NTNU
Det skapende universitet

Rekonfigurerbar maskinvare som applikasjonsakselerator ved søk i DNA

Per Andreas Gulbrandsen

Master i informatikk
Oppgaven levert: Juni 2007
Hovedveileder: Morten Hartmann, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Sammendrag

Rekonfigurerbar maskinvare er en teknologi som kan få potensielt stor innvirkning på mange fagfelt, deriblant bioinformatikk. Ved bruk av FPGA-teknologi, eller andre former for rekonfigurerbar maskinvare, kan man løse store, beregningsintensive oppgaver uten nødvendigvis å ha tilgang til svært kostbare dataanlegg. Men for å få en slik ytelse stilles det krav til algoritmen og implementasjonen. FPGA-teknologien har store fordeler, som for eksempel parallellisering av oppgaver, men også store svakheter, hvor to som kan nevner er båndbredde ut av brikken og at konstruksjon av spesialtilpasset maskinvare kan være svært komplisert. Dette er noe som kommer i fokus i oppgaven.

Oppgaven tar for seg utvikling av en modul i et større system. Denne modulen bruker post-prosessering av resultater til å velge ut mer relevant data for presentasjon. I tillegg søkes også en ytelsesøkning gjennom å forhindre overdreven bruk av begrenset båndbredde og problemer forbundet med kommunikasjon mellom FPGA og CPU. Utviklingen av denne modulen, i to varianter, skal implementeres som en fungerende modul i et system for søk i DNA. En ytelsesøkning vil bli tydelig, og bakgrunnen for denne blir forklart.

Oppgaven ser på muligheten for å utnytte parallellisering ved en flerkjerne-variant av systemet, og eventuelle forbedringer som kan gjøres i enkjerne-implementasjonen for best å utnytte tilgjengelige ressurser på FPGA-brikken, da utredningen tar utgangspunkt i at flerkjerne-implementasjonen blir tuftet på enkjernevarianten. En eksisterende modul blir implementert på nytt for å øke arealeffektiviteten til systemet, og en rekke forslag til forbedringer av andre moduler vil også bli gitt. Tilslutt vil en diskusjon rundt oppbygningen av en slik flerkjerne-variant bli gitt, og også en diskusjon rundt arealeffektivitet og ytelse. Fordeler og ulemper ved denne fremgangsmåten i forhold til en implementasjon som parallelliserer over flere noder vil også bli gitt.

Oppgaven dokumenterer også videreutviklingen av rammeverket rundt systemet for å gjøre dette klart for bruk. I dette inngår overføring av data til og fra systemet og tilpasning til eventuelle problemer forbundet med dette. Utfordringer er å takle store datamengder, konvertere data til systemets format og håndtere utdata fra systemet. Hensikten er å gjøre systemet fullt fungerende. Begrensninger i rammeverket som påvirker ytelse vil bli tydelige, og disse blir forklart. I tillegg vil resultater fra kjøringer med DNA-strenger og vektingsmatriser vises og tendenser i disse vil bli forklart.

Forord

Denne oppgaven er skrevet som hovedavhandlig under mitt mastergrads-studie ved NTNU, Trondheim. Veileder har vært Morten Hartmann ved Institutt for Datateknikk og Informasjonsvitenskap, avdeling for komplekse datasystemer. Oppgaven er skrevet på norsk, og i appendiks A finnes en liste over ord og forkortelser som er brukt.

Kreditering

Jeg vil gjerne takke min veileder Morten Hartmann for faglige råd, samt min medstudent Gaute Heimstad for faglige diskusjoner, hjelp og støtte. Også Lars Krutådal, tidligere ved instituttet, og Geir Kjetil Sandve ved IDI har vært behjelpelig i mitt arbeid.

Per Andreas Gulbrandsen
7. juni 2007

Innhold

Sammendrag **i**

Forord **iii**

Figurer **ix**

Tabeller **xi**

Algoritmer **xiii**

I Innledning **1**

1 Innledning **3**
1.1 Innledning . . . 3
1.2 Oppgaven . . . 4
1.3 Motivasjon . . . 4
1.4 Problemer og utfordringer . . . 5
1.4.1 FPWM-systemets tilstand ved prosjektstart . . . 5
1.4.2 Utfordringer av teknisk art . . . 6
1.5 Rapportens struktur . . . 6

2 Bakgrunn **9**
2.1 FPGA . . . 9
2.1.1 Introduksjon . . . 9
2.1.2 Teknisk . . . 9
2.1.3 Block SelectRAM+ . . . 11
2.1.4 FPGA som applikasjons-akselerator . . . 11
2.2 Cray XD1 . . . 12
2.2.1 Komponenter . . . 12
2.3 PWM . . . 15

3 Tidligere arbeider **17**
3.1 FPWM . . . 17
3.1.1 Presentasjon av FPWM-prosjektet . . . 17
3.1.2 Systemets oppbygging . . . 17

3.1.3 Kort om utvidelser . . . 19
3.2 Applikasjonsspesifikke brikker for akselerering av spesielle algoritmer . . 19
3.2.1 Presentasjon . . . 19
3.2.2 Eksperimenter og resultater . . . 19
3.3 Kommersielle modeller . . . 20
3.3.1 Progeniq BioBoost . . . 20
3.3.2 Mitrionics . . . 21
3.3.3 Impulse C . . . 21

II Løsning **23**

4 Utredning rundt mulige utvidelser **25**
4.1 Mulig implementasjon av filter . . . 25
4.1.1 Terskelfilter . . . 26
4.1.2 Summasjonsfilter . . . 27
4.2 Flerkjerneimplementasjon . . . 28
4.2.1 Forberedende arbeid med enkjernesystemet . . . 28
4.2.2 Endringer fra enkjernesystem til flerkjernesystem . . . 30
4.3 Utvidelse av grensesnitt . . . 34
4.3.1 Valg av filter og setting av parameter . . . 34
4.3.2 Overføring av DNA og vektingsmatrise . . . 34
4.3.3 Overføring av resultater . . . 35
4.3.4 Behandling av data . . . 35

5 Implementerte løsninger **37**
5.1 Filterimplementasjon . . . 37
5.1.1 Implementasjon av terskelfilter . . . 37
5.1.2 Implementasjon av summasjonsfilter . . . 38
5.2 Implementasjon av flerkjerneløsning . . . 39
5.2.1 Writer-modulen . . . 39
5.3 Implementasjon av rammeverk . . . 40
5.3.1 Valg av filter og overføring av parameter . . . 40
5.3.2 Overføring av DNA og vektingsmatrise . . . 41
5.3.3 Overføring av resultater . . . 42
5.3.4 Behandling av data . . . 42
5.3.5 Kort brukerveiledning . . . 43

III Analyse **45**

6 Eksperimenter og resultater **47**
6.1 Eksperimenter med enkjerneløsning . . . 47
6.1.1 Eksperimenter med terskelfilter . . . 47
6.1.2 Eksperimenter med summasjonsfilter . . . 49
6.2 Eksperimenter med flerkjerneløsning . . . 51
6.2.1 Forberedelser til flerkjernesystem . . . 51

6.3 Eksperimenter med rammeverk 52

7 Diskusjon **55**
7.1 Filterimplementasjon 55
7.1.1 Terskelfilter 55
7.1.2 Summasjonsfilter 57
7.2 Flerkjerneimplementasjon 59
7.3 Rammeverk 60

8 Konklusjon og videre arbeid **63**
8.1 Konklusjon 63
8.1.1 Filterimplementasjon 63
8.1.2 Flerkjerneimplementasjon 64
8.1.3 Rammeverk 65
8.1.4 Prosjektets verdi 65
8.2 Videre arbeid 66
8.2.1 Gjenstående arbeid 66

IV Appendiks **69**

A Ordliste og ordforklaringer **71**

B Kildekode **73**
B.1 Kildekode for terskelfilter 73
B.1.1 Kildekode for terskelfilter 73
B.1.2 Kildekode for testbenk til terskelfilter 74
B.2 Kildekode for summasjonsfilter 76
B.2.1 Kildekode for summasjonsfilter 76
B.2.2 Kildekode for testbenk til summasjonsfilter 76
B.3 Kildekode for BRAM-basert writermodul 79
B.3.1 Kildekode for writermodul 79
B.3.2 Kildekode for testbenk til writermodul 82
B.3.3 Kildekode for testbenk til system-test 86
B.4 Kildekode for `user_app.vhd` 90
B.5 Kildekode for c-rammeverk 98

C Utskrift fra kjøring **105**
C.1 Utskrift fra kjøring med terskelfilter 105
C.2 Utskrift fra kjøring med summasjonsfilter 106
C.3 Utskrift fra kjøring med rammeverk 108

D Veiledning til FPWM-systemet **111**
D.1 Kort veiledning til FPWM-systemet 111

Bibliografi **113**

Figurer

1.1 Logisk oppbygging av FPWM-systemet 4

2.1 Forenklet oversikt over oppbyggingen av en FPGA fra [1] 10
2.2 Konseptuell oversikt av en CLBs oppbygging fra [2] 11
2.3 Skisse av BRAM-blokk fra [3] 12
2.4 Cray XD1 chassis hentet fra [4] 13
2.5 Konfigurasjon av FPGA hentet fra [4] 14
2.6 To steg i en PWM-algoritme hentet fra [5] 15

3.1 Sammensetningen av moduler i FPWM-systemet 18

4.1 Informasjon fra syntetiseringsverktøy 28
4.2 Informasjon fra syntetiseringsverktøy, forkortet 28
4.3 Konseptuell skisse av forskjeller ved flerkjerneimplementasjon 31

5.1 Konseptuell tegning av en ripple-carry adder. 38
5.2 Skjematisk tegning av terskelfilter etter syntetisering 44

6.1 Utsnitt fra simulering av terskelfilter. 48
6.2 Utsnitt fra simulering av summasjonsfilter. 50
6.3 Utsnitt fra simulering av terskelfilter. 53

Tabeller

2.1 Konfigurasjoner av BRAM 11

4.1 Tidsforbruk ved syntetisering 34

5.1 Utskrift fra synteseverktøy 40
5.2 Utskrift fra synteseverktøy 40
5.3 Vektingsmatrise ved oppstart 41
5.4 Vektingsmatrise etter prosessering 41

7.1 Kjøringer med terskelfilter 56
7.2 Vektingsmatrise til terskelfilterforsøk 57
7.3 Kjøringer med summasjonsfilter 58
7.4 Vektingsmatrise til terskelfilterforsøk 58
7.5 Kjøringer med rammeverk 61
7.6 Tidsforbruk med rammeverk 61

Algoritmer

2.1 Eksempel på PWM-implementasjon 16
4.1 Implementasjon 1, terskelfilter 26
4.2 Implementasjon 2, terskelfilter 27
4.3 Implementasjon 1, summasjonsfilter 27
4.4 Implementasjon 2, summasjonsfilter 27
4.5 Implementasjon med separate filter for N-beste-filter 32
4.6 Implementasjon av et delt summasjonsfilter 33

Del I

Innledning

Kapittel 1

Innledning

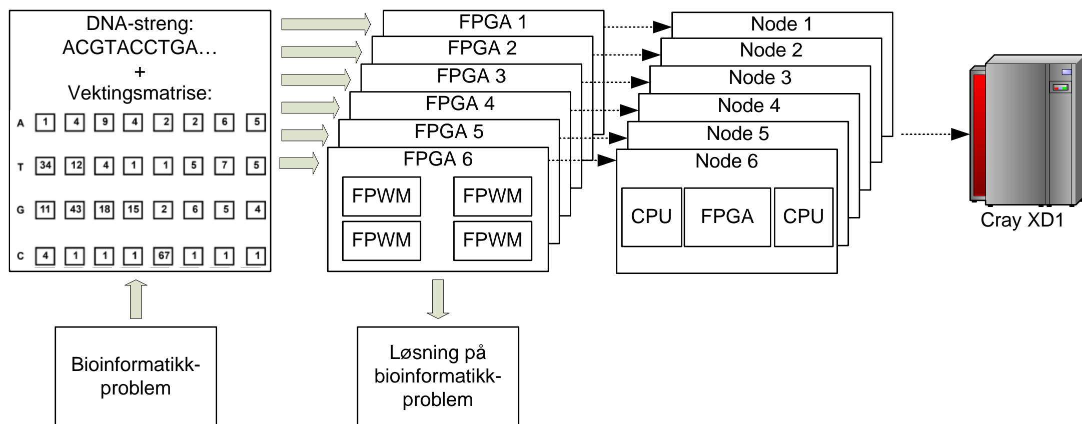
1.1 Innledning

Som et samarbeidsprosjekt mellom Institutt for Datateknikk og Informasjonsvitenskap (IDI) og Det Medisinske Fakultet (DMF) ble det i løpet av sommeren 2005 kjøpt inn en Cray XD1 [6]. Dette er en spesiell maskin med seks noder, hver node bestående av to ordinære CPUer og en Field Programmable Gate Array (FPGA) [7]. Det endelige målet med prosjektet er å gjøre FPGA-akselerert prosessering tilgjengelig for DMF, som kan bruke FPGAens potensiale innen bioinformatikk.

Cray XD1 er en svært avansert datamaskin, og for å kunne benytte den må man ha svært god kjennskap til en rekke verktøy. Eksempler på disse er Message Passing Interface (for kommunikasjon mellom noder), Hardware Description Language (for utvikling av FPGA) og programvareutvikling, fortrinnsvis i C (for programmering av kontrollprogram). Det er derfor urealistisk at noen uten god kjennskap til nevnte teknologier skal kunne benytte maskinen [8]. IDIs rolle i dette prosjektet er å utvikle moduler som kan brukes av DMF til å løse problemer innen bioinformatikk, uten å måtte besitte kunnskap om disse teknologiene. I tillegg representerer denne maskinen også et forskningsobjekt for IDI, som kan utrede mange problemstillinger ved hjelp av denne maskinvaren.

En algoritme som kan utnytte FPGAens potensiale er Posistion Weight Matrix-algoritmen (PWM) [5]. En FPGA-implementasjon av denne er utviklet ved instituttet, kalt FPGA PWM-Modul (FPWM). Denne modulen søker igjennom DNA-strenger etter motiv basert på en vektingsmatrise. Algoritmen er godt egnet til implementasjon i FPGA, med en liten og beregningsintensiv del som trenger minimalt med kommunikasjon.

Figur 1.1 viser et logisk oppsett av systemet.



Figur 1.1: Logisk oppbygging av FPWM-systemet

1.2 Oppgaven

Mønstergjenkjenning i bioinformatikk er et fagfelt i sterk vekst, og fagfeltet har et stort behov for søking i store datamengder. Ved NTNU er det utviklet en tidlig prototype på en skreddersydd FPGA-løsning for søking i genetisk data. Oppgaven består i å sette seg inn i denne løsningen og fortsette arbeidet ved å utvide løsningen på en eller flere måter, ved utredning av løsningsalternativer og eventuell implementering av disse. Noen mulige utvidelser er:

- Implementasjon og test av filterfunksjoner for reduksjon av returdatamengde
- Beskrive og eventuelt implementere utvidelser fra eksisterende kjerne til en flerkjerneløsning
- Utvide rammeverk til FPWM-systemet til lasting og prosessering av filer som inneholder DNA og PWM på systemet, i den grad tiden tillater det

1.3 Motivasjon

En FPGA er en svært annerledes type brikke enn den generelle CPUen. Derfor kan man høste betydelige fordeler av å bruke denne til applikasjonsakselerering i kjøring av spesielle algoritmer. En maskinvareutvikler kan bruke FPGAens rekonfigurerbarhet og skreddersy deler av algoritmen han/hun jobber med til å kjøre ekstremt kjapt på en FPGA. Dette gjelder i hovedsak beregningsintensive algoritmer, som gjerne krever lite eller ingen kommunikasjon i utregningen. FPGAer har også en stor fordel til i forhold til generelle CPUer. Utvikleren kan kjøre flere kjerner i parallell på FPGAen og dermed parallellisere utregningen sin. Da kan det oppnås en ytelse tilsvarende det enorme, svært kostbare dataanlegg kan gi [9].

PWM-algoritmen, en algoritme konstruert for å søke etter motiv i en DNA-streng, er svært beregningsintensiv, og har også iboende parallelle egenskaper [5]. Dette gjør den

egnet til implementering på FPGA i den hensikt å utnytte applikasjonsakseleringen som FPGA-teknologien tilbyr.

En svakhet ved FPWM-algoritmen i forhold til FPGAens egenskaper er at FPWM-algoritmen produserer ett resultat for hver posisjon i DNA-strengen. Dette vil altså tilsvare en enorm datamengde som skal skrives til CPU. Derfor kan FPGA-kjøringens ytelse økes ved hjelp av postprosessering av resultatene. Ved å implementere et eller flere filter i systemet kan man bedre velge ut hvilke resultater som skal skrives til CPU, og på den måten forbedre utnyttelse av den noe begrensede båndbredden mellom FPGA og CPU. Dette vil igjen påvirke ytelsen til hele systemet, ettersom systemet ikke behøver å vente på at resultater skal skrives tilbake til CPU. Ytelsesøkningens grad avhenger av avhengig av filterimplementasjon og eventuelle parametre og inndata til filteret.

De iboende parallelle egenskapene i FPWM-algoritmen kan det også høstes ytelsesøkning fra. Ved å implementere en flerkjerneløsning av systemet kan flere kjerner arbeide med den samme DNA-strengen og det samme motivet i parallell, og dermed potensielt øke ytelsen med en faktor lik antallet instansierte kjerner. I en flerkjerneløsning er en god implementasjon av mekanismer for postprosessering viktigere enn i en enkjerneimplementasjon, fordi det er flere kjerner som skal dele den samme båndbredden.

En annen mulig fremgangsmåte for å utnytte de iboende parallelle egenskapene i PWM-algoritmen er å bruke flere FPGAer samtidig, og fordele problemet over flere noder. Cray XD1 tilbyr i alt 6 noder med tilsammen 6 FPGAer, hvilket vil si at problemet kan fordeles over seks kjerner. Å kombinere en flernode-løsning med en flerkjerne-løsning vil gi den høyeste ytelsen. Hvis en flerkjerneløsning kan gi 8 kjerner i parallell på en FPGA, kan en kombinasjon med en flernodeløsning gi 48 kjerner i parallell.

1.4 Problemer og utfordringer

1.4.1 FPWM-systemets tilstand ved prosjektstart

Ved oppstart av dette prosjektet var allerede en implementasjon av FPWM-systemet underveis. Denne blir presentert i 3.1. Dette systemet skulle være et fungerende grunnsystem som skulle utvides med filter og mulighetene rundt en flerkjerneløsning skulle også utredes. Dessverre hadde det oppstått problemer underveis i utviklingen av FPWM-systemet, selv om hvilke er ukjent, og systemet var ikke fungerende. Problemet var at noen resultater var duplisert og noen resultater var forsvunnet når eksekveringen var avsluttet. Arbeidet med dette systemet ble, av den opprinnelige forfatter, avsluttet, men jeg valgte å fortsette mitt arbeide med systemet på tross av manglene. Dette fordi jeg mente, og fortsatt mener, at manglende eller dupliserte resultater ikke ville påvirke implementasjonen av filtermoduler i nevneverdig grad.

Et større problem med FPWM-systemet ved prosjektoppstart var svært mangelfull dokumentasjon. I [10], som er en side som beskriver FPWM-prosjektet på en wiki-side for forskjellige prosjekter ved NTNU som handler om motiv-gjenkjenning og forskning innen dette, skriver forfatteren at dokumentasjon av FPWM-systemet hovedsaklig er

representert som kommentarer i koden. Disse kommentarene er i stor grad fraværende, og den dokumentasjon som forelå ved prosjektstart ble aldri utbedret. Det har påvirket mitt arbeide i betydelig grad at jeg nesten utelukkende har måttet forholde meg til kode for å forstå systemet.

1.4.2 Utfordringer av teknisk art

Prosjektet var i begynnelsen plaget med flere problemer med syntetisering av systemet. Etterhvert viste det seg at dette var som et resultat av inkompabilitet mellom Xilinx ISE Foundation [11] utviklingsverktøy og synteseverktøyene installert på Cray XD1. Også synteseverktøyene var levert av Xilinx. Grunnen til denne inkompabiliteten er uvisst, og det er også min mening at det finnes en løsning på problemene den skapte. Men istedet for å bruke tid på å løse disse problemene fortsatte arbeidet uten Xilinx ISE Foundation, noe som var en tilstrekkelig løsning. Xilinx ISE Foundation ble imidlertidig tatt i bruk igjen i forbindelse med simulering, men det skapte da ingen problemer ettersom syntese på Cray XD1 ikke var involvert.

Da simulering med implementerte løsninger skulle starte oppstod det problemer med å få tilgang til programmet Modelsim, laget av Modeltech [12]. Dette programmet brukes for simuleringer av vhdl-konstruksjoner, og er lisensiert av IDI. Imidlertidig fikk jeg store problemer med lisens til programmet. IDI har en lisenstjener programmet kobler seg opp imot, men både på egen maskin og på NTNU sine Windows-maskiner skapte dette problemer, da Modelsim ikke godtok validiteten til lisensen. Dette problemet forsinket utviklingsprosessen endel, men løste seg tilslutt da jeg fikk tilgang til Linux-maskiner ved IDI med Modelsim installert. Denne versjonen godtok lisensene, og lot meg komme igang med simulering.

En rekke mindre problemer har også oppstått i løpet av prosjektets gang, men dokumentene ”Introduksjon til bruk av FPGA i vitenskapelige beregninger” av Lars Krutådal [1], ”Synthesis and simulation design guide” fra Xilinx [13], ”Cray XD1 Programming” fra Cray [14] og ”Cray XD1 FPGA Development” også fra Cray [15] har vært til stor hjelp.

1.5 Rapportens struktur

Kapittel 2, bakgrunn, vil gi endel bakgrunnsstoff, både om FPGA som teknologi, om Cray XD1 og iboende teknologier, og også om PWM-algoritmen.

Kapittel 3, tidligere arbeider, vil presentere tidligere arbeider innen samme felt. Først vil det bli gitt en grundigere gjennomgang av oppbyggingen i en eksisterende implementasjon av FPWM-systemet. Deretter vil et prosjekt om applikasjonsspesifikke brikker for applikasjonsakselerering av spesielle algoritmer bli presentert. Sist vil noen kommersielle løsninger presenteres for leseren.

Kapittel 4, utredning rundt mulige utvidelser, inneholder utredning rundt mulige implementasjoner av filter til FPWM-systemet, samt en diskusjon rundt svakheter og styrker

til disse. Kapittelet inneholder også diskusjon rundt problemer ved en flerkjerneimplementasjon av FPWM-systemet, og forslag til implementasjon. Tilslutt kommer en utredning rundt problemer/utfordringer rundt en utvidelse av grensesnittet.

Kapittel 5, implementerte løsninger, presenterer de valgte implementasjonene med en begrunnelse for hvorfor løsningen ble valgt. Endringer gjort i implementasjonen underveis blir begrunnet ved hjelp av oppdagelser gjort underveis i utviklingsprosessen.

Kapittel 6, eksperimenter og resultater, presenterer simuleringer av de implementerte moduler samt eksperimenter gjort med systemet etter implementasjon. Resultater fra eksperimentene presenteres også.

Kapittel 7, diskusjon, inneholder diskusjon rundt både resultater av kjøring og tilstanden til det eksisterende system etter prosjektavslutning. Fremgangsmåter for å implementere et flerkjerne-system blir også diskutert.

I kapittel 8, konklusjon og videre arbeid, konkluderes arbeidet som er gjort med filterimplementasjon og det resulterende systemet, og konklusjoner rundt en flerkjerneløsning trekkes. Arbeidet med videreutvikling av rammeverket konkluderes også. Videre arbeid blir foreslått.

