NTNU
Det skapende universitet

Selvrekonfigurering av FPGA

Fredrik Gravdal

Master i elektronikk
Oppgaven levert: Juni 2007
Hovedveileder: Kjetil Svarstad, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Bakgrunnen til denne oppgaven er å lage dynamiske hardwareløsninger på FPGA, der FPGA-en selv gjør rekonfigureringen. Oppgaven tar utgangspunkt i programmet BITAnalyse som er laget i forbindelse med en tidligere masteroppgave ved NTNU. Det skal implementeres et softwareprogram til FPGA som kan overskrive deler av sin egen oppstartsmatrise. Dette programmet kan fungere som en prototype for å vise hvordan dynamiske moduler på FPGA kan implementeres.

Mål for oppgaven:

1: Spesifisere et program for microblaze som kan rekonfigurere FPGA-en det kjører på.

2: Kartlegge tekniske rammer for selvrekonfigurering av FPGA

3: Teste tekniske krav

4: Implementere et program for microblaze som gjør dette

Oppgaven gitt: 23. januar 2007
Hovedveileder: Kjetil Svarstad, IET

Forord

Arbeidet med denne avhandlingen representerer fullførelse av mitt toårige masterstudium i elektronikk ved Norges Teknisk-Naturvitenskapelige Universitet.

Arbeidet med denne avhandlingen har vært en utfordrende og krevende prosess, men likevel veldig positiv og lærerik. Det har til tider vært prøvende og frustrerende, men jeg angrer ikke på valg av tema. Selvrekonfigurering av FPGA er et svært interessant fagtema som inkluderer både forskning og utvikling.

En rekke personer har bidratt til å muliggjøre gjennomføringen av masteroppgaven. Undertegnede ønsker spesielt å takke professor Kjetil Svarstad for konstruktive tilbakemeldinger og inspirerende innspill.

Jeg vil også takke medstudentene på grupperommet, familie og venner for all hjelp og oppmuntring.

Fredrik Gravdal
Trondheim, Juni 2007

Sammendrag

Den tradisjonelle designflyten i utviklingen av mikroelektronikk forutsetter at alle utviklingsaktivitetene er unnagjort pre-kjøretid, og at ferdiggenererte, udelelige konfigurasjonsfiler brukes for å konfigurere brikkene. De fleste systemer som benytter FPGA-teknologi i dag har derfor et begrenset utvalg forhåndsgenererte konfigurasjoner å velge mellom for å løse en oppgave.

Ideen bak denne oppgaven er ønsket om å lage et rekonfigurerbart system der det er FPGA-en selv som står for rekonfigureringen uten noe behov for ekstern tilkobling eller manipulasjon. Dette for å drive den innovative utviklingen av dynamiske hardwaresystemer.

Systemet er laget på en Suzakuplattform med en Spartan-3 XC3S1000 FPGA fra Xilinx. Det er utviklet to program, CLBRead og CLBWrite som kjøres på en microblazeprosessor. CLBRead kan lese en CLB-struktur med forskjellig størrelse, der en enkelt CLB er den minste oppdelingen, til fil. En CLB-struktur kan leses ut fra flash på FPGA-kortet, eller fra en bitstrømsfil på en PC. CLBWrite skriver en filstruktur generert av CLBRead til flashområdet der FPGA-konfigurasjonene ligger. Ved oppstart av FPGA-en vil det nye oppsettet konfigureres opp.

Systemet som er utviklet gjør at FPGA-en kan rekonfigureres helt uten behov for ekstern tilkobling eller manipulasjon. Det er FPGA-en selv som gjør hele jobben. Forskjellige moduler kan lagres og lastes inn ved behov.

Systemer er testet med to moduler, en OG-port og en ELLER-port, der disse kan byttes med hverandre og endringene kan måles med et digitalt multimeter.

Innhold

1 Introduksjon **5**
1.1 Bakgrunn for oppgaven 5
1.2 Historie 6
1.3 Målet med oppgaven 7
1.4 Arbeidsmetode 7
1.5 Rapportens struktur 8

2 Bakgrunn **9**
2.1 FPGA-arkitektur 9
2.2 Reconfigurable Computing 11
2.2.1 Partiell rekonfigurering [13] 12
2.3 Utviklingsplattform 15
2.4 Relatert arbeid 15
2.4.1 BITAnalyse: Oppbyggingen av bitstrømmen 17
2.4.2 PARBIT 20
2.5 Oppsummering 24

3 Metodikk **25**
3.1 Bakgrunn for utviklingen av CLBRead og CLBWrite 26
3.2 Konfigurasjon av FPGA 26
3.2.1 Oppbyggingen av bitstrømmen 27
3.2.2 CLB-oversikt 27
3.2.3 Adressering i bitstrømmen 30
3.2.4 Flash 31
3.3 CLBRead output file 32
3.4 Beskrivelse av funksjonalitet 32
3.4.1 CLBRead 33
3.4.2 CLBWrite 33

4 Test **37**
4.1 Teststrategier 37
4.2 Test av utviklede programmer 38
4.2.1 FlashRead og FlashWrite 38

4.2.2 CLBRead 39
4.2.3 CLBWrite 39
4.2.4 BITDiff 40
4.3 Testcase 1 40
4.4 Testcase 2 44

5 Diskusjon 49
5.1 Modulbasert rekonfigurasjon 49
5.2 Optimalisering 51
5.3 Network on chip 51
5.4 Fremtidig arbeid 53

6 Konklusjon 55

A Kildekode 59
A.1 CLBRead.h 59
A.2 CLBRead.c 59
A.3 CLBWrite.h 64
A.4 CLBWrite.c 64
A.5 FlashRead.h 69
A.6 FlashRead.c 69
A.7 FlashWrite.h 70
A.8 FlashWrite.c 70
A.9 BITDiff.c 72

Figurer

2.1 Generell FPGA-arkitektur 10
2.2 Tidslinje 12
2.3 Bussmakro for modulbasert rekonfigurering 13
2.4 Rekonfigurerbare moduler 14
2.5 Bussmakro 14
2.6 Bussmakro fysisk fremstilling 15
2.7 Suzaku-S 16
2.8 BITAnalyse GUI 20
2.9 PARBIT slice mode 22
2.10 PARBIT block mode 23

3.1 Oversikt over slice i en CLB 28
3.2 Øvre venstre slice 29
3.3 En CLB vist i BITAnalyse 30
3.4 Oppbygging av filstruktur 33
3.5 CLBRead konsollvindu 34
3.6 CLBRead outputfile 34
3.7 CLBWrite konsollvindu 35

4.1 Testcase 1: FlashRead konsollvindu 41
4.2 Testcase 1: CLBRead konsollvindu 42
4.3 Testcase 1: CLBWrite konsollvindu 43
4.4 Testcase 1: BITDiff konsollvindu 43
4.5 Testcase 2: OR-port.bit 45
4.6 Testcase 2: microblaze.bit 45

5.1 2D Mesh - Network on Chip (NoC) 52

Tabeller

2.1 Spesifikasjoner for Suzaku-S 16
2.2 Suzaku-S memory map 17
2.3 Suzaku-S flash memory map 18
2.4 Oppbyggingen av en bitstrøm 19
2.5 Oppbyggingen av en bitstrøm 21
2.6 Datarammenes rekkefølge i bitstrømmen 21

Kapittel 1

Introduksjon

Elektronikk blir i dagens samfunn mer og mer integrert i omgivelsene. Kravene blir strengere og avhengigheten større. Det er derfor nødvendig å gå nye veier og se etter nye muligheter for å bygge opp under denne utviklingen.

1.1 Bakgrunn for oppgaven

Det er en rivende utvikling i dagens teknologisamfunn og systemer blir mer og mer avanserte. For fortsatt videreutvikling er det er ønskelig med mer dynamiske hardwaresystemer. Dagens reprogrammerbare kretser har bidratt til at dette i større og større grad er mulig. Problemet er at disse vanskelig lar seg reprogrammere under kjøringen av selve applikasjonen. I de fleste av dagens dynamiske systemer er hardware en statisk ressurs, der brukeren kun kan benytte seg av dynamisk software. Software er ikke alltid gunstig med tanke på portable enheter som drives av batteri. Det er derfor ønskelig at dynamisk reprogrammerbare kretser skal kunne brukes som et botemiddel for dette.

For å lage et dynamisk system må det være mulig å syntetisere hver modul for seg, for så å legge ut en og en modul på FPGA. Brukere av et dynamisk system har ulike behov, ulik kildetilgang og ulike bakgrunnskunnskaper om systemet. For at forskjellige brukere skal kunne benytte seg av tilgjengelig programmerbar maskinvare i et dynamisk miljø er det nødvendig å bryte med tradisjonell designflyt.

Dette arbeidet er en videreutvikling av tidligere avlagte prosjektet på området. Tidligere har FPGA-ens konfigurasjonsfiler og oppbygging blitt analysert. Det er også gjort en vurdering av hvorvidt rekonfigurering kan gjøres.

Det vil derfor refereres til endel av dette arbeidet i oppgaven. Denne oppgaven tar i hovedsak for seg selve implementasjonen av problemstillingen.

1.2 Historie

Denne oppgaven er en videreføring av tidligere prosjekter ved NTNU og SINTEF.

2002 - 2004:
Det startet med et EU-prosjekt kalt ambiesense. Prosjektet er underliggende i fagområdet "Ambient Intelligence". Dette prosjektet fokuserte på å kunne tilby lokal informasjon til mobile enheter. Brukere skulle kunne hente informasjon i områdene de beveger seg inn i [7].

2004:
Ambiesense er et firma som har opprinnelse fra dette EU-prosjektet. De tilbyr kommersielle løsninger der TAG-er i omgivelsene serverer mobile klienter diverse former for TV- og radiostrømmer i tillegg til internettforbindelse [1].

2005:
Parallellt med denne utviklingen har det vært ønskelig å fortsette med utviklingen av nye metoder å tilby mobile klienter regnekraft på. Professor Kjetil Svarstad startet derfor opp prosjektgruppen AHEAD (Ambient Hardware, Embedded Architectures on Demand). AHEAD rendyrket ideen med at en enkelt brikke skulle kunne tilby regnekraft til mobile enheter.

I AHEAD har det blitt undersøkt hvordan en slik beskrivelse bør være [4]. Dette prosjektet resulterte i innkjøp av et Suzaku SZ030-U00 fra Atmark-Techno [8]. Dette kortet har en Spartan 3 FPGA fra Xilinx [15].

2006:
Ingar Hauge fortsatte i sin masteroppgave [5] en analyse av hvordan bitstrømmen til denne FPGA-en var bygd opp, da det er ønskelig å kunne manipulere FPGA-en uten å benytte komplisert DAK-verktøy. Masteroppgaven resulterte i programmet BITAnalyse. BITAnalyse er et program laget for Microsoft Windows og lager en grafisk representasjon av bitstrømsfila. Ved hjelp av et bildebehandlingsprogram kan endringer gjøres. Etter at endringene er gjort kan det genereres en ny bitstrømsfil som kan plasseres tilbake på FPGA.

2007:
Dette prosjektet er en videreføring av analysen Ingar Hauge gjorde i sin masteroppgave. Det vil implementeres et program der FPGA-en rekonfigurerer seg selv på CLB-nivå.

1.3 Målet med oppgaven

Det er ønskelig å utarbeide en ny metode å gjøre oppdateringer på en Spartan 3 FPGA fra Xilinx. I dagens systemer må et utlegg lages av et eksternt DAK-program. Dette betyr at en bruker av systemet trenger lisens på DAK-verktøy, kunnskap om FPGA-teknologi og ha tilgjengelig all informasjon som skal inn i brikken i tillegg til endringene for å gjøre en syntese. Målet er derfor å lage et program til FPGA-en som kan ta inn en ferdig syntetisert modul og plassere den i et definert miljø i seg selv. Dette gjør at brukeren bare trenger å laste inn sin modul, for så å kjøre den.

Denne metoden vil skape flere fordeler, blant annet kan en relativt liten FPGA utføre mye forskjellig arbeid da den kan bytte ut enkeltmoduler med moduler fra flash. Det vil heller ikke være nødvendig med så mye flash-minne som tidligere da en modul kun lagres en gang. Tidligere måtte hvert oppsett bli syntetisert med alle sine moduler, men med denne teknikken vil hver modul bli linket inn under kjøring av systemet. Det kan også defineres nye moduler til et allerede kjørende system.

Oppgaven vil i hovedsak være å programmere opp en brikke til å kunne reprogrammere moduler i seg selv under kjøring. Dette gjør at brukeren kan forandre på hardware og få effektiv eksikvering av dedikerte oppgaver med et fornuftig energiforbruk.

Direkte manipulasjon av bitstrømmer foregår for det meste i forskningsbaserte systemer. Det er publisert skisser og lignende til forskjellige verktøy for å kunne gjøre en slik manipulasjon, men tilgangen til programvare og tilhørende kildekode er liten. Dette er hovedgrunnen til at arbeid som dette er nødvendig.

1.4 Arbeidsmetode

Arbeidet kan organiseres i følgende hovedaktiviteter

Analyse
Vurdere tidligere dynamiske systemer og lage en oversikt over hva som allerede finnes av dynamisk rekonfigurasjon på området, samt vurdere behovet og mulighetene for teorien denne oppgaven bygger på. Det er også nødvendig å få oversikt over ressurser, nødvendige filer, utstyr og utviklingsverktøy for utviklingen på suzakuplattformen.

Design
Få en grundig oversikt over bitstrømmen og hvordan adresseringa i denne

fungerer for å vurdere om de adresseringsmetodene som tidligere er benyttet er egnet for dette prosjektet. Planleggingen av virkemåten og designet til programmet er her essensiell.

Implementasjon, Test og Verifikasjon
Denne fasen inkluderer implementasjon av hovedprogrammet og støtteprogrammer for test, analyse og validering.

Diskusjon
Fordeler og ulemper med eksisterende systemer sammenlignes med løsningene som presenteres i denne rapporten. En vurdering om dette er riktig retning å dreie rekonfigurerbare systemer. Dette prosjektets relevans i forhold til målet om dynamisk rekonfigurering. Det blir her også foreslått videreutvikling av systemet.

1.5 Rapportens struktur

Kapittel 2 inneholder en oversikt over hvordan en FPGA fungerer og hvordan slik teknolgi kan nyttes i denne sammenhengen. Resultater og analyser fra tidligere arbeider vil også bli presentert.

Kapittel 3 dokumenterer arbeidet som er nedlagt i denne oppgaven og hvordan de utviklede programmene kan nyttes.

Kapittel 4 dokumenterer testingen av de utviklede programmene og beviser for at det utviklede systemet fungerer.

Kapittel 5 inneholder diskusjonen av resultatet vurdert opp mot løsninger og muligheter. Det fremkommer også anbefalinger om videre arbeider rundt problemområdet.

Kapittel 2

Bakgrunn

Dette kapitlet vil gi leseren en rask introduksjon til fagområdet rundt FPGA-teknologi og Reconfigurable Computing samt bruk av FPGA-teknologi i dynamiske systemer. Det vil drøftes muligheter og begrensninger til teknologien.

2.1 FPGA-arkitektur

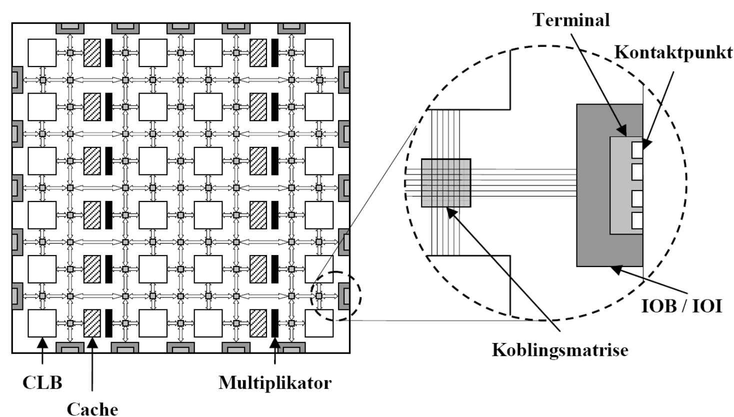
En FPGA (field programmable gate array) er en elektronisk brikke som inneholder programmerbare logiske komponenter og programmerbare interntilkoblinger. Den kan programmeres til enkle logiske porter som OG- og ELLER-porter eller mer komplekse kombinatoriske funksjoner som dekodere eller enkle matematiske funksjoner. I de fleste FPGA-er er det inkludert minneelementer som kan fungere som vipper eller mer komplekse blokkminnemoduler. Fordelene med bruk av FPGA-teknologi er mange. FPGA-er trenger ikke programmeres før de er satt inn i systemet. Dette gir kjappere "time to market". Det er også mulig å programmere de flere ganger da de er basert på RAM-teknologi. Dette gjør at feil enkelt kan rettes opp etter produksjon av hardware.

En FPGA fungerer slik at den laster inn en konfigurasjonsfil ved oppstart. Denne konfigurasjonsfila ligger i flash og lastes inn hver gang FPGA-en tilføres strøm. Konfigurasjonsfila inneholder en bitstrøm med informasjon om brikkens logiske egenskaper. FPGA-ens SRAM-baserte minne beholder denne konfigurasjonen helt til strømmen skrus av. Formatet til denne fila varierer avhengig av brikkens størrelse og arkitektur. Folk som arbeider med programmerbar logikk vil ikke direkte forholde seg til denne, men til DAK-verktøyet som genererer bitstrømsfila.

Ulempen med tradisjonell utvikling av programmerbar logikk er avhengigheten brukeren har av proporitære designverktøy. Utvikleren er også fullstendig avhengig av å ha tilgang til all informasjonen som skal inn i brikken. Dette gjelder også om man bare skal gjøre en endring på en modul. Ved en eventuell endring må det foretas ny, fullstendig syntese og implementasjon. Alt dette fører til at man må ha tilgang på designverktøy, kildekode og kunnskap om digitaldesign for å gjøre endringer i en bitstrøm.

