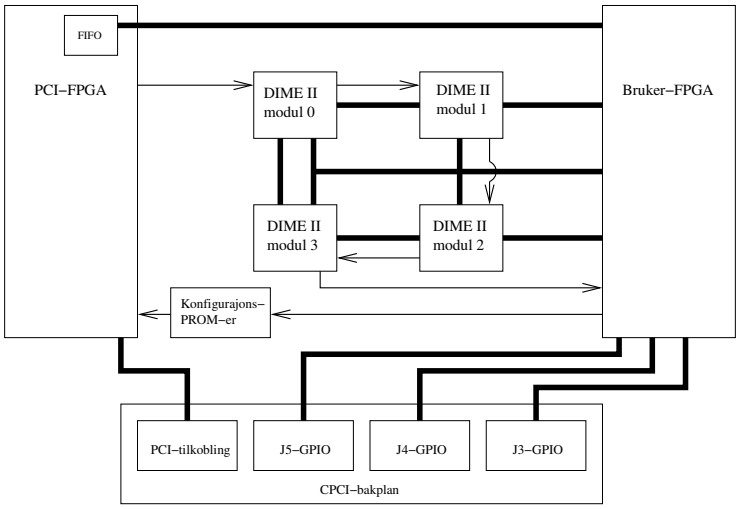
I tillegg kommer BenERA FPGA-kortet som er beskrevet i kapittel 3.3.

CompactPCI-standarden (CPCI) er en busstandard som er et supersett av den vanlige PCI-standarden. CPCI er laget med tanke på industriell bruk, og har derfor en annen fysisk tilkobling som er mer robust og enklere å håndtere. I tillegg støtter den opp til 8-kort på en CPCI-buss, mot 4 i en vanlig PCI-maskin. [18]

Maskinen kjører Debian GNU/Linux, og kontrolleres ved hjelp av ssh-tilkobling over nettverk.

3.3 BenERA FPGA-kort

BenERA FPGA-kortet fra NallaTech [14] er et CPCI-kort med en Xilinx VirtexE FPGA som kan rekonfigureres ved hjelp av programvarebiblioteket FUSE på datamaskinen BenERA-kortet sitter i.



Figur 3.3: Funksjonell oversikt over BenERA. Tykke linjer viser kommunikasjonsbusser, tynne linjer med pil viser JTAG-konfigurasjonssignaler. Kun de viktigste signaler er tatt med. Figuren er en forenklet versjon av oversiktsfiguren i brukerhåndboken til BenERA [15].

Figur 3.3 viser en funksjonell oversikt over kortet. *PCI-FPGA* er en prekonfigurert FPGA som tar seg av all konfigurasjon og generell kontroll av kortet. Denne er koblet til PCI-bussen og kommuniserer direkte med FUSE-programvaren

i datamaskinen. *Bruker-FPGA* er en VirtexE-1000 og kan konfigureres og benyttes fritt. I tillegg er det mulig å tilkoble opp til fire ekstra DIME-II ekspansjonsmoduler. Disse kan inneholde ekstra FPGA-er eller annet utstyr.

Dersom det refereres til «FPGA-en» på BenERA-kortet senere i denne rapporten, er det bruker-FPGA-en som omtales.

Det finnes tre konfigurerbare klokker som hver kan settes til en verdi mellom 20MHz og 120MHz. Alle klokkene kan benyttes samtidig av bruker-FPGA-en.

3.3.1 Konfigurering

Konfigurering av komponenter (det vil hovedsaklig si bruker-FPGA-en dersom ingen DIME-II-moduler er installert) skjer gjennom JTAG [9]. FUSE-programvaren i datamaskinen sender konfigurasjonsdata til PCI-FPGA-en som deretter konfigurerer de aktuelle komponentene.

JTAG konfigureres over en seriebuss, det vil si én bit av gangen. VirtexE-FPGA-en som sitter på kortet har mulighet for å konfigureres 8-bit av gangen over en konfigurasjonsbuss som kalles *SelectMap* [26], og er derfor raskere enn JTAG. Dette har begrenset støtte fra BenERA.

Konfigurasjonshastigheten til JTAG er på maksimalt 20MHz. Dette medfører en maksimal konfigurasjonshastighet på 2.38MB (ca. 3 fullstendige FPGA-konfigurasjoner) pr. sekund.

3.3.2 Kommunikasjon med programvare

Bruker-FPGA-en har ikke mulighet til å kommunisere direkte over PCI-bussen. All kommunikasjon over PCI-bussen må gå gjennom PCI-FPGA-en. For å gjøre det mulig for bruker-FPGA-en å kommunisere indirekte over PCI-bussen, inneholder PCI-FPGA-en et to-veis FIFO-buffer. Dette kan både skrives til og leses fra av bruker-FPGA-en via en dedikert kommunikasjonsbuss mellom bruker-FPGA og PCI-FPGA. Det samme bufferet kan også leses fra og skrives til av programvaren på datamaskinen ved hjelp av FUSE-programvarebiblioteket.

Kommunikasjonen mellom PCI-FPGA og bruker-FPGA består av en 32-bits databuss med tilhørende kontrollsignaler. Se tillegg C for beskrivelse av denne bussen.

Mer informasjon om denne bussen spesielt og BenERA kortet generelt finnes i brukerhåndboken til BenERA [15].

3.4 Utviklingsverktøy

Maskinvaren skrives i VHDL-87. VHDL-87 ble valgt i stedet for VHDL-93 på grunn av bedre støtte i utviklingsverktøyene som benyttes.

Til simulering benyttes ModelSim 5.7a, Linux-versjon. Til syntese benyttes Xilinx ISE 5.1. Disse verktøyene ble valgt fordi de begge kunne benyttes under GNU/Linux, og fordi NTNU har denne programvaren tilgjengelig.

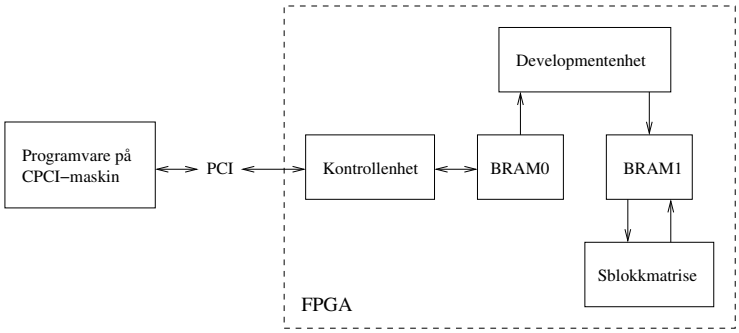
Testprogramvare skrives i ren C, kompilert med GCC, og benytter seg av FUSE programvarebibliotek [16] for styring av BenERA-kort. En oversikt over FUSE-funksjoner finnes i tillegg D.

Kapittel 4

Funksjonell beskrivelse

Kretsen skal brukes til å kjøre sblokkbaserte eksperimenter innen development. Den har derfor støtte både for å gjøre development og å kjøre en ferdigkonfigurert sblokkmatrise i maskinvare.

Kretsen kjører i en FPGA på et BenERA CPCI-kort, og kontrolleres ved hjelp av programvare på CPCI-maskinen. Denne programvaren kommuniserer med kretsen over PCI-bussen.



Figur 4.1: Abstrakt oversikt over systemet. Modulene inne i den stiplete boksen befinner seg inne i en FPGA

En abstrakt oversikt over systemet vises i figur 4.1. Systemet har disse hovedkomponentene:

- *Sblokkmatrise*: Dette er en kjørbar sblokkmatrise. Den kan konfigureres med nye LUT-verdier og tilstander, og etter kjøring kan tilstandene leses ut igjen.
- *BRAM-0* og *BRAM-1*: Dette er midlertidig lagringsplass for sblokkmatrisens LUT-er og tilstander. BRAM-0 og BRAM-1 er helt identiske i oppbygning og er som navnet antyder implementert ved hjelp av RAM-blokker. De inneholder all informasjon som er nødvendig for en fullstendig konfigurasjon av sblokkmatrisen.

  Hensikten med disse er å ha data lett tilgjengelig, og brukes under manipulering av sblokkmatrisedata (f.eks kjøring av et developmentsteg).

Det er to stykker av disse fordi det kreves av algoritmen for kjøring av developmentsteg.

- *Developmentenhet*: Denne enheten er i stand til å kjøre et developmentsteg. Denne opererer utelukkende på BRAM-ene, og ikke direkte på sblokkmatrisen.

- *Kontrollenhet*: Denne tar i mot instruksjoner fra PCI-bussen, og styrer de andre enhetene ut i fra disse instruksjonene. Alle operasjoner som kan utføres inne i FPGA-en settes i gang av denne enheten.

- *Programvare på CPCI-maskin*: Sender instruksjoner til FPGA-en og leser tilbake data.

4.1 Skrive initialverdier

Programvaren på CPCI-maskinen har tilgang (indirekte gjennom kontrollenheten) til både å skrive verdier (sblokktyper og tilstander) til-, og lese verdier fra BRAM-0. Dette vil typisk benyttes til å skrive en starttilstand før kjøring av development, og til å lese ut resultater underveis og til slutt.

4.2 Kjøring av developmentsteg

Developmentenheten er i stand til å kjøre et developmentsteg.

Regler Enheten jobber etter et sett med developmentregler som må være lastet opp til FPGA-en på forhånd av programvaren på CPCI-maskinen. Se kapittel 2.4.3 for mer informasjon om developmentregler.

Algoritme for developmentsteg Enheten følger en algoritme som leser fra én kopi av sblokkmatrisen (BRAM 0) og skriver til en annen (BRAM 1). På denne måten vil ikke sblokkmatrisen det leses fra endres underveis i developmentsteget.

For alle sblokker:

1. Sjekk om gjeldende sblokk (med naboer) svarer til venstre side i en regel

2. Dersom Change-regel slår til: Skriv ny verdi til sblokk

3. Dersom Growth-regel slår til: Skriv en nabosblokks verdi til sblokk.

4. Dersom ingen regel slår til: Skriv gammel verdi til sblokk

Dersom flere regler slår til på samme sblokk vil reglene prioriteres etter en gitt prioritet.

4.3 Kjøring av sblokkmatrise

For å kjøre sblokkmatrisen må den først konfigureres, det vil si at sblokkmatrisens LUT-er og vipper settes til en gyldig startverdi. Sblokkmatrisen kan konfigureres med verdier fra BRAM-1.

Underveis i kjøring, vil sblokkenes tilstand endre verdi. Etter at sblokkmatrisen har kjørt det nødvendige antall tilstandssteg er det antakeligvis ønskelig å undersøke den nye tilstanden til sblokkmatrisen. Dette gjøres ved å lese ut tilstanden og skrive denne tilbake til BRAM-1.

4.4 Bytte om på BRAM-0 og BRAM-1

Som vist i figur 4.1 har BRAM-0 og BRAM-1 hver sine oppgaver:

- Data kan overføres mellom programvare på CPCI-maskinen og BRAM-0
- BRAM-0 er utgangspunktet for developmentsteget
- BRAM-1 er resultatet av developmentsteget
- BRAM-1 er utgangspunkt og resultat for kjøring av sblokkmatrise

Dette er et problem dersom data befinner seg i «feil» BRAM i forhold til hva man ønsker å bruke de til.

Det kan for eksempel være ønskelig å konfigurere sblokkmatrisen med data fra CPCI-maskinen uten å gå via developmentenheten. Dette lar seg ikke gjøre fordi CPCI-maskinen (via kontrollenheten) ikke kan skrive til BRAM-1.

Eller man kan ha kjørt et developmentsteg noe som medfører at resultatdata blir plassert i BRAM-1. Problemet blir å kjøre neste developmentsteg, fordi developmentenheten ikke kan bruke BRAM-1 som utgangspunkt for developmentsteget.

Det er ikke mulig å kopiere data direkte mellom BRAM-0 og BRAM-1, men det er mulig å gi en kommando som bytter om på BRAM-ene. Det vil i praksis si at alle data i BRAM-0 vil flyttes over i BRAM-1, og alle data i BRAM-1 vil flyttes over i BRAM-0. Dette løser problemene nevnt over.

4.5 Koprosessor

PCI-bussen kan være en flaskehals. For å unngå å sende for mye data over PCI-bussen er kretsen implementert som en koprosessor; det er det mulig å laste ned små programsnutter til FPGA-en som kan kjøre enkeltvis eller i løkke uten videre interaksjon fra programvaren på CPCI-maskinen.

Et eksperiment vil typisk bestå av en liten mengde instruksjoner som gjentas mange ganger. Ofte vil det for hver iterasjon sendes data tilbake over PCI-bussen.