I appendiks A finnes en liste over ord oversatt fra engelsk samt akronymer brukt i teksten.

I appendiks B finnes kildekode utviklet i løpet av prosjektet.

I appendiks C finnes utskrift fra kjøringer av systemet etter prosjektslutt.

I appendiks D finnes en veiledning til FPWM-systemet for utviklere.

Kapittel 2

Bakgrunn

I dette kapittelet vil teknologien i systemet bli forklart. Først blir FPGA presentert i 2.1, deretter blir selve Cray-maskinen beskrevet i 2.2. Tilslutt vil Position Weigth Matrices-algoritmen (PWM) bli forklart i 2.3. I tillegg til teknisk beskrivelse vil også noe historie og bruksområder bli nevnt.

2.1 FPGA

Field Programmable Gate Array er en teknologi representert som en brikke. Brukeren kan konfigurere brikken, slik at den kan endre oppførsel. Dette kan gjøres både dynamisk (under kjøring) og statisk (kun under oppstart). Brukeren kan konfigurere brikken ved hjelp av å sette konfigurasjonsbit i brikken og endre koblingen mellom brikkens forskjellige interne enheter. Ved egnede oppgaver kan ytelsen økes drastisk ved bruk av FPGA-teknologi i forhold til konvensjonelle CPUer.

2.1.1 Introduksjon

I 1984 startet Ross Freeman, Bernie Vonderschmitt, og Jim Barnett firmaet Xilinx. I 1985 slapp dette firmaet sitt første produkt, verdens første FPGA. Denne teknologien ble oppfunnet av Ross Freeman, etter at han oppdaget et ønske i industrien etter brikker man kunne programmere og utvikle selv [16].

I de to tiårene som har fulgt har FPGAens popularitet vokst, og stadig nye bruksområder oppdages innen forskjellige fagfelt.

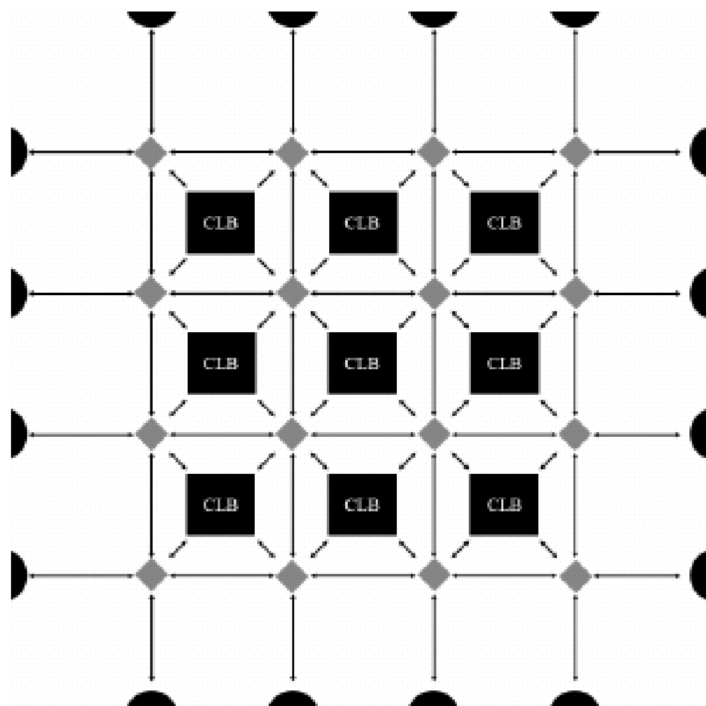
2.1.2 Teknisk

FPGA-ens sentrale enhet kalles for Configurable Logic Block (CLB), og er som navnet foreslår konfigurerbare logikk-blokker. Hver av disse er koblet til programmerbare koblingsmatriser (switch matrix), som også programmeres for å etablere kommunikasjons-

linjer mellom CLBene. Dette styres selvfølgelig av designet som lastes inn i FPGAen. I tillegg finnes også en type blokker som heter Input/Output Blocks (IOB) som tar seg av kommunikasjon inn og ut av FPGAen. Disse IOBene er også koblet til switchmatrisene. FPGAer kan også ha flere mer komplekse enheter, avhengig av type og modell. For eksempel har FPGAen i Craymaskinen to PowerPC 405 RISC-prosessorer innebygd på brikken.

Oppbygningen av FPGAer varierer selvfølgelig fra merke til merke, og modell til modell, og derfor må en slik generell presentasjon nødvendigvis bli grovkornet.

Figur 2.1 viser oppsettet av CLB, IOB og koblingsmatriser svært forenklet.

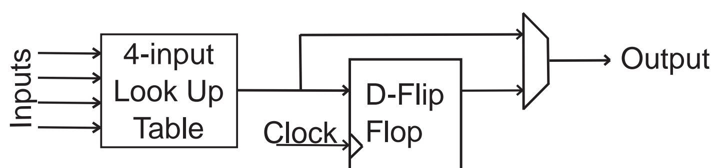


Figur 2.1: Forenklet oversikt over oppbyggingen av en FPGA fra [1]

CLB

De konfigurerbare logikk-blokkene kan benyttes på flere forskjellige måter. Den vanligste er å bruke en oppslagstabell (Look-Up Table), en slags sannhetstabell som er konfigurert av bruker. Denne kan enkeltstående brukes til å instansiere enkle logiske operasjoner, eller den kan kombineres med flere andre logikk-blokker og bygge mer komplekse blokker. Blokkene kan også brukes som styresignaler for andre blokker, eller de kan brukes som styresignaler for større og mer komplekse enheter, for eksempel en Aritmetisk-Logisk Enhet (ALU). En logisk blokk inneholder også en D-FlipFlop, hvis man krever noen form for tilstandslagring.

FPGAene i Cray XD1 inneholder 53136 konfigurerbare logikkblokker [17]. En konseptuell skisse av oppbyggingen i en CLB vises i figur 2.2.



Figur 2.2: Konseptuell oversikt av en CLBs oppbygging fra [2]

2.1.3 Block SelectRAM+

I FPGAen som er montert i Cray XD1 er det en rekke minnekomponenter. Disse heter Block SelectRAM+ (BRAM), og alt i alt er det 232 BRAMs spredt utover brikken. Hver av disse kan holde data i en kombinasjon av databredde og adressebredde fra 1 bit databuss og 11 bit adressebuss til 16 bit databuss og 8 bit adressebuss, som det tabell 2.1 viser.

Bredde	Dybde	ADDR-buss	DATA-buss
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1:0>
4	1024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

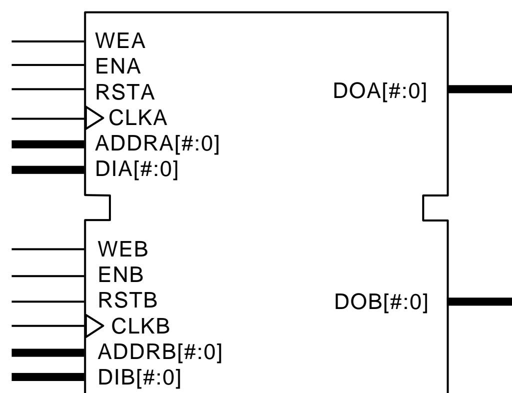
Tabell 2.1: Forskjellige mulige konfigurasjoner av forholdet adressebuss/databuss i BRAM.

Dette minnet er svært nyttig å bruke til lagring av større datamengder da det kan spare stor bruk av CLBer. Disse BRAM-blokkene er dessuten en fysisk ressurs, og det er svært hensiktsmessig å bruke disse fremfor CLBer til lagring av data.

En BRAM-blokk har to porter, og kan derfor skrive/lese to verdier hver sykel. Ved lesing dyttes adresse inn på adressebussen til den aktuelle porten mens `WEN` (write enable) holdes inaktivt. Resultatet kommer da på utgangsporten som er tilknyttet porten en klokkeflanke senere. Ved skriving må `WEN` holdes aktivt, samtidig som adresse og data settes på sine respektive porter. Begge porter opererer på samme data, og det er derfor mulig å for eksempel bruke en port til skriving og en port til lesing simultant. 2.3 viser en skisse av en BRAM-blokk, med sine to porter og tilhørende signaler.

2.1.4 FPGA som applikasjons-akselerator

En av FPGAens styrker er muligheten til å parallellisere operasjoner. Hvis en algoritme, eller en del av en algoritme, er beregningsintensiv og med lite eller ingen kommunikasjon, kan denne akselereres i en FPGA i forhold til en vanlig CPU, selv om CPUen har mye høyere klokkefrekvens. Dette på grunn av CPUens generelle natur. En FPGA formes av



Figur 2.3: Skisse av BRAM-blokk fra [3]

utvikleren til å utføre akkurat de nødvendige operasjoner, og sparer derfor på arbeidet. Men hvis algoritmen i tillegg har iboende parallelle egenskaper, kan den akselereres ytterligere ved å instansiere flere kjerner på FPGAen, og utføre arbeidet i parallell. FPWM-modulen med PWM-algoritmen er en slik algoritme. Den trenger ikke å kommunisere under eksekvering, den er beregningsintensiv, og hvert steg er helt uavhengig både av det forrige og det neste. En flerkjerneimplementasjon av FPWM-systemet har en speedup på 480x i forhold til samme algoritme på en Pentium M CPU [5], og dette viser det enorme potensialet som ligger i FPGAer som applikasjonsakseleratorer.

2.2 Cray XD1

Dette systemet fra Cray er satt opp som et relativt vanlig SMP-oppsett (Symmetric Multi Processing) utvidet med FPGAer. Kommunikasjonen mellom de forskjellige delene i maskinen går over et proprietært system, RapidArray Interconnection. Systemet er konfigurert som et chassis som inneholder 6 noder, hver node med 2 AMD Opteron CPUer og en Xilinx Virtex-II Pro FPGA. Hver node er et selvstendig system med sitt eget GNU/Linux operativsystem. Nodene kommuniserer med hverandre over RapidArray Interconnect.

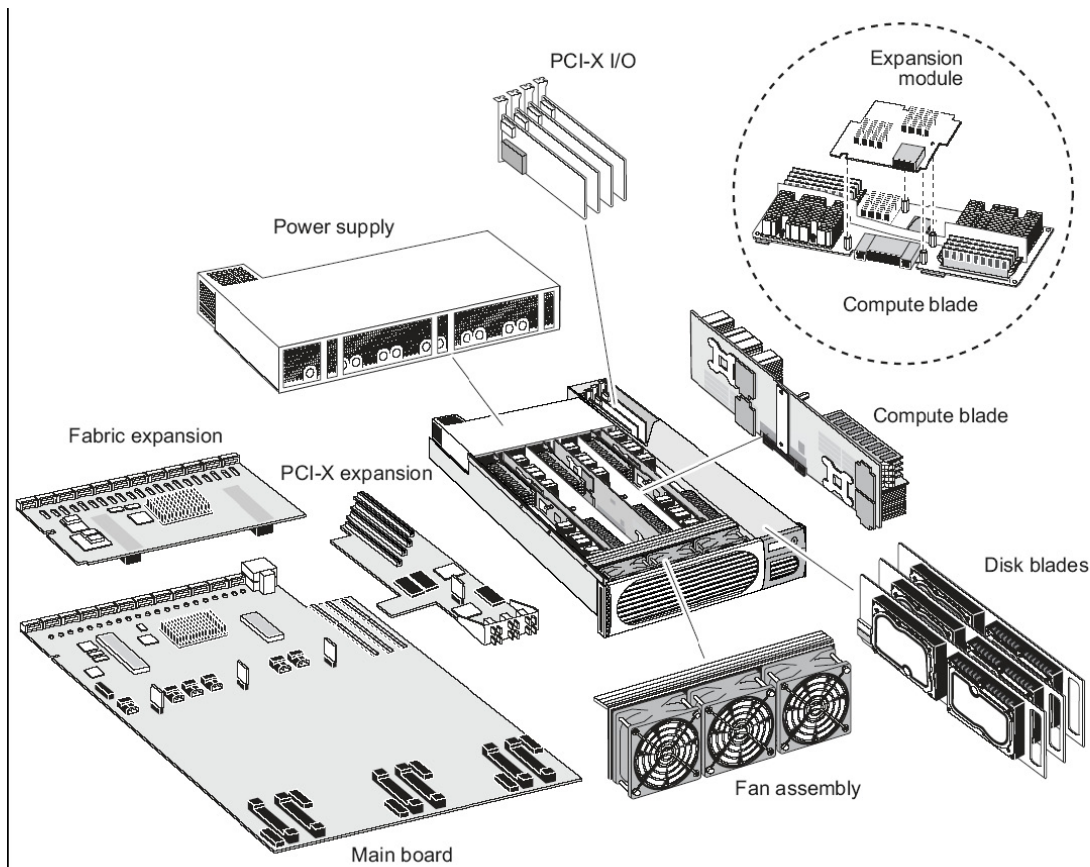
Figur 2.4 viser hovedkomponentene i et Cray XD1 chassis. Disse inkluderer et chassis, seks noder (compute blade) med to CPUer og en FPGA, lagringskapasitet (disk blades), strømforsyning (power supply) og kjølesystemer (fan assembly).

Maskinen ved NTNU heter Musculus, og har adresse musculus.hpc.ntnu.no.

2.2.1 Komponenter

RapidArray Interconnect

Grunnkonfigurasjonen av RapidArray Interconnect er en switch med 12 interne porter, brukt til å koble sammen de 6 nodene, altså 2 porter per node. Systemet har også



Figur 2.4: Cray XD1 chassis hentet fra [4]

12 eksterne porter, brukt til å koble sammen flere chassis. Antallet porter kan dobles, både interne og eksterne, ved installasjon av et ekspansjonskort, som tilbyr en ekstra switch. I grunnkonfigurasjonen kan hver node kommunisere med en kapasitet på opptil 2GB/s hver vei. Med ekspansjonmodulen dobles dette, altså en samlet båndbredde i hvert chassis på 96GB/s.

RapidArray Interconnect styres av egne prosessorer (RapidArray Processor) som kontrollerer og koordinerer trafikk på dette kommunikasjonsgrensesnittet.

I tillegg kobler også RapidArray Interconnect sammen FPGAen med SMP-noden. Dette deles i to deler, Fabric Request Interface, og User Request Interface. Førstnevnte er ikke så interessant i denne sammenhengen, og presentasjonen vil begrenses til at Fabric Request Interface brukes i transaksjoner mellom CPU og FPGA som er initiert av CPU. For ytterligere presentasjon henvises lesere til [1].

User Request Interface behandler kommunikasjon fra FPGA til CPU på FPGAens initiativ. Den fysiske bussen som brukes er 64bit bred og har en hastighet på 199MHz. Denne bussen er i stand til å overføre 64 bit per sykel, og for hver overføring trenger den 1 sykel nedkjøling. Den mest effektive måten å overføre data på er ved hjelp av buntskriving (burstwrite), altså skriving av 8x64bit. Dette tar åtte sykler, men medfører

bare en sykel nedkjøling, slik at det totalt blir 9 sykler. På denne måten kan skrivingen effektiviseres.

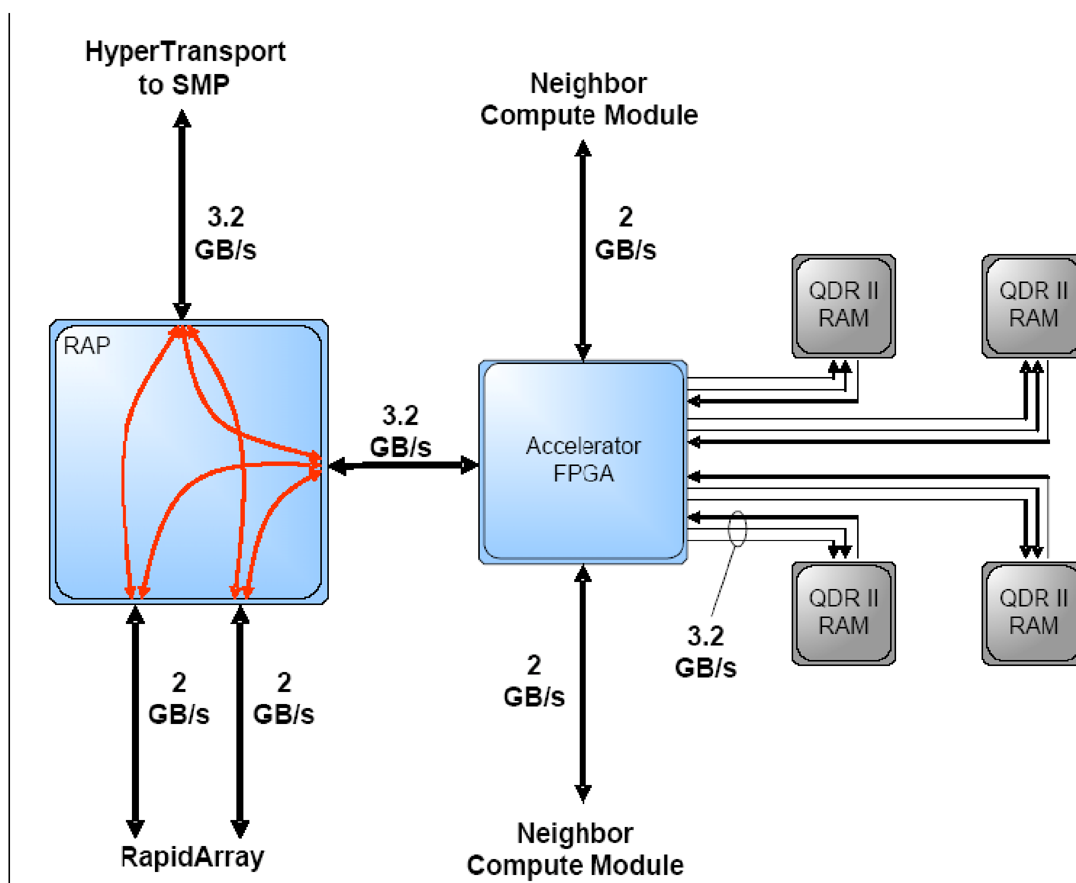
Xilinx Virtex-II Pro FPGA

I hver node sitter det en Virtex-II Pro FPGA. Denne går på hastigheter mellom 130 MHz og 200 MHz, den nedre grensen settes av minnemodulene som er koblet til. Hvis man ikke benytter disse kan man kjøre FPGAen på hastigheter ned til 63 MHz.

FPGAen er knyttet til fire separate minneblokker (QDR II SRAM), hver på 1M 36bit ord, hvorav 4 er paritetsbit. Hver minneblokk har uavhengige lese- og skrive-busser, som med sine 36 bit bredde kan gi en overføringshastighet på opptil 1.6GB/s hver vei. Gjennom bruk av grensesnittet Cray XD1 QDR II SRAM Core får brukere tilgang til minnet fra et FPGA-design.

Modellen som sitter i Cray XD1 er en XC2VP50.

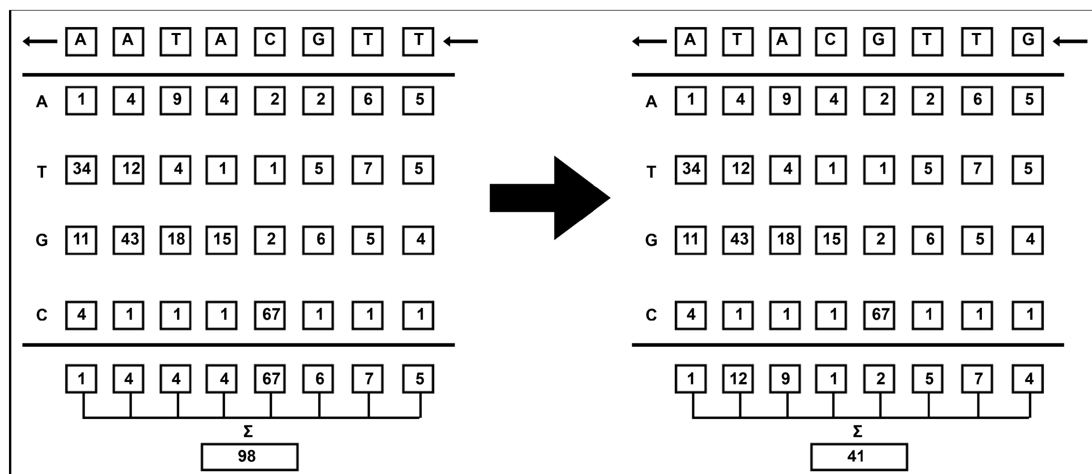
Figur 2.5 viser oppsett og kobling av en FPGA, RAP (Rapid Array Processor) og FPGAens minneblokker i en node.



Figur 2.5: Konfigurasjon av FPGA hentet fra [4]

2.3 PWM

Position Weight Matrices blir ofte tilskrevet Roger Staden, etter at han introduserte konseptet i hans artikkel [18]. Disse er grunnkomponenter i mange algoritmer for søking i nukleinsyre [19] og mye brukt innen bioinformatikk. Algoritmen består av et motiv, med en tilhørende vektingsmatrise. Denne matrisen har høyde tilsvarende lengden på alfabetet i tekststrengen, og lengde tilsvarende lengden på motivet. Et eksempel på dette er presentert i figur 2.6.



Figur 2.6: To steg i en PWM-algoritme hentet fra [5]

Denne matrisen flytter man langs tekststrengen og regner ut en verdi for hver posisjon i strengen. Verdien blir beregnet ut ifra vektene for hver bokstav i vektingsmatrisen, og hvis første synlige tegn i strengen er en A blir verdien hentet fra koordinater A,1 i matrisen. Summen av disse tilsvarer verdien for første synlige posisjon i strengen. Figur 2.6 viser to steg, hvor matrisen illustrerer et vindu man ser strengen gjennom. I steg to i figuren ser man at vinduet er flyttet ett hakk mot høyre i forhold til strengen.

I algoritme 2.1 ser vi algoritmen beskrevet med pseudokode.

FPWM-modulen [5], som er grunnsteinen i systemet denne oppgaven omhandler, er en implementasjon av PWM-algoritmen på en FPGA. Her produseres det, som i PWM-algoritmen, en verdi mellom 1 og 0 for hver posisjon i strengen. Disse verdiene sendes så fortløpende til et filter, og det er utviklingen av dette filteret som er hovedfokus for oppgaven. Selvsagt kan det utvikles og brukes forskjellige typer filtre, alt etter hvilke verdier man søker etter i tekststrengen, og denne oppgaven tar for seg utviklingen av to slike filtre.

FPWM-modulen er ved prosjektstart ikke implementert slik at den kan utnytte FPGA-ens muligheter til det fulle. Denne oppgaven skal også se litt på muligheten for å øke ytelsen ytterligere. Lars Krutådal har i [5] sett litt på problemer og utfordringer i forhold til å videreutvikle FPWM-modulen til å bli en flerkjerneløsning, og denne oppgaven forsøker, så godt tiden strekker til, å videreføre dette arbeidet og også påbegynne implementeringen av dette.

```
N : integer (Lengden på motiv);
M : integer (Lengden på streng);
S : string (DNA-streng av lengde M);
pwm_tabell : tabell[4][N] (Tabell som inneholder vektingsmatrise);
teller : integer ← 0 (Teller brukt i utregning);
sum, i : integer (Variabler brukt i utregning);
while teller ≤ M−N do
   for i=0 to N do
      sum ← sum + finn verdi i pwm_tabell for S[teller+i];
   end
   teller++;
   lagre sum;
   sum ← 0;
end
```

Algoritme 2.1 : Eksempel på PWM-implementasjon

Kapittel 3

Tidligere arbeider

I dette kapittelet vil tidligere arbeider innen samme felt presenteres. Dette innbefatter Lars Krutådal sine tidligere arbeider med FPWM (3.1), Geir Kjetil Sandve m.fl. sitt arbeid med applikasjonsspesifikke brikker for bruk innen bioinformatikk (3.2) og også noen tilgjengelige kommersielle løsninger (3.3).