Dagens alternativ til dynamisk hardware er å lage flere ferdig konfigurerte bitstrømsfiler. Ved å veksle mellom disse kan brukeren styre hvilket oppsett FPGA-en skal laste inn. Dette fører til unødvendig bruk av flash for å lagre de forskjellige oppsettene. Samtidig begrenser denne metoden seg da den ikke er veldig dynamisk med tanke på applikasjoner, og brukeren vil uansett ha et begrenset antall valgmuligheter.

Figur 2.1 viser hvordan en generell FPGA-arkitektur ser ut.



Figur 2.1: Generell FPGA-arkitektur. [5]

Configurable Logic Blocks (CLB):
CLB-ene er hoveddelen av en FPGA. Hver CLB inneholder funksjonsgeneratorer eller Look-Up Tables (LUT), synkrone vipper, og multipleksere for ruting av interne signaler. Arbeidet i en FPGA blir gjort på CLB-nivået og CLB-ene blir koblet sammen for å tilby ønsket logikk. [5]

Input/Output Blocks (IOB):
Kommunikasjon med logikken i en FPGA foregår via kontaktpunkter på undersiden eller rundt brikken avhengig av pakketype. Hvert kontaktpunkt (pad) er tilknyttet en IOB som igjen er tilknyttet en terminal. Antall kontaktpunkter per IOB og terminal varierer mellom ulike FPGA-arktitekturer. En IOB administrerer all kommunikasjon inn og ut av brikken ved hjelp av

multipleksere og synkrone vipper (flip-flops). Terminalene bestemmer de elektriske egenskapene (pullup, pulldown, weak keeper, high impedance etc.) til kontaktpunktene. Begrepet Input/Output Interconnect (IOI) brukes som en felles betegnelse på en IOB og all tilhørende, programmerbar interconnect. [5]

Multiplikatorer og cache:
De fleste FPGA-arkitekturer har et antall multiplikatorer og minneblokker (cache) tilgjengelig på brikken. For logiske systemer som utfører en eller annen form for multiplikasjon, vil bruk av de ferdige modulene redusere bruken av CLB-ressursene. Tilgangen på rask cache på FPGA-en reduserer behovet for bruk av eksternt minne for mellomlagring av midlertidige resultater. Xilinx kaller dette brikkeminnet for Block RAM (BRAM) i sine arkitekturer. [5]

Interconnect:
Mesteparten av arealet på en FPGA tas opp av brikkens programmerbare ledningsnettverk. Dette nettverket brukes til ruting av signaler internt, og ut til kontaktpunktene på brikken, via IOB-er. Utvalget av de logiske motorer på en FPGA, varierer fra arkitektur til arkitektur. En FPGA som kun tilbyr sammenføying av enkle logiske porter betegnes som “finkornet” (finegrained) mens arkitekturer som tilbyr mer komplekse byggeklosser (for eksempel multiplikatorer, LUT og cache) betegnes som “grovkornet” (coursegrained). Finkornede FPGA-er var vanlig på begynnelsen av nittitallet, og banet vei for dagens mer grovkornede løsninger. For en mer detaljert beskrivelse av FPGA-arkitektur vises det til brikkeleverandørens datablader. [5]

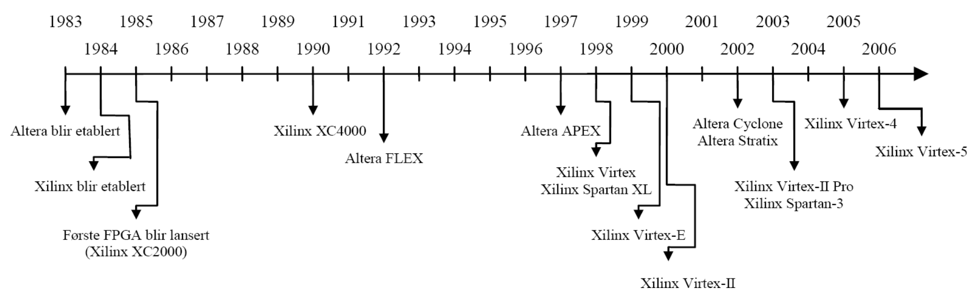
2.2 Reconfigurable Computing

Uttrykket Reconfigurable Computing omfatter reprogrammerbare logiske kretser samt teknologi og metodikk rundt dette. Det meste av forskning og utvikling på området baseres på FPGA-teknologi. De to store firmaene Xilinx og Altera er de største produsentene av FPGA-er. Disse ble etablert tidlig på åttitallet og har siden da lansert flere familier. Figur 2.2 viser en oversikt over de viktigste lanseringene.

Reconfigurable Computing er ofte FPGA-er eller lignende tenknologier sammen med en en GPP som brukes til å kontrollere den rekonfigurerbare logikken, og eksikverer kode som ikke trenger effektiv akselerasjon fra dedikert maskinvare. Dette koblet opp med RTR (runtime reconfiguration) utvider mulighetene til ikke bare å rekonfigurere maskinvare mellom to forskjellige applikasjoner, men også inne i en applikasjon.

Tradisjonelle FPGA-strukturer har primært vært seriell-programmerbare single-context enheter, der bare en konfigurasjon kan lastes av gangen. Denne type FPGA-er blir programmert ved at en seriell datastrøm av konfigurasjonsbit lastes inn i brikken, som trenger full oppdatering av hele brikken hver gang. Denne måten å rekonfigurere dynamiske systemer på gir veldig begrensede muligheter til å drive RTR på.

Som en kontrast til dette kan multi-context FPGA-strukturer inkludere flere minnebit for hver programmeringsbitlokasjon. Disse minnebitene kan bli tenkt på som flere plan av konfigurasjonsinformasjon. Bare ett plan med konfigurasjonsinformasjon kan være aktivert til en gitt tid, men brikken kan raskt bytte mellom planene eller kontekstene av allerede programmert konfigurasjon. Med tanke på dette kan multi-context enheter fungere som et multiplekset sett av singel-context enheter. Denne metoden krever dessverre mye større areal enn den andre strukturen, gitt at det må bli så mange lagringsenheter som det er kontekster, eller plan [6].



Figur 2.2: Tidslinje for Altera sin produktportefølje. [5]

2.2.1 Partiell rekonfigurering [13]

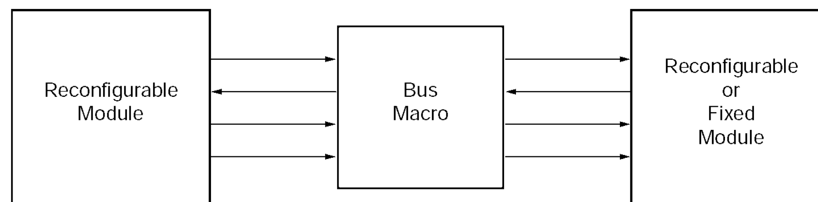
I partiell rekonfigurerbare FPGA-er vil det underliggende programmeringslaget fungere på samme måte som RAM, ved å bruke adresser til å spesifisere plassen til konfigurasjonsdataene. Samtidig kan andre deler av FPGA-en fortsette eksikvering uten å bli avbrutt.

Det finnes to typer partiell rekonfigurering. Den ene er modulbasert, mens den andre er differansebasert.

Modulbasert rekonfigurering

Modulbasert partiell rekonfigurering blir brukt når det er nødvendig med kommunikasjon mellom rekonfigurerbare moduler. For moduler som kommuniserer med hverandre brukes en spesiell bussmakro. Denne makroen

gjør det mulig å kommunisere ut over grensene til den partielle modulen. Dette er nødvendig for at moduler skal kunne kommunisere med hverandre. Hver gang partiell rekonfigurasjon blir gjort vil denne bussen være fast og håndtere ruting mellom modulene. Det er ikke nødvendig med en slik makro om det ikke er noe kommmunikasjon mellom moduler der modulene er helt uavhengig av hverandre. Figur 2.3 viser hvordan denne bussmakroen vil fungere.

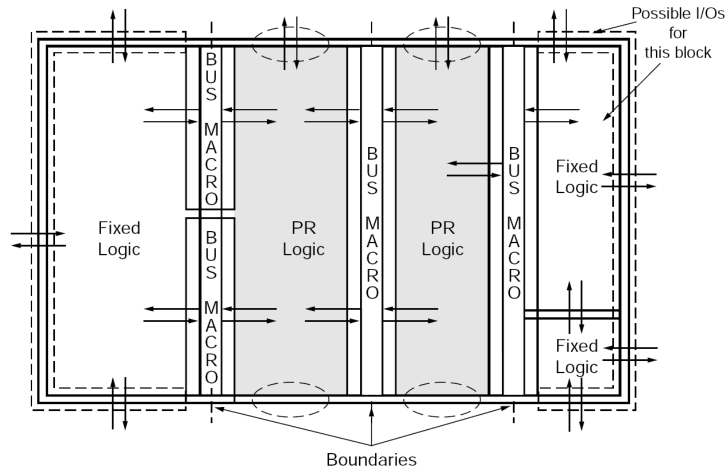


Figur 2.3: Bussmakro for modulbasert rekonfigurering [13]

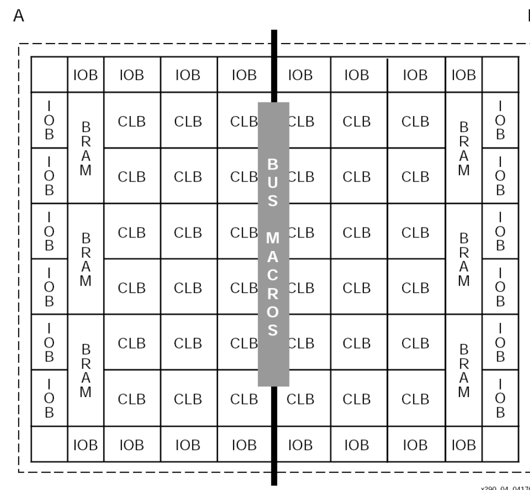
En rekonfigurerbar modul må være i full høyde av FPGA-matrisen. Dette fordi en FPGA bare kan rekonfigureres kolonnevis. En modul kan være på minimum på 4 rammedata til full størrelse av brikken. Modulstørrelsen må inkremeteres med 4 (dvs. 0, 4, 8 ...) opp til full brikkestørrelse. Klokkelogikk (BUFGMUX og CLKIOB) er ikke en del av den rekonfigurerbare modulen da klokkelogikk har egne rammedata. IOB over og under den aktuelle konfigurerbare modulen vil være en del av modulen. Det anbefales at bruk av dynamiske moduler holdes til et minimum da det gjør systemet meget komplekst. Det er ikke mulig å forandre en moduls yttergrense, altså posisjonene til utgang- og inngangslogikk på kretsen. Den er alltid statisk. Figur 2.4 viser hvordan to moduler kan være. Det kommer tydelig frem hvordan det statiske nettverket er nødvendig rundt de to rekonfigurerbare blokkene.

Når det gjelder klokkenettverk er det viktig at alt hva angår klokkelogikk defineres globalt. Klokkelogikk kan ikke være definert lokalt da klokkenettverket trengs av brikken under rekonfigurasjon.

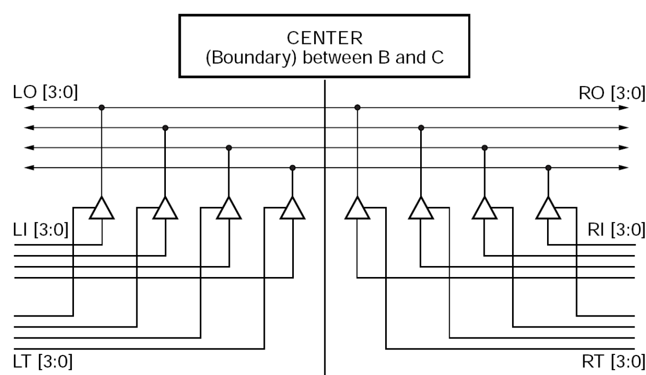
Route locking er nødvendig for at rutingen til bussmakroen ikke skal forandre seg. Forandrer denne seg vil ikke de dynamiske modulene passe inn i designet. Denne implementasjonen av bussmakroen benytter seg av åtte trestegsbuffere (TBUF) koblet slik at det tillater ett bit å gå fra venstre til høyre eller fra høyre til venstre ved å bruke en TBUF longline per bit som vist i figur 2.6 Hver enhet kan håndtere fire bit. Bussens posisjon vil alltid være midt på delingslinja mellom design A og design B ved å bruke fire kolonner med TBUF på A siden og fire kolonner med TBUF på B siden.



Figur 2.4: To rekonfigurerbare moduler med respektiv fast logikk[13]



Figur 2.5: Bussmakro[13]



Figur 2.6: Bussmakro fysisk fremstilling[13]

Differansebasert rekonfigurering

Denne form for partiell rekonfigurering blir brukt om det gjøres en liten forandring i designet (for eksempel i FPGA Editor), for så å lage en bitstrøm som kun består av forskjellen mellom de to utleggene. Dette kan være en mye kjappere måte å forandre konfigurasjonen på da forskjellene på de to utleggene kan være små.

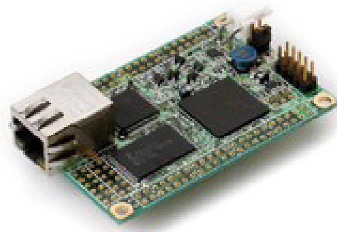
2.3 Utviklingsplattform

Dette prosjektet er en videreføring av tidligere prosjekt. Det ble da kjøpt inn et Suzaku FPGA-kort til utprøving og forskning på området. Dette FPGA-kortet tar relativt liten plass og har minimalt med komponenter. Kortet består hovedsakelig av en Spartan-3 FPGA [15], DRAM, FLASH og en ethernet kontroller. Det har også tilkoblinger til RS232 og vanlig IO. Figur 2.7 viser dette kortet og tabell 2.1 viser suzakukortets spesifikasjoner.

Minnekartet (memory map) til suzakukortet er vist i tabell 2.2 og tabell 2.3. Tabell 2.2 viser hvor Flashen ligger i prosessorens adresseområde, mens tabell 2.3 viser områdene internt på flashen.

2.4 Relatert arbeid

Her kommer en oppsummering av tidligere arbeider som relaterer seg til denne oppgaven



Figur 2.7: Suzaku-S[8]

Suzaku-S	
Produktnummer	SZ030-U00
FPGA	Xlinx Spartan-3 (XC3S1000 FT256)
CPU	MicroBlaze (Soft Core 32bit RISC)
Oscillator	3,6864 MHz
DRAM	16 MB
Flash minne	8 MB
Ethernet	10 BASE-T / 100 BASE-TX
Tilkobl.punkter for I/O	86
Operativsystem	uCLinux (kernel 2.4)
Kortets størrelse	72 x 47 (mm)

Tabell 2.1: Tekniske spesifikasjoner for Suzaku-S[8]

Start Address	End Address	Peripheral	Device
0x0000 0000	0x0000 1FFF	BRAM	
0x0000 1000	0x7FFF FFFF	Reserved	
0x8000 0000	0x80FF FFFF	OPB-SDRAM Controller	SDRAM 16MByte
0x8100 0000	0xFEFF FFFF	Free	
0xFF00 0000	0xFF7F FFFF	OPB-EMC	FLASH Memory 4MByte
0xFF80 0000	0xFFCF FFFF	Free	
0xFFE0 0000	0xFFEF FFFF	OPB-EMC	LAN Controller
0xFFF0 0000	0xFFFF 0FFF	Free	
0xFFFF 1000	0xFFFF 10FF	OPB-Timer	
0xFFFF 1100	0xFFFF 1FFF	Free	
0xFFFF 2000	0xFFFF 20FF	OPB-UART Lite	RS232C
0xFFFF 2100	0xFFFF 2FFF	Free	
0xFFFF 3000	0xFFFF 30FF	OPB-Interrupt Controller	
0xFFFF 3100	0xFFFF 9FFF	Free	
0xFFFF A000	0xFFFF A0FF	OPB-GPIO	Boot Mode Jumper LED Software Reset
0xFFFF A100	0xFFFF FFFF	Free	

Tabell 2.2: Suzaku-S memory map[11]

2.4.1 BITAnalyse: Oppbyggingen av bitstrømmen

Oppbyggingen til en bitstrøm er så kompleks at det er umulig å få oversikt over den for å gjøre endringer manuelt. Det ble derfor laget et grafisk verktøy kalt BITAnalyse [5] for å gjøre endringer på en bitstrøm uten at det er nødvendig med DAK-verktøy fra brikkeprodusenten. BITAnalyse er et verktøy laget for Microsoft Windows og figur 2.8 viser hvordan BITAnalyse ser ut.

Dette verktøyet analyserer bitfila og fremstiller den grafisk i form av et bilde. Det er mulig å lagre disse grafiske bildene som bildefiler (bmp). Bildefilene kan så endres med et bildebehandlingsprogram, for så å lastes tilbake i BITAnalyse der de kan konverteres tilbake til en bitstrømsfil. Dette gjør at moduler enkelt kan "klippes ut" av et design for så å settes inn i et annet. Dette programmet gjør det mulig å lagre flere forskjellige moduler og ved hjelp av BITAnalyse kan disse settes sammen.