I stedet for å overføre én og én instruksjon over PCI-bussen for å styre eksperimentet, så kan det i stedet lastes et lite program til FPGA-en, bestående av de samme instruksjonene. Etter nedlastning kan programmet startes og kjøre i løkke. Hele kapasiteten til PCI-bussen vil dermed kunne brukes til å overføre data fra hver iterasjon på FPGA-en over til programvaren på CPCI-maskinen.

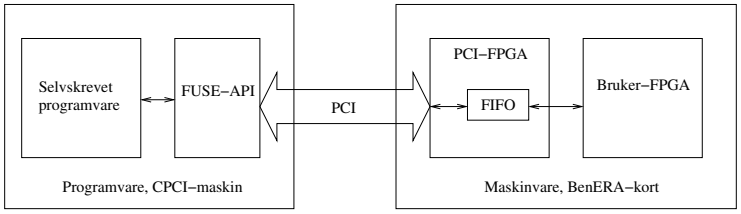
Siden denne dataoverføringen ikke stadig blir avbrutt av instruksjoner andre veien, vil man kunne benytte store DMA-overføringer. Dette er mer effektivt enn hvis man stadig sender instruksjoner over PCI-bussen midt i eksperimentet.

Kapittel 5

Implementasjon

Dette kapittelet beskriver hvordan systemet har blitt implementert.

5.1 Systembeskrivelse



Figur 5.1: Oversikt over systemet

Systemet består både av programvarekomponenter og maskinvarekomponenter. En generell oversikt over et BenERA CPCI-system er vist i figur 5.1. Boksene merket «selvskrevet programvare» og «bruker-FPGA» inneholder moduler spesiallaget for formålet.

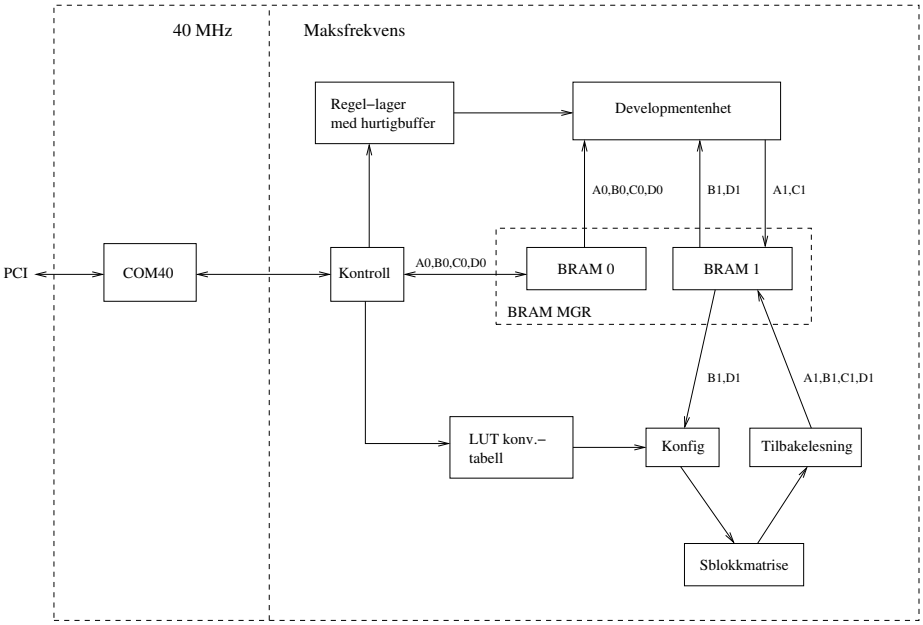
Bruker-FPGA-en inneholder selve koprosessoren som har blitt implementert. Denne vil bli beskrevet senere i dette kapittelet. Programvaren på CPCI-maskinen består av enkel styring av koprosessoren og vil avhenge av eksperimentet som ønskes utført.

5.2 Toppnivå FPGA

5.2.1 Enhetsoversikt

Figur 5.2 viser en oversikt over hvilke enheter som er lagt i FPGA-en og hvordan data flyter mellom disse. Dette er en mer implementasjonsnær fremstilling enn den som ble gitt i kapittel 4.

FPGA-en er delt inn i to frekvensdomener; et som må kjøre på 40MHz for å kunne kommunisere med PCI-FPGA-en, og et som kjører så raskt som mulig. *COM40* er en enhet som har til oppgave å overføre data mellom PCI-bussen og den raske delen av kretsen.



Figur 5.2: Enhetsoversikt FPGA

Figuren viser hvilke lese/skrive-porter som benyttes mot BRAM av de forskjellige enhetene. Hver BRAM har fire forskjellige lese/skrive-porter (se kapittel 5.9); A, B, C og D. A0, B0, C0, D0 betegner lese/skrive-portene på BRAM-0 og A1, B1, C1, D1 betegner lese/skrive-porter på BRAM-1. Retningene på pilene i figuren viser hvilken vei data flyter – lesing, skriving eller begge deler. Eksempelvis så benytter developmentenheten alle lese/skrive-portene til begge BRAM-ene, noe som betyr at ingen andre enheter kan benytte BRAM-ene mens developmentenheten jobber.

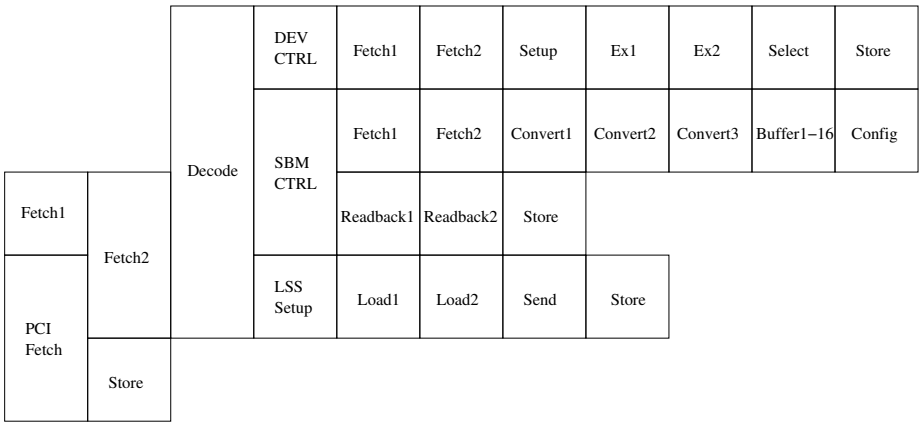
Enheten merket *Kontroll* inneholder et instruksjonslager den kan lagre og hente instruksjoner fra.

5.2.2 Samlebånd

Kontrollogikken i systemet er implementert ved hjelp av et samlebånd. En oversikt over samlebåndet er vist i figur 5.3.

Instruksjoner hentes i de to første trinnene. Disse kommer enten fra et internt instruksjonslager eller fra PCI-bussen. Disse instruksjonene flyter mot høyre i figuren, og sendes inn til én av tre mulige samlebånd:

- *Development*: DEV-samlebåndet er det øverste samlebåndet etter dekodingssteget. Representerer developmentenheten i figur 5.2, og tar seg dermed av kjøring av et developmentsteg.
- *Sblokkmatrise*: SBM-samlebåndet er det midterste samlebåndet. Styrer sblokkmatrisen, og representerer konfig-enheten og tilbakelesningsenheten i figur 5.2. Tar seg av kjøring, konfigurering og tilbakelesning av sblokkmatrisen.



Figur 5.3: Samlebånd. Instruksjoner flyter fra venstre mot høyre.

Samlebåndet deler seg i et konfigurasjonssamlebånd og et tilbakelesningssamlebånd, som representerer henholdsvis konfig-enheten og tilbakelesningsenheten i figuren.

- *Load Send Store*: LSS-samlebåndet tar seg av lesing og skriving mellom kontrollenheten og BRAM-0 i figur 5.2, samt sending av data fra FPGA-en og over PCI-bussen.

Samlebåndsstegene er valgt slik for å gjøre lengste kritiske sti minst mulig. Dette for å få opp klokkefrekvensen. Spesielt er oppslag i RAM tidkrevende i en Xilinx Virtex FPGA, noe som har gjort det nødvendig å holde nesten all kombinatorikk borte fra inngangene og utgangene til RAM-blokkene. Derfor finnes det to samlebåndssteg som tilsynelatende nærmest er tomme over alt der det gjøres oppslag i RAM.

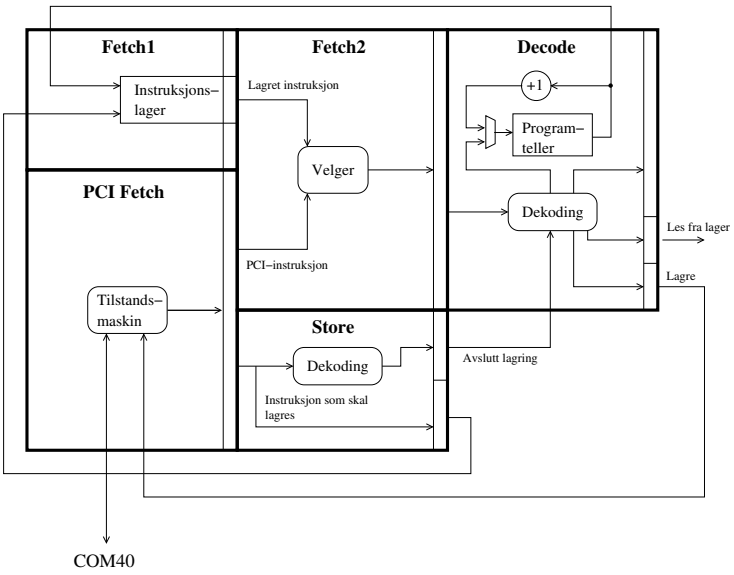
Samlebåndet vil forklares nærmere i senere delkapitler.

5.3 Instruksjonsutføring

Figur 5.4 viser en forenklet oversikt over samlebåndsstegene som tar seg av å hente instruksjoner og dekode disse.

Det finnes to kilder for instruksjoner som skal utføres av koprosessoren; Et internt instruksjonslager, og instruksjoner fra PCI-bussen. Hele instruksjonssettet (bortsett fra instruksjoner som kontrollerer skriving til instruksjonslageret) er tilgjengelige fra begge kildene. Alle operasjoner kan altså utføres både direkte over PCI-buss og ved hjelp av et lagret program.

Decode inneholder en programteller og et flagg («Les fra lager» i figuren) som signaliserer at instruksjoner leses fra det interne instruksjonslageret. Så lenge dette flagget er satt vil ingen instruksjoner aksepteres fra PCI-bussen. Flagget skrus av når det utføres en *break*-instruksjon. Da vil koprosessoren stoppe henting av instruksjoner fra lageret og begynne å vente på instruksjoner fra PCI-bussen. En *jump*-instruksjon vil sette flagget og begynne kjøring av instruksjoner fra adressen gitt i instruksjonen. *Jump*-instruksjonen kan også benyttes



Figur 5.4: Instruksjonshenting, dekoding og lagring i instruksjonslager

i et lagret program til å lage en evigvarende løkke. Under vanlig kjøring økes programtelleren med én for hver eneste sykel.

Instruksjoner fra instruksjonslageret hentes fra samlebåndssteget *Fetch1*. Instruksjoner fra PCI-bussen hentes av en tilstandsmaskin i *PCI Fetch* som kommuniserer med COM40. En av instruksjonskildene velges i *Fetch2*, og den valgte instruksjonen sendes videre til *Decode*. Der vil instruksjonene dekodes og instruksjonen sendes videre til det korrekte delsamlebåndet.

Instruksjoner som sendes over PCI-bussen vil vanligvis sendes via *Fetch2* til *Decode* og videre til resten av samlebåndet. Dersom en *store*-instruksjon utføres, settes et spesielt flagg («Lagre» i figuren) i *Decode* som signaliserer at alle etterfølgende instruksjoner fra PCI-bussen skal lagres i instruksjonslageret. Da vil instruksjoner sendes til *Store*-trinnet i stedet for *Fetch2*, og lagres i instruksjonslageret. *Store* inneholder enkel dekodingslogikk og vil avslutte lagringen av instruksjoner når *end*-instruksjonen kommer.

For å forenkle kontrollogikken er alle instruksjoner internt i koprosessoren 64-bit brede. Dette gjelder også instruksjoner som ville fått plass på 32-bit. For å unngå overflødig trafikk på PCI-bussen deles instruksjoner som sendes over PCI-bussen i to grupper: Korte og lange. Korte instruksjoner er på 32-bit, mens lange instruksjoner er på 64-bit. I *PCI Fetch* vil korte instruksjoner utvides til 64-bit for å få alle instruksjoner på samme format.

En oversikt over alle instruksjoner er vist i tabell 5.1.