3.1 FPWM

3.1.1 Presentasjon av FPWM-prosjektet

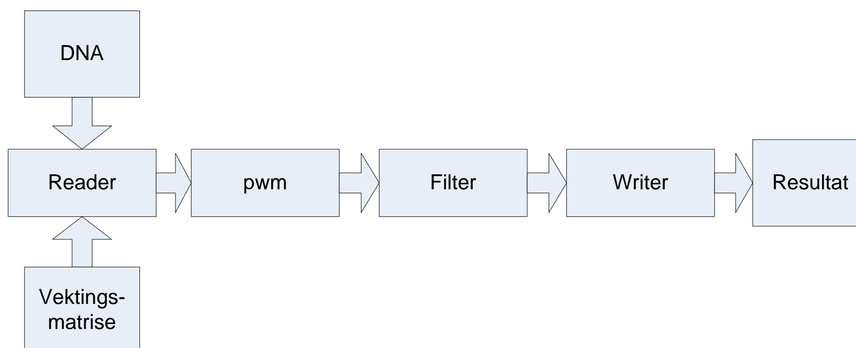
Lars Krutådal ved Institutt for Datateknikk og Informasjonsvitenskap har med avsluttende diplomoppgave i sitt sivilingeniørstudie skrevet om og implementert FPWM-algoritmen. Han gjorde en tidlig implementasjon av FPWM-systemet, riktignok med endel mangler, og han ble senere ansatt av instituttet for å ferdigstille systemet.

3.1.2 Systemets oppbygging

Systemet består av et overliggende rammeverk for kontroll, og fire underliggende moduler. Disse fire modulene er reader-, pwm-, filter- og writer-modulen, og en konseptuell skisse av disse finnes i 3.1.

user_app

Det overliggende rammeverket heter user_app. Her skjer initialisering av modulene, både ved oppstart og også ved `reset_n='0'`. I tillegg tar denne modulen seg av all kommunikasjon og data fra SMP-noden. Vel å merke initierer den ikke kommunikasjon, den bare behandler innkommende meldinger.



Figur 3.1: Sammensetningen av moduler i FPWM-systemet

reader

Lesing av DNA-strengen fra minne blir tatt hånd om av reader-modulen. Denne sender en ny verdi til pwm-modulen hver sykel.

pwm

Denne modulen utfører selve utregningen, beskrevet i 2.3. Den blir matet med substrenger fra reader-modulen, regner ut, og sender videre et resultat og en tilhørende index.

filter

Filteret prosesserer resultatene fra pwm-modulen for å begrense mengden, og hente ut nyttigere resultater. Antallet resultater fra en slik utregning kan være overveldende, og avhengig av filterets utforming kan man hente ut et subsett av disse. På denne måten begrenser man kommunikasjon mellom FPGA og CPU under eksekvering av FPWM-algoritmen. Dette er en ganske stor forbedring da båndbredde mellom FPGA og CPU er begrenset, og FPWM-systemet må gå inn i ventemodus hvis skriving til CPU ikke går fort nok, og resultatene dermed hoper seg opp.

writer

Denne modulen tar seg av all skriving til SMP-noden. Modulen instansierer to buffere med åtte elementer hver, slik at writermodulen ikke trenger å sette `stall` for hver gang den skal skrive. Den kan ta imot data samtidig som den skriver til SMP-noden. Men, modulen er ikke istand til å prosessere mer enn ett element per sykel i mer enn 16 sykler. Dette fordi User Request Interface trenger en sykel nedkjøling etter en skriveoperasjon. Derfor er også et filter nødvendig, for å begrense hyppigheten på skrivekall.

3.1.3 Kort om utvidelser

Lars Krutådal har i [5] i hovedsak sett for seg to mulige videreutviklinger av FPWM-systemet for å oppnå parallellisert eksekvering. Lokal parallellisering ved hjelp av en flerkjerneløsning er den ene retningen, og denne innebærer å kjøre flere versjoner av en FPWMkjerne på en fpga i parallell, hvor alle kjernene har samme inndata (DNA-streng og vektingsmatrise). Den andre løsningen [5] har sett for seg er en flernode-implementasjon, som CRAY XD1 tillater. Denne innebærer flere FPWM-kjerner distribuert over flere noder. En optimal løsning ville vært en kombinasjon, hvor man har lokal parallellisering på flere noder samtidig.

3.2 Applikasjonsspesifikke brikker for akselerering av spesielle algoritmer

3.2.1 Presentasjon

I [9] presenteres en abstrakt modul PAMM (Parallel Acceleration of Motif Matching) som bygger på en implementasjon av en PWM-algoritme ved navn MEME. PAMM-modulen ligner på FPWM-modulen beskrevet ovenfor, og regner ut verdier $v_{s,p}^{m}$ for hvert motiv m for hver posisjon p i hver streng s. PAMM-modulen kjøres på en brikke utviklet av Interagon [20] ved navn Pattern Matching Chip. PAMM-modulen tilbyr, også i likhet med denne oppgavens mål for FPWM-systemet, etterprosessering av resultatene. PAMM-modulen inngår som en del av et større system, MAMA (Massively parallel Acceleration of the Meme Algorithm).

3.2.2 Eksperimenter og resultater

I tillegg til å kjøre MAMA (med PAMM på PMC), testet gruppen også den sekvensielle MEME-algoritmen, samt en parallell implementasjon av denne, ParaMEME. Disse ble følgelig testet på forskjellig maskinvare:

- MAMA: 2.8 GHz Pentium4 PC med 1 GB minne og PMC tilknyttet via PCI
- MEME: 2.8 GHz Pentium4 PC med 1 GB minne
- ParaMEME: en klyngemaskin med 12 noder, hver node bestående av en 3.4 GHz Pentium4 PC med 1 GB minne

Testene gruppen utførte viste at MAMA hadde en lavere kjøretid enn de andre, og også viste en lavere økning i kjøretid ved økning i problemstørrelse enn de andre algoritmene. På det største datasettet de testet med var MAMA fire ganger kjappere enn ParaMEME, selv om den sistnevnte algoritmen kjørte på mye mer kostbar maskinvare.

Gruppen nevner også i konklusjonen at deres implementasjon kun viser at PAMM-modulen har potensiale som en applikasjonsakselerator, og at en FPGA-implementasjon

av denne vil være en naturlig fortsettelse. Denne implementasjonen vil være tilpasset praktisk bruk av det vitenskapelige samfunn, og kan gi enda større økninger i kjøretid.

3.3 Kommersielle modeller

Flere selskaper tilbyr kommersielle løsninger innen bioinformatikk, som enten er basert på FPGA-løsninger, eller med helt applikasjonsspesifikke brikker. Noen eksempler vil her bli presentert.

3.3.1 Progeniq BioBoost

Progeniq er et selskap som ble opprettet i Singapore 10. Mars 2005 [21]. Selskapet har sitt utspring fra Ngee Ann Polytechnic Digital Signal Processing (DSP) Technology Centre i Singapore. Deres målsetning er å selge dataplattformer for beregningsintensive oppgaver som kan levere resultater kjappere og til en lavere kostnad. Dette gjør de med sine tre produkter [22]:

BioBoost Lab

BioBoost Lab er en arbeidsstasjon med en FPGA på et tilleggskort. Dette produktet er hovedsaklig rettet mot utdanningsinstitusjoner og læringssituasjoner. Det leveres med diverse biologi-databaser, et web-grensesnitt og også flere vanlige verktøy for bioinformatikk, basert på åpen kildekode.

BioBoost Cluster Add-on

Cluster Add-on er rettet mot både forskningsinstitusjoner og læringsinstitusjoner som allerede har klyngemaskiner, men som trenger å utvide kapasiteten i det eksisterende systemet. Dette produktet leveres også med flere biologi-databaser og et grensesnitt som hjelper klyngeapplikasjoner høste akselerasjonen fra BioBoost-teknologien.

BioBoost Cluster

BioBoost Cluster er et komplett system som kan leveres basert på maskiner fra Apple, IBM eller Sun. Dette systemet leveres med de samme databasene som de to ovennevnte systemene, og det samme grensesnittet som Cluster Add-on.

Progeniq reklamerer med at deres system kan gi 40 ganger bedre kjøretid på parvis sekvensjustering/sekvensgruppering ved bruk av Smith-Waterman-algoritmen, sammenlignet med et enkelt-CPU-system. Den eneste spesifikasjonen de har oppgitt på dette systemet er at det er en Pentium P4 prosessor, som kjører i 3 GHz.

3.3.2 Mitrionics

Mitronics [23] er et selskap startet i 2001, og hovedkvarteret ligger i Lund i Sverige. Mitrionics leverer systemer til flere fagfelt, i tillegg til bioinformatikk. De har også vært involvert i oljebransjen (analyser av seismiske data) og bildeanalyse. Mitrionics selger ett system, basert på to deler:

Mitrion Virtual Processor

Kjernen i systemet heter Mitrion Virtual Processor, og er en mykkjerne prosessor implementert på en FPGA. Denne skal virke som et abstraksjonslag for programmereren, og ved hjelp av språket Mitrion-C skal en programmerer, uten særlige kunnskaper om maskinvareutvikling, kunne konfigurere FPGAen. Dette er ment å forkorte utviklingstiden for en fungerende konfigurasjon av FPGAen.

Mitrion Development Kit

For å hjelpe programmereren å finne beregningsintensive og/eller parallelliserbare deler har Mitrionics laget et Development Kit. Det inkluderer også en simulator og debugger som lar programmereren teste og evaluere sitt system uten å måtte benytte FPGA-ressurser.

I deres oppdrag for oljeindustrien nevnt ovenfor laget de en løsning med 25 FPGAer og fikk samme effektivitet som en klyngemaskin med 250 noder. I de tre andre løsningene de har beskrevet fikk de en hastighetsøkning på mellom 10 og 30 ganger.

3.3.3 Impulse C

Impulse Accelerated Technologies Inc. [24] er et amerikansk firma startet i 2002. Selskapet har base i Kirkland, Washington. Selskapet sier at deres mål er å hjelpe programvare- og maskinvare-utviklere ved å tilby verktøy for utvikling og verifikasjon av maskinvare gjennom programvare. Sagt på en enklere måte ønsker de å gjøre maskinvareutvikling tilgjengelig for programvareutviklere.

Produktet til Impulse er basert på to deler:

Impulse C bibliotek

Impulse tilbyr et bibliotek som lar utvikleren beskrive maskinvare i C-koden. Slik kan skillet mellom maskinvare og programvare viskes ut. Man kan også lett variere hvilke deler av applikasjonen som skal være maskinvare eller programvare, slik at forskjellige utviklingsstrategier kan testes. Dette skal føre til en enklere og kortere utviklingsprosess.

CoDeveloper

CoDeveloper er en samling verktøy for kompilering av applikasjonen, RTL-generering, testing/simulering, muligheter for parallellisering, identifisering av flaskehalser og andre måter å oppnå akselerasjon. Man kan bruke CoDeveloper som et utvilkingsmiljø, men man kan også bruke Impulse sammen med et vanlig utviklingsmiljø, f.eks. Eclipse.

Man kan også kombinere sine konstruksjoner med hdl-moduler, f.eks. tredjeparts ip-blokker eller egenutviklede blokker.

Impulse samarbeider med FPGA-produsenter (Altera, Xilinx), plattform-leverandører (SGI, Cray, Nallatech m.fl.) og programvare-produsenter (Synplicity, Symphony m.fl.) for å sørge for god støtte for produktene deres.

Del II

Løsning

Kapittel 4

Utredning rundt mulige utvidelser

Som beskrevet i 3.1.2 er filterfunksjonalitet en stor forbedring til FPWM-systemet fordi det begrenser bruken av båndbredde mellom FPGA og CPU. Dette kan gi systemet stor ytelsesøkning, fordi hele systemet må vente på skriving til CPU hvis skrivekøen blir full. Siden båndbredden mellom FPGA og CPU er begrenset er filteret en kritisk del av systemet. Derfor vil det, i 4.1, bli presentert mulige implementasjoner av terskelfilter og summasjonsfilter. Implementasjonene for de respektive filter vil også bli diskutert, målt opp imot hverandre og veid. Deretter, i 4.2, vil problemer og mulige løsninger for en flerkjerneimplementasjon av FPWM-modulen diskuteres. Tilslutt, i 4.3, vil det komme en utredning rundt mulige implementasjoner av endringer i rammeverket for å tilpasse det til bruk av filer som inneholder DNA og vektingsmatrise i utregningen og for å gi mulighet til valg av filter.

4.1 Mulig implementasjon av filter

I implementasjonen av FPWM-systemet som forelå ved prosjektstart er det ingen postprosessering av resultatene. To forslag til filter, med forskjellige fremgangsmåter for implementasjon vil her bli presentert.

Filtermodulens grensesnitt er som følger:

Inndata:

- reset_n, user_clk, stall: Kontrollsignaler delt av alle moduler. `stall` signaliserer at writer-modulen ikke er klar til å ta imot data.
- valid: Hvorvidt inndata fra pwm-modulen er gyldige
- data_in: Verdien utregnet av pwm-modulen
- index_in: Verdiens posisjon i strengen

Utdata:

- data_valid: Hvorvidt data_out er gyldig eller ikke. Reflekterer inndata `valid`
- data_out: Data som har tilfredsstilt filteret, og skal sendes til writer-modulen.

Signalene `reset_n`, `stall` og `valid` har tilhørende funksjonalitet som er lik i alle filtre, og presenteres derfor nedenfor.

Signalet `reset_n` er et felles kontrollsignal og hvis det er aktivt skal hele kretsen nullstilles. I et filter vil det si å tømme eventuelle registre og nullstille fullstendig.

Signalet `stall` settes høyt når writer-modulen ikke er i stand til å ta imot. I en enkjerne-implementasjon vil dette kun skje hvis writer-modulen får en verdi hver sykel. I så fall vil ikke buntskrivingen klare å holde tritt med produksjonen fra pwm-modulen.

Signalet `valid` verifiserer at innsignalene `data_in` og `index_in` er gyldige verdier. Hvis `valid` er lavt skal filteret ikke vurdere verdiene fra pwm-modulen. Dette kan være ved for eksempel en stall.

4.1.1 Terskelfilter

I dette filteret vil resultatene bli valgt ut på grunnlag av en terskelverdi `t`, satt av bruker ved oppstart. Verdier som tilfredsstiller terskelverdien ($\mathrm{v}_{p,s}^{m} \geq$ `t`, verdi v for hvert motiv m for hver posisjon p i hver streng s) blir sendt videre til writer-modulen, mens verdier som ikke tilfredsstiller terskelverdien ($\mathrm{v}_{p,s}^{m}$ `< t`) blir forkastet.

Mulige implementasjoner

En mulig implementasjon, algoritmeeksempel 4.1, bruker en enkel kontrollstruktur med en if. Hvis `t` $\geq \mathrm{v}_{p,s}^{m}$ og `stall='0'` blir `data_valid` $\leftarrow$ `valid` og `data_in` og `index_in` blir pakket i `data_out`. Ved `stall='1'` skal ikke verdier sendes til writer-modulen.

if $t \geq v_{p,s}^{m}$ **then**
```
   data_valid ← valid;
   data_out ← data_in & index;
```
end

Algoritme 4.1 : Implementasjon 1, terskelfilter

En annen mulig måte å implementere filteret på, algoritme 4.2, er å la `data_valid` bestemme om verdien skal skrives eller forkastes. Hvis `stall='0'` skrives verdien. Deretter, hvis `t` $\geq \mathrm{v}_{p,s}^{m}$ setter `data_valid` $\leftarrow$ `valid`. Ellers settes `data_valid` $\leftarrow$ `'0'`. Slik utnyttes logikken i writermodulen for å forkaste data som ikke er gyldig. Da blir ikke lignende logikk skapt i flere moduler.

En mulighet er at synteseverktøyet optimaliserer konstruksjonen av filteret, slik at begge disse vil bli lignende.

```
data_out ← data_in & index;
if t ≥ v^m_{p,s} then
   data_valid ← valid;
else
   data_valid ← '0';
end
```

Algoritme 4.2 : Implementasjon 2, terskelfilter

4.1.2 Summasjonsfilter

Her presenteres to implementasjoner av et filter som summerer verdiene fra pwm-modulen. På den måten kan man få en total verdi for et motiv. Med dette filteret vil kommunikasjon ut fra brikken minimeres til et minimum, da all utregning finner sted fullstendig uten kommunikasjon underveis.

Mulige implementasjoner

En mulig implementasjon, algoritme 4.3, er å bruke en variabel for å instansiere ett register. Da vil verdien `data_in` legges til dette registeret for hver sykel `valid` er aktivt, altså at verdien er gyldig. `data_out` er `std_logic_vector(63 downto 0)`, mens `data_in` er `std_logic_vector(37 downto 0)`. Av plasshensyn er det siste mest fordelaktig, men man må også tenke på overflow og at registeret muligens skal holde en stor verdi. For dette filteret må også grensesnittet utvides, da filteret må få beskjed om når modulen er ferdig med eksekveringen. Dette fordi registeret kun skal skrives tilbake til CPU etter endt kjøring.

```
sum : variable;

sum ← sum + data_in;
if ferdig = '1' then
   data_out ← sum;
end
```

Algoritme 4.3 : Implementasjon 1, summasjonsfilter

En annen mulig implementasjon, algoritme 4.4 er å bruke `data_out` istedet for ett register. Altså at verdien inn hele tiden legges til utsignalet. Det er da viktig å holde `data_valid` lavt og ikke sette det aktivt før eksekveringen er over. Et ankepunkt ved denne algoritmen er at `data_out` er 64 bit bred, og at denne summasjonen da kan ta for lang tid. En fordel er at man får 64 bits bredde på resultatverdien.

```
data_out ← data_out + data_in;
data_valid ← '0';
if ferdig = '1' then
   data_valid ← '1';
end
```

Algoritme 4.4 : Implementasjon 2, summasjonsfilter

4.2 Flerkjerneimplementasjon

I utviklingen av enkjerneversjonen av FPWM-systemet har ikke en flerkjerneimplementasjon blitt tatt hensyn til, og det er derfor noen punkter hvor enkjernesystemet kan forbedres med hensyn på en videreutvikling av denne til en flerkjerneløsning. Disse vil bli forklart i 4.2.1. Likevel har Lars Krutådal i [5] sett for seg to mulige parallelliseringmetoder av FPWM-systemet, nemlig flerkjerneimplementasjon og flernodeimplementasjon. Flerkjerneimplementasjon ved lokal parallellisering ble kort nevnt i 3.1.1 og her vil to mulige implementasjonsvalg for lokal parallellisering bli beskrevet i større detalj i 4.2.2.

4.2.1 Forberedende arbeid med enkjernesystemet

I syntese av system ved oppstart av prosjektet blir to infomeldinger, presentert i figur 4.1 og figur 4.2, gitt av synteseverktøyet. Disse meldingene gir informasjon om forbedringer som kan gjøres for å minske systemets areal.

INFO:Xst:738 - HDL ADVISOR - 512 flip-flops were inferred for signal <bufr>. You may be trying to describe a RAM in a way that is incompatible with block and distributed RAM resources available on Xilinx devices, or with a specific template that is not supported. Please review the Xilinx resources documentation and the XST user manual for coding guidelines. Taking advantage of RAM resources will lead to improved device usage and reduced synthesis time.

Figur 4.1: Informasjon fra syntetiseringsverktøy

INFO:Xst:738 - HDL ADVISOR - 1024 flip-flops were inferred for signal <pwm_table>. You may be trying to describe a RAM in a way[...].

Figur 4.2: Informasjon fra syntetiseringsverktøy, forkortet

Disse to meldingene gjelder signalet `bufr` i writermodulen, og signalet `pwm_table` i user_app-modulen. Ved å utbedre disse punktene kan BRAM komme godt med til også å minske størrelsen på brikken, og slik gjøre plass for flere mulige kjerner på FPGAen. I disse to punktene er det nevnt at 1536 flip-flop-er brukes istedet for BRAM. Som syntesen av systemet ved prosjektstart viser brukes det rundt 4500 Slice Flip Flops for å danne hele brikken. En reduksjon her på 1500 ville være en betydelig innsparing og kunne føre til et mer plassbesparende system, og også muligens noe høyere ytelse.

Signalet `bufr`

Her er to implementasjoner mulig. Den ene vedlikeholder nåværende struktur med to buffere som vekselvis skrives og leses, og en overliggende kontroll-modul som kontrollerer bufferene og lesing/skriving. Kontroll-modulen endres minimalt, men hvert buffer implementeres som to 32bits BRAM-blokker. Dette gir relativt lite endring i koden,

men innfører mye større buffere (128 plasser i motsetning til 8). Nåværende kompleksitet i koden vil også bestå, man må fortsatt velge mellom ett av to buffere for skriving og lesing.

En annen løsning som kan gi lavere kompleksitet, men som vil bety større endring i koden, er å implementere kun ett buffer bestående av 4 BRAM-blokker, hver på 16 bit. Dette vil gi samme størrelse som ovennevnte løsning, men kompleksiteten på løsningen vil bli lavere da man slipper å velge mellom flere mulige destinasjoner for dataene. Det er kun en mulig plass å lagre dataene. Grunnen til at man kan redusere fra to til ett buffer er at BRAM-blokkene tilbyr samtidig lese- og skrive-funksjonalitet. Hver BRAM har to porter, hvor man kan lese fra det ene og skrive til det andre. Selvsagt kan man ikke lese og skrive samme adresse samtidig (dette vil føre til at verdien som skal leses først blir tilgjengelig neste sykel), men med litt håndtering av adresser for skriving og lesing kan dette lett unngås.

Begge disse løsningene kan implementeres på en relativt enkel måte, men ettersom løsning to letter systemet for kompleksitet som er unødvendig er det mest hensiktsmessig å implementere løsning 2.

Signalet `pwm_table`

Signalet `pwm_table` inneholder vektingsmatrisen. Det finnes flere problemer med å implementere `pwm_table` som BRAM-blokker:
1. Det gjøres oppslag i denne tabellen hver sykel og hastigheten må opprettholdes, ellers vil hele kretsen lide av dette. En BRAM-blokk kan maksimalt lese og/eller skrive to verdier per sykel, derfor må flere BRAM-blokker instansieres, selv om dette sløser med minneplass.
2. Ved å implementere signalet som BRAM må en viss lengde på PWM-vinduet bestemmes. Det går an å implementere det slik at det går å gjøre søk på vektingsmatriser som er kortere, men ikke lengre enn en viss lengde `N`.
3. Slik det er implementert nå sendes vektingsmatrisen fra `user_app` hvor den lagres og ned til pwm-modulen. Denne utformingen må endres da BRAM-blokkene må instansieres i en modul, og all behandling må skje der. Enten må BRAM-blokkene instansieres i pwm-modulen, og verdiene som skal inn i vektingsmatrisen må sendes dit etterhvert, eller så må en egen modul lages, som kan ta seg av oppslag og lagring i matrisen. Denne siste tilnærmingen er sannsynligvis en svakere versjon av å implementere matrisen i pwm-modulen.

For å få bukt med hastighetsproblemet kan et samlebånds-steg implementeres i pwm-modulen. I denne modulen finnes det allerede flere samlebåndssteg så dette vil sannsynligvis ikke gå ut over ytelsen. Databredden på vektingsmatrisen tilsier at hver verdi må lagres over to adresser i BRAM-blokker. Siden en BRAM-blokk kun kan lese eller skrive to verdier per sykel må det instansieres N BRAM-blokker, hvor N er lengden på vektingsmatrisen. For å ivareta dagens 8 elementer lange vektingsmatrise må man altså instansiere 8 BRAM-blokker for å lagre $8 * 4$ verdier.