I arbeidet med å lage BITAnalyse ble det utarbeidet en analyse på hvordan bitstrømmen til Spartan-3 FPGA-en er oppbygd. Tabell 2.4 viser hvordan bitstrømmen er bygd opp. Tabellen inneholder kun statiske konfigurasjonsbit, databitene vises i tabell 2.5

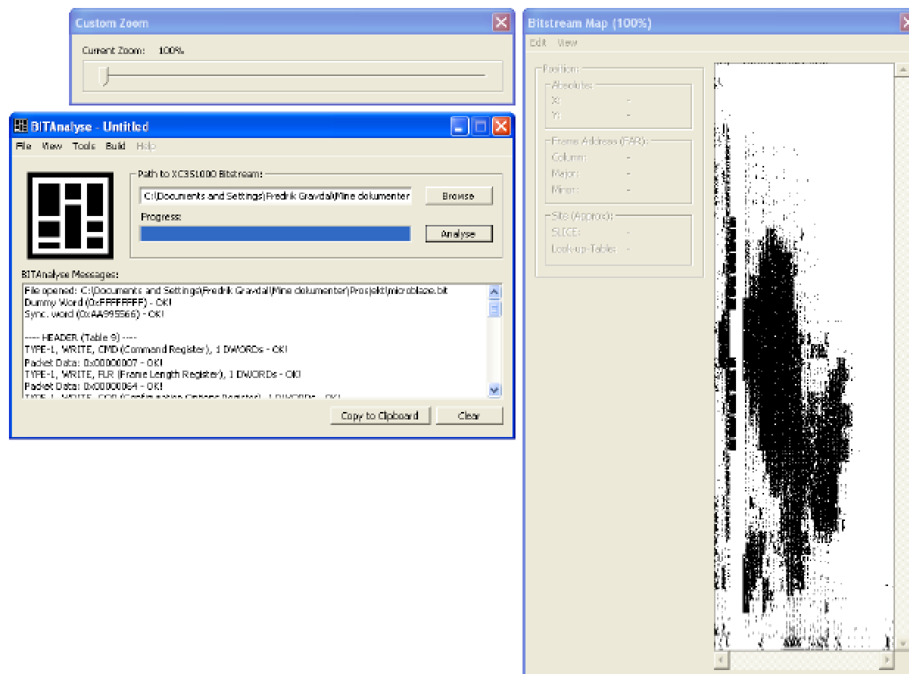
Tabell 2.5 viser dataområdet i bitstrømsfila. Tabellen er sortert etter stigen-

Address	Region	Size	Description
0x00000000 0x0000FFFF	free 1	64 KB	
0x00010000 0x0007FFFF	free 2	448 KB	
0x00080000 0x000FFFFF	fpga	512 KB	
0x00100000 0x0011FFFF	bootloader	128KB	Hermit boot loader
0x00120000 0x0041FFFF	kernel	3MB	Linux kernel
0x00420000 0x007EFFFF	user	Approx. 3.81MB	Userland
0x007F0000 0x007FFFFF	config	64KB	Configuration area

Tabell 2.3: Suzaku-S flash memory map[11]

Beskrivelse	Data(XC3S1000)
Dummy Word	0xFFFFFFFF
Synchronization Word	0xAA995566
CMD Write Packet Header	0x30008001
CMD Write Packet Data (Reset CRC)	0x00000007
FLR Write Packet Header	0x30016001
FLR Write Packet Data	0x00000064
COR Write Packet Header	0x30012001
COR Write Packet Data	0x60003FE5
IDCODE Write Packet Header	0x3001C001
IDCODE Write Packet Data (3S400)	0x11428093
MASK Write Packet Header	0x3000C001
MASK Write Packet Data	0x00000000
CMD Write Packet Header	0x30008001
CMD Write Packet Data (Switch CCLK)	0x00000009
FAR Write Packet Header	0x30002001
FAR Write Packet Data	0x00000000
CMD Write Packet Header	0x30008001
CMD Write Packet Data (WCFG)	0x00000001
FDRI Write Packet Header (Type 1)	0x30004000
FDRI Write Packet Header (Type 2)	0x500188f4
FDRI Write Packet Data (0 - 100596)	—DATA—
CMD Write Packet Header	0x30008001
CMD Write Packet Data (GRESTORE)	0x0000000A
CMD Write Packet Header	0x30008001
CMD Write Packet Data (DGHIGH/LFRM)	0x00000003
No Op (one frame's worth)	0x20000000
CMD Write Packet Header	0x30008001
CMD Write Packet Data (START)	0x00000005
CTL Write Packet Header	0x3000A001
CTL Write Packet Data	0x00000000
CRC Write Packet Header	0x30000001
CRC Write Packet Data	0x0000DEFC
CMD Write Packet Header	0x30008001
CMD Write Packet Data (DESYNC)	0x0000000D
No Op (4 words)	0x20000000

Tabell 2.4: Oppbyggingen av en bitstrøm til Spartan-3 XC3S1000[12] og [5]



Figur 2.8: BITAnalyse GUI

de rekkefølge (rammenummer) i bitstrømmen. Rammenummeret forteller hvilken plass dataene har i bitstrømmen, og kolonnenummeret forteller hvilken plass dataene har på FPGA-en. Som et eksempel vil GCLK-L være de første 3232 bitene i bitstrømmen, mens de vil bli lastet inn i kolonne 306 på FPGA-en (den vil altså bli vist i kolonne 306 i BITAnalyse). Block, Major og Minor er adresseringssystemet i bitstrømmen. Dette adresseringssystemet blir brukt av BITAnalyse, men dette systemet gjør det hele ganske komplisert.

2.4.2 PARBIT

FPGA kan bli brukt til å implementere systemer som kan rekonfigureres under kjøretid. Et verktøy kalt PARBIT [8] har blitt utviklet for å transformere FPGA-datastrømmer til partielle bitstrømmer. Med dette verktøyet er det mulig å definere partiell rekonfigurerbare områder inne på FPGA-en og laste ned disse til FPGA-enheten.

Column	Block	Major	Minor	Rammenr.	Kolonnenr.	Ant. CLB
GCLK-L	0	0	0	0	306	
GCLK-R	0	0	1	1	688	
CENTER	0	0	2	2	497	
TERM-L	0	1	0-1	3-4	0-1	
IOI-L	0	2	0-18	5-23	2-20	
CLB	0	3-42	0-18	24-783	21-58	2 * 48
					154-305	8 * 48
					307-496	10 * 48
					498-687	10 * 48
					689-840	8 * 48
					936-973	2 * 48
IOI-R	0	43	0-18	784-802	974-992	
TERM-R	0	44	0-1	803-804	993-994	
BRAM-L	1	0	0-75	805-880	78-153	
BRAM-R	1	1	0-75	881-956	860-935	
BRAM-INT-L	2	0	0-18	957-975	59-77	
BRAM-INT-R	2	1	0-18	976-994	841-859	

Tabell 2.5: Oppbyggingen Spartan-3 XC3S1000 [5]

Device	# of Frames	Frame Length in Bits	Configuration Bits	Total # of Bits (Including header)	Approx. SelectMap Download Time (50 MHz) in ms	Approx. Serial Download Time (50 MHz) in ms	Approx. JTAG Download Time (33 MHz) in ms
XC3S50	368	1,184	435,712	439,264	1.10	8.79	13.31
XC3S200	615	1,696	1,043,040	1,047,616	2.62	20.95	31.75
XC3S400	767	2,208	1,693,536	1,699,136	4.25	33.98	51.49
XC3S1000	995	3,232	3,215,840	3,223,488	8.06	64.67	97.68
XC3S1500	1223	4,384	5,205,088	5,214,784	13.04	104.30	158.02
XC3S2000	1451	5,280	7,661,280	7,673,024	19.18	153.46	232.52
XC3S4000	1793	6,304	11,303,072	11,316,864	28.29	226.34	342.94
XC3S5000	1945	6,816	13,257,120	13,271,936	33.18	265.44	402.18

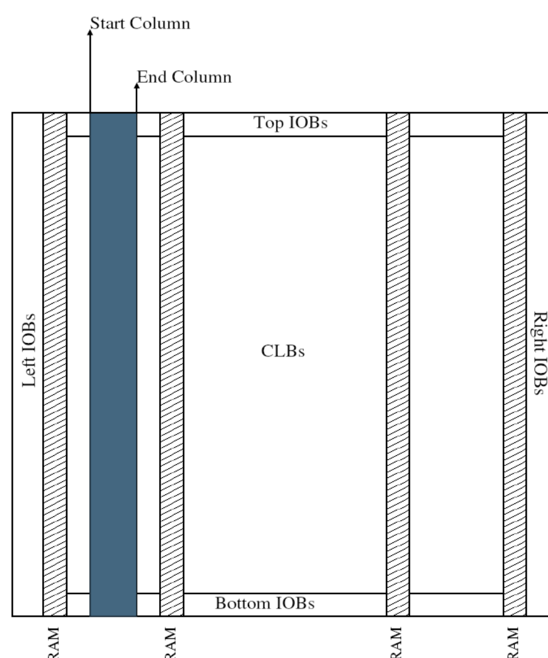
Tabell 2.6: Datarammenes rekkefølge i bitstrømmen [12]

Overføringsmetoder

Det er tre forskjellige måter å laste inn konfigurasjonsfila: Master/Slave Serial, SelectMAP og Boundary Scan. SelectMAP er en 8-bit parallell tilkobling, mens de andre er en bit serielloverføring. For å kunne generere en partiell bitstrømsfil må PARBIT lese konfigurasjonsrammene fra den originale bitstrømmen og kopiere den til en partiell bitstrøm. Den vil da kopiere inn de forskjellige områdene brukeren vil skal være mulig å forandre på senere. Så genereres det nye verdier til konfigurasjonsregisteret og adresseregisteret i forhold til det partiell rekonfigurerbare området.

Operasjonsmoduser

Slice Mode (utklippsmodus) (figur 2.9)
Her defineres et utklipp av en kolonne som inneholder en eller flere CLB-kolonner samt kontrollbitene for topp og bunn IOB-ene. Verktøyet genererer partielle bitstrømmer som er tilsvarende den originale bitstrømmen. Her trengs det bare å definere start- og sluttkolonne som skal bli rekonfigurert.



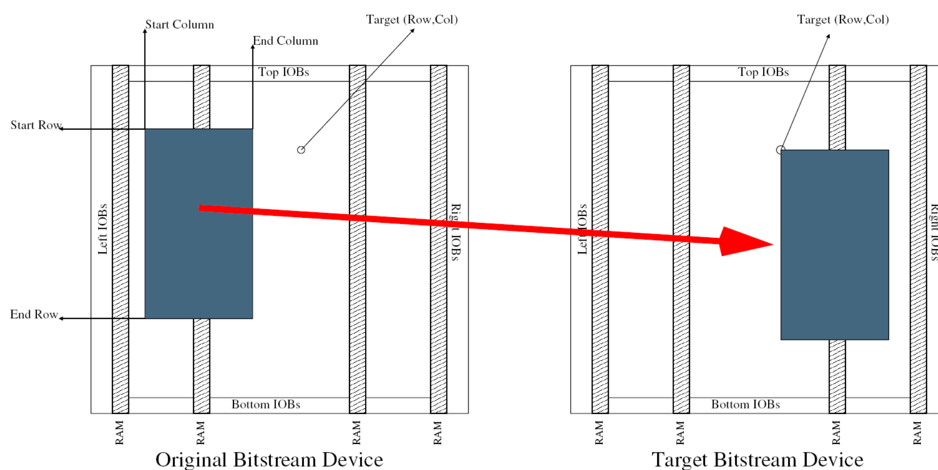
Figur 2.9: PARBIT slice mode [2]

Block Mode
Her blir rader og kolonner definert i en rektangulær blokk lokalisert på et

område inne på FPGA-en. Verktøyet genererer en bitstrømsfil som inneholder logikken som ligger i dette området. Den leser dette ut fra den originale fila og lagrer det i den nye partiell rekonfigurerbare fila. Over og under dette området genererer verktøyet logikk hentet fra den originale bitfila. Her må brukeren definere

- Startkolonne og sluttkolonne. Bredden på det rekonfigurerbare området.
- Startrad og sluttrad. Høyden på det rekonfigurerbare området.
- Den nye plasseringen av rad og kolonne.

Figur 2.10 Viser hvordan dette fungerer.



Figur 2.10: PARBIT block mode [2]

Filer i PARBIT

PARBIT verktøyet bruker tre typer filer. Den *originale* bitfila som blir generert av Xilinx verktøyet inneholder de rekonfigurerbare områdene som blir pakket ut av PARBIT-verktøyet og lagret i en partiell bitstrømsfil. *Target* bitstrømsfila er også en fil som er generert av Xilinx-verktøyet. I block mode inneholder denne fila konfigurasjonen for FPGA-en, pluss et tomt område som er reservert til å motta det rekonfigurerbare område som blir generert av PARBIT. *Partiell* bitstrømsfila er generert av PARBIT og inneholder data fra den originale bitstrømsfila. Den inneholder også data fra target bitstrømsfila, samt konfigurasjon for Block mode.

2.5 Oppsummering

PARBIT er utelukkende den største pådriften til dette prosjektet. Ideen om å lage moduler eller blokker internt på FPGA-en og hvordan dette gjøres kommer fra PARBIT-systemet og er det største argumentet til at denne metoden er gjennomførbar. BITAnalyse er et analyseprogram utviklet for å analysere bitstrømmen i en Spartan-3 FPGA fra Xilinx som blir brukt i dette prosjektet.

Bidraget i denne oppgaven vil bruke ideen fra PARBIT der en modul kan kopieres ut fra et sted på FPGA-en og legges tilbake et annet sted. I denne oppgaven brukes en Spartan-3 FPGA fra Xilinx og med kunnskapen om bitstrømmen fra BITAnalyse kan man utvikle et nytt system der FPGA-en selv gjør rekonfigureringa basert på moduler. Til forskjell fra PARBIT vil det i denne oppgaven være FPGA-en selv som gjør rekonfigureringa. Det vil derfor ikke være nødvendig med eksterne komponenter eller PC-systemer som gjør rekonfigurasjonsjobben.

Kapittel 3

Metodikk

Dagens systemer er preget av at eksternt utstyr må tilkobles for å laste opp konfigurasjonsfilene til et definert system. Ved rekonfigurasjon må ofte hele systemet kompileres eller syntetiseres på nytt på en ekstern datamaskin, som tar lang tid ved større systemer. Denne statiske systemtopologien hindrer den dynamiske utviklingen i dagens samfunn. Denne oppgaven vil prøve å bote på dette problemet.

Selvrekonfigurering er i denne sammenheng ment som en maskinvarebrikke som selv skal gjøre sin egen rekonfigurasjon. Ved selvrekonfigurasjon menes det at brikken vil får tilsendt en modul over eksempelvis internett. Brikken vil selv legge inn den mottatte modulen i sitt konfigurasjonssystem. Ved bruk av enkle kommandoer er det meningen at brikken selv kan bytte mellom lagrede konfigurasjoner. Det er ikke meningen at hele området på brikken skal kunne rekonfigureres, bare det området hvor de logiske operasjonene blir utført (CLB-nivået). Maskinvaremoduler vil skiftes inn og ut ettersom de trengs i systemet. Kommunikasjonen inn og ut av de dynamiske modulene vil foregå på statisk definerte busser. Dette vil gjøre maskinvaresystemer meget dynamiske i forhold til dagens statiske systemer.

CLBRead og CLBWrite er to uavhengige programmer som er utviklet i forbindelse med selvrekonfigurering av FPGA. Dette er to softwareprogrammer som kjøres på Microblaze for å gjøre rekonfigureringa på FPGA-en. CLBRead er et program som kan lese FPGA-ens CLB-struktur til fil, mens CLBWrite er et program som kan lese en fil generert av CLBRead og skrive denne CLB-strukturen tilbake til FPGA-en. Det vil i dette kapittelet bli gått gjennom behovet for disse programmene og hvordan de påvirker systemet. I slutten av kapittelet vil programmenes virkemåte presenteres.

3.1 Bakgrunn for utviklingen av CLBRead og CLBWrite

Det er ønskelig å lage et system der FPGA-en selv gjør rekonfigureringen. På denne måten vil det være unødvendig å på noen som helst måte koble seg til kretsen ved hjelp av en JTAG eller en RS232-tilkobling for å laste inn en modul. Dette gjør brikken selvstendig i rekonfigurasjonsarbeidet og for at enheter uten kjennskap til kretsen skal kunne laste opp sin modul er det nødvendig at FPGA-en selv håndterer konfigureringen. Denne metoden gjør at større og større deler av et digitalt system puttes inn i brikken. Dette er helt nødvendig for å drive frem utviklingen der brikkene selv skal ta del i sin egen konfigurasjon og kunne tilby regnekraft til sine omgivelser uavhengig av omgivelsene.

Kjernerelokasjon er et begrep som tidligere har blitt benyttet i omtale av modulbasert rekonfigurering. Kjernerelokasjon er i og for seg det samme som modulbasert rekonfigurasjon, men en modul kan bestå av flere sammenhengende kjerner eller omvendt. Dette kommer an på hvordan strukturene på den faste logikken er plassert på FPGA-en. Det kan også være aktuelt å kun forandre på strukturen til en inaktiv kjerne ved hjelp av modulbasert rekonfigurering.

Konfigurasjonsminnet i en FPGA er SRAM-basert, noe som innebærer at all data i minnet blir slettet hver gang strømmen skrus av. Ved oppstart vil den fullstendige bitstrømmen med konfigurasjonsdataene lastes fra flash og inn i minnet. Dataene i denne bitstrømmen definerer FPGA-ens funksjon og virkemåte. Det er denne bitstrømmen det vil være aktuelt å gjøre endringer på for å gjøre rekonfigurasjon.

CLBRead og CLBWrite er to uavhengige programmer som har blitt utviklet for å kunne lese ut eller skrive inn CLB-strukturer til og fra bitstrømsfila. CLB-nivået er der de logiske operasjonene og ruting på FPGA-en foregår. CLB-nivået er også det laveste nivået man kan legge seg på for å være innenfor definisjonen til en modul.