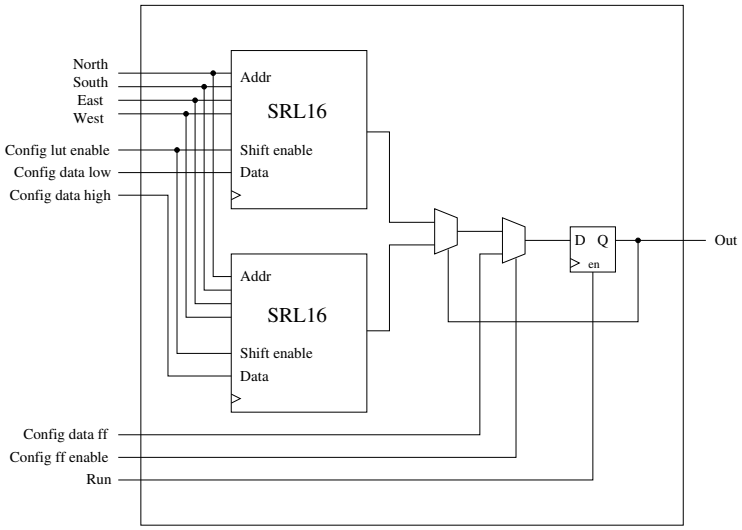
5.4 Sblokkmatrise

Selv om sblokkdata lagres i BRAM må de også kunne overføres til en ekte sblokkmatrise for å kunne kjøres. En slik kjørbar sblokkmatrise er tegnet inn i figur 5.2. Denne består av rekonfigurerbare sblokker koblet opp som en tradisjonell sblokkmatrise. Størrelsen på sblokkmatrisen er parametrisert; størrelsen

Navn	Type	Argumenter	Beskrivelse
break	Kort		Avslutt kjøring fra instruksjonslager
clearBRAM	Kort	type, tilstand	Setter alle sblokker i BRAM-0 til gitt verdi
config	Kort		Konfigurer sblokkmatrise fra BRAM-1
devstep	Kort		Kjør developmentsteg fra BRAM-0 til BRAM-1
end	Kort		Stopp lagring av instruksjoner
jump	Kort	adresse	Hopp til adresse i instruksjonslager
nop	Kort		Ingen handling
readback	Kort		Les tilbake fra sblokkmatrise til BRAM-1
readState	Kort	x, y	Send tilstand fra BRAM-0 over PCI
readStates	Kort		Send alle tilstander fra BRAM-0 over PCI
readType	Kort	x, y	Send type fra BRAM-0 over PCI
readTypes	Kort		Send alle typer fra BRAM-0 over PCI
run	Kort	sykler	Kjør sblokkmatrise gitt antall sykler
setNumberOfLastRule	Kort	prioritet	Angir regel med høyest prioritet
store	Kort	startadresse	Lagre etterfølgende instruksjoner i instruksjonslager
switch	Kort		Bytt om på BRAM-ene
writeLUTConv	Lang	lut, type	Skriv til LUT konverteringstabell
writeRule	Lang	regel, prioritet	Skriv til regellager
writeState	Kort	tilstand, x, y	Skriv tilstand til BRAM-0
writeType	Kort	type, x, y	Skriv type til BRAM-0

Tabell 5.1: Oversikt over instruksjonssett. Detaljert beskrivelse er i tillegg A.

kan settes ved å endre noen konstanter før syntese.



Figur 5.5: Sblokkimplementasjon

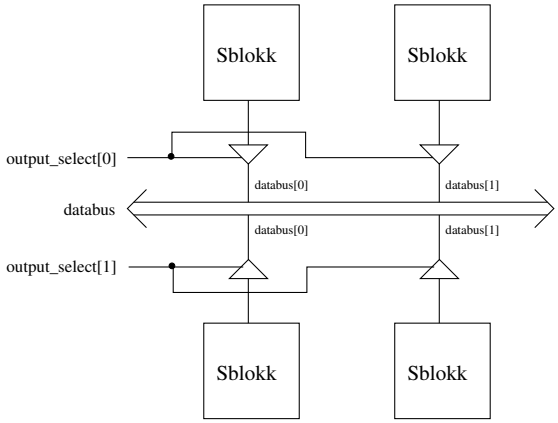
Hver enkelt sblokk implementeres som vist i figur 5.5. LUT-en er realisert ved hjelp av to spesielle 16-bits skiftregistre (SRL16 [25]). Disse fungerer som vanlige 16-bits LUT-er i normal bruk, men har egenskapen at man kan skifte inn nye data når sblokkens LUT-er skal konfigureres. De to skiftregistrene settes sammen til en 32-bits LUT ved hjelp av en multiplekser. Denne implementasjonen oppfører seg som en standard sblokk, men har i tillegg muligheten til å rekonfigurere både LUT og tilstandsvippe. Implementasjonen er valgt slik at hver sblokk opptar én Xilinx VirtexE CLB.

Konfigurering av LUT og vippe er helt uavhengig av hverandre og kan skje samtidig. Det konfigureres to LUT-bit i gangen, noe som betyr at det tar 16-sykler å fullføre konfigurasjonen av en sblokk-LUT.

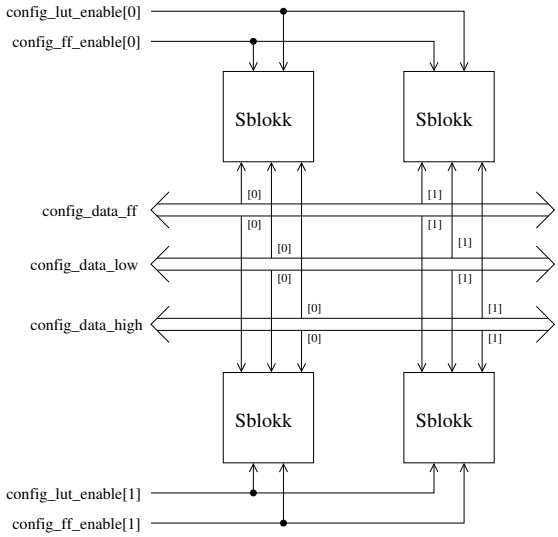
Sblokkene settes sammen til en sblokkmatrise med LUT-enes innganger og vippenes utganger koblet som i figur 2.1. Alle sblokkenes *run*-signaler kobles sammen. Når dette signalet er satt vil alle sblokkene kjøre og oppdatere sine tilstandsvipper.

I forbindelse med tilbakelesning av tilstander grupperes sblokkene inn i grupper på størrelse med databussen for tilbakelesning. Et *output_select*-signal velger hvilken gruppe med sblokker som skal drive databussen. Dette er vist i figur 5.6. Når bit 0 i *output_select* er satt vil de to øverste sblokkene drive databussen. Når bit 1 i *output_select* er satt vil de to nederste sblokkene drive databussen.

Konfigurering av sblokkene skjer også gruppevis. Sblokkene deles inn i grupper like store som databusser for konfigurasjon. Et *config_lut_enable*-signal velger hvilken gruppe som skal få konfigurert sine LUT-er. Et *config_ff_enable*-signal velger hvilken gruppe som skal få konfigurert sine vipper. Det finnes separate databusser for konfigurering av LUT og vippe, slik at begge deler kan skje samtidig. Dette er vist i figur 5.7. Det konfigureres alltid to bit i en LUT av gangen, derfor er det to konfigurasjonsdatabusser (*Config_data_low* og *config_data_high*) for konfigurering av LUT i figuren.



Figur 5.6: Tilbakelesning av tilstander fra sblokkmatrise. For enkelhets skyld er det vist en $2 \cdot 2$-sblokkmatrise med en utlesningsdatabuss på to bit.



Figur 5.7: Konfigurering av sblokkmatrise. For enkelthets skyld er det vist en $2 \cdot 2$-sblokkmatrise med konfigurasjonsbusser på to bit.

5.4.1 LUT konverteringstabell

BRAM-ene inneholder fullstendig konfigurasjonsdata for sblokkmatrisen. For å spare plass og forenkle logikk lagres hver sblokks type som en 5-bits verdi. Dette gir mulighet for 32 forskjellige sblokktyper.

En sblokk LUT inneholder 32 bits, noe som gjør det nødvendig med en konvertering fra 5-bits typeverdi til 32-bits LUT-verdi. Dette tas hånd om av en LUT konverteringstabell. Dette er en oppslagstabell lagret i RAM der sblokkens typeverdi brukes som adresse. To uavhengige LUT-er kan hentes ut pr. sykel.

5.4.2 SBM-samlebånd

Samlebåndet som tar seg av styring av sblokkmatrisen er vist i figur 5.8. *SBM Control* inneholder en tilstandsmaskin (FSM) som aktiveres av instruksjoner fra *Decode*. Tilstandsmaskinen er vist i figur 5.9. Tilstandsmaskinen aksepterer instruksjoner når den er i tilstand *Ledig*. Etter å ha mottatt en instruksjon, skifter den til den tilsvarende tilstanden. *Vent*-tilstanden får tilstandsmaskinen til å vente til samlebåndet er tømt, og benyttes til å garantere at tilstandsmaskinen ikke går til *Ledig* før den er klar til å akseptere nye instruksjoner.

Konfigurering

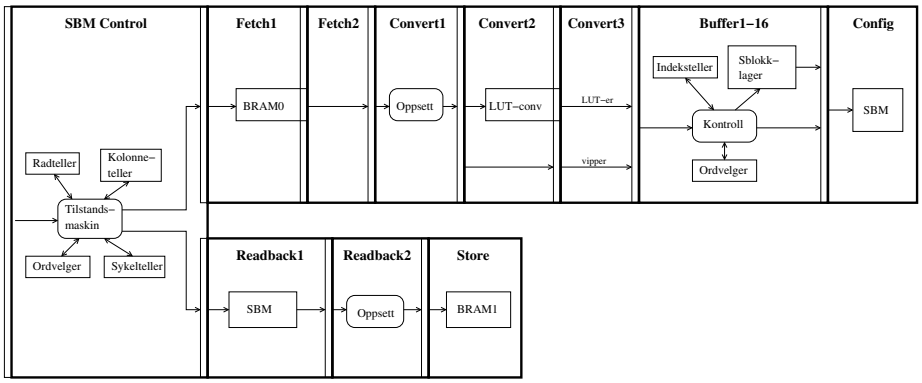
Konfigurering av sblokkmatrisen skjer i konfigurasjonsdelen av sblokkmatrise-samlebåndet (øverst på figur 5.8). To sblokker behandles i parallell i dette samlebåndet. Tilstandsmaskinen i *SBM Control* styrer denne prosessen som foregår slik:

- *Fetch1* og *Fetch2* leser ut sblokkdata for to sblokker. For hver sykel leses det ut to nye sblokker. En radteller og en kolonneteller brukes til å gå gjennom sblokkmatrisen systematisk. Adressen til sblokkene i BRAM-1 genereres av verdiene fra rad- og kolonnetellerene.
- *Convert1*, *Convert2* og *Convert3* konverterer sblokktype til 32-bits LUT. Dette gjøres ved oppslag i en LUT-konverteringstabellen. Sblokkenes tilstand sendes uforandret gjennom disse trinnene.
- *Buffer1-16* bufrer 32 sblokker. Det tar 16 sykler å konfigurere én enkelt sblokk-LUT. For å være i stand til å ha en gjennomstrømming på to sblokker pr. sykel må det konfigureres 32 sblokker i parallell. Dette trinnet samler opp 32 sblokker før de sendes videre til *Config*-trinnet. Det er med andre ord 16 syklers forsinkelse i dette trinnet, men gjør at *Config* alltid har data til 32 sblokker klare.
- *Config* konfigurerer to bits av 32 sblokker i parallell. Hver 16. sykel vil i tillegg sblokkenes tilstand konfigureres.

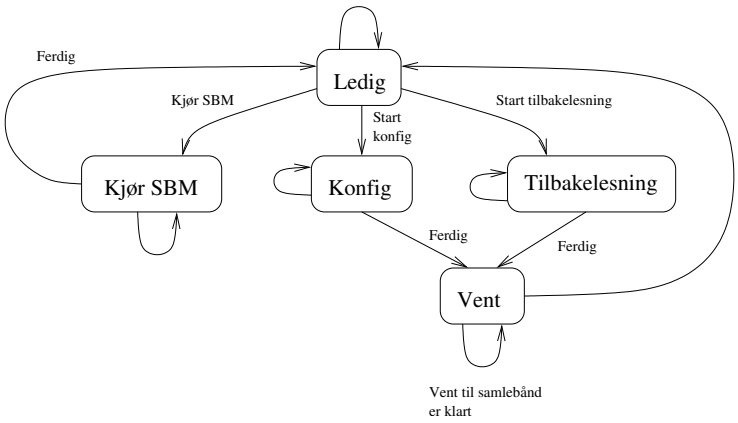
Tilbakelesning

Tilbakelesning styres av tilstandsmaskinen i SBM Control. 8 sblokker behandles i parallell. Kun sblokk-tilstand leses ut og skrives til BRAM-1. Dette foregår slik:

- *Readback1* og *Readback2* leser ut vipper fra sblokkmatrisen



Figur 5.8: Samlebånd for SBM-operasjoner (kjøring, konfigurering, tilbakelesning)



Figur 5.9: Tilstandsmaskin i SBM Control

- *Store* skriver disse tilbake til BRAM-1. Siden 8 sblokker skal skrives, benyttes alle skriveportene på BRAM-1.