Det er ganske tydelig at dette legger store begrensninger på en mulig lengde. Ikke

bare for å unngå resyntetisering ved endring, men også for å bevare ressurser i en flerkjerneløsning. Alt i alt er det 232 BRAM-blokker. Med en lengde på vektingsmatrisen lik 16 og 16 kjerner i parallell brukes 256 blokker. To faktorer som kan begrense antallet tilgjengelige BRAM-blokker er at blokkene også brukes av andre moduler, slik at det ikke er 232 blokker tilgjengelig. Dessuten er det uvisst hvordan det vil påvirke kretsen å bruke så mange blokker, ettersom disse er fysiske ressurser på brikken og signalene må rutes deretter. Men med en maksimal lengde på vektingsmatrisen på 8 er det ikke lenger antallet BRAM-blokker som er begrensningen på det maksimale antall kjerner i parallell.

En endring i implementasjonen er uansett nødvendig, og som det tidligere ble påpekt er sannsynligvis den beste løsningen å implementere tabellen i pwm-modulen. Dette burde la seg gjøre ved å sende dataene, som ankommer som en verdi per sykel, til pwm-modulen hvor de lagres i BRAM. Plassering av verdiene i BRAM-blokkene bestemmes av adressen verdien har ved ankomst til minnet i FPGAen, og kan lett dekodes.

Angående alfabetstørrelse, som er på 4 i DNA, kan dette lett utvides. Dette fordi hver BRAM-blokk holder på alle verdiene for en gitt posisjon. Altså kan kretsen lett utvides til et alfabet på 128 uten at man må endre på bruken av BRAM til lagring av vektingsmatrisen.

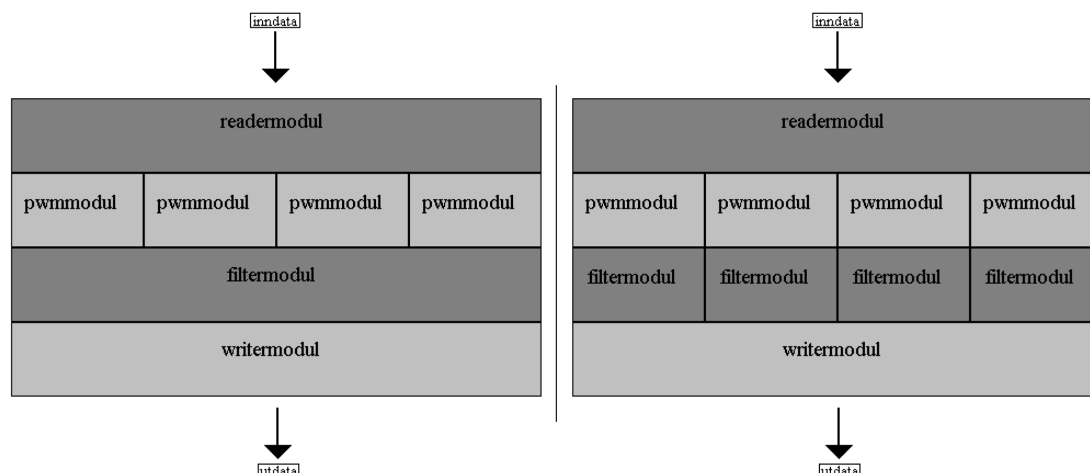
4.2.2 Endringer fra enkjernesystem til flerkjernesystem

I utformingen av en flerkjerneversjon av systemet er et ankepunkt i utviklingen hvor mye av systemet som skal instansieres flere ganger, og hvor mye man kan kjøre sekvensielt. Krutådal [5] har sett for seg to løsninger, begge med felles reader-modul, hvor forskjellen er at alle kjernene enten har en felles filter-modul, eller at kjernene inkluderer filter, og at writer-modulen er felles, og fungerer som samlende punkt for dataflyten på brikken.

Som sagt inkluderer begge løsningene felles reader-modul. Denne modulen må utvides til å spre data til flere kjerner samtidig, og dette kan for eksempel gjøres ved bruk av et skiftregister. Størrelsen på dette kommer an på antallet kjerner og lengden på motivet. Et skiftregister på størrelse 64 vil kunne mate 32 kjerner med en motivlengde på 32 tegn. Dette hvis hver kjerne henter elementer `(0 + egen_rang) .. (32 + egen_rang)`, hvor `egen_rang` er nummeret til hver instans. Altså henter kjerne 2 element 2 til 34 og kjerne 17 henter element 17 til 51, hvor både data og kjerner er nullindeksert.

Figur 4.3 viser konseptuellt to 4-kjerners oppsett av systemet, det ene med en delt filtermodul, det andre med separate filter for alle fire kjernene.

Med 32 kjerner som Lars Krutådal [10] har foreslått vil det potensielt kunne produseres 32 resultat for skriving hver sykel. Og ettersom Cray XD1 kun kan skrive 64 bit (ett resultat) hver sykel vil disse altså hope seg opp med svært stor hastighet. I beste fall, med bruk av buntskriving, vil åtte resultater kunne skrives på 9 sykler (grunnet en sykel nedkjøling etter skriving). Valg av filter og/eller parameter vil være avgjørende for kretsens ytelse. Ved bruk av terskelfilteret med en (relativt sett) for lav terskel vil svært mange resultater tilfredsstille terskelnivået, og antallet resultater som skal skrives vil være alt for høyt. Kretsen kan risikere å bruke $N + N/8$ sykler med kommunikasjon



Figur 4.3: Konseptuell skisse av forskjeller ved flerkjerneimplementasjon

for hver sykel utregning, hvor N er antall kjerner (og resultater til skriving) og $N/8$ er nedkjølingskostnader forbundet med skriving. Dette ved buntskriving.

En løsning, som kan tillate periodevis mange resultater til skriving, er å øke bufferstørrelsen i writermodulen betraktelig. Det er helt opplagt at denne i det minste må være lik antallet kjerner, men ved å bruke BRAM-blokkene som ligger på FPGA-brikken kan antallet elementer i writer-buffer økes. En diskusjon rundt dette er gjort i 4.2.1, men det er også en mulighet å utvide skrivebufferet til det maksimale av det ressursene tillater, slik at brikkens evne til å lagre opp resultater og tillate periodevis høyt antall verdier til skriving kan forbedres.

En mulig løsning for å unngå for lang prosesseringstid ved bruk av terskelfilter og en relativt sett for lav terskelverdi, kan være å lage en variant av dette filteret som kontinuerlig sorterer ut de beste resultatene og sender disse til CPU. Altså at man ikke sender flere verdier enn man kan uten å måtte stoppe eksekveringen. Da kan man tillate seg å sette for lav terskelparameter og likevel få en hurtig eksekvering. De fleste høye resultatverdier vil komme igjennom, mens en rekke lavere verdier vil bli forkastet.

N-beste-filter

Delt filter N-beste-filter er et filter Krutådal presenterte i [5], og det sorterer til enhver tid de åtte høyeste verdiene. Dette filteret kan være et ideelt filter for en flerkjerneimplementasjon, fordi kommunikasjonsproblemet skissert ovenfor ikke er vesentlig her. I en delt versjon av dette filteret er utfordringen å ta imot resultater fra alle kjernene samtidig og klare å sortere de. Dette har Krutadål [5] kort diskutert, og hans forslag var å ha en delt modul som er koblet til hver pwm-modul i de separate kjernene. Sorteringen består av en kjede med de N beste resultatene. For å sikre hastigheten på kretsen må det også velges hvilke resultater som skal bli sortert. Dette gjøres ved hjelp av mekanismer på hvert signal inn i datainntaket i resultatmodulen som forkaster resultater som er mindre enn det minste elementet i den sorterte kjeden. Dette er mindre

tidkrevende enn selve sorteringen, og slik sikrer man at alle elementer som sorteres vil komme inn i kjeden.

Separate filter Dette filteret egner seg svært godt som et filter implementert i hver av kjernene. Her kan hver kjerne gjøre sine beregninger og holde en sortert liste over de X (for eksempel 8) høyeste resultatene. Når pwm-modulen har gjort ferdig sine beregninger kan så en felles del av filteret trå i kraft, og samle sammen de Y (for eksempel 16) beste resultatene fra alle filterene. Dette kan gjøres med en teknikk lik den som brukes i flettesortering (mergesort). Pekere til hvert filters resultatliste opprettes, og ved sammenligning hentes det største og legges i en ny liste. Pekeren som pekte på dette elementet dekrementeres.

```
N ← antall resultater vi søker etter;
teller : variabel ← 0;
resultatliste : matrise[0..N-1];
while teller ≠ N do
   finn største element E;
   resultatliste[teller] ← E;
   teller++;
   peker_til_E++;
end
```

Algoritme 4.5 : Implementasjon med separate filter for N-beste-filter

En svakhet ved denne fremgangsmåten er at hvis de $X+1$ beste resultatene genereres av en kjerne, vil ikke det $X+1$. resultatet bli skrevet til CPU. Ved å lage reader-modulen som beskrevet i 4.2.2 vil de forskjellige kjerne operere på nøstede data, og det er derfor lite sannsynlig av et slikt tilfelle vil inntreffe.

Terskelfilter

Delt filter Et terskelfilter med delt implementasjon vil måtte ta imot store mengder data hver sykel, og sjekke disse mot terskelverdien. Det må også sende videre alle tilfredsstillende resultater. En logisk måte å se for seg dette er en buss, med bredde lik antall kjerner. Denne bussen går fra pwm-modulen, igjennom filteret og videre til writer-modulen. I writermodulen er det kun data som er gyldig og tilfredsstillende. Writer-modulen må da også selvsagt modifiseres til å kunne ta imot flere data per sykel. Den må også sette `stall` aktivt hvis det er færre ledige plasser i buffer enn det er kjerner instansiert.

Separate filter Ved terskelfilter med separat implementasjon kan man se for seg at bussen nevnt i avsnittet over begynner i filter-modulen og går til writer-modulen. Writer-modulen må klare de samme oppgavene som over.

Disse to løsningene vil sannsynligvis være svært like etter realisering.

Summasjonsfilter

Delt filter For summasjonsfilteret er det lett å se for seg en løsning med delt filter, hvor hver av kjernene kan sende sin verdi til et eget register i filteret. Her kan man bruke en løsning hvor man for hver sykel legger sammen to og to registre, til man tilslutt sitter igjen med en sluttsum. Denne metoden ville kreve en nedkjølingsperiode etter endt eksekvering, men denne er forsvinnende liten.

Et litt omstendig eksempel med åtte kjerner:

```
sum1, sum2,..., sum8, sum1_2, sum3_4, sum5_6, sum7_8, sum12_34,
sum56_78, sum1234_5678 : variable ← '0';
```

Fase 1

```
sum1234_5678 ← sum12_34 + sum56_78;
```

Fase 2

```
sum12_34 ← sum1_2 + sum3_4;
sum56_78 ← sum5_6 + sum7_8;
```

Fase 3

```
sum1_2 ← sum1 + sum2;
sum3_4 ← sum3 + sum4;
sum5_6 ← sum5 + sum6;
sum7_8 ← sum7 + sum8;
```

Fase 4

```
sum1 ← verdi1;
sum2 ← verdi2;

...

sum8 ← verdi8;
```

Algoritme 4.6 : Implementasjon av et delt summasjonsfilter

Det er lett å påpeke flere klare feil i pseudokode-biten over, men teorien den forsøker å illustrere er hovedpoenget. Det er heller ikke tatt hensyn til at filteret til slutt ikke får flere inndata i `verdi1,..., verdi8`.

Separate filter En implementasjon av summasjonsfilteret hvor man bruker separate filter er også lett å se for seg. Her ville hvert separate filter fungere likt enkjerne-implementasjonen av filteret, men man måtte også lage en felles del for å legge sammen alle tallene etterpå, i nedkjølingsfasen beskrevet ovenfor. Denne løsningen er sannsynligvis svakere, ettersom hver instans av filteret inkluderer en summering, samt at den delen som samler alle undersummer (verdiene som er regnet ut i hvert filter) har en tilsvarende mengde summasjoner som hele løsningen med delt filter. Altså vil konstruksjonen sannsynligvis ta større plass og flere ressurser.

4.3 Utvidelse av grensesnitt

Ettersom systemets hensikt er å gjøre maskinvare-akselerert DNA-søk tilgjengelig for personer uten dyp maskinvare-teknisk bakgrunn, som nevnt i 1.1, må det lages ett grensesnitt som gjør eksekvering på FPWM-systemet lett tilgjengelig. Ett forslag er å lage et webgrensesnitt hvor jobber lett kan lastes inn og legges klar for kjøring. Dette er imidlertidig utenfor denne oppgavens omfang, og for å gjøre de forskjellige filterene tilgjengelig og enkelt kunne endre terskelverdien i terskelfilteret kan man gjøre endringer i grensesnittet til FPWM-systemet, som ved prosjektoppstart er representert av c-rammeverket `fpwm.c`. Dette rammeverket er ansvarlig for lasting av konfigurasjonsfil og data til FPGAen og det leser resultatene som FPGAen produserer. Dessuten må det håndtere valg av filterimplementasjon og overføring av terskelverdi, ved valg av terskelfilter. Rammeverket som forelå ved prosjektoppstart genererte en DNA-streng og en vektingsmatrise, og rammeverket må derfor utvides for å kunne benytte det til søk i DNA. Dessverre er det en rekke problemer forbundet med dataoverføring mellom CPU og FPGA. Disse vil bli beskrevet nedenfor.

4.3.1 Valg av filter og setting av parameter

Ettersom resyntetisering av systemet og generering av konfigurasjonsfil tar svært lang tid, jamfør tabell 4.1, må valg av filter og lasting av eventuelle parametre foregå gjennom rammeverket til systemet. Et mulighet er å syntetisere og generere ferdig konfigurasjonsfiler for hver filterimplementasjon, og laste den etterspurte versjonen avhengig av parametre lagt inn i oppstart av c-rammeverket.

real	62m10.804s
user	61m25.957s
sys	0m20.124s

Tabell 4.1: Utskrift fra unix-verktøy `time` som viser tidsforbruk ved syntetisering av FPWM-systemet på Cray XD1.

Tabell 4.1 viser tiden det tar å syntetisere FPWM-systemet på Cray XD1. Rad 1 (real) viser hvor lang tid jobben tok, mens rad 2 og rad 3 viser hvor mye av denne tiden som ble brukt i CPU i henholdsvis brukermodus og kernelmodus. Vi ser av differansen mellom real og user+sys at syntetiserings-verktøyet har fått jobbe nesten uavbrutt. Likevel har tider fra 30 minutter og helt opp til 120 minutter blitt observert. Det er heller ikke forutsigbart hvor lang tid en syntetisering tar.

4.3.2 Overføring av DNA og vektingsmatrise

Mekanismer for overføring av ett ekte DNA og vektingsmatrise må lages, dette innbefatter også omkoding av DNA-streng og vektingsmatrise, slik at de får riktig format for FPWM-systemet. Et DNA i en fil er kodet som tekst med elementer ”a”, ”c”, ”g”,

”t” og ”N”. Sistnevnte betyr at den posisjonen av DNAet ikke er kartlagt, og kan derfor ignoreres i forbindelse med FPWM-systemet. Løsningen i FPWM-systemet koder hvert element i åtte bits og pakker åtte elementer sammen i en 64 bits streng. Derfor må en mekanisme for å bygge 64 bits pakker av en fil med DNA-koder lages. Enten kan dette lages separat, som et program for å konvertere filer, eller det kan inkluderes i rammeverket, altså at rammeverket konverterer mens det kjører og bygger 64 bits pakker.

Vektingsmatrisen må utregnes etter en bestemt formel, ettersom det er en forenklet utgave som blir lastet inn. Den må også kodes om ettersom flyttall ikke kan representeres FPWM-systemet.

FPGAen har også en kraftig plassbegrensing. Kun 16MB data kan overføres og prosesseres. Derfor er det nødvendig å dele opp prosesseringen av større datamengder til flere bolker. Ved datamengder over 16MB må dataoverføringer til FPGAen deles opp i bolker så store som 16MB, og resultater må lagres midlertidig for å kunne presentere disse tilslutt. For eksempel ved bruk av summasjonsfilteret må resulatet fra hver kjøring av FPGAen lagres for så å summeres og skrives ut etter at DNAet er gjennomsøkt. Dessuten må man ta høyde for at databolkene må overlappe hverandre, slik at alle posisjoner i DNAet blir prosessert.

4.3.3 Overføring av resultater

Når resultatene skal overføres tilbake til CPU opprettes først ett minneområde som deles av FPGA og CPU. Størrelsen på dette minneområdet er begrenset til 2MB, og dette setter en alvorlig begrensning på eksekveringen ettersom kun 262144 resultater, hver på 64 bits, kan lagres før minneområdet er fullt. Cray [14] har beskrevet en annen funksjon for å bygge et felles minneområde på størrelser helt opp til 1GB. Men ettersom det eksisterer en begrensning på inndata på 16MB, som nevnt ovenfor, fordrer dette et minneområde på 16MB inndata / 1 byte per element $*$ 8 byte per resultat = 128MB.

Denne funksjonen er noe mer avansert ettersom den stiller større krav til allokering av minneområdet. Den er likevel, i [14], oppført som foretrukket framfor funksjonen opprinnelig implementert i rammeverket.

4.3.4 Behandling av data

Data som kommer fra FPGA behandles ikke på noen måte. C-rammeverket som foreligger ved prosjektoppstart skriver de ut på skjerm etter endt eksekvering. En ønskelig måte å lagre, behandle eller fremstille resultatene må implementeres. Representasjon av data er utenfor denne oppgavens omfang, ettersom oppgaveteksten kun etterspør mulighet til å laste og eksekvere et DNA med tilhørende vektingsmatrise. Som nevnt i dette underkapittels innledning er ett forslag å lage ett webgrensesnitt hvor bruker kan legge inn DNA, vektingsmatrise og eventuelle parametre, som for eksempel valg av filter og terskelverdi. For å muliggjøre for et slikt utvidet grensesnitt er en mulighet å lagre alle resultater i en fil. På denne måten vil også c-rammeverket fungere som et

abstraksjonslag til et ytre grensesnitt, slik at et eventuelt web-grensesnitt ikke behøver å behandle lasting av data og konfigurasjon til FPGA.

Kapittel 5

Implementerte løsninger

I dette kapittelet vil de implementerte løsningene beskrives, og valgene tatt vil diskuteres på bakgrunn av redegjørelsen i 4. Først, i 5.1.1 og 5.1.2, vil løsningene for terskelfilter og summasjonsfilter presenteres. Deretter, i 5.2, vil en flerkjerneimplementasjon bli presentert. Tilslutt vil en implementasjon av rammeverket presenteres i 5.3

5.1 Filterimplementasjon

5.1.1 Implementasjon av terskelfilter

I 4.1.1 ble to forskjellige fremgangsmåter for implementasjon av et filter basert på et terskelnivå presentert og diskutert. Siden forskjellen på disse ikke er så stor, ble begge kjapt implementert. Figur 5.2, lagt til i slutten av kapittelet, viser en skjematisk tegning tegnet av Xilinx ISE Foundation [11] etter syntetisering. Det viste seg at syntetiseringsverktøyet optimaliserer kretsen på en slik måte at begge løsningene blir identiske, slik det også er foreslått i 4.1.1. Derfor er det kun tatt med en løsning videre, og denne vil her bli presentert.

Som nevnt overføres terskelverdien fra c-rammeverket, gjennom `user_app` og til filtermodulen. For å gjøre dette ble grensesnittet til filtermodulen utvidet med ett signal, `threshold`. Filteret lagrer denne og sjekker hver verdi som passerer gjennom filteret om den tilfredstiller terskelen. Dersom den gjør det settes `data_valid` aktiv og dataene sendes til writer-modulen. Dersom en verdi ikke tilfredsstiller terskelverdien, settes `data_valid` inaktiv, og dataene blir forkastet i writer-modulen som ugyldige. Filteret overholder `stall` og er synkront i forhold til resten av kretsen, med asynkron reset.

I syntese oppgir synteseverktøyet at antatt klokkehastighet, for hele kretsen, er 214.549MHz (med BRAM writer-modul beskrevet i 5.2.1).

Endelig kildekode er vedlagt i B.1.

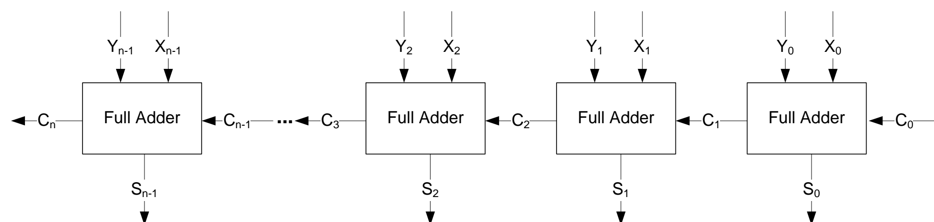
5.1.2 Implementasjon av summasjonsfilter

For å lage dette filteret måtte grensesnittet til filtermodulen utvides med et signal, som foreslått i 4.1.2. Dette for å si ifra når pwm-modulen var ferdig med sine beregninger, og nedkjølingsfasen begynner. I `user_app.vhd` (rammeverket som kontrollerer de fire undermodulene, se 3.1.2) er det lagt til et signal ved navn `filter_comp_finished`. Dette signalet settes aktivt når nedkjølingsfasen begynner, og filtermodulen sender da innholdet av register `sum` til writer-modulen for skriving. alle verdier som kommer til filteret etter at `filter_comp_finished` har gått aktivt vil bli forkastet, ettersom det ikke skal forekomme gyldige verdier på det tidspunkt. Verdiene skrives til CPU når `write_flush` settes aktivt. Writer-modulen skriver da alle gjenstående verdier i bufferene tilbake til CPU.

I filteret ble det implementert et signal `sum` til å holde på resultatet fra summeringen. Dette signalet har en bredde på 48 bits. Ved å sjekke det mest signifikante bitet kan man se om signalet `sum` nærmer seg overflyt. Hvis dette bitet er satt til 1, blir signalet sendt videre til writer-modulen med aktiv `data_out`. Ved en eventuell gyldig inndata i samme sykel blir denne lagt til som første verdi i neste samling summeringer. At modulen skriver ut sin verdi selv om eksekveringen ikke er ferdig og overlater til c-rammeverket å legge disse sammen er fordelaktig. Ved å kunne begrense bredden på signalet `sum` blir eksekveringen på brikken lettere, samtidig som det ikke vil representere noen tung operasjon for c-rammeverket å legge sammen verdiene ettersom det er et begrenset antall. Hvor mange som blir skrevet ut i løpet av en eksekvering avhenger av størrelsen på tallene inn, men det er lite realistisk at disse tallene vil være i størrelsesorden 2^{38} som databredden kan tillate.

Selve summeringen i modulen blir instansiert som en ripple-carrier, hvor hvert bit i to strenger blir lagt sammen, og et mente-signal bærer eventuelt overflyt oppover, derav navnet. Figur 5.1 viser hvordan denne kretsen er bygd opp, konseptuelt.

Først ble denne forsøkt implementert med en bredde på 64 bits, altså at man brukte hele bredden tilgjengelig i `data_out`. Dette viste seg å være for krevende, og hastigheten på kretsen ble for lav. Med hensyn til overflytsfeil er også valgte løsning bedre, da den har innebygde mekanismer som forebygger dette.



Figur 5.1: Konseptuell tegning av en ripple-carry adder.

Endelig kildekode er vedlagt i B.2.

5.2 Implementasjon av flerkjerneløsning

Implementering av et flerkjerne-system er et ganske stort arbeide, og tiden i min oppgave strekker ikke til til annen implementering enn en ny writer-modul, som vil minske enkjerne-systemets overflate betraktelig, og vil styrke grunnlaget for implementering av en flerkjerneløsning.