3.2 Konfigurasjon av FPGA

Oppbyggingen av selve FPGA-en er det vanskelig å gjøre noe med, men konfigurasjonsfilen derimot er det fullt mulig å gjøre endringer på så lenge man vet hva og hvor man skal gjøre disse. Foreløpig vil det være denne metoden modulbasert rekonfigurering kan bli gjort på for at kun enkelte moduler skal kunne bli oppdatert av brikken selv. For mer fullstendig informasjon omkring oppbyggingen av bitstrømsfila vises det til [12] og [5].

3.2.1 Oppbyggingen av bitstrømmen

For å navigere i en bitstrømsfil er det nødvendig å se på hvordan den er bygget opp og hvor de aktuelle dataene befinner seg. Bitstrømsfila har tidligere blitt analysert og er oppsummert i kapittel 2. For å forstå hvordan det hele henger sammen vil vi se nærmere på strukturen til fila. Dette spesielt for å forstå hvordan adressering lettest kan gjøres.

Som det fremkommer av tabell 2.6 ser vi at FPGA-en XC3S1000 har 995 kolonner og 3,232 rader. Dette betyr at dataområdet i bitstrømsfila er på 3,215,840 bit. Tabellen viser også at det totalt skal være 3,223,488 bit i hele fila, header inkludert. Det vil da si at headeren alene er på 7,648 bit.

$$3,223,488 - 3,215,840 = 7,648$$

$$995 * 3,232 = 3,215,840$$

Ved å åpne bitfila i en hexeditor kan man lese ut størrelsen på 0x625F7 = 402935. Dette må multipliseres med 8 da det er åtte bit på hver plass 402935*8 = 3223480. Siden vi her teller med plass 0, kan vi legge til 8 bit og vi får 3223488 som er det samme som databladet. Det interessante i bitstrømsfila er CLB-ene. Det er derfor nødvendig å navigere seg frem til dette området i bitstrømmen. Tabell 2.5 viser hvordan dataområdet i bitstrømmen er satt sammen. For å komme seg frem til CLB-ene må man hoppe over all header data før dataområdet begynner. Dette er 80 byte (vist i tabell 2.4). Så må man hoppe over CLK-L, CLK-R, CENTRE, TERM-L og IOI-L dataene som tilsammen er på 9696 byte for å komme til CLB-området (vist i tabell 2.5). Til slutt er det nødvendig å hoppe over IOI og TERM dataene i den aktuelle kolonna som er på 10 byte.

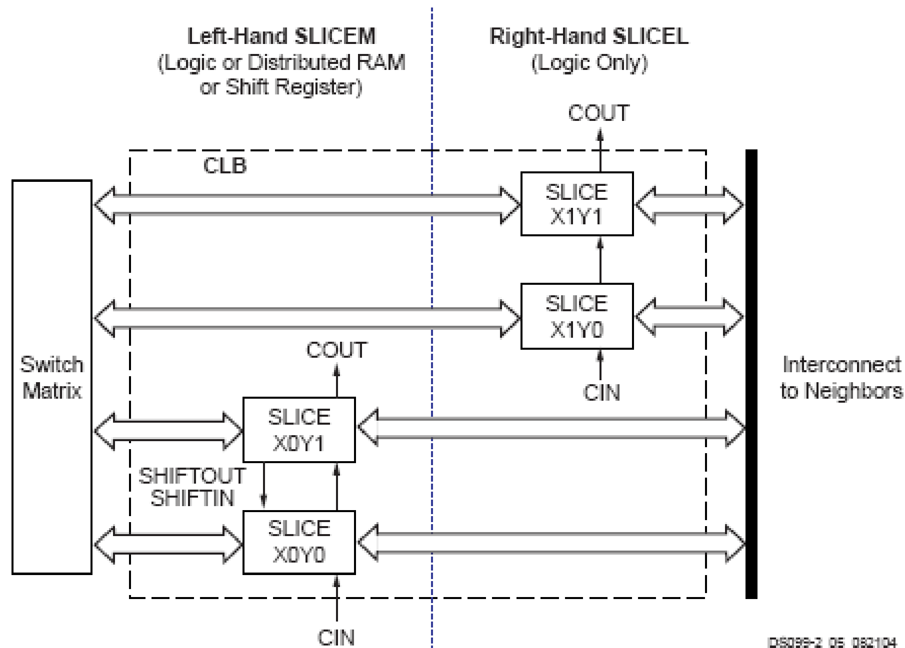
$$80 + 9696 + 10 = 9786 = 0x263A$$

Utregningen over viser hvor langt det er nødvendig å hoppe for å komme til første bit i første CLB. Dette er i programkoden definert som START-CLB.

3.2.2 CLB-oversikt

De konfigurerbare logiske blokkene (CLB-ene) bygger opp hovedstrukturen av den implementerte logikken. Hver CLB inneholder fire sammenkobla slices. Figur 3.1 viser hvordan disse henger sammen. Disse slicene er

gruppert i par og hvert par er gruppert kolonnevis med en uavhengig carry chain [15].

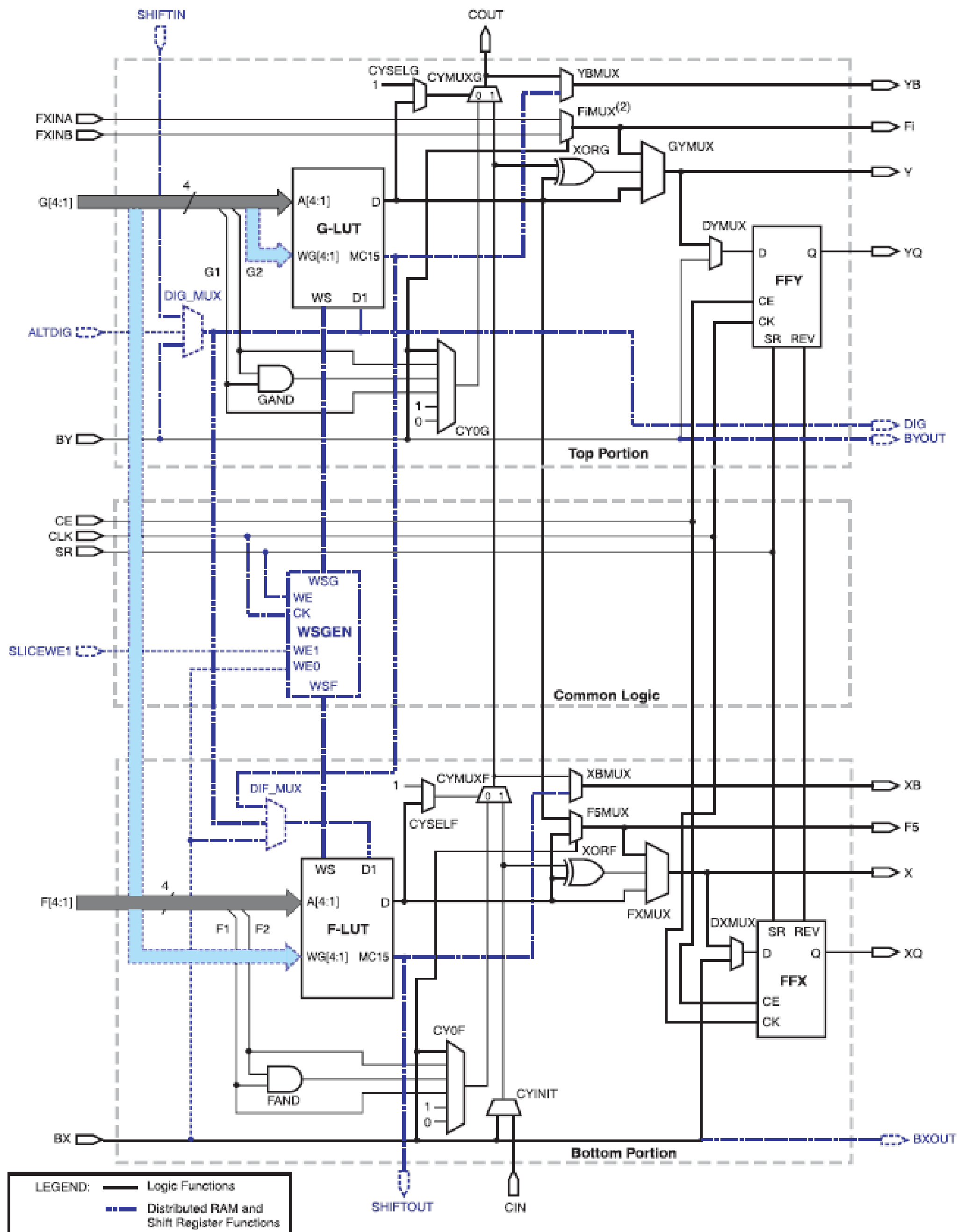


Figur 3.1: Oversikt over slice i en CLB. [15]

Alle fire slicene har følgende elementer til felles: to logiske funksjonsgeneratorer, to lagringselementer, wide function mulitipleksere, carry logikk og aritmetisk logikk. Både venstre og høyre side bruker de samme elementene til å tilby logikk, aritmetikk og ROM funksjonalitet. Figur 3.2 viser et forenklet skjema til den øvre venstre slicen. I tillegg har venstre side to ekstra funksjoner for lagring av data i distribuert RAM og skifting av data med et 16-bit register [15].

Funksjonsgeneratorene, også kjent som look-up tables eller LUT er hovedressursen for oppbygging av logiske funksjoner. LUT-ene i venstre slice kan også konfigureres som distribuert RAM eller et 16-bit skiftregister. Funksjonsgeneratorene plassert i øvre del av slicen blir referert som "G-LUT" og nedre del refereres som "F-LUT" [15].

CLB-nivået kan ses på som byggesteinene til FPGA-ens struktur og oppbygging. Det er dette nivået CLBWrite- og CLBRead-programmene ligger på i abstraksjon. CLB-nivået er det laveste nivået i strukturen på en FPGA der blokkene er like. Lavere abstraksjonsnivå kan ikke formaliseres. Hvordan intern ruting inne i CLB-en er gjort i praksis på bitnivå er ikke dokumentert fra Xilinx. Ved bruk av BITAnalyse kan man også se at innholdet

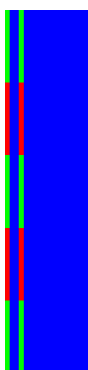


Figur 3.2: Forenklet bilde av øvre venstre slice. [15]

i en CLB ikke er organisert symetrisk (vist i figur 3.3). Dette gjør at oppdeling på et lavere nivå ikke er praktisk mulig uten mer forståelse for den oppbyggingen av CLB-strukturen på bitnivå.

Xilinx har ikke dokumentert sammenhengen mellom den logiske oppkoblingen og bitstrømsfila, derfor er det umulig å vite hvilke bit som setter hva inne i CLB-strukturen. Det er bare mulig å se hvordan en CLB ser ut i BIT-Analyse når alle bitene i CLB-en er satt. Figur 3.3 viser hvordan CLB-en vil se ut. En CLB er 19 bit bred og 64 bit lang. Det er 40 CLB-er i bredden og det er 48 CLB-er i lengden på den aktuelle Spartan-3 FPGA-en fra Xilinx (XC3S1000).

Siden alle CLB-ene på brikken er like kan en CLB flyttes fra et område til et annet inne på FPGA-en forutsatt at den flyttede CLB-en kommer inn i samme miljø, det vil si at den må ha den samme rutingen av signaler inn og ut eksternt. Dette betyr at en eller flere CLB-er i rektangulær form kan flyttes fra et sted til et annet uten at full syntese må gjøres. Det eneste som er viktig å passe på er at modulen som plukkes ut kommer tilbake i samme omgivelser og får like "naboer" slik den tidligere har hatt. Flytende signaler vil forekomme i tomme (ubrukte) modulposisjoner. Disse bør derfor inneholde logikk som definerer utgangssignalene høye eller lave så de ikke er flytende.



Figur 3.3: En CLB vist i BITAnalyse

3.2.3 Adressering i bitstrømmen

BITAnalyse bruker et adresseringssystem der det loopes opp Block, Major og Minor for kontroll på hvilke data som skal hvor i bitstrømmen. Dette er en ganske komplisert metode og ikke en gunstig måte å gjøre det på inne på FPGA-en. FPGA-en har relativ lite regnekraft og det er ønsket at koden skal eksikveres så raskt som mulig. Flashen som er tilgjengelig på

FPGA-kortet kan lagre 16-bit data på hver adresse så den raskeste metoden for adressering er å hoppe direkte til den adressen som skal leses eller skrives. Adresseringssystemet til BITAnalyse ble en stund utprøvd i forbindelse med CLBWrite og CLBRead der et BMP-bilde fra BITAnalyse ble benyttet i stedet for fra egengenererte filer. Dette viste seg meget ugunstig og ble ikke gjennomførbart.

Spartan-3 har 40 CLB-er i bredden og 48 CLB-er i lengden. En CLB består av 19 bit i bredden og 64 bit i lengden. For å lese inn en hel CLB er det derfor nødvendig å hoppe til startpunktet til valgt CLB for å lese inn de første 64 bitene. For å lese de neste 64 bitene i kolonna til høyre er det nødvendig å hoppe ytterligere 3232 bit (lengden på en rad) for å være på startadressa til de neste 64 bitene i CLB-en. Koden under viser hvordan en CLB blir lagret:

```
for (k=0;k<19*2;k+=2){

        iData = (*(volatile unsigned short *) (CLBaddr+k*2)) << 16;
        iData|= (*(volatile unsigned short *) (CLBaddr+2 +k*2));
        oneCLB.iCLB[k] = iData;

        iData = (*(volatile unsigned short *) (CLBaddr+4 +k*2)) << 16;
        iData|= (*(volatile unsigned short *) (CLBaddr+6 +k*2));
        oneCLB.iCLB[k+1] = iData;

        CLBaddr += 100*4;
}
```

3.2.4 Flash

Flashteknologi er litt spesielt med tanke på lagring av data. I motsetning til lesing og skriving av RAM og lignende registre kan en adresse kun skrives til en gang. Flashteknologi kan nemlig bare skrive verdien null, det er ikke mulig å lagre en ener. Dette gjør at ekstra funksjonalitet må til for å lagre forskjellige verdier. Skal det lagres en ny verdi på en adresse er det nødvendig å slette hele sektoren adressen befinner seg i. Flashen som er plassert på Susaku-S kortet har 128 sektorer og alle er 65536 byte store. Den første sektoren er fra 0x0 til 0xFFFF, neste fra 0x10000 til 0x1FFFF osv.

For å lagre data på en adresse i flashen er det nødvendig å kopiere alt innholdet i hele sektoren adressen befinner seg i til et buffer. Endringene som skal gjøres må gjøres i dette bufferet. Før bufferet kan kopieres tilbake til flash må sektoren slettes, så kan dataene skrives. Koden under viser hvordan slette-metoden fungerer:

```
void flash_sector_erase(int sectorAddr)
```

```
{
        printf("Erasing sector: %x\n", sectorAddr);

        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);
        flash_write_cmd(FLASH_START + 0xaaa, 0x80);
        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);

        flash_write_cmd(FLASH_START + sectorAddr, 0x30);

        while ((*(volatile unsigned short *) (FLASH_START+sectorAddr
+FLASH_SECTOR_SIZE - 1)) != 0xFFFF);

}
```

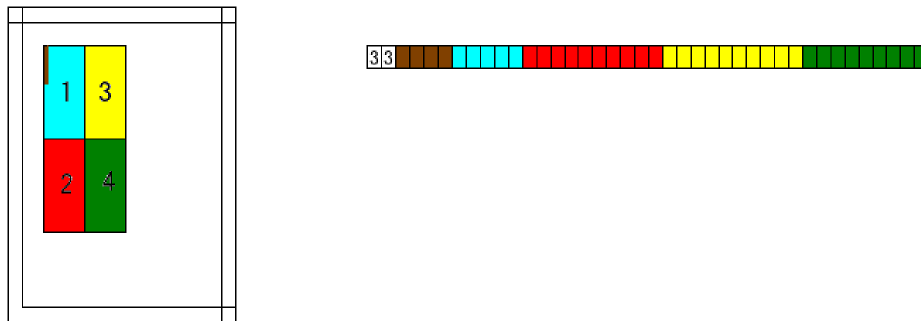
Atmark Techno har ikke dokumentert hvilken type flash de har montert på kortet, det eneste databladene sier er at det er 8 megabyte flash, det har derfor vært vanskelig å finne ut hva de forskjellige kommandoene gjør. Denne koden er kopiert fra softwareprosjekter man kan laste ned fra Atmark Techno sin webside. Koden er testet og fungerer.

3.3 CLBRead output file

For å kopiere ut CLB-strukturer er det nødvendig å lagre disse til fil, slik at de kan tas vare på. Først i denne fila ligger det to tall av typen int (32 bit), der det første tallet er antall kolonner med CLB-er og det andre tallet er antall rader med CLB-er. Etter disse to talla følger alle CLB-dataene til den første CLB-en, så den andre CLB'en. Størrelsen på en CLB er på 152 byte. Figur 3.4 illustrerer hvordan CLB-ene ligger i FPGA-en og hvordan disse blir lagret i denne fila. Den brune fargen representerer noen av de første dataene i den første kolonna i den første CLB-en

3.4 Beskrivelse av funksjonalitet

Programmene er implementert som kommandolinjeprogrammer, slik at de enkelt kan inkluderes i andre programmer og skript. De er skrevet i standard C, slik at de kan kompileres for forskjellige arkitekturer. De påfølgende underkapitlene beskriver funksjonaliteten til de utviklede programmene.