Kjøring

Kjøring av sblokkmatrisen styres av tilstandsmaskinen i *SBM Control*. Denne inneholder en teller som teller sykler som skal kjøres. En kontrollinje som aktiverer sblokkmatrisen holdes aktiv så lenge telleren ikke er ferdig. Det trengs ikke et samlebånd til dette; alt gjøres direkte av tilstandsmaskinen.

5.5 Developmentenhet

Developmentenheten har som oppgave å kjøre et developmentsteg etter algoritmen gitt i kapittel 4.2. Enheten jobber med to sblokker i parallell, og arbeider systematisk fra venstre mot høyre, rad for rad nedover sblokkmatrisen.

Samlebåndet som representerer developmentenheten er vist i figur 5.10. En tilstandsmaskin i DEV Control (se figur 5.11) styrer kjøringen av developmentsteg.

Tilstandsmaskinen oppfører seg på denne måten:

- Tilstandsmaskinen venter på en instruksjon i tilstanden *Ledig*. Etter å ha mottatt instruksjonen *start devstep* går den inn i tilstand *Klargjør samlebånd*

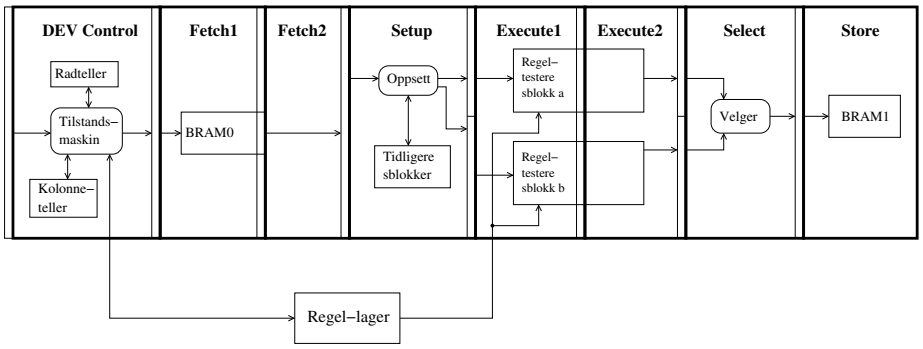
- I *Klargjør samlebånd* settes samlebåndet opp riktig før kjøring (noen registre må settes slik at de er klare til bruk).

- Tilstandsmaskinen går umiddelbart videre til *Kjør*-tilstanden der den er helt til den har kommet til den siste sblokken. Det er i denne tilstanden selve jobben gjøres. To tellere som angir gjeldende rad og kolonne benyttes til å holde kontroll på hvilke sblokker som prosesseres. Når siste sblokk er prosessert, går tilstandsmaskinen videre til *Neste regelsett*.

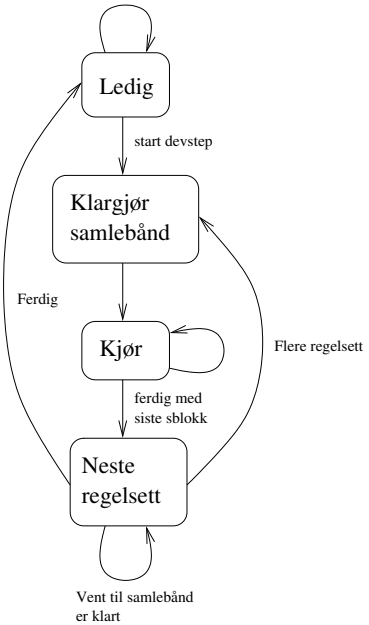
- I tilstand *Neste regelsett* vil først tilstandsmaskinen vente til samlebåndet er helt ferdig (tomt). Deretter vil enten et nytt regelsett hentes inn fra regellageret, og prosessen gjentas, eller kjøringen avsluttes dersom det ikke er flere regelsett igjen.

Developmentenheten jobber etter regler som ligger lagret i et regellager. Det som er viktig å merke seg er at regellageret bare klarer å ha 8 regler (ett regelsett) klare av gangen. Dersom det finnes flere regler enn dette totalt, må developmentenheten skifte ut de 8 reglene med nye regler og kjøre prosessen om igjen. Det vil altså bli én iterasjon for hvert sett med 8 regler som finnes før developmentsteget er ferdig. Regellageret kan inneholde en stor mengde (256) regler lagret i Block SelectRAM, men har altså bare 8 regler tilgjengelige til enhver tid i et hurtigbuffer.

Etter første iterasjon vil BRAM-0 fremdeles være uforandret, men BRAM-1 vil inneholde alle endringer bestemt av de første 8 reglene i regellageret. Alle iterasjoner deretter må i tillegg til å lese fra BRAM-0 for å sjekke betingelsesdelen av reglene, også lese ut verdier fra BRAM-1 for ikke å miste data som ble



Figur 5.10: Samlebånd for developmentsteg



Figur 5.11: Tilstandsmaskin i DEV Control

produsert i forrige iterasjon. Derfor må developmentenheten både kunne lese og skrive til BRAM-1 (se figur 5.2).

Flyten gjennom samlebåndet (i tilstand *Kjør*) kan beskrives slik:

- *Fetch1* og *Fetch2* leser ut alle nye sblokker fra BRAM-ene som er nødvendige for å behandle to nye sblokker i parallell.
- *Setup* samler sammen sblokkdata nettopp lest ut sammen med tidligere utlest data lagret i registre. Siden sblokkdata kan komme fra forskjellige kilder (både BRAM og registre), setter dette samlebåndstrinnet opp alle data slik at de er klare for neste trinn.
- *Execute1* og *Execute2* inneholder 16 enheter (regeltestere) som hver tar seg av å teste betingelsen til en gitt regel på en bestemt sblokk med naboskap. Disse regner også ut resultatet dersom betingelsen slår til. Det finnes 16 stykker fordi 8 regler må testes ut på 2 forskjellige sblokker.
- Dersom noen av reglene over slår til, velger *Select* ut resultatet fra én av disse ut i fra en gitt prioritet. Slår ingen til velges den gamle verdien til sblokken (ingen forandring).
- *Store* lagrer valgt verdi til BRAM-1

5.6 Overføring av sblokkdata over PCI-buss

Sblokktype og sblokktilstand kan både leses og skrives av programvare over PCI-bussen. Begge deler håndteres delvis av *Load Send Store*-samlebåndet, vist i figur 5.12.

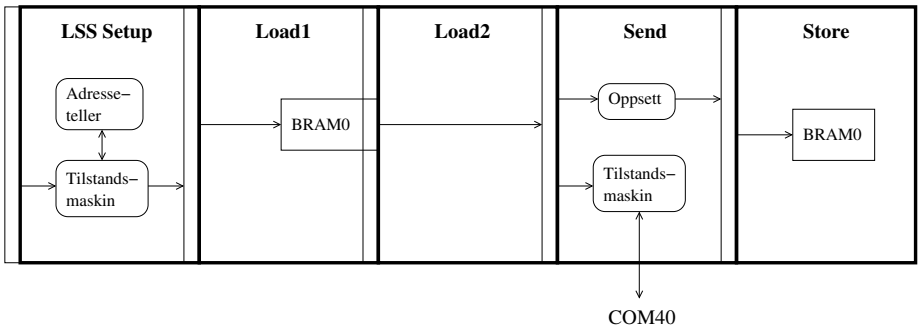
LSS Setup inneholder en tilstandsmaskin som utfører alle instruksjoner som har med LSS-samlebåndet å gjøre. Denne aksepterer nye instruksjoner når den er i *Ledig*-tilstanden. Alle fler-sykel instruksjoner medfører at tilstandsmaskinen hopper til den tilsvarende tilstanden. *Vent*-tilstanden benyttes til å vente til samlebåndet er klart til å motta nye instruksjoner. Én-sykel-instruksjoner (lesing og skriving av enkelt-ord) går direkte til *Vent*-tilstanden.

Lesing

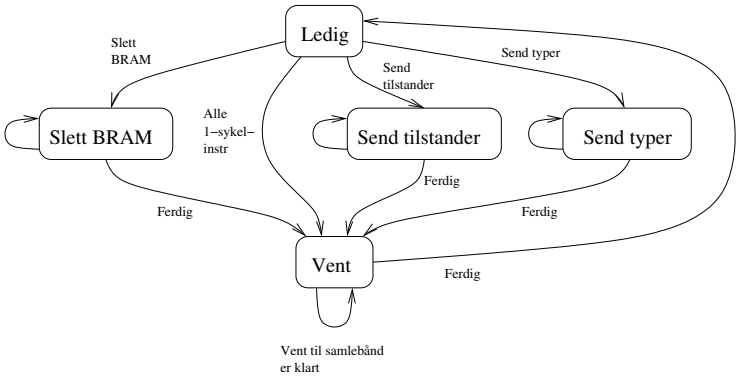
Når programvaren på CPCI-maskinen skal lesing ut sblokkdata, skjer det på denne måten:

- En instruksjon aktiverer LSS
- Data leses ut fra BRAM-0 i *Load1* og *Load2*
- Data sendes over PCI-buss til programvare på CPCI-maskin i *Send*. Dette samlebåndssteget inneholder en tilstandsmaskin som kommuniserer med COM40.

I tillegg er det mulig å lese ut hele sblokkmatrisen (type eller tilstand) med én enkelt instruksjon (håndteres av tilstandene *Send tilstander* og *Send typer*). For å utnytte kapasiteten på PCI-bussen maksimalt pakker denne sammen flere sblokker til ett ord slik at det totale antallet ord som sendes minimeres. Dette styres av tilstandsmaskinen i *LSS Setup*. Det vil altså være én instruksjon som medfører at flere ord sendes over PCI-bussen, pakket med sblokkdata.



Figur 5.12: Samlebånd for LSS



Figur 5.13: Tilstandsmaskin i LSS Setup

Skriving

Når programvaren på CPCI-maskinen skal skrive sblokkdata til BRAM-0, skjer det på denne måten:

- En instruksjon som inneholder sblokkdata som skal skrives sendes over PCI-bussen og aktiverer LSS
- Data leses ut fra BRAM i *Load1* og *Load2*
- Utlest data og data fra instruksjonen flettes sammen i *Send*
- Data skrives tilbake til BRAM-0 i *Store*

Siden databussbredden er lik to sblokker må man alltid skrive minimum to sblokker til BRAM. Har man bare én sblokk som skal skrives må man først lese ut eksisterende data og flette sammen med nye data før de skrives til BRAM.

I tillegg er det mulig å sette alle sblokker i BRAM-0 til en gitt verdi ved hjelp av én instruksjon. Dette håndteres av tilstanden *Slett BRAM*.

5.7 Hazardenhet

Korrekt oppførsel garanteres ved hjelp av en enkel regel som hindrer alle dataavhengigheter i å oppstå.

Hvert delsamlebånd (DEV, SBM, LSS) garanterer at de ikke har noen instruksjoner i sitt delsamlebånd så lenge de er i tilstanden *Ledig*. Dette vil si at tilstandsmaskinene må gå inn i en ventetilstand etter at de er ferdige med sin instruksjon, men før de siste dataene har forlatt enden av samlebåndet.

Dersom ett av delsamlebåndene ikke er i *Ledig*-tilstanden, vil samlebåndstrinnene *PCI Fetch*, *Fetch1*, *Fetch2* og *Decode* «stalles», det vil si at de stopper opp og venter uten å forandre noen av sine registre. Ingen nye instruksjoner vil altså kunne forlate *Decode*-trinnet før alle delsamlebåndene er klare. Dette gjør at det aldri vil befinne seg to instruksjoner samtidig i samlebåndene etter *Decode*-trinnene, og ingen dataavhengigheter vil ha mulighet til å oppstå.

Dette kan virke som en lite effektiv hazard-håndtering, men vil i praksis fungere bra. De fleste instruksjoner bruker svært mange sykler på å fullføre og vil holde hele sitt delsamlebånd i aktivitet i lengre tid. Et lite opphold med tømming og fylling av samlebåndet mellom hver instruksjon vil ha lite å si totalt sett.