5.2.1 Writer-modulen

For å minske kretsens størrelse ble BRAM implementert i writer-modulen. Begge fremgangsmåter beskrevet i 4.2.1 ble implementert og testet.

2 buffer

Modulens oppbygning ble forandret minimalt, men istedet for å bruke ett signal til lagring i hver av bufferene, ble to BRAM-blokker implementert i hvert buffer. Grunnen til å implementere to er at hver blokk kun kan skrive eller lese 32 bit per sykel, og derfor trenger man to til å lagre eller skrive signalene `data_in, data_out : std_logic_vector(63 downto 0)` på en sykel. Dette gir mye større buffer, ettersom hvert av bufferene nå kan holde 128 verdier. De er designet slik at de skriver så ofte som mulig, ved bruk av buntskriving. De er konstruert sykliske, altså at de vil skrive rundt (verdi `B"0000000"` følger etter verdi `B"1111111"`). Overskriving av verdier som ikke er ennå er skrevet tillates, men signalet `overflow` vil settes aktivt for å varsle om dette. Det skal egentlig ikke skje, men hvis filteret ikke overholder et aktivt `stall` signal, vil kretsen likevel fungere, dog med et tapt resultat.

1 buffer

Denne fremgangsmåten implementerer minnet i writer-modulen som ett stort minne med plass til 256 elementer ved hjelp av 4 BRAM-blokker. Dataene distribueres jevnt over de 4, med 16 bit i hver. Hver BRAM-blokk har to porter, hvorav port A brukes til å skrive verdier og port B brukes til å lese verdier. På denne måten kan minnet leses og skrives samtidig. Også dette minnet er syklisk, og buntskriving brukes alltid, såfremt `flush` er inaktivt. Når `flush` blir aktivt skrives resten av elementene i minnet i bunter på 8 sålenge det er mulig. Den siste overføringen til CPU er ofte på færre elementer.

Denne fremgangsmåten bruker en teller, beregnet ut ifra skrive/lese-adresse, til å styre lesing fra BRAM/skriving til CPU. På denne måten får man en enkel og effektiv komponent.

Begge implementasjonene tilbyr enkel utvidelse av minnet, men den lavere kompleksiteten i denne konstruksjonen gjorde at den ble valgt for videre arbeid.

I tabell 5.1 og 5.2 presenteres utskrift fra synteseverktøyet som beskriver bruk av ressurser. Ved skifte til ny BRAM-basert writer-modul synker forbruket av ressurser, til tross for at et terskelfilter er introdusert.

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization
Number of Slices	3264	23616	13%
Number of Slice Flip Flops	4647	47232	9%
Number of 4 input LUTs	2783	47232	5%
Number of bonded IOBs	17	692	2%
Number of BRAMs:	4	232	1%
Number of GCLKs	2	16	12%
Number of DCMs	2	8	25%

Tabell 5.1: Utskrift fra synteseverktøy som beskriver utnyttelse av ressursene på FPGAen ved syntetisering av FPWM ved oppstart av prosjekt.

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization
Number of Slices	2356	23616	9%
Number of Slice Flip Flops	3515	47232	7%
Number of 4 input LUTs	2058	47232	4%
Number of bonded IOBs	14	692	2%
Number of BRAMs:	4	232	1%
Number of GCLKs	2	16	12%
Number of DCMs	2	8	25%

Tabell 5.2: Utskrift fra synteseverktøy som beskriver utnyttelse av ressursene på FPGAen ved syntetisering av FPWM med terskelfilter og BRAM-basert writer-modul.

Endelig kildekode er vedlagt i B.3.

5.3 Implementasjon av rammeverk

5.3.1 Valg av filter og overføring av parameter

For å lette innlasting av terskelverdien er den satt som ett parameter ved oppstart av programmet. For mer detaljert forklaring av oppstart se påfølgende del 5.3.5. Ved å bruke det som valgfritt parameter ved oppstart angir bruker terskelverdien ved oppstart av programmet om bruker ønsker å benytte terskelfilter. Hvis terskelparameter ikke blir oppgitt blir eksekveringen startet med summasjonsfilter. Terskelen blir overført til `user_app.vhd` ved hjelp av funksjonen `fpga_wrt_appif_val()` [14] som overfører en enkelt verdi direkte til FPGAens minneområde. Denne minneaksessen blir fanget opp i `user_app.vhd` og verdien blir så ved hjelp av to signaler overført til filter-modulen hvor den blir lagret.

5.3.2 Overføring av DNA og vektingsmatrise

Filen som inneholder DNAet er som nevnt kodet som bokstaver (a, c, g, t). Disse leses inn i en streng som så omkodes for å tilpasses FPWM-systemet. Dette gjøres ved hjelp av en case-løkke. En streng bestående av 8 tegn, eller 64 bits, blir konstruert før den overføres. I den siste strengen i en overføring blir strengen tillagt 0-bits i de minst signifikante posisjonene for at strengen skal fylle 64 bits. Lesing og omkoding av strengen foregår inntil det enten 1) er slutt på strengen, eller 2) det er overført 16MB. Hvis tilfellet er det sistnevnte, gjentas prosedyren inntil hele strengen har blitt overført. Slik får man delt opp store datasett i bolker som FPWM-systemet klarer å håndtere.

Også for hver bolk må vektingsmatrisen overføres. Disse verdiene leses også inn fra fil, som flyttall (float). Disse må også ha litt preprosessering før de kan overføres. Et eksempel på en vektingsmatrise som lastes inn i programmet finnes i tabell 5.3.

	A	C	G	T
1	4	4	3	0
2	2	5	4	0
3	3	2	4	2
4	2	0	9	0
5	0	11	0	0
6	11	0	0	0
7	0	0	11	0
8	1	2	8	0

Tabell 5.3: Vektingsmatrisens utseende ved oppstart og lasting i rammeverket.

Først blir 0.01 lagt til hvert element for å forhindre at $log(0)$ blir utregnet, ettersom dette ikke går. Deretter blir matrisen så prosessert etter formelen $log((i_j / \sum_{k=1}^{4} i_k)/0.25)$, hvor i er posisjon horisontalt i tabellen, mens j er vertikalt. Et eksempel hvor $i = 1$: $log((4/4+4+3+0)/0.25) = 0.538934$. Dette konstruerer verdiene som blir overført til FPWM-systemet. Tabell 5.4 viser det ferdige resultatet.

	A	C	G	T
1	0.538934	0.538934	0.125095	-8.108524
2	-0.457473	0.860142	0.538934	-8.108524
3	0.125095	-0.457473	0.538934	-0.457473
4	-0.457473	-8.108524	1.706859	-8.108524
5	-8.108524	1.996074	-8.108524	-8.108524
6	1.996074	-8.108524	-8.108524	-8.108524
7	-8.108524	-8.108524	1.996074	-8.108524
8	-1.450313	-0.457473	1.537134	-8.108524

Tabell 5.4: Vektingsmatrisens utseende etter prosessering i rammeverket.

Etter utregningen konverteres verdiene fra floating point til fixed point. Hovedforskjellen på disse er at floating point er et kodet format, i den forstand at den ikke kan leses som en vanlig bit-vektor. Fixed point setter av M bits til heltallsdelen og N bits til frak-

sjonsdelen. Dette begrenser tallets nøyaktighet, særlig i tilfeller med stor fraksjonsdel, men til gjengjeld kodes tallet som en vanlig bitvektor, hvor det minst signifikante bitet er den laveste verdi, og det mest signifikante bitet er den høyeste verdi. Dette tillater vanlig bitmessig aritmetikk, og i kombinasjon med 2s kompliment kan tall adderes i FPWM-systemet uten å måtte ta hensyn til hvorvidt tallet er negativt eller positivt.

Hvis ett datasett krever prosessering i flere bolker (>16MB) blir FPGAen nullstilt for hver bolk. Dette gjøres med kommandoene `fpga_reset()` og `fpga_start()`. Disse kommandoene henholdsvis aktiverer og deaktiverer `reset_n`. Sistnevnte kommando kan også brukes for å bekrefte at FPGAen er klar til eksekvering, men brukes i dette tilfellet for å hente FPGAen ut av nullstillingsfasen.

Ettersom dette må gjøres for å sikre eksekveringen i FPGAen, må alle verdier overføres for hver bolk. Dataene fra DNA-strengen er unike for hver gang, og også verdien som angir antall elementer i DNA-strengen må nødvendigvis overføres for hver bolk. Vektingsmatrisen, eventuell terskelverdi og adresse til det delte minneområdet er derimot overflødig i overføringen, men må overføres ettersom alle data i FPGAen er nullstilt. Men sammenlignet med DNA-strengen er denne datamengden minimal, og representerer ikke noen reell flaskehals.

5.3.3 Overføring av resultater

For å sikre at alle resultater blir oppfanget av c-rammeverket blir ett minneområde på størrelse med 16.777.216 elementer $*$ 64 byte allokert. Dette fordi minneområdet må bestå av N hele sidetabeller i minnet, hvis størrelse på Cray XD1 er 4096 byte. For alle verdier mellom $16.777.216 * 64$ og $(16.777.216 * 64) - 4096$ må nevnte størrelse brukes. For å finne sidetabell-størrelsen brukes funksjonen `getpagesize()`.

Det allokerte minnet må også justeres slik at det okkuperer N hele sidetabeller. Altså at området begynner på starten av en sidetabell og ender på slutten av en sidetabell. Dette gjøres med funksjonen `posix_memalign()`.

Disse operasjonene står beskrevet nærmere i [14].

Når c-rammeverket har startet eksekveringen på FPGAen, ved å overføre en variabel hvis innhold er antall elementer som skal prosesseres, venter den i 0.2 sekunder. Deretter leses det allokerte og delte minneområdet ved hjelp av en iterator som beveger seg fra start til slutt. Verdiene som ligger der blir behandlet og noen verdier blir utregnet. Eksempel på dette siste kan være den totale sum for alle bolker i en større kjøring ved bruk av summasjonsfilter. Alle verdier som leses fra det delte minnet må konverteres fra fixed point og tilbake til floating point.

5.3.4 Behandling av data

Data behandles ikke i noen særlig stor grad i denne utgaven av rammeverket. Verdiene som leses fra det delte minnet har litt forskjellig format avhengig av hvilket filter de stammer fra. Ved bruk av summasjonsfilter er alle 64 bits ett tall, selv om de øverste

16 alltid er 0. Dette tallet skrives rett til fil, samtidig som tallet legges til en sum som gjelder hele DNAet. Ved bruk av terskelfilter er de 64 bits kodet til å inneholde en index (63-38) og en verdi (37-0). Disse deles opp og skrives til fil på en linje. Altså vil hver linje i utdata-filen inneholde ett resultat.

5.3.5 Kort brukerveiledning

For å starte eksekvering av et DNA må man, i tillegg til tilgang til prosjektfilene, ha det ønskede DNA og vektingsmatrise tilgjengelig i to filer. Det er anbefalt å logge inn på en av undernodene på musculus.hpc.ntnu.no og ikke eksekvere på innloggingsnoden (musculus403-6). Dette for å sikre at man fortsatt har tilgang til de resterende nodene, selv om en feil skulle føre til at noden man jobber på krasjer.

I katalogen som inneholder binær-filene og den kompilerte, kjørbare versjonen av `fpwm.c` (vanligvis prosjektkatalog/bin/) starter man programmet ved å skrive:
`./fpwm [sti til DNA-fil] [sti til vektingsmatrise-fil] [eventuell terskel-verdi]`
De to første parametre er obligatoriske, det siste er valgfritt. Skulle en terskelverdi ikke oppgis vil eksekveringen skje med summasjonsfilter. Etterhvert som flere filter blir implementert, skulle dette skje, kan dette grensesnittet lett utvides, for eksempel med funksjonsflagg. Eksempel på dette kan være `./fpwm -t ...` for terskelfilter.

Data som omhandler tilstanden til kjøringen vil skrives ut på skjerm fortløpende, mens resultater fra kjøringen vil skrives til filen `output-fpwm.txt` i rent tekstformat.

Figur 5.2: Skjematisk tegning av terskelfilter etter syntetisering

Del III

Analyse

Kapittel 6

Eksperimenter og resultater

I dette kapittelet vil eksperimenter fra forskjellige faser av arbeidet bli presentert, i en rekkefølge som følger en naturlig utvikling i implementasjonsarbeidet. I delkapittel 6.1 vil først eksperimenter med terskelfilter bli presentert i 6.1.1, deretter følger eksperimenter med summasjonsfilter 6.1.2. I delkapittel 6.2 kommer eksperimenter med flerkjerne-forberedelser og flerkjerneimplementasjon beskrevet i 5.2. I 6.3 vil eksperimenter med rammeverket bli presentert.

6.1 Eksperimenter med enkjerneløsning

6.1.1 Eksperimenter med terskelfilter

Simulering av terskelfilter

Terskelfilteret er både simulert som egenstående modul og som en modul i det helhetlige systemet. Simuleringene med terskelfilter som egenstående modul er grundigere når det gjelder filterets funksjonalitet, toleranse og ytelse, mens system-simuleringene bekrefter filteret som en del av hele systemet. I B.1.2 finnes kildekode for testbenk konstruert til terskelfilter-simulering. I simulering av hele systemet er det en generell testbenk for begge filterimplementasjoner og kildekode for denne finnes i B.3.3.

Under simuleringene av terskelfilter som egenstående modul ble ingen alvorlige feil oppdaget, og derfor ble ingen store endringer gjort. Dette skyldes nok konstruksjonens lave kompleksitet.

Følgende forsøk ble gjort:

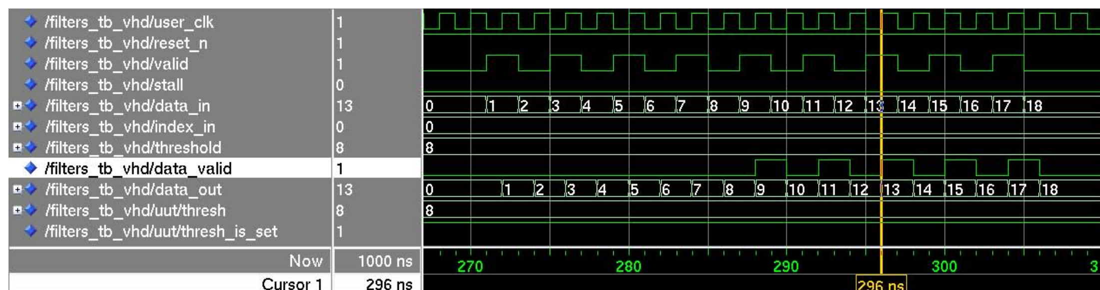
Lasting av terskelverdi (1/4) Terskelverdien ble forsøkt lastet inn til filteret for å verifisere at den ble lagret som en lokal kopi. Dette var vellykket.

Terskelverdiens funksjon (2/4) Filterets funksjonalitet ble testet ved å sende igjennom en strøm av elementer med økende verdi. Terskel-funksjonaliteten fungerte, og satte `data_valid` aktiv idet elementene var høyere i verdi enn terskelen.

`data_valid` som et produkt av `valid` og `data_in` (3/4) Med valgte implementasjon er det `data_valid` som gjenspeiler om `data_in` tilfredsstiller terskelverdien. En strøm med elementer, inkrementert etterhvert, ble sendt gjennom filteret. Samtidig ble `valid` satt fluktuerende. Korrekt resultat er altså at alle elementer mindre eller lik terskelverdi får `data_valid` ← `'0'`, mens alle elementer større enn terskel får `data_valid` ← `valid`. `data_valid` blir da fluktuerende, som `valid`. Filteret besto testen.

Tilbakestilling av filter, med påfølgende kjøring av test 3 (4/4) Ved å sette `reset_n` aktiv nullstilles alle verdier, deriblant terskelverdien. For å starte ny eksekvering lastes deretter terskelverdi igjen. Tilslutt i testen ble test 3 kjørt på nytt for å forsikre at filteret fungerte som det skulle. Testen ble korrekt fullført.

Figur 6.1 viser et utsnitt fra simuleringen, og beskriver forsøk 3. Vi ser her hvordan `data_valid` først er lav inntil `data_in` overstiger terskelverdien (8). Deretter fluktuerer det i takt med `valid`.



Figur 6.1: Utsnitt fra simulering av terskelfilter.

Funksjonaliteten til signalene `stall` og `threshold` ble også verifisert i simulering av helhetlig system. `stall` ble testet ved å sette terskelverdien lavt, slik at mange elementer ble lagt i kø for skriving.

Under simulering med hele systemet ble det oppdaget en alvorlig feil. Hvis `stall` ble satt høyt forsvant ett element fra eksekveringen, mellom pwm-modul og filter-modul. Derfor ble filter-modulen endret til ikke å inkludere sjekk etter `stall`. Ansvaret for å håndheve stopp i eksekveringen ved aktivt `stall` hviler nå på reader-modulen og pwm-modulen. Data som allerede er utregnet vil bli sendt gjennom filter-modulen og til writer-modulen. Writer-modulen har kapasitet til å håndtere dette uten at data mistes, da den kontinuerlig leser fra BRAM. Dessuten ble sammenkoblingen av modulene endret litt, men det vises nok bedre i koden, og ikke i den ferdig syntetiserte implementasjonen.

Kjøring med terskelfilter

I C.1 finnes utskrifter fra kjøring av systemet. Disse utskriftene er svakt redigert for å bedre lesbarheten. Tre kjøringer ble gjort. Alle tre ble gjort med rammeverket som forelå ved prosjektoppstart og prosesserte 262144 elementer. Forskjellen på de tre kjøringene er terskelverdien som er satt.

I kjøring 1 ble terskelen satt til 0. Antallet resultater ble 262136. Dette tallet er lavere enn antall elementer fordi pwm-vinduet må være fullt for å kunne prosessere ett element, jamfør 2.3.

I kjøring 2 ble terskelverdien satt til 11. Her ble antall resultater 196606.

I kjøring 3 ble terskelverdien satt 15. Her ble antall resultater 8192.

6.1.2 Eksperimenter med summasjonsfilter

Simulering med summasjonsfilter

I likhet med terskelfilteret ble summasjonsfilteret simulert både egenstående og som en helhet med resten av systemet.

Følgende forsøk ble gjort i simulering av modulen:

Summering og `filter_comp_fininshed` (1/4) 100 elementer med `valid` aktiv ble sendt inn i filteret. Deretter ble `filter_comp_finished` satt aktivt, for å sette `data_out` ← `sum`. Filteret besto denne testen, og summasjonen var også riktig.

Inndata med `filter_comp_fininshed` aktivt (2/4) 104 elementer, med gyldig `valid` ble sendt inn i filteret. Etter element 100 ble `filter_comp_fininshed` ← `'1'`. Dette for å sjekke at ingen flere elementer enn de allerede summerte blir summert eller sendt til writer-modulen. Filteret besto forsøket.

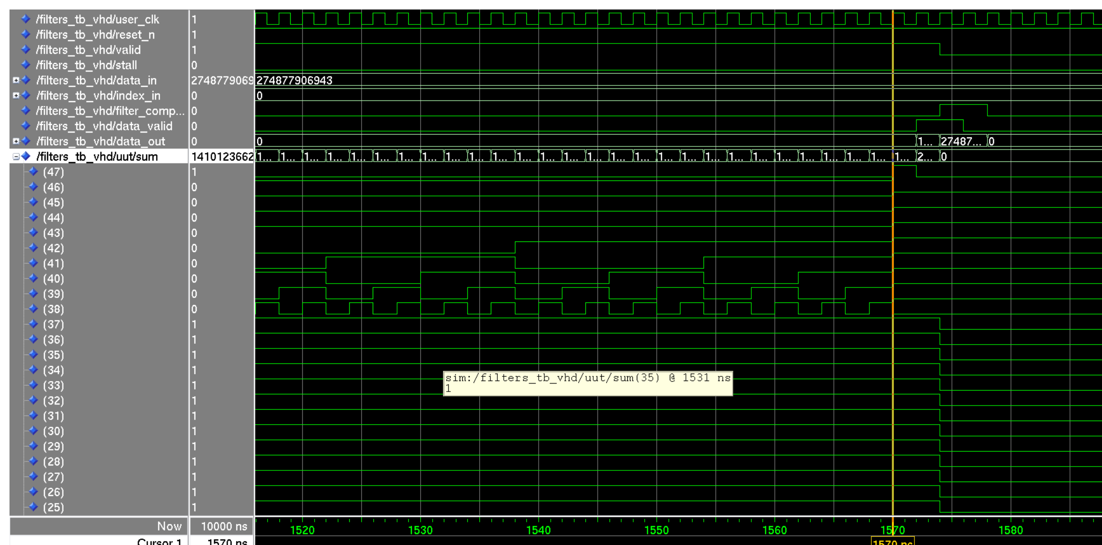
Utskrift for å forhindre overflyt (3/4) For å forhindre overflyt ved summering skrives signalet `sum` til writer-modulen hvis det mest signifikante bit tilsvarer 1, som nevnt i 5.1.2. Gyldige elementer med en størrelse på $2^{3}8$ ble sendt inn i filteret kontinuerlig til `sum` ble skrevet ut. Filteret besto forsøket og sum ble skrevet ut da det mest signifikante bitet gikk fra 0 til 1.

Nullstilling og videre funksjon (4/4) 100 elementer ble skrevet til filteret. Deretter ble `reset_n` satt aktivt. Dette skal da nullstille alle signaler i filter-modulen. Deretter ble en ny eksekvering av 100 elementer påbegynt og tilslutt ble `filter_comp_fininshed` ← `'1'`. Filteret skrev ut summen av de 100 siste elementene, og alle utgående og lokale signaler ble nullstilt.

Under simulering av helhetlig system ble det oppdaget en situasjon som krevde endring. Signalet `filter_comp_finished` ble endret, slik at det blir satt aktivt i steg 8 av i alt 16 steg i nedkjølingsfasen, istedet for i steg 1 av nedkjølingsfasen. Dette for å la pwm-modulen fullføre og sende alle tall til filter-modulen. I pwm-modulen er det benyttet en samlebånds-teknikk for å regne ut verdien for en gitt posisjon, slik at denne ennå gir fra seg utdata i 5 sykler etter at nedkjølingsfasen har begynt. Et tidligere forsøk sendte data rett gjennom filteret til writer-modulen når `filter_comp_finished` var aktivt, men denne fremgangsmåten var ikke like robust som førstnevnte endring, og den ga også en høyere kompleksitet i konstruksjonen.

I tillegg har også filteret blitt endret til å utelate `stall`, slik det er beskrevet 6.1.1.

Figur 6.2 viser et utsnitt fra simuleringen, og beskriver forsøk 3. Det ene signal hvis navn ikke er fullstendig i figuren er `filter_comp_fininshed`. Vi ser her at det mest signifikante bitet i `sum` blir 1, og i påfølgende sykel skrives `sum` til writer-modulen. To sykler med skriving forekommer. Den andre er et resultat av at `filter_comp_finished` blir aktivt, og dermed skrives verdien som ble lagret i samme sykel som den første skrivingen fant sted.



Figur 6.2: Utsnitt fra simulering av summasjonsfilter.

Kjøring med summasjonsfilter

I C.2 finnes utskrifter fra kjøring av systemet. Disse utskriftene er svakt redigert for å bedre lesbarheten. I kjøring av FPWM-systemet med summasjonsfilter ble 4 forskjellige kjøringer foretatt. Forskjellen på de tre første er antallet elementer som ble prosessert. Den siste testen har svært høye verdier i vektingsmatrisen for å forsøke å fremtvinge skriving av resultater for å forhindre overflytsfeil.

I kjøring 1 ble antallet elementer satt til 100. Etter kjøring forelå ett resultat i det delte minneområdet.

I kjøring 2 ble antallet elementer økt til 1100. Etter kjøring forelå ett resultat i det delte minneområdet.