Figur 3.4: Illustrasjon på hvordan filstrukturen er oppbygd

3.4.1 CLBRead

Dette programmet leser ut et gitt antall CLB-er fra flash, eller fra en inputfil om det er ønskelig. Dette gjør at det er mulig å lage moduler av filer som ikke er implementert på FPGA, men som kun er generert for å lage den spesifikke modulen.

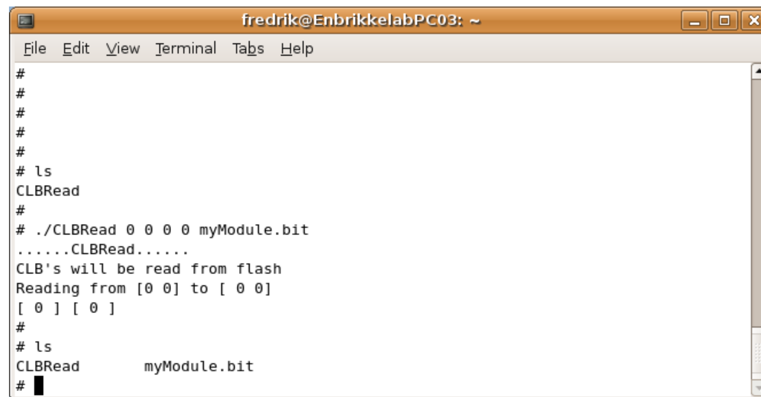
For å bruke dette programmet til å hente ut en modul fra FPGA-en er det nødvendig med 5 inputparametere; CLBstartkolonne, CLBstartrad, CLBstoppkolonne, CLBstopprad og outputfilename. Dette programmet kan også ta inn 6 inputparametere; CLBstartkolonne, CLBstartrad, CLBstoppkolonne, CLBstopprad, outputfilename og inputfilename. Ved 6 inputparametere vil programmet lese bitstrømmen fra en fil istedetfor flash på suzakukortet. Denne funksjonaliteten er implementert slik at det ikke skal være nødvendig å laste opp bitfila på FPGA-en for å kopiere ut en modul. Dermed kan programmet kjøres på en hvilken som helst linuxmaskin for å kopiere ut en modul fra fil. Gyldige inputverdier er fra 0 til 39 i bredden og 0 til 47 i høyden siden FPGA-en består av 40*48 CLB-er.

```
./CLBRead 0 0 0 0 myModule.bit
```

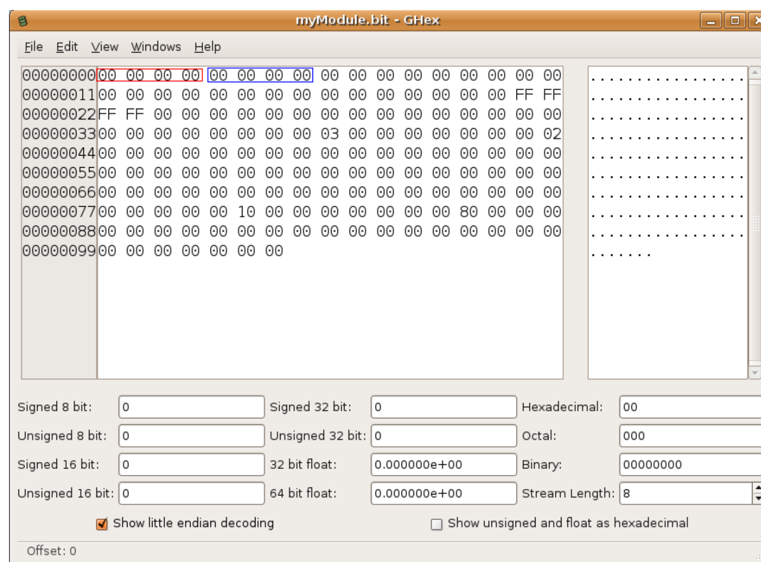
Kommandoen over vil lese ut den øverste CLB-en i venstre hjørne på FPGA-en og lagre den i myModule.bit før programmet avsluttes. Figur 3.5 viser et eksempel på dette. Filen myModule.bit har blitt generert og figur 3.6 viser hvordan myModule.bit ser ut. Den røde og den blå firkanten forteller hvor mange CLB-er fila inneholder og strukturen på disse.

3.4.2 CLBWrite

Dette programmet skriver et visst antall CLB-er til bitstrømmen i flashområdet som FPGA-en laster inn ved oppstart. Programmet har 3 inputpara-



Figur 3.5: CLBRead konsollvindu: Utlesning av CLB-en øverst til venstre

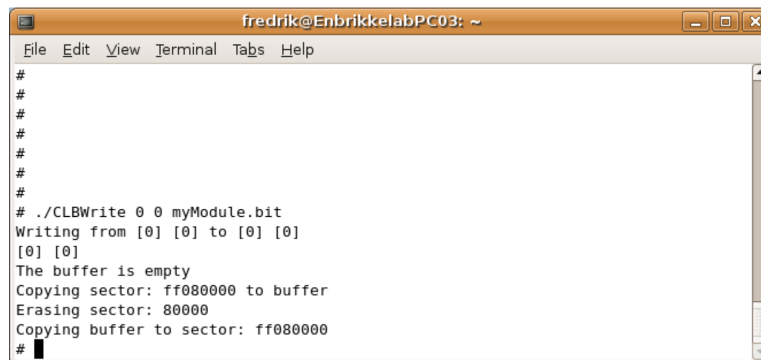


Figur 3.6: CLBRead outputfile: Utlesning av CLB-en øverst til venstre

metere; startCLBkolonne, startCLBrad og inputfilename. Inputfilename er navnet på fila som har blitt generert fra CLBRead. Denne trenger bare startposisjonene da størrelsen på området er definert i inputfila. Kommandoen under viser hvordan et kall til dette programmet kan se ut.

```
./CLBWrite 0 0 myModule.bit
```

Figur 3.7 viser et eksempel på dette



Figur 3.7: CLBWrite konsollvindu: Skriving av CLB-en øverst til venstre

Kapittel 4

Test

Programmene CLBRead og CLBWrite har blitt laget for å bringe utviklingen rundt dynamiske systemer videre og for å belyse nye muligheter i form av dynamisk rekonfigurasjon av FPGA. CLBRread og CLBWrite er verktøy for videreutvikling på dette området. Det er fortsatt nødvendig med forskning omkring hardwaremoduler og kommunikasjonen mellom disse. Derfor er det nødvendig med forståelse omkring verifiseringsverktøyene som er utviklet og konstruksjonen av CLBRead og CLBWrite slik at også dette arbeidet fortsettes ved en eventuell videreføring.

Det er utviklet tre støtteprogrammer for utvikling og test av CLBRead og CLBWrite; FlashRead, FlashWrite og BITDiff. FlashRead kan lese et område, på størrelse med en bitstrømsfil, fra flash til fil. FlashWrite kan skrive et område, på størrelse med en bitstrømsfil, fra fil til flash. BITDiff kan sammenligne to binærfiler.

Det vil i dette kapittelet komme frem hvordan tilnærmingen har vært i utviklingsprosessen og verifiseringsverktøyene som har blitt utviklet vil bli beskrevet.

4.1 Teststrategier

Det ledige flashområdet fra adresse 0xFF010000 til 0xFF080000 er stort nok til å inneholde en FPGA-konfigurasjon. Dette området er derfor veldig godt egnet som et testområde av systemet. Ved å bruke FlashWrite kan en bitstrømsfil skrives til dette området og brukes til testing uten at konfigurasjonsfila til FPGA-en berøres.

Det vil i hovedsak være to hovedtester for CLBRead og CLBWrite (testcase 1 og testcase 2). Testcase 1 går ut på å kopiere ut store logikkområder av

microblazen, for så å skrive disse tilbake på samme plass. Testcase 2 går ut på å forandre en logisk port på FPGA-en som kan måles med et multimeter på IO-portene til Suzakukortet ved bruk av CLBRead og CLBWrite. I underkapitlene som følger vil teststrategiene for utviklingen av CLBRead og CLBWrite diskuteres. Til slutt vil dokumentasjon av testcase 1 og testcase 2 gjennomgås.

4.2 Test av utviklede programmer

4.2.1 FlashRead og FlashWrite

FlashRead og FlashWrite er to programmer som er utviklet for å kunne lese og skrive et helt område i flash.

Dokumentasjonen fra Atmark Techno er noe mangelfull i forhold til skriving og lesing til og fra flash. Det var derfor nødvendig å benytte seg av kode som fulgte med suzakuprosjektet [9] for å finne navn på registre og annen nødvendig informasjon om lesing og skriving til og fra flash. For å teste disse funksjonene har det ledige området på FPGA-en blitt brukt (se tabell 2.3) (adresseområdet fra 0xFF010000 til 0xFF080000). I dette flashområdet har lese- og skrivefunksjonene blitt testet.

FlashRead

FlashRead leser et område på størrelse med konfigurasjonsfila til FPGA-en til fil. FlashRead genererer en bitstrømsfil som vil inneholde dataene som leses fra flashområdet. Ved å lese ut FPGA-konfigurasjonen til en bitstrøm fra flash og sammenligne den med bitstrømmen som er lastet opp til FPGA-kortet kan korrektheten til FlashRead bevises. Om det er ønskelig å sette FlashRead til å lese testområdet i stedet for FPGA-området må denne koden endres i FlashRead.h:

```
#define FPGA_START 0x80000 //fpga area
//#define FPGA_START 0x10000 //test area
```

FlashWrite

FlashWrite går først gjennom sektorene som skal skrives og sletter disse for data slik at ny data kan skrives inn. Så skrives data til det aktuelle området fra en inputfil. For test kan FlashRead brukes til å lese ut de innskrevne

dataene. Om det er ønskelig å skrive til testområdet i stedet for FPGA-området er det nødvendig å gjøre de samme endringene i FlashWrite som beskrevet i FlashRead over.

4.2.2 CLBRead

For å forsikre seg om at de utleste dataene er riktige er det nødvendig å lage noen testbitstrømsfiler. Til dette er BITAnalyse godt egnet. Ved å bruke et bildebehandlingsprogram til å fargelegge hele bitstrømskartet (sette alle bitene svarte) vil alle bitene i bitstrømsfila bli satt høye. Den genererte bitstrømmen fra dette kartet vil bli vist som FF i en hexeditor. Dette betyr at alle databitene i bitstrømmen er satt til å være høye. Med utgangspunkt i en bitstrøm der alle databit er satt høye kan det ved bruk av et bildebehandlingsprogram "viskes ut" en CLB. Ved å generere en bitstrøm av denne fila blir det en bitstrøm der alle bitene er høye bortsett fra dataene i den CLB-en som ble visket ut. Ved å laste denne bitstrømsfila opp på testområdet på den ledige delen i flash med FlashWrite kan CLBRead testes. Om ikke fila generert av CLBRead kun inneholder data med null, fungerer ikke CLBRead. Det er også mulig å legge inn bitkombinasjoner istedenfor bare null og bruke CLBRead til å lese ut disse. Dette er gjort på ulike CLB-er og på CLB-strukturer med ulik størrelse. Om programmet skal benyttes på testområdet er det nødvendig å endre CLBRead.h på samme måte som FlashRead.h vist over

4.2.3 CLBWrite

Det er mulig å teste CLBWrite på samme måte som CLBRead, bare at den aktuelle CLB-en skrives til testområdet i stedet for å leses. Ved å skrive en CLB til testområdet for så å lese ut dette området med FlashRead kan det vurderes om det er de rette adressene som er oppdatert. For å kunne observere at CLBWrite kun skriver til den CLB-en som er satt må all annen data i bitstrømmen være satt høy. Det er viktig å huske at dette ikke er en god nok test og at denne ikke dekker full test av CLBWrite. For å skrive til flashen er det nødvendig å buffre opp en hel sektor av data. Denne testen tester kun de skrevne CLB-dataene og ikke dataene som buffres opp av sektorbufferet og som bare skrives tilbake i områdene rundt den testede CLB-en. Testcase 1 dekker dette. Om programmet skal benyttes på testområdet er det nødvendig å endre CLBWrite.h på samme måte som FlashRead.h vist over.

4.2.4 BITDiff

BITDiff er et program som er utviklet for å validere resultatet av FlashRead. Dette programmet leser to bitstrømsfiler og sammenligner disse. Det leses en og en byte av gangen fra hver fil som sammenlignes. Er de like leses neste byte, er de ikke like skrives adressen til ulikheten ut sammen med dataene. I koden kan det velges om det skal avsluttes etter at en ulikhet er funnet, etter ti ulikheter eller når alle ulikhetene er funnet. Til forskjell fra andre slike program skriver dette programmet ut hvilken adresse ulikhetene ligger på. Dette kan brukes til debugging av programmet for å lokalisere hva som har feilet.

BITDiff er et program som kan brukes sammen med FlashRead. Ved å lese ut dataene i flash med FlashRead før og etter en oppdatering med CLB-Write kan BITDiff brukes til å sammenligne de to filene som er generert av FlashRead. Dette er nyttig i forbindelse med feilsøking av programmet.

4.3 Testcase 1

I denne testen vil noen av CLB-ene i microblazen leses ut av CLBRead til fil, for så å skrives tilbake til de opprinnelige plassene med CLBWrite. Dette vil teste om området som leses ut vil komme tilbake på riktig plass. Det vil også testes om området på utsiden av den utleste modulen vil skrives riktig tilbake etter at flashsektoren har blitt buffret opp, endret med ønskede endringer, slettet fra flash og skrevet tilbake til sektorområdet. Denne testen vil inkludere flere av testprogrammene som er utviklet for dette prosjektet. De vil kjøres i følgende rekkefølge:

1. FlashRead
2. CLBRead
3. CLBWrite
4. FlashRead
5. BITDiff

Figur 4.1 viser konsollvinduet når FlashRead blir kjørt. FlashRead leser hele orginalkonfigurasjonen til fil for å ha et sammenligningsgrunnlag slik at det kan konkluderes med at testen godkjennes eller ikke. I denne testen kalles fila fpgaTest1.bit.

Figur 4.2 viser konsollvinduet der CLBRead kjøres. CLBRead leser fra CLB [10 10] til CLB [14 14] (totalt 16 CLB-er) og lagrer disse i myModule.bit. CLBRead skriver kontinuerlig ut de innleste CLB-ene.

fredrik@EnbrikkelabPC03: ~

File Edit View Terminal Tabs Help

```
#
#
#
#
#
#
#
#
#
#
#
#
#
#
# ls
FlashRead
#
# ./FlashRead fpgaTest1.bit
Reading FPGA part of Flash
Finished reading..
#
# ls
FlashRead       fpgaTest1.bit
# 
```

Figur 4.1: Testcase 1: FlashRead konsollvindu

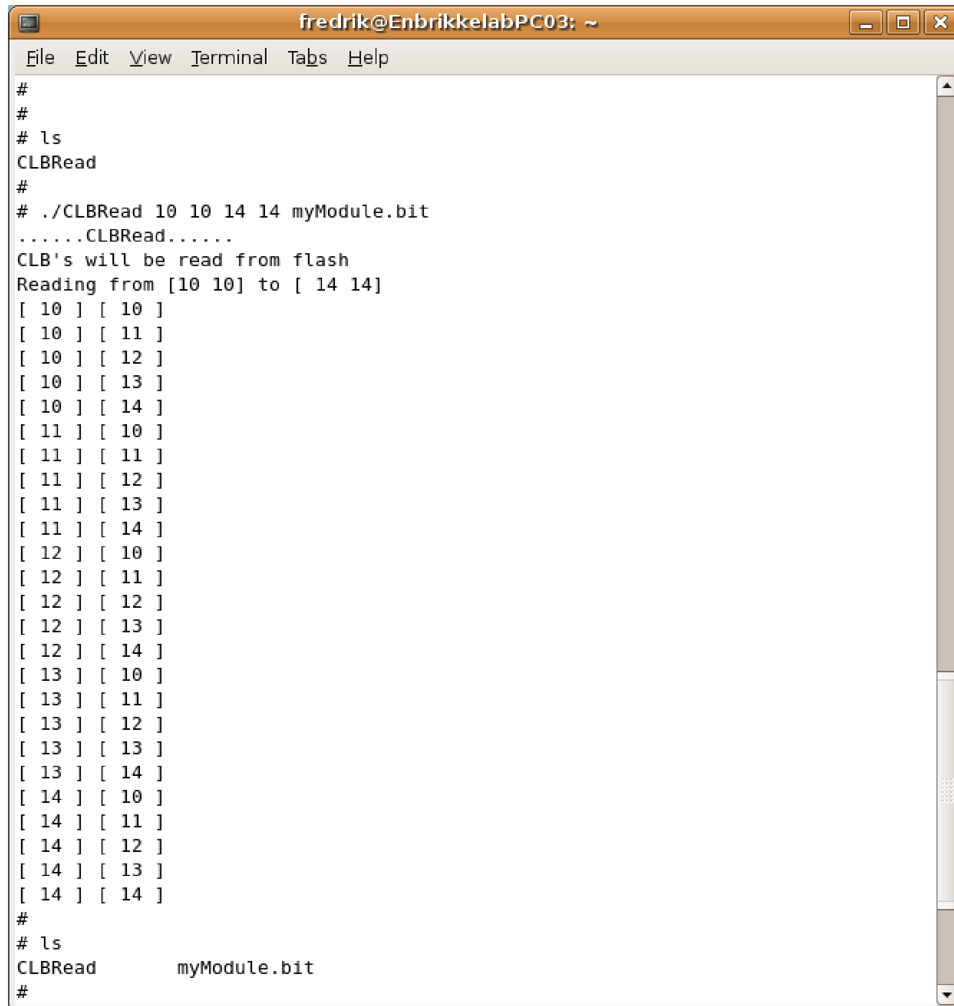
Figur 4.3 viser konsollvinduet til CLBWrite. CLBWrite settes til å starte og skrive fra og med posisjon [10 10] som er det samme som CLBRead for å skrive tilbake modulen til samme posisjon som der den ble lest ut. Programmet vil lese ut fra bitstrømsfila dens struktur, for så å skrive dataene til dette flashområdet. Som utskriften fra figur 4.3 viser kopieres først sektoren som skal skrives til inn i et buffer. Så gjøres endringene i dette bufferet før sektoren i flash slettes og bufferet blir skrevet tilbake.