En annen ting kunne vært å tillate kjøring av flere delsamlebånd i parallell. Men siden samlebåndene deler de samme enhetene er det vanskelig å tenke seg reelle situasjoner der dette ville hatt noen positiv effekt. Dersom man likevel må vente halvparten av syklene på grunn av dataavhengigheter vil man ikke spare noe på det. Derfor ble det besluttet å unngå dette for ikke å gjøre hazardenheten tregere og mer kompleks enn nødvendig.

Siden koprosessoren er implementert ved hjelp av samlebånd, vil utlesningen av instruksjoner være flere sykler fremfor dekodingen av instruksjoner. Dette vil si at når en gitt instruksjon dekodes (i *Decode*-trinnet) vil det allerede finnes instruksjoner etter denne i samlebåndstrinnene *Fetch1* og *Fetch2*. I forbindelse med kontrollflyt (hopp-instruksjoner) vil disse instruksjonene ikke være gyldige, og må fjernes. Dette kalles «flushing» og utføres av *Decode*-trinnet hver

gang den mottar en kontrollflytinstruksjon (*jump* eller *break*). Ved flushing vil instruksjonene i *Fetch1* og *Fetch2* fjernes.

5.8 Synkronisering mellom frekvensdomener

Kommunikasjon med PCI-FPGA-en må foregå på 40MHz. Siden det er ønskelig å kjøre både development og sblokkmatrise så fort som mulig, er FPGA-en delt inn i to frekvensdomener. Bare en liten tilstandsmaskin (COM40) kjører på 40MHz, resten er lagt til det raske frekvensdomenet.

Datautveksling mellom de to frekvensdomenene skjer ved hjelp av to busser; én for sending av data fra FPGA-en til PCI-bussen, og én for mottaking av data fra PCI-bussen til FPGA-en. Kommunikasjonen er asynkron og bruker en enkel *Request–Acknowledge*-protokoll. Kretsen i det raske frekvensdomenet setter i gang alle overføringer; COM40 er med andre ord alltid slave i forhold til det raske frekvensdomenet. Synkroniseringsprotokollen garanterer at databusser holdes konstante lenge nok, og at kretsen venter til data er klare til mottaking eller sending.

Koblingen mellom de to frekvensdomenene vises i figur 5.14. For å hindre metastabilitet blir alle kontrollsignaler synkronisert til frekvensdomenet de leses i ved hjelp av to etterfølgende vipper.

COM40 er en enkel tilstandsmaskin som bare oversetter forespørslene fra det raske frekvensdomenet til tilsvarende forespørsler over kommunikasjonsbussen mellom PCI-FPGA-en og bruker-FPGA-en. *PCI Fetch* leser instruksjoner. *LSS Send* sender sblokkdata.

Synkroniseringsprotokollen for lesing av data fra PCI-bussen vises i tabell 5.2. PCI Fetch starter en leseoperasjon ved å sette «motta»-signalet. Når dette oppdages av COM40 setter COM40 igang med å hente data fra PCI-FPGA. Når dette er gjort, legger COM40 data på databussen og setter signalet «data mottatt». PCI Fetch leser data fra databussen, og resetter «motta» for å signalisere at databussen ikke er i bruk lengre. COM40 svarer med å resette «data mottatt», og dataoverføringen er avsluttet.

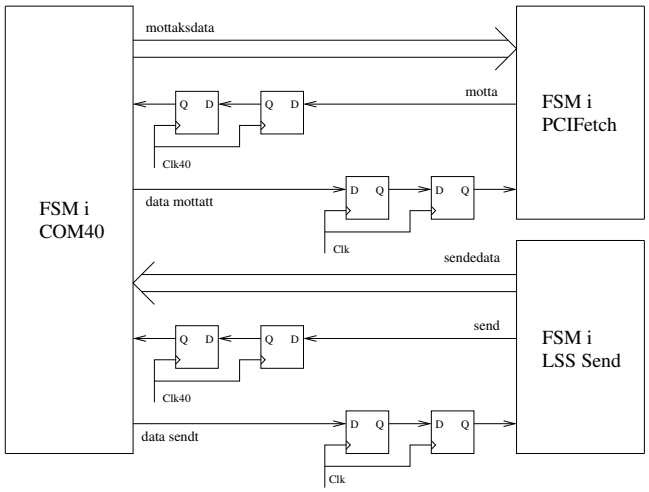
Synkroniseringsprotokollen for skriving av data til PCI-bussen vises i tabell 5.3. Denne fungerer på samme måte som for lesing, bortsett fra at dataoverføringen går motsatt vei.

5.9 BRAM-organisering

Måten sblokkdata lagres på i BRAM blir i stor grad bestemt av måten developmentenheten jobber på. Det finnes to BRAM-er. Hver enkelt av disse BRAM-ene er helt like i oppbygning, og forklares i dette delkapittelet.

Developmentenheten er avhengig av å ha klar data for 8 forskjellige sblokker for hver eneste sykel. Dette er vist i figur 5.15. Det er de svarte sblokkene som skal prosesseres, og da kreves det at data for disse, i tillegg til alle nabosblokker (gråe sblokker), må være tilgjengelige for hver sykel. Da det kun finnes toports RAM i Xilinx VirtexE, vil en naiv implementasjon bare klare å lese ut to sblokker.

Ved å utnytte egenskaper ved developmentsteg-algoritmen, kan man likevel få dette til. Viktige kjennetegn ved kjøring av developmentsteg:



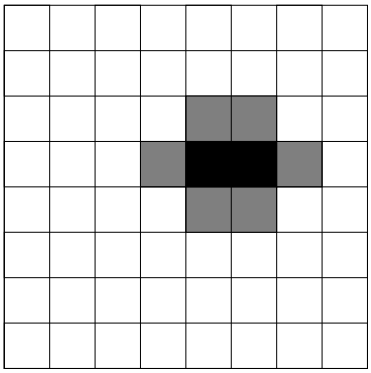
Figur 5.14: Kobling mellom frekvensdomener. Clk40 = 40MHz, Clk = Maksfrekvens

PCI Fetch	COM40
motta = '1'	
	henter data fra PCI-FPGA data mottatt = '1'
leser data fra databuss motta = '0'	
	data mottatt = '0', ferdig
ferdig	

Tabell 5.2: Synkronisering av leseoperasjoner

LSS Send	COM40
send = '1'	
	leser data fra databuss sender data til PCI-FPGA data sendt = '1'
send = '0'	
	data sendt = '0', ferdig
ferdig	

Tabell 5.3: Synkronisering av skriveoperasjoner



Figur 5.15: Sblokker som leses ut fra BRAM

- Developmentenheten prosesserer to nabo-sblokker av gangen
- Den trenger kjennskap til sentersblokkene og alle deres naboer
- Developmentenheten beveger seg alltid fra venstre mot høyre, rad for rad

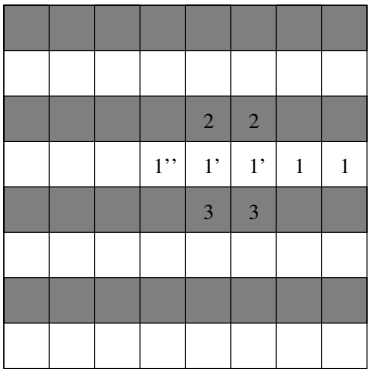
Første observasjon er at alle sblokker som er nødvendige ligger samlet i ett område (riktignok spredd utover tre forskjellige rader). Dette medfører at leser man ut én gitt sblokk er det stor sannsynlighet for at man også trenger sblokken som kommer umiddelbart etterpå. En ting man kan gjøre for å minske antall RAM-oppslag er da å sette databussbredden til BRAM-ene til $2 \cdot$ sblokkstørrelse. Da vil hvert RAM-oppslag alltid gi to etterfølgende sblokker.

Neste observasjon er at man alltid jobber med sblokker fra tre forskjellige rader, og at bare to sblokker trengs fra hver av første og tredje rad. Ved å benytte to uavhengige RAM-blokker kan man legge alle oddetallsrader med sblokker i den ene, og alle partallsrader i den andre. På den måten vil alltid sblokkene i første og tredje rad tilhøre samme RAM-blokk, og sblokkene i midt-raden vil tilhøre den andre RAM-blokken. To leseporter som hver gir ut to sblokker vil dermed være nok til å ta seg av utlesning av alle sblokkene i første og tredje rad.

Siste observasjon er at developmentenheten beveger seg systematisk fra venstre mot høyre, to sblokker av gangen. Ved å ta vare på tidligere utleste sblokker i registre, vil man med en eneste leseport kunne skaffe alle slokkdata nødvendige for den midterste raden.

Tilsammen gir dette developmentenheten tilgang til de data som er nødvendige for hver sykel. Et spesialtilfelle er hva som skjer i begynnelsen og slutten av en rad. Der vil den siste leseporten og et ekstra register sørge for at nok data er tilgjengelig.

Et eksempel er vist i figur 5.16. Figuren viser en sblokkmatrise med hver sblokk representert som kvadrater. Gråe sblokker legges i én RAM-blokk og hvite i en annen. Bruk av leseporter vises for prosessering av to gitte sblokker (merket «1'») midt i sblokkmatrisen. Sblokker merket «1» leses ut fra den ene leseporten i den ene RAM-blokken. Sblokker merket «2» eller «3» leses ut fra leseportene til den andre RAM-blokken. Sblokker merket «1'» er forrige sykels verdi fra leseport 1, og sblokker merket «1"» er den to-sykler gamle verdien fra leseport 1. «1'» og «1"» leses ut fra registre.



Figur 5.16: Organisering av BRAM

Merk at selv om hver BRAM egentlig består av to toports RAM-blokker, behandles de i denne rapporten som om de egentlig består av én fireports RAM-blokk.

5.10 Testing

Systemet har blitt testet underveis i utviklingsprosessen. Funksjonell testing har blitt gjennomført ved å teste enkeltmoduler og sammensatt system i ModelSim ved hjelp av enkle testbenker skrevet i ModelSim-skript.

PCI-kommunikasjonsmodulen og instruksjonsdekodingen var blant de første tingene som kom i kjørbar stand. Dette ga flere fordeler under testing:

- Alle modulene kunne styres gjennom det samme PCI-modulgrensesnittet. Det ble laget et C-program som setter sammen instruksjoner og konverterer disse til PCI-bussaksesser. Dette gjorde det svært enkelt å sette sammen forskjellige instruksjonssekvenser som tester spesifikke deler av kretsen. C-programmet genererer automatisk testbenker i ModelSim-skript ut i fra disse instruksjonssekvensene, noe som lettet jobben med å lage testbenker for alle mulige moduler og situasjoner betraktelig.

- Kretsen kunne tidlig bli testet i ekte maskinvare (på FPGA-en). Dette gjorde det mulig å bruke C-programmet nevnt ovenfor til å kjøre tester på FPGA-en. Når noe ikke virket som det skal, kunne det genererte testbenkskriptet benyttes til å kjøre samme testen i ModelSim for lett å lokalisere og utbedre feilen.

Kapittel 6

Resultat

Dette kapittelet presenterer synteseresultater og viser utfallet av tester som har blitt utført på den ferdige kretsen.

6.1 Syntese

Den ferdige VHDL-koden syntetiseres. Synteseresultater vises i tabell 6.1 for forskjellige verdier for sblokkmatrisens størrelse. De tre siste kolonnene viser ressursforbruk av FPGA-komponenter (TBUF er tristatebuffer).

X-størrelse	Y-størrelse	Hastighet	BRAM	slicer	TBUF
8	8	80MHz	26 (27%)	2905 (23%)	207 (1%)
16	8	80MHz	26 (27%)	3060 (24%)	289 (2%)
16	16	80MHz	26 (27%)	3342 (27%)	435 (3%)
32	16	80MHz	26 (27%)	3920 (31%)	709 (5%)
32	32	80MHz	26 (27%)	5052 (41%)	1239 (9%)

Tabell 6.1: Synteseresultater

Tabellen tar bare med seg et utvalg av alle x- og y-størrelser. Sblokkmatriser større enn $32 \cdot 32$ lar seg ikke syntetisere fordi det er en maksimal grense for antall tristate-buffer som kan drive ett enkelt signal. Denne grensen overstiges av utlesningslogikken når sblokkmatrisen blir større enn $32 \cdot 32$. Sblokkmatrisens størrelse må angis som en toerpotens på grunn av begrensninger i konstruksjonen av koprosessoren.