I kjøring 3 ble antallet elementer ytterligere økt til ett maksimum på 262144 elementer. Etter kjøring forelå ett resultat i det delte minneområdet.

I kjøring 4 ble antall elementer vedlikeholdt fra kjøring 3, mens verdiene i vektingsmatrisen ble drastisk økt. Dette for å fremtvinge en test av overflytsbeskyttelsen implementert i summasjonsfilteret. Etter kjøring forelå 22 resultater i det delte minneområdet.

6.2 Eksperimenter med flerkjerneløsning

6.2.1 Forberedelser til flerkjernesystem

Simulering av writer-modul

I likhet med de to filterene er også writer-modulen simulert som egen modul og som en helhet av systemet. Simulering med helhetlig system er i motsetning til de to filterene ganske viktig, da en slik test kan avdekke endel feil som ikke blir oppdaget i simulering av kun writer-modulen. Ettersom writer-modulen har et større grensesnitt og er en langt mer kompleks modul, viser test med resten av systemet da klarere hvordan endringer i signalene i andre moduler påvirker writer-modulen.

Følgende tester ble gjort i egenstående simulering av writer-modulen:

Test av `flush` (1/11) Seks elementer ble lagt inn i minnet med gyldig valid. Deretter ble `flush` ← `'1'`. Modulen tømte alle elementer, og signaler i forbindelse med overføring til CPU ble korrekt satt.

Test av `flush` (2/11) Seksti elementer ble lagt inn i minnet med gyldig valid. Deretter ble `flush` ← `'1'`. Formålet var å se hvordan kretsen reagerte med utskrift allerede pågående. Kretsen tømte minnet på raskest mulig måte.

Test av buntskriving (3/11) Denne testen er nedtegnet som 3.1 og 3.2 i kildekoden til testbenken. Først, i 3.1, overføres ni elementer inn i minnet. Åtte av disse skrives til CPU ved hjelp av buntskriving. Ett ligger igjen. Deretter, i 3.2, overføres femten verdier inn i minnet. Alle disse (1+15) skrives tilbake i to omganger ved hjelp av buntskriving.

Test av buntskriving (4/11) 24 elementer ble overført inn i minnet. Kretsen skriver til CPU i tre bunter, hver på 8 elementer, med en gang 8 elementer ble tilgjengelig i minnet.

Test overføring av data og base-adresse etter reset (5/11) `reset_n` ble satt aktivt og kretsen nullstilt. Deretter ble en ny adresse overført (for bruk i skriving til CPU). Deretter ble 24 elementer skrevet til minnet. Disse ble skrevet tilbake til CPU i tre bunter, hver på 8 elementer.

Test av buntskriving med 64 elementer (6/11) 64 elementer ble sendt inn i writer-modulen. Dette for å se hvor kjapt modulen klarer å skrive tilbake til CPU. 15 sykler etter endt overføring til minnet var minnet igjen tomt.

Test av buntskriving med 64 elementer og `flush` (7/11) Hensikten med denne testen var å se om det var noen forskjell i forhold til test 6. Det var det ikke, og forløpet var helt identisk.

Test av `flush` med 1, 2,.., 7 elementer (8/11) Først ble ett element sendt inn i minnet, deretter ble `flush` ← `'1'`. Kretsen ble tømt og `flush` ← `'0'`. Deretter ble det hele gjentatt med 2 elementer, 3 elementer helt opp til 7 elementer. Dette for å se at kretsen klarer å håndtere ett aktivt `flush` i alle minnets tilstander.

Test at minnet er syklisk (9/11) 270 elementer ble skrevet inn i minnet. Dette for å se at både skrive-adresse og lese-adresse var syklisk og roterte rundt fra `1111111` til `0000000`. Modulen besto testen.

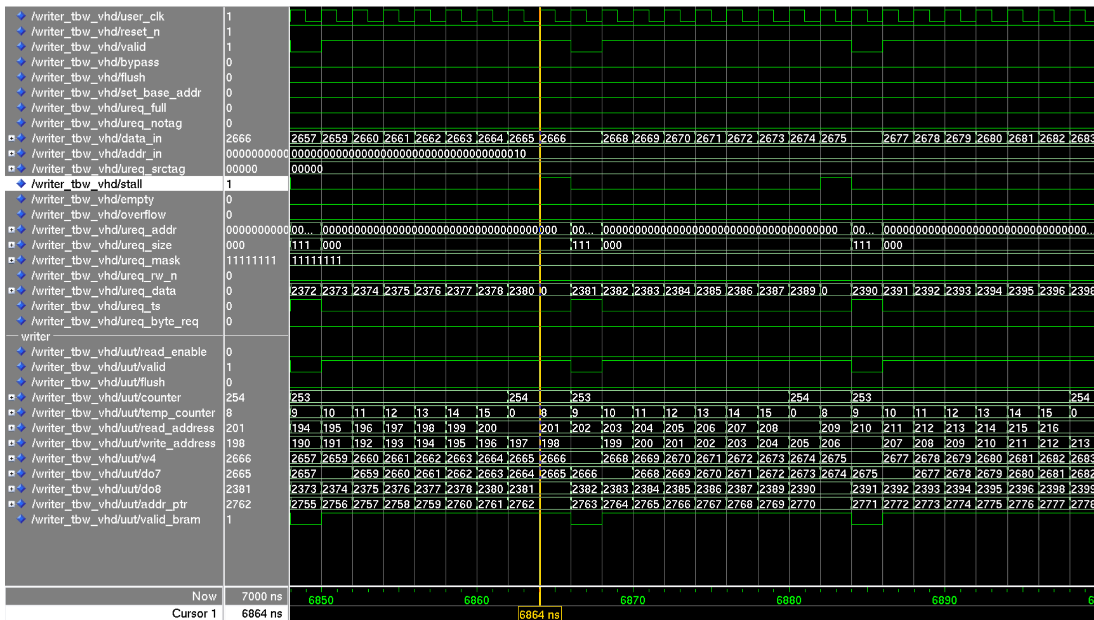
Test med aktiv `flush` og gyldig inndata (10/11) `flush` ← `'1'` og gyldig inndata ble sendt inn i modulen. Dette for å sjekke at minnet ikke blir låst når `flush` ← `'1'`.

Test etter `overflow/stall` (11/11) Data ble ført inn i modulen helt til `stall = '1'`. Når `stall` igjen ble inaktivt fortsatte data-overføringen. Dette ble gjentatt en rekke ganger for å se om signalet `overflow` ville bli aktivt. Det ble det ikke, og modulen besto testen.

Figur 6.3 viser et utsnitt fra simulering av forsøk 11. Her ser vi at signalet `stall` går opp sykelen før en ny skriving til minnet påbegynnes (se signal `ureq_ts`).

6.3 Eksperimenter med rammeverk

Under kjøring av større datamengder, fordelt over flere bolker, ble det oppdaget en feil med terskelfilter-versjonen av systemet. Ved nullstilling av FPGAen låste hele systemet seg, og noden systemet kjørte på krasjet. Dette skjedde konsekvent ved store datamengder, men det var ingen synlige mønstre i når systemet låste seg. Det har blitt observert krasj etter 7 bolker såvel som krasj etter 30 bolker. Ettersom dette kun inntraff ved



Figur 6.3: Utsnitt fra simulering av terskelfilter.

bruk av terskelfilter, ble overføring av terskelverdien undersøkt. Som nevnt i 5.3 var to signaler brukt for å overføre verdien fra `user_app.vhd`. Denne fremgangsmåten ble revurdert, og konstruksjonen ble forenklet til kun å bruke ett signal i transport av terskelverdi. Dette løste problemet og store kjøringer med terskelfilter ble mulig.

Terskelfilteret ble også såvidt endret for å ta høyde for 2s komplement, det vil si måten tallene er formatert på.

I C.3 finnes utskrift fra tre kjøringer med det ferdige rammeverket. Filteret brukt i alle tre kjøringer er terskelfilter. Vektingsmatrise brukt i de tre forsøkene er identisk, mens DNA i kjøring 1 og kjøring 3 er identisk. Det samme DNAet er brukt i kjøring 2 men det er firedoblet, altså 4 ganger sekvensielt. En annen forskjell på de tre kjøringene er terskelnivået.

I kjøring 1 ble terskelnivået satt til 3. Rammeverket ble startet med kommandoen `./fpwm chr21.fa chr21.pwm 3`. Antall resultater var 57120.

I kjøring 2 ble ikke terskelnivået endret. DNAet som ble brukt som inndata var en firedobbel sekvens av DNAet brukt i kjøring 1. Rammeverket ble startet med kommandoen `./fpwm quadchr21.fa chr21.pwm 3`. Antall resultater var 242704.

I kjøring 3 ble terskelnivået endret til 5.1. DNAet og vektingsmatrisen brukt var identisk med de i kjøring 1. Rammeverket ble startet med kommandoen `./fpwm chr21.fa chr21.pwm 5.1`. Antall resultater var 2767.

Kapittel 7

Diskusjon

I dette kapittelet følger diskusjon rundt resultater av kjøring i 7.1. En diskusjon rundt mulige valg ved en flerkjerneimplementasjon følger i 7.2. Tilslutt diskuteres valgene gjort i implementasjonen av rammeverk i 7.3.

7.1 Filterimplementasjon

7.1.1 Terskelfilter

I simulering av filteret som egen komponent ble ingen feil avdekket. Til tross for dette ble to alvorlige feil avdekket da filteret ble simulert som en del av systemet, og under kjøring av systemet. Likevel vil jeg påstå at dette ikke skyldes dårlige forsøk under simulering av modulen. Simulering av filteret som egenstående komponent var for å avdekke hvordan filteret reagerte på variasjoner og forskjellige kombinasjoner i inndata, mens feilene som ble avdekket i de to andre nevnte testene gjaldt systemet som en helhet. Riktignok ble det gjort endringer i filterimplementasjonen ved en av disse anledningene, men det var for å korrigere en misforståelse av systemets oppbygning, og dreide seg i så måte ikke om funksjonaliteten i filteret.

Modulens enkle oppbygning og dens få innsignaler begrenser antall forskjellige situasjoner som kan oppstå. Filtermodulen har 5 innsignaler som er i bruk:

- `threshold`: Signalet bærer terskelverdien ned til filteret hvor denne lagres lokalt.
- `data_in`: Signalet bærer inndata til filteret fra pwm-modulen.
- `index_in`: Signalet bærer i likhet med `data_in` inndata fra pwm-modulen.
- `valid`: Signalet forteller hvorvidt inndata fra pwm-modulen er gyldig.
- `reset_n`: Signalet setter igang nullstilling av alle verdier.

De to sistnevnte signalene er tilstandstyrende i modulen, og styrer i så måte modulens reaksjon. `reset_n` er det signalet som er høyest i rang. Hvis dette signalet er aktivt

nullstillles alle interne og utgående signaler, og ingen annen operasjon vil foretas i kretsen, uavhengig av andre signaler. Signalet `valid` deltar, sammen utvelgelse basert på terskelverdien, i settingen av utsignalet `data_valid`, som forteller hvorvidt utdataene i `data_out` er gyldige.

Signalet `threshold` styrer ikke kretsen på samme måte som de to signalene nettopp beskrevet, men et internt tilstandsregister velger hvorvidt dette signalet skal lagres internt. Hvis dette tilstandsregisteret tillater lagring, sjekkes signalet `threshold`. Dette signalet må nemlig være større enn 0 for at det skal lagres, ettersom nullstilling av hele systemet fører til at `threshold` vil være 0. Hvis terskelverdien er satt til 0 av bruker vil ikke dette påvirke systemet, ettersom den interne kopien av terskelverdien er initiert til null.

Simuleringenes dekningsgrad er, basert på det lave antallet tilstander som er mulig, god. Fire forsøk ble simulert, og disse dekker alle lovlige tilstander. Det er en tilstand som ikke er testet, men som kan nås. Dette er hvis tilstandsregisteret for lagring av terskelverdi tillater det og en ny terskelverdi ankommer filteret samtidig som inndata fra pwm-modulen ankommer. Da vil data velges på grunnlag av en utdatert terskelverdi, ettersom denne først oppdateres påfølgende sykel. Denne tilstanden kan kun komme som en følge av en feil ved oppstart av kretsen, ettersom all lasting av verdier til kretsen må skje før lasting av verdien som sier hvor mange elementer som skal prosesseres. Denne sist nevnte verdien starter også prosesseringen i kretsen. Ettersom pwm-modulen er bygd på samlebåndsprinsipp skal terskelverdien nå frem til filteret i god tid før første inndata fra pwm-modulen. Et forsøk på å fremprovosere denne tilstanden ble gjort under kjøring ved å redigere rammeverket. Resultatet varierte noe, men hovedsaklig ble ikke terskelverdien fanget opp i filteret. Som sagt er denne tilstanden et resultat av feil oppstart og sekvens i innlasting av data fra rammeverket.

Tabell 7.1 viser forholdet mellom terskelverdi og antall resultater i kjøringene som ble foretatt.

Som denne tabellen vitner om kan bruken av et terskelfilter drastisk redusere mengden data som blir skrevet fra FPGA. Som vi ser av tabellen, basert på utskriftene fra kjøring i C.1, reduseres mengden resultater etterhvert som terskelen stiger. Dette bekrefter filtermodulens funksjon. Dataene i vektingsmatrisen for disse forsøkene er gjengitt i 7.2.

Terskelverdi	Antall resultat
0	262136
11	196606
15	8192

Tabell 7.1: Terskelverdi og antall resultat ved kjøring med terskelfilter.

Strengen brukt for å representere DNAet er identisk i alle tre kjøringer. Denne forsøker å skape en forskjellig sammensetning av de fire elementer, men siden det er ønskelig å bruke samme streng i alle tre forsøk er denne ikke generert tilfeldig. Strengen repeterer 32 elementer, noe som gir en høy nok forskjell til at resultatene som genereres ikke er ensformige eller identiske, og lengden på 32 elementer tillater stor variasjon i kombinasjonen av elementer.

	A	C	G	T
1	1	2	3	4
2	1	2	3	4
3	1	2	3	4
4	1	2	3	4
5	1	2	3	4
6	1	2	3	4
7	1	2	3	4
8	1	2	3	4

Tabell 7.2: Vektingsmatrisen som ble brukt ved kjøring med terskelfilter.

Basert på disse tallene kan kretsens effektivitet teoretiseres rundt. Ved beste ytelse, altså når kretsen får eksekvere uavbrutt, kan en ytelse på 200 millioner elementer per sekund ventes. Dette ettersom uavbrutt eksekvering tilsvarer ett element per sykel og kretsens klokkehastighet er 200MHz.

Dårligste hastighet er avhengig av tilstanden til minnet i writer-modulen og ved fullt minne tilsvarer ytelsen 200 millioner elementer per sekund / 9 (antallet elementer som prosesseres ved tomt minne) $*$ 8 (antallet elementer som kan prosesseres ved fullt minne). Med andre ord er dårligste ytelse 1/9 under beste ytelse. Dette er fordi hele kretsen må stoppes ved fullt minne i writer-modulen til minnet igjen er klart til å ta imot data. Kretsen er i stand til å prosessere 9 elementer, mens den på samme tid kun klarer å skrive 8 elementer til CPU. Dette tilsvarer 200.000.000 / 9 $*$ 8 = 177.777.776 elementer per sekund. Filteret kan altså øke ytelsen til beste ytelse hvis hvert 9. resultat blir forkastet, og disse er spredt utover med noenlunde faste intervall. Ettersom writer-modulen har ett ganske stort buffer kan den tolerere en viss samling i DNA-strengen av disse forkastede resultater, så allerede på en så liten utsorteringsrate finnes det muligheter for å oppnå beste ytelse. Ved en høyere forkastningsrate stiger sjansene for å oppnå beste ytelse.

7.1.2 Summasjonsfilter

I likhet med terskelfilteret ble ingen problemer oppdaget under simulering av summasjonsfilteret som egenstående modul. Også i likhet med terskelfilteret ble det oppdaget et problem under simulering av systemet som helhet. Dette problemet gjaldt filterets funksjonalitet som en del av systemet og endringen ble gjort utenfor filteret, i user_app.

Filterets innsignaler er som følger:

- `data_in`: Signalet bærer inndata til filteret fra pwm-modulen.
- `index_in`: Signalet bærer i likhet med `data_in` inndata fra pwm-modulen.
- `valid`: Signalet forteller hvorvidt inndata fra pwm-modulen er gyldig.
- `reset_n`: Signalet setter igang nullstilling av alle verdier.

- `filter_comp_finished`: Signalet setter igang skriving av sum til writermodul etter endt eksekvering.

I denne filterimplementasjonen er det tre signaler som styrer modulens tilstand. Signalet `reset_n` nullstiller kretsen og alle signaler, både interne og utgående. Heller ikke her vil noen annen operasjon skje så lenge dette signalet er aktivt. Signalet `valid` tilsier at `data_in` er gyldig og kan summeres med `sum`. `filter_comp_finished` utløser en skriving av `sum` til writer-modulen. Eventuelle innkommende data, gyldige eller ikke, vil bli ignorert. Dette fordi `filter_comp_finished` kun skal forekomme etter endt eksekvering, og det skal da ikke lenger forekomme gyldige data i kretsen.

Simuleringens dekningsgrad er god, basert på det lave antallet tilstander som kan forekomme. De fire forsøk gjort i simuleringen dekker de nødvendige hendelser og tilstander. Det mest spennende forsøket var forsøk 3, som undersøkte overflytssikringen i modulen. Her fremkommer det at overflyt senest skjer når `sum` er lik 141.012.366.262.272, som tilsvarer $2^{47} + 2^{38}$.

Kjøring av summasjonsfilteret viser også de samme tendensene. Fire kjøringer ble foretatt. De første tre hadde identisk vektingsmatrise med den som er gjengitt i tabell 7.2. Disse varierte i antall elementer de prosesserte. Tabell 7.3 gjengir forholdet mellom antall elementer som ble prosessert, sum og antall resultater skrevet til CPU.

Antall elementer	Sum	Antall resultat
100	1270	1
1100	14474	1
262.144	3.473.347	1
2621.44	2.957.076.369.801.187	22

Tabell 7.3: Antall elementer, sum og antall resultat ved kjøring med summasjonsfilter.

I kjøring fire ble vektingsmatrisens verdier drastisk økt. Disse er gjengitt i tabell 7.4. Resultatet var, som tabell 7.3 vitner om, en drastisk økning i antall resultater skrevet fra FPGA. Dette vitner om at overflytsbeskyttelsen implementert i filtermodulen fungerer. Hver av de 22 resultatene, bortsett fra det siste som er et resultat av at eksekveringen er gjennomført, tilsvarer omtrent $2^{47} + 2^{33}$. 2^{33} tilsvarer omtrent verdien til ett element fra vektingsmatrisen.

	A	C	G	T
1	10000000000	10000000001	10000000002	10000000003
2	10000000000	10000000001	10000000002	10000000003
3	10000000000	10000000001	10000000002	10000000003
4	10000000000	10000000001	10000000002	10000000003
5	10000000000	10000000001	10000000002	10000000003
6	10000000000	10000000001	10000000002	10000000003
7	10000000000	10000000001	10000000002	10000000003
8	10000000000	10000000001	10000000002	10000000003

Tabell 7.4: Vektingsmatrisen som ble brukt ved kjøring med terskelfilter.

Strengen brukt for å representere ett DNA er identisk med den beskrevet i 7.1.1.

Ytelsen til summasjonsfilteret er alltid lik beste ytelse, altså 200 millioner elementer prosessert per sykel. Dette fordi filteret i svært stor grad begrenser antallet elementer som skrives fra FPGA til CPU.

7.2 Flerkjerneimplementasjon

Det ble dessverre ikke påbegynt en konkret flerkjerneimplementasjon, men enkjernesystemet har blitt oppdatert med en ny writer-modul som er mye bedre egnet til en flerkjerne-variant. Denne nye modulen har redusert brikkens overflateareal betraktelig, som påpekt i 5.2.1, og den er også med sin mye større kapasitet mye bedre egnet til å håndtere en større datamengde enn den tidligere implementasjonen. Det som mangler til en flerkjerneimplementasjon er en sammenknytting av datastier gjennom de forskjellige kjernene, som nødvendigvis må være samlet til en sti før de kommer til writer-modulen. Teoretisk sett går det å ha flere writer-moduler og la en megler styre hvilken som til enhver tid har tilgang til skrive-linjer ut av FPGAen, men dette er en langt mer kompleks og også langt mindre effektiv løsning enn å samle datastiene før writer-modulen.

Hvorvidt filteret skal inkluderes i hver av kjernene er noe jeg har redegjort for i 4.2.2, og her kommer det frem at et klart valg mellom å inkludere filter i kjernen eller å la det stå alene ikke finnes. Men gitt terskelfilterets lave kompleksitet kan det lett inkluderes i hver enkel kjerne. Da kan man tenke seg at writer-modulen blir utvidet med et lite buffer for å samle sammen dataene og legge de klar til overføring til CPU. Når det gjelder summasjonsfilteret er det naturlig for meg å se dette som en delt løsning, med samlebånds-basert summering. Det vil gi en effektiv summering, ettersom antallet summasjoner vil være $log_2 N$, hvor N er antall kjerner, i motsetning til en løsning med separate filter hvor antallet summasjoner kommer opp i N, ettersom det er N kjerner.

Inkluderingen av pwm-modulen i hver og en av kjernene er selvsagt. Denne modulen okkuperer en relativt stor del av brikkens overflate-areal, men er samtidig den eneste av de større enhetene som må dupliseres. Samtidig fremkommer det av syntese-verktøyets utskrift at denne modulen muligens kan endres litt for å la den bedre tilpasse seg makroer. Makroer er ressurser og ferdige implementasjoner av komponenter (alt fra små, enkle komponenter til større komponenter, som for eksempel BRAM) levert av Xilinx. Makroer er svært effektive, og hvis man tilpasser konstruksjonen sin slik at synteseverktøyet klarer å identifisere disse, kan konstruksjonens areal minskes. For eksempel er skiftregister (også nevnt nedenfor) en slik komponent.

En mer plasseffektiv løsning i forhold til PWM-tabellen hadde også vært gunstig, i og med at denne er den mest plasskrevende delen av pwm-modulen, i den grad den kan sies å tilhøre denne. I nåværende system er PWM-tabellen en del av user_app-modulen, men i en flerkjerneimplementasjon vil en flytting av tabellen til pwm-modulen være gunstig, tatt i betraktning at user_app-modulen ikke skal dupliseres.

Reader-modulen, som leser data fra minnet og sender de til pwm-modulen må rekonstrueres på en slik måte at den kan spre data til flere kjerner samtidig. Ett forslag

er å lage et skiftregister, som kan mate alle kjerner. Xilinx [15] tilbyr plassmessig og effektivitetsmessig svært gunstige implementasjoner av skiftregistere. Implementasjonsdetaljer rundt spredningen av data er et viktig punkt i implementasjonen og en god løsning må her finnes. En god egenskap som denne løsningen har er at dataene blir spredd på en sammenflettet måte, slik det er beskrevet i 4.2.2.

Å inkludere readeren i den dupliserte kjernen vil være et begrensende punkt, ettersom modulen leser fra SRAM-blokkene og disse kun kan overføre 72 bit (64 bit data og 8 paritetsbit) hver sykel per blokk. Dermed begrenses antallet kjerner til antallet SRAM-blokker, altså 4. Denne løsningen har ikke den egenskap at dataene blir distribuert på en sammenflettet måte. Her vil hver kjerne operere på sitt eget unike datasett, og det må dessuten taes grep for å spre dataene på en slik måte at DNA-strengen blir prosessert i sin helhet. Med N elementer i SRAM-blokk 1 må SRAM-blokk 2 inneholde elementer $N - 7$ til $N * 2 - 7$. SRAM-blokk 3 må inneholde elementer $N * 2 - 14$ til $N * 3 - 14$ og SRAM-blokk 4 må inneholde $N * 3 - 21$ til $N * 4 - 21$. Dette fordi hver kjerne ikke kan prosessere de siste 7 elementene i sitt datasett ettersom pwm-vinduet (se 2.3) inneholder 8 elementer.