For å teste at flashområdet er uendret etter at CLBWrite gjorde oppdateringen kjøres FlashRead en gang til for å lese av utseende på flashområdet. Dette for å sammenligne utseende før og etter etter oppdateringen. I denne testen blir denne fila kalt fpgaTest2.bit.

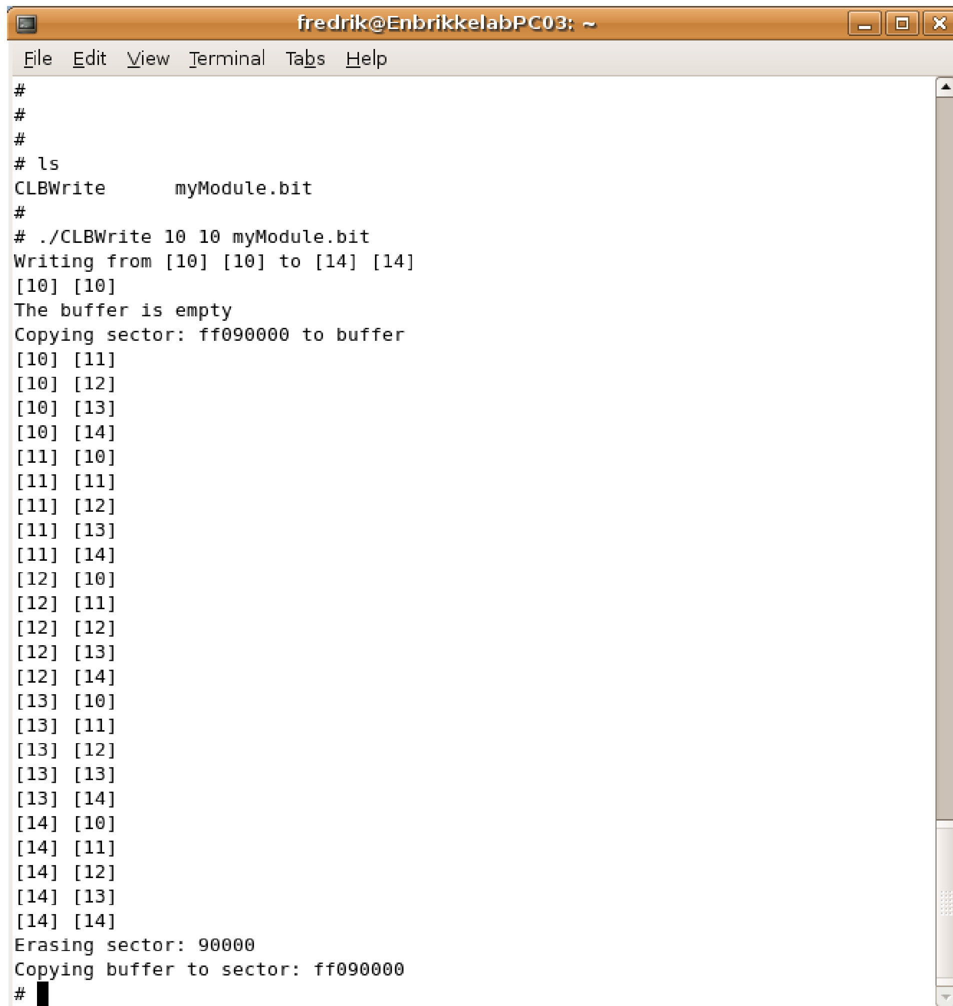
Programmet BITDiff er laget for å sjekke likheten til to binærfiler og det leses en byte av gangen som sammenlignes før neste byte leses. Figur 4.4 viser hvordan konsollvinduet til dette programmet ser ut når det blir kjørt. Som figuren viser blir dette programmet kjørt på en vanlig linuxmaskin og ikke på suzakukortet.

Figur 4.4 refererer denne testen veldig godt. Som vist er det ingen forskjell på de to filene som ble lest ut før og etter at de to programmene CLBRead og CLBWrite ble eksikvert. Med dette resultet kan det konkluderes:

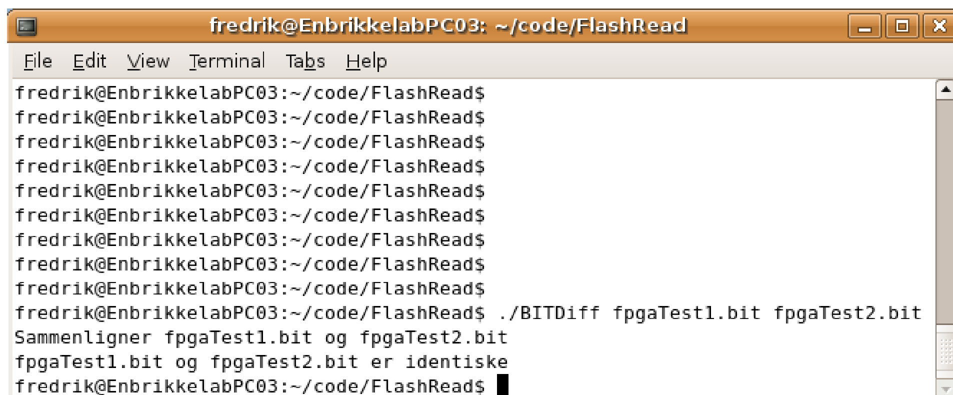
Testcase 1: OK



Figur 4.2: Testcase 1: CLBRead konsollvindu



Figur 4.3: Testcase 1: CLBWrite konsollvindu



Figur 4.4: Testcase 1: BITDiff konsollvindu

4.4 Testcase 2

Denne testen går i hovedsak ut på å bytte ut funksjonalitet på FPGA-en for å bevise at bytte av moduler er mulig. I denne testen vil en OG-port inne på FPGA-en byttes ut med en ELLER-port. Resultatet vil måles på en utgang av IO ved bruk av et multimeter. Kretsen inne på FPGA-en har to innganger og en utgang til IO på kretskortet.

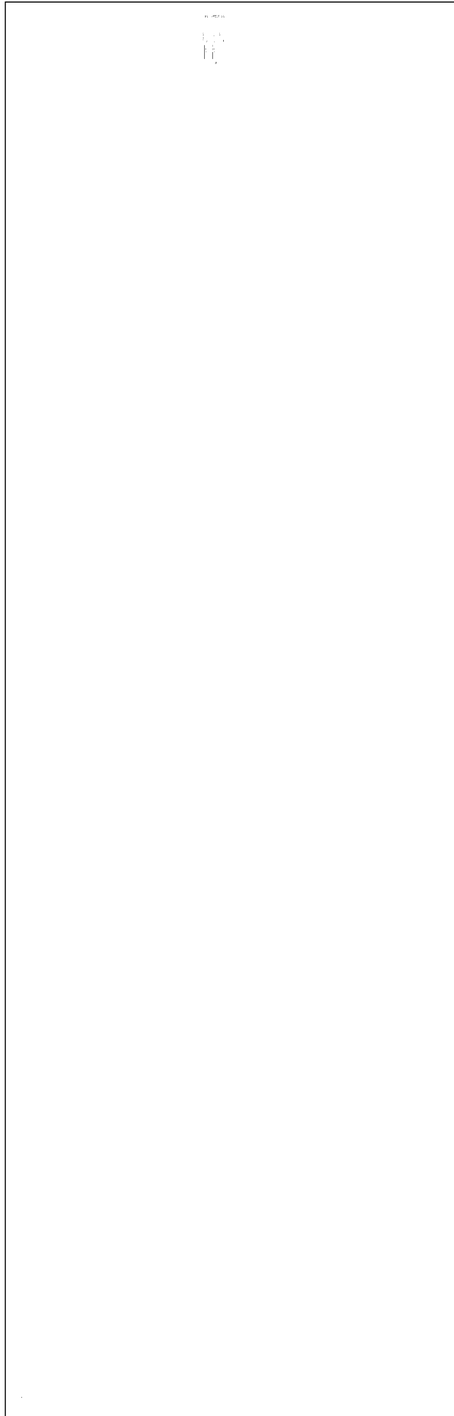
Utgangspunktet til denne testen er fra tidligere tester gjort med BITAnalyse, der testen gikk ut på å gjøre seg kjent med BITAnalyse og test av dette programmet. Dette ble gjort under forprosjektet til denne oppgaven [3].

CLBRead og CLBWrite kan kun endre CLB-nivået eller CLB-laget på FPGA-en, mens et slikt system som en ELLER-port, består av både IOI, TERM og CLB. Det er derfor nødvendig å på en eller annen måte kompilere disse to prosjektene sammen i ett og samme utlegg. Etter noe utforskning med Xilinx EDK utviklingsverktøy viste dette seg litt knotete. En ting er å lage en IP (Intellectual Property) til microblazearkitekturen, en annen er å få denne modulen til å ha egen IO uavhengig av microblazearkitekturen. Enda vanskeligere var det å bruke EDK og ha to hundre prosent uavhengige strukturer på FPGA-en og plassere OG/ELLER porten på et ønsket sted i utlegget. Det ble i dette prosjektet ikke brukt veldig mye tid på dette, men i stedet ble denne jobben kjapt gjort med bruk av BITAnalyse.

BITAnalyse viste seg å være et veldig hendig program til dette. Ved å passe på at de to utleggene microblaze.bit og OR.bit ikke hadde noen felles biter i bitstrømmen kunne disse ved bruk av et bildebehandlingsprogram plasseres over eller oppå hverandre. Ved å laste det nye utlegget tilbake til BITAnalyse kan en ny bitstrøm genereres med innholdet av begge de to utleggene. Figur 4.5 og figur 4.6 viser de to bitstrømmene som kan settes sammen til en. Det er ikke uten risiko å sette sammen bitstrømmer på denne måten da “long lines” (lange ledninger) kan forekomme. “Longlines“ er ledninger som går på tvers i FPGA-strukturen og kobler sammen CLB-strukturer på tvers i utlegget. Om CLB-ene ikke benytter dette i området kan det gå bra å gjøre det på denne måten, noe det her gjør. ”Long lines“ er beskrevet i kapittel 2 i sammenheng med bussmakroer.

Figur 4.5 test1.bit er en OR-port med to utganger og en inngang. Det er også laget en høy og en lav utgang fra FPGA-en til å koble på inngangene og for å kunne sjekke porten opp mot sin logiske funksjonstabell. Under vises test1 i sin helhet.

```
entity test1 is
port  (IO1: out std_logic;
   IO2: out std_logic;
   IO3: in std_logic;
```



Figur 4.5: Testcase 2: OR-port.bit

Figur 4.6: Testcase 2: microblaze.bit

```
   IO4: in std_logic;
   IO5: out std_logic);
end test1;

architecture Behavioral of test1 is

begin
IO1 <= '0';
IO2 <= '1';
IO5 <= IO3 or IO4;

end Behavioral;
```

UCF fila som setter hvilke interne signal som går til hvilke IO på FPGA-en er slik:

```
NET "IO1" LOC = "E7"; # pin 29N_0
NET "IO2" LOC = "D7"; # pin 29P_0
NET "IO3" LOC = "C7"; # pin 30N_0
NET "IO4" LOC = "B7"; # pin 30P_0
NET "IO5" LOC = "D8"; # pin 31N_0
```

Ved å legge disse to utleggene inn på FPGA-en som et utlegg, vil det være mulig å gjennomføre denne testen. Testen går ut på at programmet som kjører på microblazeprosessoren kan bytte ut ELLER-porten med en OG-port. Dette vil kunne måles med en digitalt mulitimeter på utgangen. For å laste inn det nye utlegget ble Hermit benyttet. Hermit er bootloaderen til FPGA-kortet. En grundig beskrivelse av hvordan man oppdaterer bitstrømmer via Hermit er godt beskrevet i Suzaku-kortet sin software manual [10]

Test2.bit inneholder OG-funksjonalitet og er ellers helt lik Test1.bit som inneholder ELLER-porten.

Ved å bruke CLBRead programmet kan det genereres to moduler, en med OG-funksjonalitet og en med ELLER-funksjonalitet. ELLER-funksjonaliteten kan lages på to forskjellige måter. Modulen kan leses ut fra flashen på det nye utlegget, eller den kan genereres direkte fra Test1.bit. Da Test2.bit ikke er inkludert i et utlegg sammen med microblaze må denne modulen genereres fra fil. CLBRead inneholder funksjonalitet for å kunne lese moduler rett ut fra en bitstrømsfil eller fra flash. Dette vil teste at funksjonaliteten fungerer både ved å lese ut en modul fra en bitstrømsfil på PC og lese ut en modul fra flash på Suzakukortet.

For å lese ut en modul er det nødvendig å vite hvor funksjonaliteten du skal lese ut er lokalisert i bitstrømmen, altså hvilke CLB-koordinater som inneholder den ønskede modulen. Det kan være vanskelig å finne ut av dette, men her er en metode; Åpne bitstrømmen i BITAnalyse og lagre den

som en bildefil slik at den kan åpnes i et bildebehandlingsprogram. Her kan pikselkoordinatene enkelt leses ut ved å sette musepekeren over den pikselen der modulen starter. Eksempelvis ligger den aktuelle dataen i Test1.bit i koordinatene 443, 80 som betyr i kolonne 443 rad 80. Rad 80 er starten på den øverste CLB-raden, for utregning av kolonne brukes tabell 2.5. Under vises utregningen:

$$443 - 307 = 136$$

$$136/19 = 7,16 = 7$$

$$2 + 8 + 7 = 17$$

Utregningen over viser at de ønskede dataene ligger i CLB-kolonne 17. For å lese ut denne CLB-en kan CLBRead kjøres:

```
./CLBRead 17 0 17 0 OR.bit

./CLBRead 17 0 17 0 AND.bit Test2.bit
```

Som vist leses ELLER-porten ut av flash (for å gjøre dette må programmet kjøres på microblaze), mens OG-porten leses fra Test2.bit (her kjøres programmet på en linuxmaskin).

Ved å kjøre CLBWrite kan de to modulene som har blitt laget enkelt skrives tilbake. Det er viktig å huske at hele bitstrømmen må lastes inn på nytt for at endringene skal fungere.

```
./CLBWrite 17 0 OR.bit

./CLBWrite 17 0 AND.bit
```

Ved å benytte et digitalt multimeter kan man teste logikken opp mot funksjonstabellene til en OG-port og en ELLER-port.

Testcase 2: OK

Kapittel 5

Diskusjon

Dette kapitelet vil evaluere løsninger og erfaringer som er blitt produsert gjennom arbeidet med denne masteroppgaven.

5.1 Modulbasert rekonfigurasjon

Som nevnt i introduksjonen må et dynamisk system kunne syntetiseres modulvis uavhengig av hverandre, for så å bytte mellom disse under kjøring av systemet. Det utviklede systemet gjør akkurat dette. Test1.bit og Test2.bit som blir omtalt i kapittel 4 er FPGA-utlegg som ble syntetisert under forprosjektet til denne masteroppgaven [3]. Disse FPGA-utleggene er utviklet helt uavhengig av microblazen og som testene viser fungerer dette fint. Det som derimot er nødvendig å arbeide mer med er hvordan en modul kan koble seg til resten av systemet via en bussmakro og utarbeide en standard eller definisjon omkring dette.

Modulbasert partiell rekonfigurering er det systemet som i dag ligger nærmest den utviklede metoden å gjøre rekonfigurering på. Denne metoden skifter ut en eller flere kolonner i FPGA-matrisen under kjøring. Virtex-4 familien fra Xilinx støtter denne form for rekonfigurasjon. Ulempen med denne metoden er at hele kolonnen skiftes ut, noe som gjør kommunikasjon via modulen umulig mens konfigureringen pågår. Eventuell logikk som ligger i samme kolonne vil heller ikke kunne kjøres under rekonfigureringa. Denne metoden kan kalles "realtime reconfiguration".

Den utviklede form for rekonfigurering vil ikke direkte kunne kalles "realtime reconfiguration" da endringene ikke direkte gjøres i logikken som kjøres på FPGA-en, men på bitstrømsfila FPGA-en laster inn ved oppstart. Enkelte vil kanskje tenke på dette som en tilbakegang fra hva som tidligere er

gjort inne på brikken i forhold til partiell rekonfigurering. For å drive den innovative utviklingen er det nødvendig å ”think outside the box“, og det er akkurat dette som her blir gjort. Dette prosjektet er startet for å drive en videreutvikling og da er det ofte viktig å starte med blanke ark og nye ideer samtidig som man har kjennskap til teknologiens historie.

Selve brikken som er laget av Xilinx er det vanskelig å gjøre noe med, men bruken av denne er det mulig å endre, som gjort i denne oppgaven. For å kunne sette denne oppgaven i et større perspektiv og sammenligne den med tidligere arbeider, vil den utviklede metoden for å gjøre rekonfigurasjon på kombinert med multi-context FPGA design (beskrevet i kapittel 2), være en mulig vei å gå for utviklingen. Disse to teknologiene vil sammen kunne lage et veldig dynamisk system for morgendagens teknologi, der FPGA-en har to plan og kan endre det ene planet, samtidig som det andre planet kjører, for så å bytte og kjøre det første planet igjen. Det er heller ikke nødvendig å kjøre ett helt plan, men bare deler av et plan. På denne måten kan forskjellige moduler raskt byttes inn og ut også uten at rekonfigurasjon av hele brikken er nødvendig.