6.2 Test A: Test av konfigurasjonshastighet

I prosjektet denne hovedoppgaven bygger på [4] ble sblokkmatrisen konfigurert ved å endre FPGA-ens bitfil og så bruke delvis rekonfigurering over JTAG til å endre sblokkmatrisen på selve FPGA-en. I denne hovedoppgaven blir sblokkmatrisen konfigurert ved hjelp av egenkonstruert logikk internt på FPGA-en. Konfigurasjonsdata kan enten sendes over PCI-bussen eller den kan genereres av developmentenheten inne på FPGA-en.

For å sammenligne disse to metodene ble det kjørt et liten test. 1000 fullstendige rekonfigureringer av sblokkmatrisen ble foretatt både av den gamle metoden med JTAG-basert delvis rekonfigurering og av kretsen beskrevet i denne rapporten.

Resultater:

- 1000 rekonfigureringer av en $32 \cdot 32$-sblokkmatrise ved hjelp av JTAG-basert delvis rekonfigurering: *170,38s*
- 1000 rekonfigureringer av en $32 \cdot 32$-sblokkmatrise ved hjelp av kretsen i denne rapporten der alle sblokkmatrisene sendes fullstendig over PCI-bussen: *0,55s*

Dette vil si at JTAG-basert rekonfigurering slik det ble gjort i [4] bruker ca. 309 ganger så lang tid på å rekonfigurere en $32 \cdot 32$-sblokkmatrise som koprosessoren i denne hovedoppgaven.

6.3 Test B: Funksjonell test

Som en endelig test for å sjekke at kretsen fungerer som den skal ble det kjørt et testeksperiment på den. Eksperimentet ble utformet av Svein Arne Aase for å få det på den formen som benyttes i hans hovedoppgave [1]. Det samme eksperimentet ble kjørt på Svein Arne Aases simulator for å se om disse ga de samme resultatene.

Sblokkmatrisen settes til en størrelse på $8 \cdot 8$ sblokker. Det finnes tre sblokktyper:

- Z; Tom, sblokken vil ikke endre sin tilstand
- J; Logisk «eksklusiv eller» av sblokkens fire naboer
- I; Logisk «eller» av sblokkens fire naboer

Prioritet	Type	Senter	Sør	Øst	Nord	Vest
1	Growth fra sør	Z	D	D	D	D
2	Growth fra øst	Z	D	D	D	D
3	Growth fra nord	Z	D	D	D	D
4	Growth fra vest	Z	D	D	D	D
5	Change til J	D	Z	D	I	D
6	Change til Z	D	I	D	D	J

Tabell 6.2: Developmentregler i testeksperiment. D angir at «Don't care», og betyr at sblokkens type ikke har noe å si

Det finnes 6 developmentregler, vist i tabell 6.2. Disse reglene tar kun hensyn til sblokkenes type; sblokkenes tilstand har med andre ord ikke noe å si under kjøring av developmentsteg.

Koprosessoren settes først slik at BRAM-1 inneholder startverdiene i figur 6.1, LUT-konverteringstabellen lastet med definisjonene av alle sblokktypene og regellageret lastet med reglene i tabell 6.2. Deretter settes den til å kjøre følgende løkke, i pseudokode:

0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1

(a) Tilstander

Z	Z	Z	Z	Z	Z	Z	Z
Z	Z	Z	Z	I	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z	Z

(b) Typer

Figur 6.1: Startverdier for testeksperiment

```
for i = 0 to 150
  kjør 77 tilstandssteg
  kjør ett developmentsteg
```

Denne algoritmen blir av Svein Arne Aase kalt «Simple Resident States Algorithm». Eksperimentet kjører i en løkke der sblokkmatrisen først kjøres i 77 sykler (77 tilstandssteg), før det kjøres ett enkelt developmentsteg. Denne løkken kjører 150 ganger; med andre ord totalt 150 developmentsteg. Instruksjonene for koprosessoren som kjører denne løkken er:

0. `config`
   konfigurer sblokkmatrise fra BRAM-1

1. `run 77`
   kjør sblokkmatrise 77 sykler

2. `readback`
   les tilbake tilstander fra sblokkmatrise til BRAM-1

3. `switch`
   bytt om BRAM-0 og BRAM-1

4. `readTypes`
   send alle sblokktyper over PCI-bussen til programvaren på CPCI-maskinen. Dette gjøres hver iterasjon for å kunne se om kretsen oppfører seg helt likt simulatoren.

5. `readStates`
   send alle sblokktilstander over PCI-bussen til programvaren på CPCI-maskinen. Dette gjøres hver iterasjon for å kunne se om kretsen oppfører seg helt likt simulatoren.

6. `devstep`
   kjør et developmentsteg fra BRAM-0 til BRAM-1

7. `jump 0`
   hopp tilbake til instruksjon 0

1	1	1	1	1	1	1	1
0	0	1	0	1	1	1	1
0	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1
1	1	1	1	0	1	0	0
1	1	1	0	0	1	0	1
0	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0

(a) Tilstander

J	I	I	I	I	J	J	I
J	J	J	J	J	J	J	J
Z	Z	Z	Z	Z	Z	J	Z
I	I	I	I	I	I	J	I
I	J	J	J	J	J	J	Z
J	J	J	J	J	J	J	I
J	I	I	I	I	I	I	J
I	I	I	I	I	I	J	J

(b) Typer

Figur 6.2: Sluttverdier for testeksperiment

Etter kjøring hadde sblokkmatrisen verdiene i figur 6.2. Resultatene fra kjøringen på FPGA ble sammenlignet med resultatene fra kjøringen på simulatoren og funnet identiske.

6.4 Test C: Hastighetstest

Testeksperimentet i kapittel 6.3 var for lite til å gi noen brukbare mål på hastighet. Derfor ble det samme eksperimentet kjørt med 50 000 tilstandssteg og 10 000 developmentsteg.

Dette ga følgende tidsforbruk:

- Simulator: 18 minutter og 6 sekunder
- Koprosessor i FPGA: 6,2 sekunder

Dette vil altså si at simulatoren bruker ca. 175 ganger så lang tid på å kjøre akkurat dette eksperimentet.

Kapittel 7

Diskusjon

7.1 Testresultater

Simuleringer i ModelSim antyder at koprosessoren fungerer som den skal. Dette støttes av testene B og C som kjører større eksperimenter uten å gi feilaktige svar. Disse testene representerer ikke en fullstendig test av systemet, men viser at kretsen fungerer tilsynelatende bra. Det finnes ingen kjente feil.

Test A viser at systemet presentert i denne oppgaven kan konfigurere en sblokkmatrise raskere enn ved å bruke JTAG og delvis rekonfigurering. Den store forskjellen kommer delvis av at JTAG-bussen er tregere enn PCI-bussen, delvis av at det må sendes mer data over JTAG-bussen for hver sblokkmatrisekonfigurasjon (på grunn av måten VirtexE er konstruert på) og delvis fordi FUSE biblioteket bruker en del tid på å starte hver JTAG-konfigurasjon.

Test C viser også at hastighetsøkningen kan være stor i forhold til å kjøre eksperimentet i programvare. Hastighetsøkningen vil riktignok variere avhengig av eksperiment, men for eksperimenter denne kretsen er optimalisert for vil hastighetsøkningen være betydelig. Test C var av en type som passer svært godt for denne kretsen fordi den har mange tilstandssteg. Noen omfattende ytelsestesting av kretsen har ikke blitt gjennomført.

7.2 Arkitektur

Koprosessoren er konstruert med tanke på å kjøre et stort antall tilstandssteg, og kan derfor gjøre dette effektivt. Dersom antall tilstandssteg i hver iterasjon av eksperimentløkken er lite, vil mye av tiden gå med til konfigurering og tilbakelesning av sblokkmatrisedata. Et problem relatert til dette er I/O. Slik koprosessoren fungerer nå er det lagt opp til at starttilstanden til sblokkmatrisen er inndata, og slutt-tilstanden er utdata. Med et stort antall tilstandssteg mellom disse vil dette kjøres effektivt. Dersom det er nødvendig å gi nye inndata underveis i kjøringen (for hver sykel i verste tilfelle), må dette gjøres ved gjentatt tilbakelesning og konfigurering noe som raskt vil dominere tidsbruken. Dersom tilstander må leses ut flere ganger underveis i kjøring av tilstandssteg vil dette medføre gjentatte tilbakelesninger av data. Begge deler burde likevel være raskere enn tilsvarende operasjoner i programvare, gitt samme generasjon av FPGA og CPU.

Siden kretsen styres av enkle instruksjoner og ikke av en hardkodet tilstandsmaskin, er det relativt enkelt å variere oppførselen til kretsen avhengig av hvilket eksperiment som ønskes kjørt. Dette krever ingen modifisering av VHDL-koden. Fleksibiliteten er likevel ikke som i en simulator. Må oppførselen endres betydelig, for eksempel endre algoritme for developmentsteg, kreves det desto større innsats. VHDL-kode er tyngre å sette seg inn i og modifisere enn kode skrevet i et høynivå programmeringsspråk.

7.3 Optimalisering

Kretsen er ikke optimal. Det er fullt mulig å redusere mengden logikk uten å redusere funksjonaliteten. Flere plasser er det mulig å gjenbruke registre og tellere. Plass har ikke vært noe problem i denne hovedoppgaven, så det har i stedet vært fokusert på oversiktlig arkitektur og kode.

Tilsvarende kan man spare en god del logikk ved å slå sammen samlebåndsstegene *Decode*, *Dev Control*, *SBM Control* og *LSS Setup*. Dette kan gjøres uten å skape timingproblemer. Koprosessoren vil faktisk bli (marginalt) raskere fordi samlebåndet blir kortet ned med ett trinn. Ønsket om å holde disse adskilt på grunn av oversiktlighet veide likevel tyngre.

Det er også mulig å optimalisere kretsen ved å spesifisere strengere «constraints» til synteseverktøyet. Dette vil si å gi tydeligere beskjed om hvor komponenter kan plasseres fysisk, og hvilke timing-krav som stilles. Det er forventet at man kan få opp klokkefrekvensen ved å gjøre dette.

7.4 Videre arbeid

Flere forbedringer av kretsen er mulig. Koprosessoren er ikke generell og kan ikke utføre generelle oppgaver. Dersom mer fleksibilitet kreves, for eksempel for å kjøre en genetisk algoritme for å finne developmentregler, vil dette måtte kjøres i programvare. Ulempen er trafikk over PCI-bussen. Én løsning på dette kan være å utvide koprosessoren med et turing-komplett instruksjonssett. En annen løsning er å benytte en Virtex II-Pro FPGA. Denne FPGA-en inneholder PowerPC-prosessorkjerner som kan kjøre et hvilket som helst program. Siden PowerPC-kjernen er tett knyttet til den egenkonstruerte logikken på FPGA-en, vil kommunikasjon mellom disse være svært effektivt, sett i forhold til kommunikasjon over PCI-bussen slik det er nå.

En annen forbedring kan være en mer fleksibel måte å sette sblokkmatrisestørrelsen på. Nå må størrelsen gis som en toerpotens på grunn av begrensninger i kontroll-logikken. Dersom dette ikke er godt nok kan man relativt enkelt utvide kretsen slik at det er mulig å benytte en hvilken som helst størrelse. Kostnaden er noe større kontroll-logikk. Det er også mulig å endre utlesningslogikken slik at antall tristate-buffer som kan drive en linje ikke vil hindre syntetisering av store sblokkmatriser. Dette er en nødvendig utvidelse dersom $32 \cdot 32$-sblokkmatriser ikke er store nok.

Tillegg A

Instruksjonsmanual

Alle instruksjoner har følgende generelle format:

operander	størrelse	opkode
n–6	5	4–0

- *Opkode*; Unik identifikator for hver instruksjon
- *Størrelse*; Signaliserer om instruksjonen er på 32 eller 64 bit
- *Operander*; Varierer fra instruksjon til instruksjon

64-bit instruksjoner sendes over PCI-bussen i to skriveoperasjoner; først sendes det minst signifikante ordet, deretter det mest signifikante ordet.