Modulen user_app er topp-modul, og instansierer de andre modulene. Denne er det derfor ikke aktuelt å inkludere i de dupliserte kjernene.

Med hensyn til arealeffektivitet syntetiserer systemet etter prosjektslutt til ca 8% av FPGAens totale ressurser. Å tro at man kan utnytte noe særlig mer enn 50% av brikkens totale ressurser er urealistisk, ettersom det kreves forskjellige mengder av forskjellige ressurser. Dessuten skal det hele kunne plasseres på brikken under syntetisering, og det kan medføre at endel ressurser på brikken blir utilgjengelig for konstruksjonen. Likevel er det min mening at det er realistisk å oppnå 8 kjerner i parallell. Dette fordi ikke hele systemet skal dupliseres, kun deler. Den klart mest plasskrevende modulen er user_app, deretter følger pwm-modulen. Modulen user_app står alene for omtrent halvparten av arealet. Dette fordi det er her alle signaler som rutes videre ned i undermodulene instansieres. I tillegg kobler denne modulen systemet sammen med kommunikasjonsmodulene `RTCore` og `QDRCore` fra Cray. Ved en flerkjerne-implementasjon vil denne modulen vokse noe, men dens arealforbruk vil ikke dobles slik som modulene som inkluderes i den dupliserte kjernen. Altså vil user_app og reader-modulen vokse noe, mens pwm og filter-modul vil bli duplisert. Dette vil trolig ikke tilsvare 8% av ressursene $*$ 8 kjerner $=$ 64% av ressursene. Det er vanskelig å anslå akkurat hvor store ressurskrav en slik implementasjon vil ha, men jeg tror den vil være endel lavere enn 64, kanskje rundt 50%. Men dette er ren gjetting.

7.3 Rammeverk

Som utskrift fra kjøring med terskelfilter viser er antallet resultater høyere enn tidligere antatt. Vektingsmatrise og DNA brukt i testene er begge to overlevert av Geir Kjetil Sandve ved IDI. DNAet er kromosom 21 fra det menneskelige genom. Ifølge han skal en eksekvering med disse to og en terskelverdi på 3 gi treff på noen promille av posisjonene i DNAet. Tabell 7.5 viser antall elementer, terskelverdi, antall treff, treffrate og

tidsforbruk for de to kjøringene.

Antall elementer	Terskelverdi	Antall resultat	Treffrate	Tidsforbruk
15.756.742	3	57.120	0.36	0m15.403s
63.026.989	3	242.704	0.38	1m1.138s
15.756.742	5.1	2767	0.01	0m5.419s

Tabell 7.5: Antall elementer, terskelverdi, antall resultat, treffrate og tidsforbruk ved kjøring med rammeverk og ekte DNA og vektingsmatrise. Tidsangivelsene kan variere noe.

Som vi ser av tabell 7.5 gir kjøring 1 0.36% treff i kromosomet, akkurat slik Sandve forventet. Dette er basert på søk han har gjort tidligere i programvare, men kun på en mindre del av kromosomet. Ved kjøring med kromosomet sekvensielt fire ganger øker treffraten med kun 0.02%. Samtidig ser vi at hvis vi sammenligner antall treff og tidsforbruk er det en sterk sammenheng. Dette er fordi lagring av resultater til fil er den mest tidskostbare operasjonen i rammeverket, og denne kostnaden øker i takt med antall resultater.

Som vi ser av tidsangivelsene, selv om disse er unøyaktige og gjelder kun den spesifikke eksekveringen, er tidsforbruket relativt lavt. Ved å implementere lagring av resultater mer effektivt kan tidsforbruket senkes, ettersom lagring av resultater er en stor andel av tidsforbruket. Det er avhengig av antallet resultater som skal skrives til fil, men vil i de fleste tilfeller (fra 0.1%) representere en klar hoveddel av tidsforbruket. Tabell 7.6 viser hvordan tidsforbruket øker i takt med antall resultater og ikke så mye i sammenheng med økende antall elementer. Terskelen brukt er 3.

Antall elementer	Antall resultater	Med lagring	Uten lagring
15.756.742	57120	0m14.853s	0m4.677s
63.026.989	242.704	0m58.033s	0m13.201s
126.053.985	485.408	1m53.615s	0m26.604s

Tabell 7.6: Antall elementer, antall resultater og tidsforbruk med og uten lagring av resultater. Tidsangivelsene kan variere noe.

Mellom overføring av data og lesing av resultater i rammeverket brukes kommandoen `usleep(200000)`. Denne kommandoen fører til at prosessen legger seg til å sove i minimum 0.2 sekunder. Vanligvis vil dette tilsvare nøyaktig 0.2 sekunder, men uforutsette hendelser kan føre til at prosessen sover litt lenger. Grunnen til denne soveperioden er at FPGAen må kjøre ferdig før lesing av resultater begynner. Som nevnt i 7.1.1 prosesserer FPGAen minimum 177.777.776 elementer per sekund. Altså prosesserer den minimum 35.555.555 elementer på 0.2 sekund. Dette representerer det dobbelte av antall elementer som kan prosesseres av FPGAen på en eksekvering, og tilfører en sikkerhetsmargin slik at man vet at FPGAen er ferdig, og ikke har blitt forsinket av eventuelle uforutsette tilstander eller oppstartskostnader. Ettersom FPGAen ikke har noen måte å signalisere at den er ferdig på, for eksempel ved hjelp av et signal eller ett flagg, må rammeverket vente til man er sikker på at FPGAen er ferdig.

Ved større datamengder som krever prosessering i bolker blir FPWM-systemet nullstilt mellom hver bolk. Dette for å forsikre at verdiene i systemet er oppdatert og at systemet

er klart til prosessering.

Kapittel 8

Konklusjon og videre arbeid

I dette kapittelet skal arbeidet konkluderes i 8.1, før noen tanker om arbeidets fremtid kommer i 8.2.

8.1 Konklusjon

8.1.1 Filterimplementasjon

Hensikten bak implementasjon av filter-moduler var i første rekke å redusere mengden resultater, ved å fjerne uønskede eller uinteressante resultater, slik at verdien av resultatene ble større. En annen hensikt var også å øke ytelsen til systemet ved å begrense kommunikasjon mellom FPGA og CPU og sørge for at kretsen fikk arbeide uten avbrytelse i størst mulig grad.

Når det gjelder terskelfilteret er disse to punktene svært avhengig av at terskelnivået er satt strengt nok, men som jeg har blitt forsikret av Geir Kjetil Sandve ved IDI skal et filter kun slippe igjennom høyst 1% av resultatene. På denne måten får man høyest mulig ytelse fra kretsen.

Summasjonsfilteret begrenser kommunikasjonen i svært stor grad ved å skrive ut kun noen få resultater, og vil derfor alltid sørge for høyest mulig ytelse.

Utviklingen av filterene har gått gjennom flere faser. Planlegging og foreløpig implementasjon ble unnagjort uten store problemer, med unntak av vanskeligheter med kompabilitet mellom Xilinx ISE og programvare på Cray XD1, som beskrevet i 1.4. Simuleringsfasen fant flere problemer med implementasjonene, men disse ble korrigert. Under kjøring ble nok et problem funnet, og også dette ble korrigert. Resultatet ved prosjektslutt er to moduler, hvor en modul begrenser antall resultater ved å fjerne mindre interessante verdier basert på en terskelverdi, og en modul som summerer alle verdier. En ønsket effekt av begresningen av resultater er økt ytelse i systemet som en følge av lavere forbruk av båndbredde mellom FPGA og CPU.

8.1.2 Flerkjerneimplementasjon

En flerkjerneimplementasjon med åtte kjerner vil kunne prosessere et enorm antall elementer per sekund. Dessverre er det problemer med datatilførselen som begrenser dette tallet i svært stor grad. Enkjerneimplementasjonen som er et resultat av dette prosjektet kan kjøre uavbrutt i under 0.1 sekund før nye data må lastes inn. I rammeverket er denne verdien riktignok satt til 0.2 sekund for å ta høyde for dårligere ytelse og eventuelle oppstartskostnader utover lasting av data. Dette er diskutert i 7.3. En flerkjerne-implementasjon vil redusere denne mulige kjøretiden ytterligere, med samme faktor som antall kjerner implementert, ettersom alle kjernene prosesserer samme datasett.

En annen løsning, svært kort nevnt i 3.1.3, er en flernode-løsning. Dette innebærer å kjøre programmet på flere noder samtidig, noe som er en mulighet på CRAY XD1. Ved en slik fremgangsmåte må rammeverket utvides til å inkludere distribuering av data på tvers av nodene ved hjelp av Message Passing Interface-teknologi. MPI kan være svært avansert, men akkurat dette problemet er ikke avansert, ettersom man kun skal dele data mellom nodene, for så å samle inn resultatene igjen. Da kan man bruke FPWM-systemet slik det foreligger ved prosjektslutt, og øke hastigheten med en faktor tilsvarende antall noder man bruker. Det vil forekomme noe oppstartstid, men ved bruk av en flernodeløsning kan man også parallellisere lesing av DNAet, noe som kan øke ytelsen samlet sett, ettersom hver instans av FPWM-systemet skal lese N/M byte av DNAet, hvor N er lengden på DNAet og M er antallet noder. Ved å bruke en hovednode, som regner ut hvor mye som skal prosesseres og aktiverer X antall noder deretter, kan forholdet mellom ytelse og ressursforbruk optimaliseres. Dette forutsetter at datamengden som skal prosesseres er tilstrekkelig stor, for eksempel kan størrelser over 16MB rettferdiggjøre kjøring på 2 noder. Generelt kan man si at størrelser over 16MB $*$ M kan rettferdiggjøre kjøring på $M + 1$ noder, men den tidsmessige fordelen bruker opplever inntreffer i realiteten først ved større datamengder. Dette fordi en enkel kjøring tar svært kort tid, og behovet for å minske denne er av mer teoretisk art en praktisk.

Ved implementasjon av konvertering av filer (nevnt i 8.2.1) vil en MPI-implementasjon fungere optimalt, ettersom det ikke finnes noen områder i en konvertert fil som ikke er kartlagt. Hvis man skal prosessere en ikke konvertert DNAfil parallelt, kan man risikere at en prosess får svært lite data å jobbe med som en følge av store områder i DNAet som ikke er kartlagt. Ved hjelp av konvertering før kjøring kan en parallell implementasjon gi beste ytelse.

Et annet punkt som taler for en flernodeimplementasjon istedet for en flerkjerneimplementasjon er utviklingsprosessen for de to alternativene. Som tidligere sagt er tidskostnaden ved utvikling av en flernodeimplementasjon lav, ettersom det kun er rammeverket som må modifiseres. Disse modifikasjonen er heller ikke komplekse og en slik implementasjon vil være relativt enkel. Utviklingen av et flerkjernesystem vil derimot være en svært tidsmessig kostbar implementasjon, med langt høyere kompleksitet.

En kombinasjon av de to nevnte parallelliseringsmetoder er også en mulighet.

Bakgrunnen til problemet med overføring av data til FGPAen vil jeg si er databredden valgt i kodingen av DNA-strengen. Hvert element er i FPWM-systemet kodet med 8 bits. Tatt i betraktning at ett DNA består av 4 elementer ser vi at hvert element kunne vært kodet med 2 bits, noe som ville gitt en langt høyere tetthet. For hver 64 bits streng som overføres til FPGA med oppløsningen i FPWM-systemet overføres 8 elementer. Med en oppløsning på 2 bits per element kunne 32 elementer overføres i hver 64 bits streng. Dette ville gitt en firedobling i antallet elementer i minnet ved oppstart av eksekvering. Dette er fortsatt ikke nok data til å kunne tillate FPWM-systemet et fullt sekund uavbrutt kjøring, men det ville minsket det administrative arbeidet rundt overføring av data til FPGAen. Ikke minst ville man, avhengig av størrelsen på datasettet, unngått 3 av 4 nullstillinger av FPGAen som skjer i rammeverket før hver dataoverføring (se 7.3 for detaljer).

8.1.3 Rammeverk

Rammeverket ved oppstart av prosjektet var svært begrenset. Utover å starte FPGAen og laste FPWM-systemet hadde det nesten ikke funksjonalitet. Videreutviklingen av dette rammeverket var svært viktig for å muliggjøre praktisk bruk av systemet, noe som jo selvsagt er det endelige målet. I den tilstand rammeverket er ved prosjektslutt er systemet tilgjengelig for søk i reelle DNA.

I hovedsak har tre ting blitt implementert.

- Lesing av DNA og vektingsmatrise fra fil, noe som også innbefatter omkoding for å tilpasse data til FPWM-systemet
- Mulighet for kjøring av store datamengder
- Lagring av resultater til fil

Imidlertidig representerer disse tre punktene en stor videreutvikling av rammeverket. De to første punktene bød på noen utfordringer, spesielt var omkoding og prosessering av vektingsmatrisen problematisk. Men spesielt takket være hjelp fra Geir Kjetil Sandve ved IDI løste disse problemene seg og ved prosjektslutt fungerer rammeverket slik det skal. Det er nå mulig å kjøre søk i DNA med vektingsmatriser og få ut resultater av interesse.

8.1.4 Prosjektets verdi

Da jeg gikk inn i dette prosjektet hadde Lars Krutådal en ikke fungerende implementasjon av FPWM. Denne var også dessverre svært mangelfullt dokumentert. Det var en stund tvil rundt verdien av denne implementasjonen, ettersom Krutådal ikke fant feilen som forårsaket manglende såvel som dupliserte resultater. Men etter implementasjon av ny writer-modul forsvant problemene med resultatene, og systemet begynte å oppføre seg optimalt.

Effektiviteten til systemet er svært høy. Som nevnt i 7.1.1 kan systemet optimalt prosessere 200 millioner elementer i sekundet, noe som er langt over hva et program kjørt på

en vanlig CPU klarer. Problemer med overføring av data til CPU kan føre til at ytelsen faller til 177.777.776 elementer per sekund. Likevel, disse problemene tatt i betraktning, er systemets ytelse svært god. Dette gjelder særlig ettersom implementasjonen av filtermoduler forsøker å opprettholde høyeste ytelse.

Arbeidet som er gjort i prosjektet har noen svakheter. Spesielt den manglende dokumentasjonen er et ankepunkt, men også verifisering av moduler er en stor mangel. Det finnes ingen dokumentasjon på hvilke tester som er gjort på de forskjellige delene av systemet, bortsett fra de tester jeg har gjort på de deler jeg har implementert. Men som ytelsen gjenspeiler er den valgte løsningen god. I løpet av arbeidet gjort i forbindelse med utviklingen av filter-moduler og writer-modul har resten av systemet blitt testet, men ikke i tilstrekkelig grad til annet enn å si at det virker som de fungerer tilfredsstillende. Også kjøring med det ferdige system tilsier at det fungerer, men dette er heller ikke nok til å verifisere funksjonaliteten til hver enkelt modul.

For å forsøke å kompensere for manglende dokumentasjon har jeg skrevet en liten introduksjon for utviklere i prosjektet. Denne introduksjonen forsøker å kortfattet oppsummere oppbygningen av prosjektet, som for eksempel hvilke filer som inneholder designet, hvordan modulene henger sammen og grovt hvordan modulene er bygd opp. Det må vektlegges at denne filen er mine betraktninger etter å ha jobbet med systemet i lang tid, og ikke den første utvikleren, Lars Krutådal, sine ord. Derfor må informasjonen i denne filen tas med forbehold om at det kan være uriktig. Men det er ihvertfall en pekepinn, ment for å lette arbeidet for fremtidige utviklere i dette prosjektet. Denne filen finnes i D.1.

8.2 Videre arbeid

I dette underkapittelet skal jeg forsøke å gi tips til videre arbeid med systemet. Punktene vil også rangeres etter viktighetsgrad og kostnad i tid.

8.2.1 Gjenstående arbeid

Parallell implementasjon

Som nevnt i 8.1.2 kan en parallell implementasjon av rammeverket ved hjelp av MPI øke ytelsen betydelig. Ved å spre eksekveringen over flere noder kan ytelsen økes med en faktor lik antallet noder som brukes. Parallellisering av lesing av DNA og skriving av resultater kan øke ytelsen ytterligere ut over dette. Implementasjonen er redegjort for i 8.1.2, og jeg vil derfor ikke gå nærmere inn på dette her.

En parallell implementasjon er ikke vanskelig, og vil heller ikke kreve betydelig arbeid.

Web-grensesnitt

For å gjøre systemet lett tilgjengelig for brukere kunne et web-grensesnitt lages. Her kunne brukere lett legge inn jobber, med tilhørende data, og web-grensesnittet kunne kommunisere med rammeverket. Også presentasjon av data kunne her blitt implementert.

Vanskelighetsgraden tidskostnaden på dette avhenger av funksjonaliteten implementert, men i enkleste form er det ganske enkelt. Med dette menes en passordbeskyttet side hvor jobber kan lastes inn og startes. Hvilke presentasjoner av resultat som er ønskelig vet jeg ikke, men mulighetene er enorme.

Manglende partier

Slik rammeverket fungerer ved prosjektavslutning ignoreres alle områder i DNAet som ikke er kartlagt. Det vil si at det heller ikke tas hensyn til disse ved gjennomgang av resultater. En modifikasjon av rammeverket for å kompensere for dette er viktig. Følgende må gjøres:

- Endre index-verdien i resultater for å ta høyde for manglende partier i DNAet.
- Muligens forkaste verdier rett før manglende partier, ettersom disse er resultater av DNA både før og etter det manglende partiet.
- Ved konvertering (se punkt nedenfor) må informasjon om manglende partier lagres for bruk senere.

Dette punktet er ikke svært tidkrevende og er enkelt å implementere.

Konvertering av DNA

Ettersom rammeverket må konvertere DNAet før det kan overføres til FPGA kan en funksjon som konverterer ett DNA og lagrer det for senere bruk implementeres. Denne kan integreres i rammeverkets funksjonalitet, slik at hvis det ikke finnes en konvertert variant av DNAet som skal brukes blir en konvertert variant laget etterhvert som DNA konverteres for overføring til FPGA. Dette vil kunne øke ytelsen til rammeverket noe, ettersom man ved overføring kan lese 64 byte og sende de rett til FPGA, istedet for å måtte gå igjennom byte etter byte og konstruere en vektor for overføring til FPGA.

Vanskelighetsgraden på dette er relativt lav. Den største utfordringen er å gjøre det på en måte som sikrer at det riktige DNA blir brukt, spesielt hvis rammeverket brukes som et abstraksjonslag i samhandling med for eksempel et web-grensesnitt, slik at rammeverket ikke kommuniserer direkte med bruker. Viktigheten er heller ikke avgjørende.

Effektivisere lagring av resultater

Lagring av resultater er den mest tidkrevende operasjon i rammeverket, jamfør 7.3. Selv ved relativt små treffrater (0.1%) er lagring den mest tidkrevende operasjon. Hvis denne kunne vært implementert på en mer effektiv måte ville man kuttet ned tidsforbruket til FPWM-systemet. Et forslag her kan være å bruke en databasetilkobling for å lagre resultater. Dette vil også være en løsning som gjør dataene mer tilgjengelig for manipulering ved senere anledninger.

For å sikre effektiviteten til systemet er dette ganske viktig. Ved søk i det menneskelige genom (10^9 elementer) vil antallet resultater være høy selv om treffraten er lav.

Databasetilkobling

I tillegg til å laste data fra fil kan en databasetilkobling implementeres slik at DNA og/eller vektingsmatrise kan hentes fra en ekstern database. Databasetilkoblingen kan også utnyttes i punktet over. En kombinasjon mellom database og fil bør også være en mulighet.

Vanskelighetsgraden på dette er lav, og viktigheten for systemet er også lav.

Oppstartsvalg og avanserte konfigurasjoner

Ved å implementere flere oppstartsvalg kan funksjonaliteten til systemet utvides. For eksempel kan man oppgi navn på utdata-fil og man kan angi hvorvidt denne filen skal skrives over eller om det skal føyes til på slutten av denne. I kombinasjon med valgmuligheter for forskjellige kombinasjoner av flere DNA/flere vektingsmatriser kan avanserte søk kjøres uten stor utvidelse i rammeverket som foreligger etter prosjektslutt.

Vanskelighetsgraden på dette avhenger av hvilke muligheter som implementeres. Viktighetsgraden avhenger også av det.

Utredning rundt ImpulseC og VHDL

Impulse C, beskrevet i 3.3.3, tilbyr en lettere måte å benytte FPGA som applikasjonsakselerator ved å innlemme maskinvareutvikling i programvareutvikling. En utredning med bakgrunn i FPWM-systemet kunne kartlagt forskjellene mellom VHDL og Impulse C. Selv så er det min mening at å utvikle et system tilsvarende FPWM i ytelse ikke kan gjøres slik Impulse C forespeiler. Jeg klarer ikke å se for meg at ett verktøy skal kunne gjøre utviklingen av et slikt system så lett som [24] antyder. Men mine kunnskaper rundt Impulse C er begrenset og en sammenligning av ytelse, utviklingstid, kompleksitet og brukbarhet mellom FPWM og en Impulse C-implementasjon hadde vært interessant å se.

Impulse C støtter Cray XD1 som FPWM er utviklet på. Det foreligger dessverre ikke noen lisens for Impulse C per 7. juni 2007.

Del IV

Appendiks

Tillegg A

Ordliste og ordforklaringer

Her vil ord oversatt til norsk samt akronymer brukt i teksten forklares:

Brikke	: eng. chip, integrert krets.
Buntskriving	: eng. burstwrite, skriving av flere elementer i en operasjon.
CLB	: Configurable Logic Block, enhet i en FPGA.
CPU	: Central Processing Unit.
DMF	: Det Medisinske Fakultet ved NTNU.
Flerkjerne	: eng. multicore, flere instanser av en kjerne på en brikke.
Flettesortering	: eng. merge sort, sorteringsalgoritme som benytter fletting [25].
FPGA	: Field Programmable Gate Array, rekonfigurerbar maskinvare.
FPWM	: FPGA-implementasjon av PWM-algoritmen.
HDL	: Hardware Description Language.
IDI	: Institutt for Datateknikk og Informasjonsvitenskap ved NTNU.
Koblingsmatrise	: eng. switch matrix, koblingsenhet i en FPGA.
MAMA	: Massively parallel Acceleration of the Meme Algorithm, beskrevet i 3.2.
MEME	: Har ikke funnet navn, nevnt i 3.2.
Motiv	: motif, grunnlaget i en PWM-algoritme.
MPI	: Message Passing Interface, grensesnitt mellom prosesser.
Mykkjerne	: eng. soft-core, kjerne realisert i et HDL.
Nedkjøling	: eng. cool down, perioden etter bruk en ressurs er utilgjengelig.
NTNU	: Norges Teknisk Naturvitenskapelige Universitet.
Nøstede	: eng. Interleave, å arrangere data på en ikke kontinuerlig måte.
PAMM	: Parallel Acceleration of Motif Matching, beskrevet i 3.2.
ParaMEME	: Parallell implementasjon av MEME-algoritmen, nevnt i 3.2.
PMC	: Pattern Matching Chip, nevnt i 3.2.
PWM	: Position Weight Matrix, algoritme forklart i 2.3.
Skiftregister	: eng. shift register, registergrupperings-arkitektur.
SMP	: Symmetric Multi Processing, når to eller flere CPUer deler minne.
VHDL	: VLSIC Hardware Description Language, utviklingsspråk for maskinvare.