Modulbasert partiell rekonfigurering benytter seg av en bussmakro som i sammenheng med den utviklede metoden også er nødvendig. Akkurat hvordan en bussmakro bør lages for dette systemet er ikke utredet og vil nevnes i forslag til fremtidige arbeider innenfor temaet. Det vil også være nødvendig å gjøre en definisjon av en modul.

Det er vanskelig å komme over artikler og dokumentasjon rundt dynamisk rekonfigurerbar logikk uten å treffe på brikkefamilien XC6200 fra Xilinx. Dette var en “åpen” brikkearkitektur som ble mye brukt i forskningsmiljøer. Alt hva angår dynamisk rekonfigurasjon blir linket opp mot denne familien og dens muligheter. Her menes optimaliseringsalgoritmer, reallokeringsmuligheter og defragmentering. Dette er gammel teknologi og er ikke portabelt til dagens systemer. Spesielt ikke om man ser på hvordan strukturen på CLB-nivået ligger.

Systemet som er utviklet er laget på denne måten for å unslippe ulempene med partiell rekonfigurasjon. Det vil ikke være mulig å kjøre et NoC (network on chip) på en plattform med partiell rekonfigurasjon da hele kolonner med moduler vil bli rekonfigurert og ikke bare en modul. Denne oppgaven er skrevet for å belyse andre former for dynamisk rekonfigurasjon. I denne metoden dukker det opp nye problemer som følge av at dagens FPGA-er ikke er designet for dette formålet. En mulighet kan være å lage et nytt FPGA-design der det nevnte systemet i kombinasjon med multi-context FPGA-design utgjør byggesteinene.

5.2 Optimalisering

Den utviklede koden for rekonfigurasjon er relativt liten, spesielt om man sammenligner systemet med for eksempel BITAnalyse. Dette er et poeng da microblazen har lite ressurser og relativt lav klokkefrekvens for kodeeksikvering. Det meste av tiden går til sette registere i flash for å kunne skrive til den.

Det er mulig å optimalisere de utviklede programmene noe, helst før et slikt produkt blir "kommersielt". Slik CLBRead genererer bitfila ligger alle dataene i CLB[1,1] lagret, før dataene i CLB [1,2] [1], osv. For å optimalisere dette maksimalt må hele feltet med CLB-er lagres samtidig. Dataene til hele CLB-området må da ligge kolonnevis og ikke CLB-vis som her er gjort. Grunnen til at det her ligger CLB-vis er for å kunne debugge dataene og analysere dem. Om dataene leses kolonnevis uavhengig av hvor mange rader som leses blir det nødvendig med et analyseverktøy. Det presiseres derfor at den nåværende versjonen ikke er optimal i forhold til hastighet, men i stedet gjør manuelle endringer i en hexeditor mulig.

5.3 Network on chip

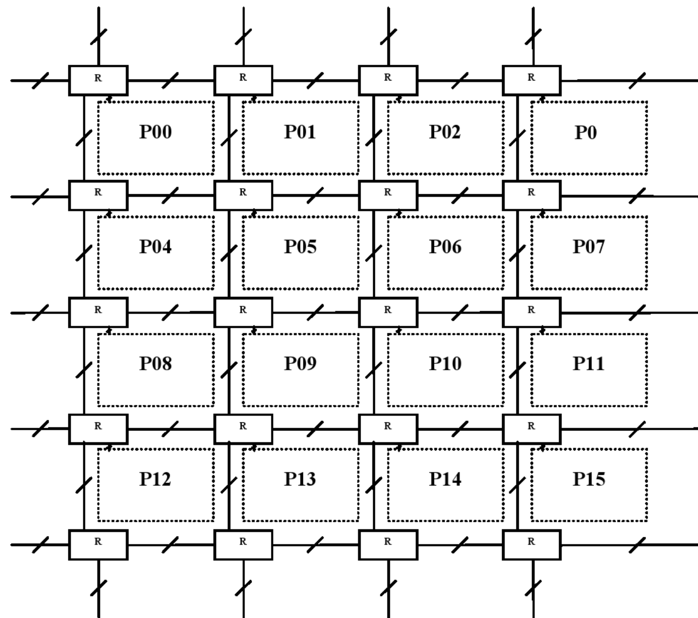
NoC (Network on chip) er en struktur som blir mer og mer aktuell i FPGA-systemer. Jo større og raskere FPGA-ene blir, jo mer naturlig vil det være å ha en bussbasert nettverksstruktur internt i brikken. Figur 5.1 viser en mulig struktur på dette nettverket der hver modul også har en "kommunikasjonsmodul" som tar seg av nettverksruting. Kommunikasjonen vil foregå på delte nettverkslinjer, noe som vil lage en flaskehals i systemet.

Microblazen opptar veldig mye av de logiske ressursene på FPGA-en. Figur 4.6 viser et bilde av et FPGA-utlegg som inneholder en microblaze arkitektur. Som bildet viser er det ikke veldig mye plass til annen logikk, spesielt ikke et NoC (network on chip). Dette kommer selvfølgelig an på størrelsen hver modul har og hvor plasskrevende NoC i seg selv er.

Figur 5.1 tar for seg hvordan en tenkt NoC-topologi kan se ut, mens figur 4.6 viser hvordan et FPGA-utlegg som inneholder en microblaze ser ut i praksis. Figurene viser at disse strukturene passer relativ dårlig inn i hverandre. Det meste av publikasjon på området omkring NoC og ideene om hvordan dette er tiltenkt er stort sett lik den som er illustrert i figur 5.1.

Det kan nok være vanskelig å få en NoC-skisse til i praksis på en FPGA da FPGA-strukturer ofte har en helt annen fasong. Tabell 2.6 viser at den

[1] CLB [i,j], i = kolonne, j = rad



Figur 5.1: 2D Mesh - Network on Chip (NoC) [5]

største spartan FPGA-en (XC3S5000) er over dobbelt så stor som den dette prosjektet har hatt tilgjengelig (XC3S1000), noe som betyr at et NoC kan være realiserbart. Det vil også bli mer og mer aktuelt ettersom FPGA-ene vokser i størrelse. Samtidig kan det også være mulig å benytte seg av en picoblaze istedenfor en microblaze for kontroll på modulene. Picoblaze er en 8-bit RISC prosessor og benytter kun 0.3 prosent av en XC3S5000 [14]. Alt dette kommer selvfølgelig an på hvordan NoC ser ut og hvor stor plass som behøves, men på en stor brikke vil NoC absolutt være mulig å få til.

Hva angår optimalisering, reallokering og defragmentering av moduler har i denne rapporten ikke blitt diskutert. På dette nivået i utviklingen vil det være vanskelig da strukturene og mulighetene til teknologien ikke er definert. Det er tidligere utført endel arbeid omkring dette, men det er i hovedsak gjort for XC6200 arkitekturen [6]. NoC som her er omtalt vil antagelig føre til at modulene i seg selv blir så store som mulig og kommunikasjonen over nettverket så liten som mulig da flaskehalsen ligger i kommunikasjonen mellom modulene. Det vil være mer naturlig med algoritmer for optimalisering, reallokering og defragmentering i definisjonen av NoC.

5.4 Fremtidig arbeid

Hensikten med den punktvise fremstillingen under er å sette fokus på begrensninger i det eksisterende systemet, og komme med ideer og anbefalinger til utvidelser og tilpasninger

Definere en modul og utarbeide et system for hvordan en modul kan lages:
For utarbeidelse av en modul til et system er det nødvendig med retningslinjer for hvordan denne kan lages. Utviklingsverktøy må settes spesielt opp og eksterne tilkoblinger må defineres. Det er nødvendig å utarbeide strukturen til en bussmakro og dens tilkoblinger, både fra modulen sin side og det statiske nettverket.

Definere kommunikasjon mellom microblaze og modul:
For at en modul skal kunne kommunisere med microblaze er det nødvendig med en busstruktur. Det vil være nødvendig å definere denne og få den realisert ved bruk av Xilinx EDK utviklingsverktøy. En utredelse av kommunikasjon med microblaze og retningslinjer for hvordan dette kan realiseres på FPGA vil være nødvendig.

Inkludere CLBRead i BITAnalyse:
Moduler vil være enkelt å lage om CLBRead er en del av funksjonaliteten i BITAnalyse. Kombinert med retningslinjene for genereringen av en modul vil dette lette jobben med å utvikle en modul betraktelig.

Utredelse av klokkenettverk og klokkeproblematikk internt i moduler:
Det er noe uklart hvordan det globale klokkenettverket er distribuert og det er nødvendig å utrede dette nærmere. Problematikken er nevnt i forhold til partiell rekonfigurasjon, men burde undersøkes i sammenheng med den utviklede metoden.

Ytelsesbetraktninger, kritiske stier og maksimal klokkefrekvens:
Ved å kombinere bitstrømmer fra forskjellige prosjekter vil informasjonen om lengden på den kritiske stien og den øvrige klokkefrekvensen gå tapt. Bruken av regulær struktur vil gjøre det mulig å estimere lengden på den kritiske stien, men resultatene vil være forbundet med usikkerhet. Muligheten for å beregne ytelsen til et slikt system bør derfor utredes nærmere.

Utvidet støtte for ulike brikkearkitekturer og brikketyper:
Tilrettelegging for ulike brikkefamilier og modeller har ikke vært prioritert i arbeidet med denne oppgaven. Den nåværende versjonen støtter kun systemet på Suzakuplattformen med FPGA-en XC3S1000. Skal andre plattformer og FPGA-er støttes er det nødvendig å utvide koden.

Optimalisering av CLBRead og CLBWrite:
For at den utviklede koden skal kjøre med optimal utnyttelse vil det være

nødvendig å optimalisere koden med tanke på cache-størrelser og lagringsmetoder. Det bør også analyseres hvor effektive skrive- og lesefunksjonene er og vurdere om mer data kan skrives eller leses samtidig.

Tilpasse NoC for et slikt system:
For at NoC skal fungere med dette systemet må nettverket passe i forhold til CLB-laget. Det er derfor nødvendig å tilpasse NoC for at strukturene skal passe sammen.

Utvikle standard for en bussmakro:
En standard for en bussmakro bør utvikles og realiseres på FPGA.

Kapittel 6

Konklusjon

Denne masteroppgaven er et bidrag til dynamiske selvrekonfigurerbare maskinvaresystemer. Det er bevist at det er mulig å gjøre selvrekonfigurering av FPGA på den måten at brukeren laster opp sin modul og kan kjøre den uten kjennskap til hele FPGA-utlegget. Dette er bevist ved at to forskjellige moduler er lastet opp og byttet ut med hverandre og resultatet er testet på IO ved bruk av et digitalt multimeter.

Det er utviklet to programmer, CLBRead og CLBWrite som tilbyr denne funksjonaliteten. CLBRead kan lese en CLB-struktur med forskjellig størrelse, der en enkelt CLB er den minste oppdelinga, til fil. En CLB-struktur kan leses ut fra flash på FPGA-kortet, eller fra en bitstrømsfil på en PC. CLBWrite skriver en filstruktur generert av CLBRead til flashområdet der FPGA-konfigurasjonene ligger. Ved oppstart av FPGA-en vil det nye oppsettet konfigureres opp.

Systemet som er utviklet kan rekonfigureres helt uten behov for ekstern tilkobling eller manipulasjon, det er FPGA-en selv som gjør hele jobben. Forskjellige moduler kan lagres og lastes inn ved behov.

Bibliografi

[1] Ambiesense. http://www.ambiesense.com.

[2] John W. Lockwood2 Edson L. Horta1 and S´ergio T. Kofuji1. Using parbit to implement partial run-time reconfigurable systems.

[3] Fredrik Gravdal. Fleksible moduler på xilinx fpga. Technical report, NTNU, 2006. Prosjektoppgave.

[4] Ingar Hauge. Arkitekturbeskrivelser for ahead. Technical report, NTNU, 2005. Prosjektoppgave.

[5] Ingar Hauge. Analyse, dekomponering og rekonstruksjon av fpga-konfigurasjoner for ahead. Master's thesis, NTNU, 2006.

[6] IEEE Zhiyuan Li James Cooley Member IEEE Stephen Knol Katherine Compton, Student Member and IEEE Scott Hauck, Senior Member. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 10, NO. 3*, June, 2002.

[7] AmbieSense EU IST project. http://www.ambiesense.net.

[8] Atmark Techno. http://www.atmark-techno.com/en/products/suzaku/suzaku-s.

[9] Atmark Techno. Suzaku prosjekt. http://download.atmark-techno.com/suzaku/fpga-proj/8.2i/sz030/sz030-20070420.

[10] Atmark Techno. *SUZAKU Hardware Manual*. http://download.atmark-techno.com/suzaku/manual/suzaku-s-hardware-manual-en-1.0.4.pdf, 1.0.4 edition, December 2004.

[11] Atmark Techno. *SUZAKU Software Manual*. http://download.atmark-techno.com/suzaku/manual/suzaku-software-manual-en-1.1.0.pdf, 1.0.4 edition, December 2004.

[12] Xilinx. Spartan-3 advanced configuration architecture. Xilinx.com, 2004.

[13] Xilinx. Development system reference guide. Xilinx.com, 2005.

[14] Xilinx. Picoblaze 8-bit embedded microcontroller user guide. http://www.xilinx.com/bvdocs/userguides/ug129.pdf, November 2005.

[15] Xilinx. Spartan-3 fpga family: Complete data sheet. Xilinx.com, April 26, 2006.

Tillegg A

Kildekode

A.1 CLBRead.h

```
//************************************************************
// Filename:     CLBRead.h
//
// Brief:
// Author:        Fredrik Gravdal, 2007, NTNU
//************************************************************
#define FLASH_START 0xFF000000
#define FPGA_START 0x80000    //fpga area
//#define FPGA_START 0x10000 //test area
#define CLB_START 0x263A

//one CLB matrix
typedef struct{
        unsigned int iCLB [19*2];
}clb;

unsigned int endian(unsigned int c);
clb read_clb_file(FILE * clbFile, unsigned int row, unsigned int col);
int read_file(int startCol, int startRow, int endCol, int endRow, char*
outPutFile, char* inPutFile);
clb read_clb_flash(unsigned int row, unsigned int col);
int read_flash(int startCol, int startRow, int endCol, int endRow,
char* outPutFile);
int main(int numArgs, char* argc[]);
```

A.2 CLBRead.c

```
//*************************************************************
// Filename:     CLBRead.c
//
// Brief:        This program reads a CLB structure
//               from file or flash on the suzaku board
//
// Author:       Fredrik Gravdal, 2007, NTNU
//*************************************************************
#include <stdio.h>
#include <stdlib.h>
#include "CLBRead.h"

//change endian if printing data to screen at x86 arcitectures
unsigned int endian(unsigned int c){
        return ((c&0xff)<<24)+((c&0xff00)<<8)+((c&0xff0000)>>8)+((c>>24)&0xff);
}

//reads one defined CLB from a inputfile
clb read_clb_file(FILE * clbFile, unsigned int row, unsigned int col){
        int i,j,k=0;
        unsigned int iData;
        unsigned int clbStart = CLB_START/4;
        clb oneCLB;

        //haxx to read 16bit exstra. (0x263A/4=2446,5) this does the tric :)
        fread(&iData, 2,1, clbFile);

        //loops the file pointer to start at CLB (0,0)
        for(j=0; j < clbStart; j++){
                fread(&iData, 4,1, clbFile);
                iData=endian(iData);
        }

        for(j=0; j < (101*19*col)+(row*2); j++){
                fread(&iData, 4,1, clbFile);
                iData = endian(iData);
        }

        for (k=0;k<19*2;k+=2){

                fread(&iData, 4,1, clbFile);
                //iData=endian(iData);
                oneCLB.iCLB[k] = iData;
                //printf("k: %d data: %x \n", k,endian(iData));

                fread(&iData, 4,1, clbFile);
                //iData=endian(iData);
                oneCLB.iCLB[k+1] = iData;
                //printf("k: %d data: %x \n", k+1,endian(iData));
```

```
                //loop pointer to next CLB line (101*4byte)
                for(i=0;i<99;i++){
                        fread(&iData, 4,1, clbFile);
                //      iData=endian(iData);
                }
        }
        return oneCLB;
}

//reads a CLB-structure from file
int read_file(int startCol, int startRow, int endCol, int endRow, char*
outPutFile, char* inPutFile){

        FILE * outFile;
        FILE * inFile;

        outFile = fopen (outPutFile,"wb");
        inFile = fopen (inPutFile,"rb");
        if (outFile == NULL){printf("Error opening outFile\n"); return 0;}
        if (inFile == NULL){printf("Error opening inFile\n"); return 0;}

        //write matrix structure and size to output file
        int data = endCol - startCol;
        fwrite(&data, 4, 1, outFile);
        data = endRow - startRow;
        fwrite(&data, 4, 1, outFile);

        clb oneCLB;
        clb CLBTable[endRow-startRow+1][endCol-startCol+1];

        int col,row=0;
        printf("Reading from [%d] [%d] to [%d] [%d]\n",startCol,startRow,
endCol, endRow);

        //loop col and row. save it in CLBTable[row][col]
        for(col = startCol; col <= endCol; col++){
                for(row = startRow; row <= endRow; row++){
                printf("[ %d ] [ %d ]\n",col,row);
                        CLBTable[row - startRow][col - startCol] =
read_clb_file(inFile, row, col);
                        rewind (inFile);

                }
        }

        //save CLBTable to outFile
        fwrite(CLBTable, sizeof(oneCLB.iCLB[0]),
sizeof(CLBTable)/sizeof(oneCLB.iCLB[0]), outFile);
```

```
        fclose (outFile);
        fclose (inFile);

        return 0;
}

//reads one CLB from flash
clb read_clb_flash(unsigned int row, unsigned int col){

        int k=0;
        unsigned int iData;
        unsigned int CLBaddr;
        clb oneCLB;

        CLBaddr = FLASH_START + FPGA_START + CLB_START + (row*4*2) +
(col*101*4*19);

        for (k=0;k<19*2;k+=2){

                iData = (*(volatile unsigned short *) (CLBaddr
+k*2)) << 16;
                iData|= (*(volatile unsigned short *) (CLBaddr+2 +k*2));
                oneCLB.iCLB[k] = iData;
                iData = (*(volatile unsigned short *) (CLBaddr+4 +k*2)) << 16;
                iData|= (*(volatile unsigned short *) (CLBaddr+6 +k*2));
                oneCLB.iCLB[k+1] = iData;

                CLBaddr += 100*4;
        }
        return oneCLB;
}

//reads a CLB-structure from flash
int read_flash(int startCol, int startRow, int endCol, int endRow, char*
outPutFile){

        FILE * outFile;

        outFile = fopen (outPutFile,"wb");
        if (outFile == NULL){printf("Error opening outFile\n"); return 0;}

        //write matrix size to output file
        int data = endCol - startCol;
        fwrite(&data, 4, 1, outFile);
        data = endRow - startRow;
        fwrite(&data, 4, 1, outFile);

        clb oneCLB;
```

```
        clb CLBTable[endRow-startRow+1][endCol-startCol+1];

        int col,row=0;
        printf("Reading from [%d] [%d] to [%d] [%d]\n",startCol,startRow,
endCol, endRow);

        //loop col and row. save it in CLBTable[row][col]
        for(col = startCol; col <= endCol; col++){
                for(row = startRow; row <= endRow; row++){
                        printf("[ %d ] [ %d ]\n",col,row);
                        CLBTable[row - startRow][col - startCol] =
read_clb_flash(row, col);
                }
        }

        //save CLBTable to outFile
        fwrite(CLBTable, sizeof(oneCLB.iCLB[0]),
sizeof(CLBTable)/sizeof(oneCLB.iCLB[0]), outFile);

        fclose (outFile);

        return 0;
}

//main method
int main(int numArgs, char* argc[]){

        printf("......CLBRead......\n");

        int startCol = atoi(argc[1]);
        int startRow = atoi(argc[2]);
        int endCol = atoi(argc[3]);
        int endRow = atoi(argc[4]);

        if(numArgs == 7){
                printf("CLB's will be read from input file\n",numArgs);
                read_file(startCol, startRow, endCol, endRow, argc[5],
argc[6]);
        }
        else if(numArgs == 6){
                printf("CLB's will be read from flash\n");
                read_flash(startCol, startRow, endCol, endRow, argc[5]);
        }
        else{
                printf("Number of statements does not mach: %d\n", numArgs);
                return 0;
        }

        return 0;
```

```
}
```