A.1 break

Avslutt kjøring av instruksjoner fra instruksjonslager. Benyttes for å avslutte et program og begynne å akseptere instruksjoner fra PCI-bussen i stedet.

ubrukt	0	01101
31–6	5	4–0

A.2 clearBRAM

Sett hele BRAM 0 til en gitt verdi (både tilstand og type).

tilstand	ubrukt	type	ubrukt	0	10011
31	30–29	28–24	23–6	5	4–0

- *type*; Type alle sblokker skal settes til
- *tilstand*; Tilstand alle sblokker skal settes til

A.3 config

Konfigurer sblokkmatrise med data fra BRAM 1.

ubrukt	0	00111
31–6	5	4–0

A.4 devstep

Kjør et developmentsteg. Data leses fra BRAM 0 og skrives til BRAM 1. Sblokker med type 0 regnes som tomme sblokker de stedene der dette er relevant for oppførselen.

ubrukt	0	01010
31–6	5	4–0

A.5 end

Stopp lagring av instruksjoner. Denne instruksjonen vil ikke bli lagret i instruksjonslageret.

ubrukt	0	01111
31–6	5	4–0

A.6 jump

Start utføring av instruksjoner som er lagret i instruksjonslageret.

ubrukt	adresse	ubrukt	0	01100
31–16	15–8	7–6	5	4–0

- *adresse*; Hopp til denne adressen i instruksjonslageret

A.7 nop

Ingen operasjon.

ubrukt	0	00000
31–6	5	4–0

A.8 readback

Les tilbake tilstandsdata fra sblokkmatrisen og lagre disse i BRAM 1.

ubrukt	0	01000
31–6	5	4–0

A.9 readState

Les en enkelt sblokks tilstand fra BRAM 0 og send den til PCI-bussen.

ubrukt	x	y	ubrukt	0	00101
31–24	23–16	15–8	7–6	5	4–0

- x; Sblokkens x-posisjon
- y; Sblokkens y-posisjon

A.10 readStates

Les alle tilstander ut fra BRAM 0 og send de til PCI-bussen. 32 tilstander sendes av gangen i et ord. I de tilbakeleste ordene vil mest signifikante bit (31) inneholder sblokken som er lengst til venstre av de 32, og minst signifikante bit (0) inneholder sblokken som er lengst til høyre av de 32.

ubrukt	0	10001
31–6	5	4–0

A.11 readType

Les en enkelt sblokks type fra BRAM 0 og send den til PCI-bussen.

ubrukt	x	y	ubrukt	0	00010
31–24	23–16	15–8	7–6	5	4–0

- x; Sblokkens x-posisjon
- y; Sblokkens y-posisjon

A.12 readTypes

Les ut alle typer fra BRAM 0 og send de til PCI-bussen. 4 typer sendes av gangen i et ord, og kun 20 bit vil altså brukes i hver ord. I de tilbakeleste ordene vil de mest signifikante bits (15–19) tilsvare sblokken lengst til venstre av de 4, og de minst signifikante bits (0–4) tilsvare sblokken lengst til høyre av de 4.

ubrukt	0	10000
31–6	5	4–0

A.13 run

Kjør sblokkmatrise.

sykler	ubrukt	0	01001
31–8	7–6	5	4–0

- *sykler*; Antall sykler sblokkmatrisen skal kjøres

A.14 setNumberOfLastRule

Sett nummeret på den regelen i regellageret som har høyest verdi. Dette avgrenser antall regler developmentenheten må ta hensyn til under kjøring av developmentsteg. Alle regler opp til og med dette nummeret vil bli brukt.

ubrukt	nummer	ubrukt	0	10010
31–16	15–8	7–6	5	4–0

- *nummer*; Nummer på den høyest prioriterte regelen i regellageret.

A.15 store

Alle etterfølgende instruksjoner fra PCI-bussen skal lagres i instruksjonslageret, helt til det kommer en *end*-instruksjon.

ubrukt	adresse	ubrukt	0	01110
31–16	15–8	7–6	5	4–0

- *adresse*; Adressen hvor programmet skal lagres fra

A.16 switch

Bytt om på de to BRAM-ene.

ubrukt	0	00011
31–6	5	4–0

A.17 writeLUTConv

Skriv til LUT konverteringstabell. Denne tabellen benyttes for å oversette et gitt sblokktypenummer til en 32-bits LUT når sblokkmatrisen konfigureres.

lut	ubrukt	nummer	ubrukt	1	01001
63–32	31–13	12–8	7–6	5	4–0

- *lut*; 32-bits LUT som skal skrives til tabellen
- *nummer*; Typenummeret denne LUT-en skal gjelde for

A.18 writeRule

Skriv regel til regellageret. Disse benyttes av developmentenheten.

regel	nummer	ubrukt	1	01011
63–15	14–7	6	5	4–0

- *regel*; Regel som skal skrives. Denne følger formatet gitt i tillegg B.
- *nummer*; Nummer på regel. Høyt nummer betyr høy prioritet. Hver regel må ha sitt unike nummer, og prioritet mellom regler er dermed entydig gitt.

A.19 writeState

Skriv en enkelt sblokks tilstand til BRAM 0.

tilstand	ubrukt	x	y	ubrukt	0	00100
31	30–24	23–16	15–8	7–6	5	4–0

- *tilstand*; Tilstand som skal skrives
- x; Sblokkens x-posisjon
- y; Sblokkens y-posisjon

A.20 writeType

Skriv en enkelt sblokks type til BRAM 0.

tilstand	ubrukt	type	x	y	ubrukt	0	00001
31	30–29	28–24	23–16	15–8	7–6	5	4–0

- *type*; Type som skal skrives
- x; Sblokkens x-posisjon
- y; Sblokkens y-posisjon

Tillegg B

Regelformat

Regler beskrives internt på samme måte som de kodes i *writeRule*-instruksjonen (se tillegg A.18).

Hver regel består av følgende felt:

gyldig	type	betingelse	resultat
48	47	46–7	6–0

- *Gyldig*; Angir om denne regelen er gyldig eller ikke. Dersom den ikke er gyldig vil ikke developmentenheten ta hensyn til denne regelen under kjøring av developmentsteg.
- *Regeltype*; Angir hvilken type regel dette er. Det finnes to typer:
  - Type 0 – Change; Regelen er av type *Change*
  - Type 1 – Growth; Regelen er av type *Growth*

  Regeltypen bestemmer til dels hvordan betingelsen skal tolkes og hvordan resultatet skal beregnes.
- *Betingelse*; Angir hvilken betingelse som må være oppfylt for at regelen skal slå til. Se B.1 for fyldigere beskrivelse.
- *Resultat*; Hva som skal skje dersom denne regelen slår til. Se B.2 for mer informasjon.

B.1 Betingelse

Betingelsen består av 5 identiske felter, som spesifiserer hvordan sblokkene i naboskapet til sblokken som undersøkes skal være for at regelen skal slå til:

nord	sør	øst	vest	senter
46–39	38–31	30–23	22–15	14–7

Hver av disse spesifiserer hver sin sblokk, og kodes på samme måte:

overse tilstand	tilstand	overse type	type
7	6	5	4–0

- *Overse tilstand*; Sblokkens tilstand vil ikke ha noe å si for om betingelsen er oppfylt eller ikke.
- *Tilstand*; Dersom *Overse tilstand* ikke er satt, må sblokkens tilstand ha samme verdi som dette feltet
- *Overse type*; Sblokkens type vil ikke ha noe å si for om betingelsen er oppfylt eller ikke.
- *Type*; Dersom *Overse type* ikke er satt, må sblokkens type ha samme verdi som dette feltet.

I tillegg er det et krav at for *Change*-regler så må senter-sblokken ha en type forskjellig fra 0 (som tolkes som tom). Tilsvarende må *Growth*-regler ha en sblokktype i sblokken det skal kopieres fra som er forskjellig fra 0.

B.2 Resultat

Resultatfeltet sier hva som skal skje med sblokken dersom regelen slår til. Resultatfeltet avhenger av hvilken regeltype det er snakk om.

Change For *Change* kodes resultatet slikt:

ikke forandre tilstand	forandre tilstand til	forandre type til
6	5	4–0

- *Ikke forandre tilstand*; Angir at denne regelen ikke skal oppdatere tilstand. Det er altså kun sblokkens type som blir forandret.
- *Forandre tilstand til*; Dersom *Ikke forandre tilstand* ikke er satt, vil sblokkens tilstand forandres til denne verdien.
- *Forandre type til*; Sblokkens type vil forandres til denne verdien.

Growth For *Growth* kodes resultatet slikt:

ikke forandre tilstand	ubrukt	kopiér fra
6	5–2	1–0

- *Ikke forandre tilstand*; Angir at denne regelen ikke skal oppdatere tilstand. Det er altså kun sblokkens type som blir forandret.
- *Kopiér fra*; Kopier sblokk type (og tilstand dersom *Ikke forandre tilstand* ikke er satt) fra sblokken som ligger i angitt retning:
  - *00*; Kopiér fra nord
  - *01*; Kopiér fra sør
  - *10*; Kopiér fra øst
  - *11*; Kopiér fra vest

Tillegg C

Grensesnitt mellom PCI-FPGA og bruker-FPGA

Kommunikasjonsbussen mellom PCI-FPGA og bruker-FPGA består av en 32-bits databuss/adressebuss (ADIO), samt følgende kontrollsignaler:

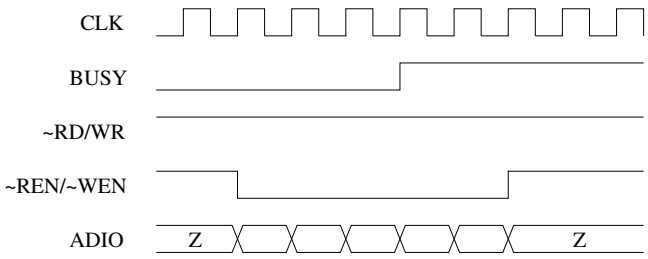
- *AS/˜DS*: Høy: adresse sendes på ADIO, lav: data sendes på ADIO. Dette signalet sier hvilken dataordtype som sendes over ADIO. To typer finnes: Adresse og data. Bruker-FPGA-en og programvaren som sender disse bestemmer hvordan disse skal tolkes.
- *EMPTY*: FIFO-buffer er tomt
- *BUSY*: FIFO-buffer er fullt. Etter at BUSY går høy er det mulig å skrive opp til to dataord uten at data går tapt.
- *˜R/W*: Høy: bruker-FPGA skal skrive til buffer, lav: bruker-FPGA skal lese fra buffer
- *˜REN/˜WEN*: «Enable»-signal. Aktivt lav

˜R/W og ˜REN/˜WEN styres av bruker-FPGA-en. Resten styres av PCI-FPGA-en. Bruker-FPGA-en fungerer som herre og setter igang alle overføringer.

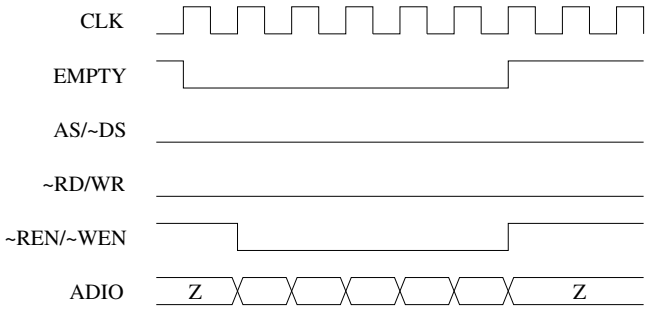
Klokken må gå på 33MHz–40MHz. Dette betyr at kommunikasjonshastigheten begrenses av hastigheten på CPCI-bussen som har en øvre teoretisk grense ved 132MB pr. sekund [18].

Figur C.1 viser et eksempel på skriving til FIFO-buffer. Eksempelet demonstrerer også at bufferet kan «flyte over» med to dataord uten at data går tapt. Figur C.2 viser et eksempel på lesing fra FIFO-buffer.

Se brukermanualen til BenERA [15] for mer informasjon.



Figur C.1: Eksempel på skriving til FIFO-buffer



Figur C.2: Eksempel på lesing fra FIFO-buffer

Tillegg D

FUSE

FUSE er et programvarebibliotek som benyttes til å kommunisere med BenERA-kortet. Det er hovedsaklig to ting FUSE brukes til; Konfigurering av bruker-FPGA og kommunikasjon med bruker-FPGA.

Tabell D.1 viser en oversikt over relevante FUSE-funksjoner. Mer informasjon om disse finnes i brukermanualen til FUSE [16].

D.1 Oppsett

Før bruk må bibliotek og kort initialiseres. Først må det søkes etter tilgjengelige kort med `DIME_LocateCard()`. Deretter må ett av de åpnes med `DIME_OpenCard()`. `DIME_SetOscillatorFrequency()` må brukes til å sette frekvensen på klokkene som går inn til bruker-FPGA-en.

Ressurser må frigis etter bruk ved å kalle `DIME_CloseCard()` og `DIME_CloseLocate()`.

D.2 Konfigurering

Konfigurering av bruker-FPGA skjer med funksjonen `DIME_ConfigSetBitsFilenameAndConfig()` der filnavnet til bitfilen angis. FUSE vil automatisk laste inn bitfilen og konfigurere kortet. Etter konfigurasjon må kortet resettes med `DIME_CardResetControl()`.