A.3 CLBWrite.h

```
//************************************************************
// Filename:     CLBWrite.h
//
// Brief:
// Author:       Fredrik Gravdal, 2007, NTNU
//************************************************************
#define FLASH_START 0xFF000000
#define FPGA_START 0x00080000   //fpga area
//#define FPGA_START 0x00010000 //test area
#define CLB_START 0x263A
#define FLASH_SECTOR_SIZE 0x10000

typedef struct{
        unsigned long iCLB [19*2];
}clb;

void flash_program(unsigned long addr, unsigned short val);
void flash_sector_erase(int sectorAddr);
void copy_sector_to_buf(unsigned short* buf, unsigned int secStartAddr);
void copy_buf_to_sector(unsigned short* buf, unsigned int secStartAddr);
void update_flash(unsigned short* buf, unsigned int addr, unsigned int data,
int finish);
void dump_sector_buf_to_file(unsigned short* buf);
unsigned int endian(unsigned int c);
int main(int numArgs, char* argc[]);
```

A.4 CLBWrite.c

```
//************************************************************
// Filename:     CLBWrite.c
//
// Brief:        This program writes a CLB structure
//               in flash on the suzaku board
//
// Author:       Fredrik Gravdal, 2007, NTNU
//************************************************************
#include <stdio.h>
#include <stdlib.h>
#include "CLBWrite.h"

typedef unsigned short __u16;

//changing little endian with big endian
```

```
#define ___swab16(x)
        ({
                __u16 __x = (x);
                ((__u16)(
                        (((__u16)(__x) & (__u16)0x00ffU) << 8) |
                        (((__u16)(__x) & (__u16)0xff00U) >> 8) ));
        })

#define flash_write(addr, b)
        (void)((*(volatile unsigned short *) (addr)) = ((b)))

#define flash_write_cmd(addr, b)
        (void)((*(volatile unsigned short *) (addr)) = (___swab16(b)))

//global variables to remember which sector that is in the buffer
int sectorInBuf=128;
unsigned int bufferAddr;
unsigned int oldSectorStartAddr;

//writes data to flash
void flash_program(unsigned long addr, unsigned short val)
{
        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);
        flash_write_cmd(FLASH_START + 0xaaa, 0xa0);
        flash_write(addr, val);
        while ((*(volatile unsigned short *) (addr)) != val);
}

//eraseing the sector in flash
void flash_sector_erase(int sectorAddr)
{
        printf("Erasing sector: %x\n", sectorAddr);

        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);
        flash_write_cmd(FLASH_START + 0xaaa, 0x80);
        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);

        flash_write_cmd(FLASH_START + sectorAddr, 0x30);

        while ((*(volatile unsigned short *) (FLASH_START + sectorAddr +
FLASH_SECTOR_SIZE - 1)) != 0xFFFF);
}

//copying the sector in flash to buffer
void copy_sector_to_buf(unsigned short* buf, unsigned int secStartAddr)
{
```

```
        printf("Copying sector: %x to buffer\n", secStartAddr);
        int i;
        for(i = 0; i < FLASH_SECTOR_SIZE; i+=2){
                buf[i/2] = (*(volatile unsigned short *) (secStartAddr + i));
        }
}

//copying the buffer to flash
void copy_buf_to_sector(unsigned short* buf, unsigned int secStartAddr)
{
        int i;
        flash_sector_erase(secStartAddr-FLASH_START);
        printf("Copying buffer to sector: %x\n", secStartAddr);
        for(i = 0; i < FLASH_SECTOR_SIZE; i+=2){
                flash_program(secStartAddr + i, buf[i/2]);
        }
}

//updates the buffer, and writes it to the flash if needed
void update_flash(unsigned short* buf, unsigned int addr, unsigned int data,
int finish){

        int sector = addr/FLASH_SECTOR_SIZE;
        unsigned int sectorStartAddr = sector*FLASH_SECTOR_SIZE;
        unsigned short tmpData1, tmpData2;

        if (finish == 1){
                copy_buf_to_sector(buf, oldSectorStartAddr);
        }
        else {
                //If the buffer is empty
                if(sectorInBuf == 128){
                        printf("The buffer is empty\n");
                        copy_sector_to_buf(buf, sectorStartAddr);
                        oldSectorStartAddr = sectorStartAddr;
                        sectorInBuf = sector;
                        bufferAddr = addr;
                }
                //If the address is in an another buffer
                else if(sectorInBuf != sector){
                        printf("The address is in an another buffer\n");
                        copy_buf_to_sector(buf, oldSectorStartAddr);
                        copy_sector_to_buf(buf, sectorStartAddr);
                        oldSectorStartAddr = sectorStartAddr;
                        sectorInBuf = sector;
                        bufferAddr = addr;
                }

                tmpData2 = data;
```

```
                tmpData1 = data >>16;

                buf[(addr-sectorStartAddr)/2]   = tmpData1;
                buf[(addr-sectorStartAddr+2)/2]   = tmpData2;
        }
}

//FOR DEBUGGING
//writes the flashbuffer to file
void dump_sector_buf_to_file(unsigned short* buf){

        printf("Dumping sector to sectorDump.bit \n");

        FILE * pFile;
        pFile = fopen ("sectorDump.bit","wb");

        int i;
        if (pFile!=NULL){
                for(i = 0; i < FLASH_SECTOR_SIZE/2 ; i+=2){
                        fwrite(&buf[i/2], 2,1, pFile);
                }
        }
        else {
                printf("Error! Dumping sector to sectorDump.bit\n");
        }
        printf("Finished dumping sector to sectorDump.bit\n");
        fclose (pFile);
}

//main method. runs throug cols and rows in the input file and
//updates the flash
int main(int numArgs, char* argc[]){

        int startCol = atoi(argc[1]);
        int startRow = atoi(argc[2]);

        if(numArgs != 4){
                printf("Number of arguments does not mach. Should be 4, you
entered: %d\n", numArgs);
                return 0;
        }
        else{
                FILE * inFile;
                inFile = fopen (argc[3],"rb");
                if (inFile == NULL){printf("Error opening inFile\n"); }

                int endCol, endRow;
                unsigned int iData;
```

```
                fread(&iData, 4,1, inFile);
                endCol = iData;
                fread(&iData, 4,1, inFile);
                endRow = iData;

                endRow += startRow;
                endCol += startCol;

                printf("Writing from [%d] [%d] to [%d] [%d]\n", startRow,
startCol, endRow, endCol);

                unsigned short buffer[FLASH_SECTOR_SIZE];
                clb oneCLB;
                clb CLBTable[endRow-startRow+1][endCol-startCol+1];

                fread(CLBTable, sizeof(oneCLB.iCLB[0]),
sizeof(CLBTable)/sizeof(oneCLB.iCLB[0]), inFile);

                //writes the CLBTable to flash
                int k,col,row=0;
                for(col = startCol; col <= endCol; col++){
                        for(row = startRow; row <= endRow; row++){

                                printf("[%d] [%d]\n",col,row);
                                oneCLB = CLBTable[row - startRow][col -
startCol];

                                 for (k=0;k<19*2;k+=2){

                                        iData = oneCLB.iCLB[k];
                                        update_flash(buffer, FLASH_START +
FPGA_START+CLB_START+(row*4*2)+(col*101*4*19)+(k/2)*(101*4), iData, 0);

                                        iData = oneCLB.iCLB[k+1];
                                        update_flash(buffer, FLASH_START +
FPGA_START+CLB_START+(row*4*2)+(col*101*4*19)+(k/2)*(101*4)+4, iData, 0);
                                }

                        }
                }

                //write buffer back to flash
                update_flash(buffer, 0, 0, 1);
                fclose (inFile);
        }
        return 0;
}
```

A.5 FlashRead.h

```
//************************************************************
// Filename:     FlashRead.h
//
// Brief:
// Author:       Fredrik Gravdal, 2007, NTNU
//************************************************************
#define FLASH_START 0xFF000000
#define FPGA_START 0x80000        //fpga area
//#define FPGA_START 0x10000     //test area
#define FLASH_SECTOR_SIZE 0x10000

void readFPGAPart(int start, int end, char* fileToStore);
int main(int numArgs, char* argc[]);
```

A.6 FlashRead.c

```
//************************************************************
// Filename:     FlashRead.c
//
// Brief:        This program reads flash and stores
//               the data to inputfile.
//               The program reads the size of an
//               XC3S1000 configuration file
//
// Author:       Fredrik Gravdal, 2007, NTNU
//************************************************************
#include <stdio.h>
#include <stdlib.h>
#include "FlashRead.h"

void readFPGAPart(int start, int end, char* fileToStore){
        printf("Reading FPGA part of Flash\n");

        FILE * pFile;
        pFile = fopen (fileToStore,"wb");

        unsigned short tmp_data;
        if (pFile!=NULL){
                for(start; start<end ; start+=2){
                        tmp_data = (*(volatile unsigned short *) (start));
                        fwrite(&tmp_data, 2,1, pFile);
                }
                fclose (pFile);
                printf("Finished reading.. \n");
        }
        else {
```

```
                printf("Error! :(\n");
        }
}

int main(int numArgs, char* argc[]){
        readFPGAPart(FLASH_START+FPGA_START, FLASH_START+FPGA_START+0x725F8,
argc[1]);
        return 0;
}
```

A.7 FlashWrite.h

```
//************************************************************
// Filename:    FlashRead.c
//
// Brief:       This program writes a bitstreamfile to flash.
//
// Author:      Fredrik Gravdal, 2007, NTNU
//************************************************************
#define FLASH_START 0xFF000000
#define FLASH_SECTOR_SIZE 0x10000
#define FPGA_START 0x80000       //fpga area
//#define FPGA_START 0x00010000 //test area

void flash_program(unsigned long addr, unsigned short val);
void flash_sector_erase(int sectorAddr);
void readFPGAPart(int start, int end, char* fileToStore);
void writeFPGAPart (int start, char* fileToWrite);
int main(int numArgs, char* argc[]);
```

A.8 FlashWrite.c

```
//************************************************************
// Filename:    FlashRead.c
//
// Brief:       This program writes a bitstreamfile to flash.
//
// Author:      Fredrik Gravdal, 2007, NTNU
//************************************************************
#include <stdio.h>
#include <stdlib.h>
#include "FlashWrite.h"

typedef unsigned short __u16;

//changing little endian with big endian
#define ___swab16(x)
```

```
        ({
                __u16 __x = (x);
                ((__u16)(
                        (((__u16)(__x) & (__u16)0x00ffU) << 8) |
                        (((__u16)(__x) & (__u16)0xff00U) >> 8) ));
        })

//write data
#define flash_write(addr, b)
        (void)((*(volatile unsigned short *) (addr)) = ((b)))

//write command
#define flash_write_cmd(addr, b)
        (void)((*(volatile unsigned short *) (addr)) = (___swab16(b)))

//writes data to flash
void flash_program(unsigned long addr, unsigned short val) {

        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);
        flash_write_cmd(FLASH_START + 0xaaa, 0xa0);
        flash_write(addr, val);
        while( (*(volatile unsigned short *) (addr)) != val){};
}

//erase flash sector
void flash_sector_erase(int sectorAddr)
{
        printf("Erasing sector: %x\n", sectorAddr);

        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);
        flash_write_cmd(FLASH_START + 0xaaa, 0x80);
        flash_write_cmd(FLASH_START + 0xaaa, 0xaa);
        flash_write_cmd(FLASH_START + 0x554, 0x55);

        flash_write_cmd(FLASH_START + sectorAddr, 0x30);

        while ((*(volatile unsigned short *) (FLASH_START + sectorAddr + FLASH_S

}

//writes a bitstreamfile to the flash
void writeFPGAPart (int start, char* fileToWrite){

        int i,size;
        unsigned char a,b;
        unsigned short tmp;
        unsigned int address = start;
```

```
        flash_sector_erase(FPGA_START);
        flash_sector_erase(FPGA_START + FLASH_SECTOR_SIZE);
        flash_sector_erase(FPGA_START + (FLASH_SECTOR_SIZE*2) );
        flash_sector_erase(FPGA_START + (FLASH_SECTOR_SIZE*3) );
        flash_sector_erase(FPGA_START + (FLASH_SECTOR_SIZE*4) );
        flash_sector_erase(FPGA_START + (FLASH_SECTOR_SIZE*5) );
        flash_sector_erase(FPGA_START + (FLASH_SECTOR_SIZE*6) );

        printf("Writing to flash \n");

        FILE * pFile;
        pFile = fopen (fileToWrite , "rb");

        if (pFile == NULL){
                printf("Error opening file");
        }
        else {
                fseek (pFile, 0, SEEK_END);
                size=ftell (pFile);
                rewind (pFile);

                printf("Starting at address: %x\n", start);
                printf("Writing to address: %x\n", start+size);

                for(i=address;i <= start+size ; i+=2){
                        a = fgetc (pFile);
                        b = fgetc (pFile);
                        tmp = a<<8;
                        tmp|= b;
                        flash_program(i, tmp);
                }
                fclose (pFile);
                printf("Finished writing..\n");
        }
}

//main method
int main(int numArgs, char* argc[]){

        writeFPGAPart (FLASH_START + FPGA_START, argc[1]);

        return 0;

}
```

A.9 BITDiff.c

```
//**************************************************************
```

```
// Filename:     BITDiff.c
//
// Brief:        This program compares to bitstreamfiles.
//
// Author:       Fredrik Gravdal, 2007, NTNU
//*************************************************************
#include <stdio.h>
#include <stdlib.h>

int main(int numArgs, char* argc[]){

        printf("Sammenligner %s og %s \n",argc[1], argc[2]);

        int size1, size2, i, feil;
        unsigned int a,b;


        FILE * inFile1;
        FILE * inFile2;

        inFile1 = fopen (argc[1] , "rb");
        inFile2 = fopen (argc[2] , "rb");

        if (inFile1 == NULL){printf("Error opening %s\n", argc[1]); return 0;}
        if (inFile2 == NULL){printf("Error opening %s\n", argc[2]); return 0;}

        //finds number of elements in each file
        fseek (inFile1, 0, SEEK_END);
        size1=ftell (inFile1);
        rewind (inFile1);

        fseek (inFile2, 0, SEEK_END);
        size2=ftell (inFile2);
        rewind (inFile2);

        feil =0;

        if (size1 != size2){
                printf("Ulik størrelse\n");
                printf("%s = %d\n", argc[1], size1);
                printf("%s = %d\n", argc[2], size2);
                return 0;
        }
        else{
                for(i = 0; i < size1; i++){
                        a = fgetc (inFile1);
                        b = fgetc (inFile2);
                        if(a != b){
                                printf("at addr = %x\n",i);
```

```
                                printf("%s: %x\n",argc[1],a);
                                printf("%s: %x\n",argc[2],b);
                                //return 0;
                                feil++;
                                if(feil == 10){
                                        i =size1;
                                }
                        }
                }

        }
        if(feil == 0){
                printf("%s og %s er identiske\n",argc[1],argc[2]);
        }
        else{
                printf("Det er %d forskjeller\n",feil);
        }
        fclose (inFile1);
        fclose (inFile2);
        return 0;
}
```