D.3 Kommunikasjon

Kommunikasjon mellom C-programmer på CPCI-maskinen og kretsen på bruker-FPGA-en må skje gjennom FUSE. DMA-overføringer er mest effektive. Disse krever et lagerområde spesielt satt opp for formålet med funksjonen `DIME_LockMemory()` og en åpen DMA-kanal satt opp med `DIME_DMAOpen()`. Deretter kan det skrives til og leses fra FIFO-bufferet i PCI-FPGA-en ved hjelp av funksjonene `DIME_DMAWriteFromLockedMem()` og `DIME_DMAReadToLockedMem()`.

Funksjonsnavn	Beskrivelse
DIME_DMAWriteFromLockedMem()	Skriv til FIFO (DMA-overføring)
DIME_DMAReadToLockedMem()	Les fra FIFO (DMA-overføring)
DIME_LocateCard()	Finn kort koblet til maskinen
DIME_OpenCard()	Åpne kort
DIME_LockMemory()	Lås hukommelse (for bruk i DMA-overføringer)
DIME_DMAOpen()	Åpne DMA-kanal til kort
DIME_JTAGControl()	Sett JTAG-hastighet (konfigurasjonshastighet)
DIME_SetOscillatorFrequency()	Set FPGA-klokkefrekvens
DIME_CloseCard()	Lukk kort
DIME_CloseLocate()	Frigi ressurser ifm. DIME_LocateCard()
DIME_ConfigSetBitsFilenameAndConfig()	Konfigurer kort med gitt bitfil
DIME_CardResetControl()	Reset FPGA

Tabell D.1: FUSE-funksjoner

Tillegg E

Oppsett, syntetisering og kjøring

Dette tillegget beskriver hvordan man skal gå frem for å syntetisere koprosessoren og kompilere testprogramvaren.

E.1 Filhierarki

Følgende filer finnes:

- *Makefile*: Toppnivå makefil, brukes både til syntetisering av maskinvare og til kompilering av programvare. Her settes konstanter som konfigurerer systemet (størrelse på sblokkmatrise).
- **sblock_matrix**: Underkatalog for maskinvare.
  - *Makefile*: Makefil for maskinvare, kalles fra toppnivå-makefil og tar seg av syntese.
  - *VHDL-filer*: Se tillegg F.
  - *sblock_matrix.ucf*: User Constraint File, inneholder blant annet pinneallokeringer og tidskrav til FPGA-en.
- **sblocktest**: Underkatalog for testprogram.
  - *Makefile*: Makefil for testprogram, kalles fra toppnivå-makefil og tar seg av kompilering.
  - *C-filer*: Se tillegg G.

E.2 Spesifisering av konfigurasjon

Noen egenskaper ved den ferdige kretsen kan spesifiseres før syntese. Dette gjøres i toppnivå-makefilen.

Det er tre ting som kan konfigureres bare ved å sette en konstant i makefilen:

- SYNT_DEVICE: Velger hvilken FPGA det skal syntetiseres til. Bruk xcv1000e-6-fg860 for å syntetisere til BenERA med VirtexE-1000.

- COORD_SIZE_X: Velger størrelse i x-retning på sblokkmatrisen. Størrelsen vil være toerpotensen av dette tallet. Velges f.eks 5 vil sblokkmatrisen være $2^5 = 32$ sblokker i x-retning.
- COORD_SIZE_Y: Velger størrelse i y-retning på sblokkmatrisen.

Minimal sblokkmatrisestørrelse er $8 \cdot 8$. Maskinvarebegrensninger i VirtexE-1000 begrenser størrelsen oppad til $32 \cdot 32$.

E.3 Syntese

Syntese forutsetter et GNU-miljø med Xilinx ISE installert. Dette kan være GNU/Linux med Wine, eller MS Windows med Cygwin. Start syntese slik (øverst i filhierarkiet):

```
make hardware
```

Dette produserer en bitfil «sblock_matrix.bit» i sblock_matrix-katalogen. Denne kan benyttes til å konfigurere FPGA-en med koprosessoren.

E.4 Testprogram

Testprogrammet kjører eksperimentet beskrevet i kapittel 6.3. Kompilering og kjøring forutsetter et GNU-miljø, f.eks GNU/Linux. Start kompilering (øverst i filhierarkiet) slik:

```
make software
```

Dette produserer et kjørbart program i sblocktest-katalogen kalt «sblocktest». Sblocktest tar følgende argumenter:

```
sblocktest <bitfil> <kort> <modelsim skriptfil> <utdatafil>
```

- `bitfil`: sti og filnavn til bitfilen som skal brukes til å konfigurere bruker-FPGA-en med
- `kort`: nummer på BenERA-kort som skal benyttes, f.eks 1 eller 2
- `modelsim skriptfil`: navn på fil modelsim testbenk skal skrives til
- `utdatafil`: navn på fil som utdata skal skrives til. Resultatene fra eksperimentet skrives hit

Tillegg F

Oversikt over VHDL-filer

Her er en oversikt over alle VHDL-filer som benyttes til syntetisering av koprosessoren.

F.1 Toppnivå

- *package.vhd.in*: Pakke med konstanter og komponentdeklarasjoner. Denne blir preprosessert med GNU-verktøyet m4 for å få satt noen konstanter fra Makefile.
- *funct_package.vhd*: Pakke med noen funksjoner.
- *toplevel.vhd*: Toppnivå.

F.2 Sblokkmatrise

Dette er den kjørbare sblokkmatrisen:

- *sblock_matrix.vhd*: Sblokkmatrise, satt sammen av sblokker.
- *sblock.vhd*: En enkelt sblokk.

F.3 Samlebånd

Dette er alle samlebåndsstegene:

- *fetch.vhd*: Samlebåndsstegene *Fetch1*, *Fetch2*, *PCI Fetch* og *Store*.
- *decode.vhd*: *Decode*-samlebåndssteg.
- *dev.vhd*: DEV-samlebånd.
- *sbm_pipe.vhd*: SBM-samlebånd.
- *lss.vhd*: LSS-samlebånd.

F.4 Andre enheter

Enheter som ikke havner direkte under et samlebåndsteg:

- *hazard.vhd*: Hazard-enhet.
- *rule_storage.vhd*: Regel-lager.
- *sbm_bram_mgr.vhd*: SBM BRAM MGR, grensesnitt mot de to BRAM-ene som inneholder sblokkmatrisedata.
- *instrmem.vhd*: Instruksjonslager.
- *lutconv.vhd*: LUT-konverteringstabell.
- *com40.vhd*: Tilstandsmaskin for kommunisering med PCI-FPGA-en.

F.5 Småmoduler

VHDL-kode som benyttes flere ganger, og som derfor er trukket ut i småmoduler som kan instansieres på mer enn én plass:

- *addr_gen.vhd*: Setter sammen en BRAM-adresse gitt x- og y-posisjon til sblokk.
- *counter.vhd*: Generell teller.
- *rule_exec.vhd*: Tester og kalkulerer resultat for en gitt regel på en gitt sblokk (med naboskap). Del av DEV-samlebåndet.
- *rule_select.vhd*: Velger ut ett resultat fra et sett med regler som har testet mot samme sblokk. Del av DEV-samlebåndet.
- *sbm_bram.vhd*: BRAM brukt til lagring av sblokkmatrisedata, del av SBM BRAM MGR.
- *word_select.vhd*: Et skiftregister som benyttes for å aktivere et sett med enable-signaler, en etter en.

Tillegg G

Oversikt over C-filer

Her er en oversikt over alle C-filer som benyttes til kompilering av testprogrammet. Dette testprogrammet kjører eksperimentet i kapittel 6.3, men kan enkelt endres til å kjøre andre eksperimenter.

- *types.h*: Generelle typer og konstanter
- *sblocklib.h*: Header-fil til sblocklib
- *sblocktest.c*: C-fil for testprogram
- *sblocklib.c*: C-fil for programvarebiblioteket sblocklib

For å gjøre det lettere å styre koprosessoren har det blitt laget et lite bibliotek kalt *sblocklib*. Dette fungerer som et abstraksjonslag over FUSE API og tilbyr noen skreddersydde funksjoner for styring av koprosessoren. Funksjonene er dokumenterte i *sblocklib.h*. Se kildekoden til testprogrammet for eksempel på bruk av dette biblioteket.

Bibliografi

[1] Aase, S. A. Investigations into a Knowledge Rich Approach to Rule Based Development on an Sblock Platform. Teknisk rapport, Norges teknisk-naturvitenskapelige universitet, April 2003. Hovedoppgave.

[2] Banks, E. R. Information Processing and Transmission in Cellular Automata. Teknisk rapport, MIT/LCS/TR-81, Massachusetts Institute of Technology, januar 1971. PhD thesis.

[3] de Garis, H. Artifical Life: Growing an Artificial Brain with a Million Neural Net Modules Inside a Trillion Cell Cellular Automata Machine. I *4th International Symposium on Micro Machine & Human Science*. 1993.

[4] Djupdal, A. Sblokkmatrise på BenERA. Teknisk rapport, Norges teknisk-naturvitenskapelige universitet, desember 2002. Prosjektrapport.

[5] Haddow, P. og Tufte, G. An Evolvable Hardware FPGA for Adaptiv Hardware. I *Congress on Evolutionary Computation, CEC2000*, ss. 553–560. 2000.

[6] Haddow, P. og Tufte, G. Bridging the Genotype-Phenotype Mapping for Digital FPGAs. I *The third NASA/DoD Workshop on Evolvable Hardware, EH'01*, ss. 109–115. 2001.

[7] Haddow, P. og Tufte, G. Building Knowlage into Developmental Rules for Circuit Design. I *5th International Conference on Evolvable Systems (ICES03), Lecture Notes in Computer Science*. Springer, 2003.

[8] Haddow, P., Tufte, G. og van Remortel, P. Shrinking the Genotype: L-systems for EHW? I *The 4th International Conference on Evolvable Systems: From Biology to Hardware, ICES2001*, ss. 128–129. 2001.

[9] IEEE. *Standard 1149.1, Boundary Scan (JTAG).*

[10] Keymeulen, D., Zebulum, R. S., Jin, Y. og Stoica, A. Fault-Tolerant Evolvable Hardware Using Field-Programmable Transistor Arrays. *IEEE Transactions on Reliability*, bind 49(3):ss. 305–316, September 2000.

[11] Kitano, H. Building complex systems using development process: An engineering approach. I *In Evolvable Systems: from Biology to Hardware, ICES, Lecture Notes in Computer Science*, ss. 218–229. Springer, 1998.

[12] Lindenmayer, A. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, bind 18:ss. 280–315, 1968.

[13] Miller, J. F. og Hartmann, M. Untidy Evolution: Evolving Messy Gates for Fault Tolerance. I *4th International Conference on Evolvable Systems: From Biology to Hardware*, ss. 14–25. Springer, 2001.

[14] BenERA Product Highlights.
URL `http://www.nallatech.com/solutions/products/embedded_systems/dime2/rc_systems/benera/`

[15] NallaTech. *BenERA Users Guide*.

[16] NallaTech. *FUSE Users Guide*.

[17] Ortega-Sánches, C. og Tyrrell, A. Fault-Tolerant Systems: The Way Biology Does it! I *Proceedings Euromicro*, ss. 146–151. IEEE CS Press, Budapest, September 1997.

[18] PCI Industrial Computer Manufacturers Group (PICMG). *CompactPCI Specification*.
URL `http://www.picmg.org/compactpci.stm`

[19] The National Technology Roadmap for Semiconductors. Teknisk rapport, Semiconductor Industry Association, 1997.

[20] Sipper, M. *Evolution of Parallel Cellular Machines. The Cellular Programming Approach*. Springer, 1997.

[21] Thompson, A. *Hardware Evolution. Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*. Springer, London, UK, 1998.

[22] VMIACC-0320 Product Overview.
URL `http://www.vmic.com/products/embeddedpc/products/hw_sbc_acc_0320.html`

[23] VMICPCI-7760 Product Overview.
URL `http://www.vmic.com/products/embeddedpc/products/hw_sbc_cpci_7760.html`

[24] von Neumann, J. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL, USA, 1966. Fullført etter von Neumanns død av Arthur Burks.

[25] Xilinx. *Libraries Guide*.
URL `http://toolbox.xilinx.com/docsan/xilinx5/manuals.htm`

[26] Xilinx. *VirtexE Complete Datasheet*.
URL `http://direct.xilinx.com/bvdocs/publications/ds022.pdf`

[27] Xilinx. *Xilinx Application Note 138, V2.7*.
URL `http://support.xilinx.com/xapp/xapp138.pdf`

[28] Xilinx. *Xilinx Application Note 151, V1.5*.
URL `http://support.xilinx.com/xapp/xapp151.pdf`

Register