Tillegg B

Kildekode

B.1 Kildekode for terskelfilter

B.1.1 Kildekode for terskelfilter

```
--------------------------------------------------------------------------------
-- This file implements the threshold functionality of the filter module.
-- Threshold + 1 = the smallest value sent to writer.
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.user_pkg.all;


entity filter is
   port (
    reset_n      : in  std_logic;
    user_clk     : in  std_logic;
    valid        : in  std_logic;
    stall      : in  std_logic;
    data_in      : in  std_logic_vector(37 downto 0);
    index_in     : in  std_logic_vector(25 downto 0);
    threshold    : in  std_logic_vector(37 downto 0);

    data_valid     : out std_logic;
    data_out     : out std_logic_vector(63 downto 0)
   );
end filter;

architecture rtl of filter is
  signal thresh : std_logic_vector(37 downto 0);
  signal thresh_is_set : std_logic;
begin
  do_stuff : process (reset_n, user_clk)
  begin
    if reset_n='0' then
      data_valid <= '0';
      data_out <= (others => '0');
      thresh <= (others => '0');
      thresh_is_set <= '0';
    elsif user_clk='1' and user_clk'event then
      if thresh_is_set/='1' then
        if threshold > 0 then
          thresh <= threshold;
          thresh_is_set <= '1';
        end if;
      end if;
      data_out <= index_in & data_in;
      if thresh(30 downto 0) < data_in(30 downto 0) and valid='1' then
        if thresh(31) = '0' then
          data_valid <= not data_in(31);
        else
          data_valid <= '1';
        end if;
```

```
      else
        data_valid <= '0';
      end if;
    end if;
  end process do_stuff;

end rtl;
```

B.1.2 Kildekode for testbenk til terskelfilter

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   13:07:45 05/10/2007
-- Design Name:   filter
-- Module Name:   /home/dmlab/perandg/kode/fpwmv3/src/80-0037_vhdl/par/xc2vp50/filtert_tb.vhd
-- Project Name:  top
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: filter
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY filtert_tb_vhd IS
END filtert_tb_vhd;

ARCHITECTURE behavior OF filtert_tb_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT filter
  PORT(
    reset_n : IN std_logic;
    user_clk : IN std_logic;
    valid : IN std_logic;
    stall : IN std_logic;
    data_in : IN std_logic_vector(37 downto 0);
    index_in : IN std_logic_vector(25 downto 0);
    threshold : IN std_logic_vector(37 downto 0);
    data_valid : OUT std_logic;
    data_out : OUT std_logic_vector(63 downto 0)
    );
  END COMPONENT;

  --Inputs
  SIGNAL reset_n :  std_logic := '0';
  SIGNAL user_clk :  std_logic := '0';
  SIGNAL valid :  std_logic := '0';
  SIGNAL stall :  std_logic := '0';
  SIGNAL data_in :  std_logic_vector(37 downto 0) := (others=>'0');
  SIGNAL index_in :  std_logic_vector(25 downto 0) := (others=>'0');
  SIGNAL threshold :  std_logic_vector(37 downto 0) := (others=>'0');

  --Outputs
  SIGNAL data_valid :  std_logic;
  SIGNAL data_out :  std_logic_vector(63 downto 0);

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut: filter PORT MAP(
    reset_n => reset_n,
    user_clk => user_clk,
    valid => valid,
    stall => stall,
    data_in => data_in,
    index_in => index_in,
```

```
    threshold => threshold,
    data_valid => data_valid,
    data_out => data_out
  );

  tb : PROCESS
  BEGIN

    -- Wait 100 ns for global reset to finish
    reset_n <= '0';
    wait for 100 ns;

    -- Place stimulus here

---------------------------------------------
--             Threshold testbench         --
---------------------------------------------

    reset_n <= '1';
    -- Test that the threshold value is saved in the filter module
    report "Test that the threshold value is saved in the filter module (1/4)";
    threshold <= B"00000000000000000000000000000000000100";
    wait for 10 ns;

    -- Test that the threshold value discards the correct inputs
    report "Test that the threshold value discards the correct inputs (2/4)";
    valid <= '1';
    data_in <= B"00000000000000000000000000000000000000";
    while data_in < 18 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      wait for 1 ns;
    end loop;

    -- Test that data_valid (out) corresponds to valid (in) and threshold compliance
    report "Test that data_valid (out) corresponds to valid (in) and threshold compliance
         (3/4)";
    wait for 5 ns;
    valid <= '0';
    data_in <= B"00000000000000000000000000000000000000";
    wait for 5 ns;
    while data_in < 18 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      valid <= not valid;
      wait for 1 ns;
    end loop;

    wait for 10 ns;

    -- Test the functionality of reset_n with reload of values afterwards and that data_valid
          (out) corresponds to valid (in) and threshold compliance
    report "Test the functionality of reset_n with reload of values afterwards and that
         data_valid (out) corresponds to valid (in) and threshold compliance (4/4)";
    reset_n <= '0';
    threshold <= (others => '0'); -- Nullstill verdier som ville blitt nullstilt i andre
         moduler ved reset_n=0
    data_in <= (others => '0');
    index_in <= (others => '0');
    valid <= '0';
    stall <= '0';
    wait for 10 ns;
    reset_n <= '1';
    wait for 2 ns;
    threshold <= B"00000000000000000000000000000000001000";

    wait for 10 ns;

    valid <= '1';
    data_in <= B"00000000000000000000000000000000000000";
    while data_in < 18 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      wait for 1 ns;
    end loop;

    wait for 5 ns;
    valid <= '0';
    data_in <= B"00000000000000000000000000000000000000";
    wait for 5 ns;
    while data_in < 18 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      valid <= not valid;
      wait for 1 ns;
    end loop;

    wait; -- will wait forever
```

```
  END PROCESS;
END;
```

B.2 Kildekode for summasjonsfilter

B.2.1 Kildekode for summasjonsfilter

```
--------------------------------------------------------------------------------
-- This file implements the summation functionality of the filter module.
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.user_pkg.all;


entity filter is
   port (
    reset_n      : in   std_logic;
    user_clk     : in   std_logic;
    valid        : in   std_logic;
    stall      : in   std_logic;
    data_in      : in   std_logic_vector(37 downto 0);
    index_in     : in   std_logic_vector(25 downto 0);
    filter_comp_finished   : in   std_logic;

    data_valid     : out std_logic;
    data_out     : out std_logic_vector(63 downto 0)
   );
end filter;

architecture rtl of filter is
  signal sum : std_logic_vector(47 downto 0); -- The sum

begin

  do_stuff : process (reset_n, user_clk)
  begin
    if reset_n='0' then -- reset
      data_valid <= '0';
      data_out <= (others => '0');
      sum <= (others => '0');
    elsif user_clk='1' and user_clk'event then
      if filter_comp_finished='1' then -- if done
        if sum>0 then -- write sum if > 0
          data_out <= B"0000000000000000" & sum;
          data_valid <= '1';
          sum <= (others => '0');
        else
          data_valid <= '0';
        end if;
      elsif sum(47) = '1' then
        data_out <= B"0000000000000000" & sum;
        data_valid <= '1';
        if valid = '1' then -- if incoming value while writing
          sum <= B"0000000000" & data_in;
        else
          sum <= (others => '0');
        end if;
      elsif valid='1' then
        sum <= sum + data_in;
        data_valid <= '0';
      else
        data_out <= (others => '0');
        data_valid <= '0';
      end if;
    end if;
  end process do_stuff;

end rtl;
```

B.2.2 Kildekode for testbenk til summasjonsfilter

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
```

```
-- Create Date:    12:19:00 05/01/2007
-- Design Name:    filter
-- Module Name:    /home/dmlab/perandg/kode/fpwmv3/src/80-0037_vhdl/par/xc2vp50/filters_tb.vhd
-- Project Name:   top
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: filter
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
USE ieee.numeric_std.ALL;

ENTITY filters_tb_vhd IS
END filters_tb_vhd;

ARCHITECTURE behavior OF filters_tb_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT filter
  PORT(
    reset_n : IN std_logic;
    user_clk : IN std_logic;
    valid : IN std_logic;
    stall : IN std_logic;
    data_in : IN std_logic_vector(37 downto 0);
    index_in : IN std_logic_vector(25 downto 0);
    filter_comp_finished : IN std_logic;
    data_valid : OUT std_logic;
    data_out : OUT std_logic_vector(63 downto 0)
    );
  END COMPONENT;

  --Inputs
  SIGNAL reset_n :  std_logic := '0';
  SIGNAL user_clk :  std_logic := '0';
  SIGNAL valid :  std_logic := '0';
  SIGNAL stall :  std_logic := '0';
  SIGNAL data_in :  std_logic_vector(37 downto 0) := (others=>'0');
  SIGNAL index_in :  std_logic_vector(25 downto 0) := (others=>'0');
  SIGNAL filter_comp_finished : std_logic := '0';

  --Outputs
  SIGNAL data_valid :  std_logic;
  SIGNAL data_out :  std_logic_vector(63 downto 0);

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut: filter PORT MAP(
    reset_n => reset_n,
    user_clk => user_clk,
    valid => valid,
    stall => stall,
    data_in => data_in,
    index_in => index_in,
    filter_comp_finished => filter_comp_finished,
    data_valid => data_valid,
    data_out => data_out
  );

  tb : PROCESS
  BEGIN

    -- Wait 100 ns for global reset to finish
    reset_n <= '0';
    wait for 100 ns;

    -- Place stimulus here
---------------------------------------------
--          Summation filter testbench       --
---------------------------------------------
```

```
    reset_n <= '1';
    wait for 10 ns;

    -- Test functionality of filter_comp_finished and summation
    report "Test functionality of filter_comp_finished and summation (1/4)";
    data_in <= (others => '0');
    valid <= '1';
    while data_in < 100 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      wait for 1 ns;
    end loop;

    filter_comp_finished <= '1';
    valid <= '0';
    wait for 4 ns;
    filter_comp_finished <= '0';
    wait for 10 ns;

    -- Test valid input while filter_comp_finished is active
    report "Test valid input while filter_comp_finished is active (2/4)";
    data_in <= (others => '0');
    valid <= '1';
    while data_in < 104 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      if data_in >100 then
        filter_comp_finished <= '1';
      else
        filter_comp_finished <= '0';
      end if;
      wait for 1 ns;
    end loop;

    valid <= '0';
    wait for 4 ns;
    filter_comp_finished <= '0';
    wait for 10 ns;

    -- Test write of sum to ensure overflow prevention
    report "Test write of sum to ensure overflow prevention(3/4)";
    data_in <= (others => '1');
    valid <= '1';
    while data_out = 0 loop
      wait for 2 ns;
    end loop;

    filter_comp_finished <= '1';
    valid <= '0';
    wait for 4 ns;
    filter_comp_finished <= '0';
    wait for 10 ns;

    -- Test reset
    report "Test reset (4/4)";
    data_in <= (others => '0');
    valid <= '1';
    wait for 2 ns;
    while data_in < 100 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      wait for 1 ns;
    end loop;

    reset_n <= '0';
    valid <= '0';
    wait for 2 ns;
    reset_n <= '1';
    wait for 10 ns;

    data_in <= (others => '0');
    valid <= '1';
    wait for 2 ns;
    while data_in < 100 loop
      wait for 1 ns;
      data_in <= data_in + 1;
      wait for 1 ns;
    end loop;

    filter_comp_finished <= '1';
    valid <= '0';
    wait for 4 ns;
    filter_comp_finished <= '0';
    wait for 10 ns;

    wait; -- will wait forever
  END PROCESS;
```

```
END;
```

B.3 Kildekode for BRAM-basert writermodul

B.3.1 Kildekode for writermodul

```
-------------------------------------------------------------------------------
-- todo: make bypass functionality
-- todo: make sticky flush
-- note: after flush is flagged, no writes should be issued before the writer flags "empty".
-- note: base address setting should ONLY be attempted when the circuit is quiescent (empty,
    no other in flags)
-------------------------------------------------------------------------------

--------------------
-- Write and read means read from BRAM (and write to CPU) and write to BRAM.
--------------------

library ieee, unisim;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use work.user_pkg.all;
use unisim.VCOMPONENTS.all;

entity writer is
  port (
    reset_n        : in  std_logic;
    user_clk        : in  std_logic;
    valid          : in  std_logic;
    bypass         : in  std_logic;
    flush          : in  std_logic;
    set_base_addr    : in  std_logic;

    data_in        : in  std_logic_vector(63 downto 0);
    addr_in        : in  std_logic_vector(39 downto 3);

    stall          : out std_logic;
    empty          : out std_logic;
    overflow        : out std_logic;

    ureq_srctag        : in  std_logic_vector(4 downto 0); -- unused
    ureq_full      : in  std_logic;
    ureq_notag       : in  std_logic; -- unused
    ureq_addr      : out std_logic_vector(39 downto 3);
    ureq_size      : out std_logic_vector(2 downto 0);
    ureq_mask      : out std_logic_vector(7 downto 0);
    ureq_rw_n      : out std_logic;
    ureq_data      : out std_logic_vector(63 downto 0);
    ureq_ts        : out std_logic;
    ureq_byte_req    : out std_logic
  );
end writer;

architecture rtl of writer is
  component RAMB4_S16_S16
        port (
                ADDRA:  in std_logic_vector(7 downto 0);
                ADDRB:  in std_logic_vector(7 downto 0);
                DIA:    in std_logic_vector(15 downto 0);
                DIB:    in std_logic_vector(15 downto 0);
                WEA:    in std_logic;
                WEB:    in std_logic;
                CLKA:   in std_logic;
                CLKB:   in std_logic;
                RSTA:   in std_logic;
                RSTB:   in std_logic;
                ENA:    in std_logic;
                ENB:    in std_logic;
                DOA:    out std_logic_vector(15 downto 0);
                DOB:    out std_logic_vector(15 downto 0)
  );
  end component;

  signal reset           : std_logic; -- reset for brams
  signal read_enable         : std_logic; -- read_enable allows reading from brams
  signal blank_enable        : std_logic; -- blank enable disallows writin to b-port of every
     bram
  signal enable_brams_a, enable_brams_b      : std_logic; -- signals for enabling brams (doesn
     't really work)
  signal counter          : std_logic_vector(7 downto 0); -- counter of elements in bram
```

```
  signal temp_counter        : std_logic_vector(3 downto 0); -- used in reading from bram/
      writing to smp
  signal size          : std_logic_vector(2 downto 0); -- used in reading from bram/writing to
       smp
  signal read_address, write_address     : std_logic_vector(7 downto 0); -- addresses for
      brams
  signal r1, r2, r3, r4, w1, w2, w3, w4    : std_logic_vector(15 downto 0); -- signals for
      brams
  signal do1, do2, do3, do4, do5, do6, do7, do8    : std_logic_vector(15 downto 0); -- signals
       for brams
  signal addr_ptr          : std_logic_vector(39 downto 3); -- address for writing to smp
  signal valid_bram        : std_logic; -- valid signal when writing data to bram

begin

  write_main : process (reset_n, user_clk)
  variable inc_wa, inc_ra : std_logic;
  begin
    reset <= not reset_n; -- Modelsim won't allow negators on bram port map
    if reset_n='0' then
      ureq_addr      <= (others => '0');
      ureq_size      <= (others => '0');
      ureq_mask      <= (others => '1');
      ureq_rw_n      <= '0';
      ureq_data      <= (others => '0');
      ureq_ts        <= '0';
      ureq_byte_req  <= '0';

      write_address <= (others => '1');
      read_address <= (others => '0');
      counter <= (others => '0');
      temp_counter <= (others => '0');
      read_enable <= '0';
      enable_brams_a <= '0';
      enable_brams_b <= '0';
      blank_enable <= '0';

      addr_ptr <= (others => '0');

      stall <= '0';
      overflow <= '0';
      empty <= '1';

    elsif user_clk='1' and user_clk'event then
     -- Initiation
      enable_brams_a <= '1';
      blank_enable <= '0';
      inc_ra := '0';
      inc_wa := '0';

     -- Set the different state-signals
      overflow <= and_reduce(counter&valid);
      stall <= and_reduce(counter) or and_reduce(counter+valid) or ureq_full;
      empty <= not or_reduce(counter&valid);
      enable_brams_b <= or_reduce(counter&valid);

     -- write to brams
      if valid='1' then
        w1 <= data_in(63 downto 48); -- Distribute data to different brams
        w2 <= data_in(47 downto 32);
        w3 <= data_in(31 downto 16);
        w4 <= data_in( 15 downto 0);
        inc_wa := '1'; -- Increase write_address
        valid_bram <= '1';
      else
        valid_bram <= '0';
      end if;

     -- read from bram and write to smp
      if temp_counter = 8 then -- Burstwrite startup
        ureq_ts <= '1'; -- Assign signal for burstwrite startup
        ureq_rw_n <= '0'; -----||-----
        ureq_byte_req <= '0'; ----||-----
        ureq_addr <= addr_ptr; ------||------
        ureq_size <= not size;
        temp_counter <= (B"1" & size) + 1;
        ureq_data <= do2 & do4 & do6 & do8; -- Only even numbers because we are reading port
            b of every bram
        inc_ra := or_reduce(size+1);
        addr_ptr <= addr_ptr + 1;
      elsif temp_counter = 0 then -- No ongoing transfer to smp, possibly except for cooldown
          -phase
        if counter > 7 then -- If the memory is ready to transfer (no matter what flush is
            like)
          temp_counter <= B"1000";
          size <= B"000"; -- set size to 0 (which actually is 7)
          inc_ra := '1';
```

```
        elsif (flush = '1' and counter(2 downto 0) > 0) then -- If flush = 1 and 8>counter
            >0...
          temp_counter <= B"1000"; -- Initiate transfer
          size <= not counter(2 downto 0) + 1; -- set size to 1 and nr of elements in bram
          inc_ra := '1';
        elsif set_base_addr = '1' then -- If new addr_ptr ...
          addr_ptr <= addr_in;
        end if;
        ureq_ts <= '0'; -- Acts as cooldown fase in addition to setting temp_counter for next
             cycle
        ureq_addr <= (others => '0');
        ureq_size <= (others => '0');
        ureq_data <= (others => '0');
      else -- Push data
        ureq_data <= do2 & do4 & do6 & do8; -- Only even numbers because we are reading port
            b of every bram
        if temp_counter(2 downto 0) < 7 then -- If not the last cycle of the transfer,
            increment read_address
          inc_ra := '1';
        end if;
        addr_ptr <= addr_ptr + 1; -- Increment addr_ptr for every iteration of transfer to
            get correct next address
        temp_counter <= temp_counter + 1; -- Iterator
        ureq_ts <= '0';
        ureq_addr <= (others => '0');
        ureq_size <= (others => '0');
      end if;

      counter <= counter + inc_wa - inc_ra; -- Update counter and addresses
      read_address <= read_address + inc_ra;
      write_address <= write_address + inc_wa;

    end if;
  end process write_main;

bram1 : RAMB4_S16_S16
  port map (
    ADDRA    => write_address,
    ADDRB    => read_address,
    DIA => w1,
    DIB => r1,
    WEA => valid_bram,
    WEB    => blank_enable, -- null
    CLKA   => user_clk,
    CLKB   => user_clk,
    RSTA   => reset,
    RSTB   => reset,
    ENA => enable_brams_a,
    ENB => enable_brams_b,
    DOA => do1,
    DOB => do2
  );
bram2 : RAMB4_S16_S16
  port map (
    ADDRA    => write_address,
    ADDRB    => read_address,
    DIA => w2,
    DIB => r2,
    WEA => valid_bram,
    WEB    => blank_enable, -- null
    CLKA   => user_clk,
    CLKB   => user_clk,
    RSTA   => reset,
    RSTB   => reset,
    ENA => enable_brams_a,
    ENB => enable_brams_b,
    DOA => do3,
    DOB => do4
  );
bram3 : RAMB4_S16_S16
  port map (
    ADDRA    => write_address,
    ADDRB    => read_address,
    DIA => w3,
    DIB => r3,
    WEA => valid_bram,
    WEB    => blank_enable, -- null
    CLKA   => user_clk,
    CLKB   => user_clk,
    RSTA   => reset,
    RSTB   => reset,
    ENA => enable_brams_a,
    ENB => enable_brams_b,
    DOA => do5,
    DOB => do6
  );
bram4 : RAMB4_S16_S16
  port map (
```

```
    ADDRA    => write_address ,
    ADDRB    => read_address ,
    DIA => w4 ,
    DIB => r4 ,
    WEA => valid_bram ,
    WEB   => blank_enable , -- null
    CLKA  => user_clk ,
    CLKB  => user_clk ,
    RSTA  => reset ,
    RSTB  => reset ,
    ENA => enable_brams_a ,
    ENB => enable_brams_b ,
    DOA => do7 ,
    DOB => do8
  ) ;
end rtl ;
```

B.3.2 Kildekode for testbenk til writermodul

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   21:01:31 04/06/2007
-- Design Name:   writer
-- Module Name:   /home/dmlab/perandg/kode/fpwmv3/src/80-0037_vhdl/par/xc2vp50/writer_tbw.vhd
-- Project Name:  top
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: writer
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY writer_tbw_vhd IS
END writer_tbw_vhd;

ARCHITECTURE behavior OF writer_tbw_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT writer
  PORT(
    reset_n : IN std_logic;
    user_clk : IN std_logic;
    valid : IN std_logic;
    bypass : IN std_logic;
    flush : IN std_logic;
    set_base_addr : IN std_logic;
    data_in : IN std_logic_vector(63 downto 0);
    addr_in : IN std_logic_vector(39 downto 3);
    ureq_srctag : IN std_logic_vector(4 downto 0);
    ureq_full : IN std_logic;
    ureq_notag : IN std_logic;
    stall : OUT std_logic;
    empty : OUT std_logic;
    overflow : OUT std_logic;
    ureq_addr : OUT std_logic_vector(39 downto 3);
    ureq_size : OUT std_logic_vector(2 downto 0);
    ureq_mask : OUT std_logic_vector(7 downto 0);
    ureq_rw_n : OUT std_logic;
    ureq_data : OUT std_logic_vector(63 downto 0);
    ureq_ts : OUT std_logic;
    ureq_byte_req : OUT std_logic
    );
  END COMPONENT;

  --Inputs
  SIGNAL reset_n :  std_logic := '0';
```

```
  SIGNAL user_clk : std_logic := '0';
  SIGNAL valid : std_logic := '0';
  SIGNAL bypass : std_logic := '0';
  SIGNAL flush : std_logic := '0';
  SIGNAL set_base_addr : std_logic := '0';
  SIGNAL ureq_full : std_logic := '0';
  SIGNAL ureq_notag : std_logic := '0';
  SIGNAL data_in : std_logic_vector(63 downto 0) := (others=>'0');
  SIGNAL addr_in : std_logic_vector(39 downto 3) := (others=>'0');
  SIGNAL ureq_srctag : std_logic_vector(4 downto 0) := (others=>'0');

  --Outputs
  SIGNAL stall : std_logic;
  SIGNAL empty : std_logic;
  SIGNAL overflow : std_logic;
  SIGNAL ureq_addr : std_logic_vector(39 downto 3);
  SIGNAL ureq_size : std_logic_vector(2 downto 0);
  SIGNAL ureq_mask : std_logic_vector(7 downto 0);
  SIGNAL ureq_rw_n : std_logic;
  SIGNAL ureq_data : std_logic_vector(63 downto 0);
  SIGNAL ureq_ts : std_logic;
  SIGNAL ureq_byte_req : std_logic;

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut: writer PORT MAP(
    reset_n => reset_n,
    user_clk => user_clk,
    valid => valid,
    bypass => bypass,
    flush => flush,
    set_base_addr => set_base_addr,
    data_in => data_in,
    addr_in => addr_in,
    stall => stall,
    empty => empty,
    overflow => overflow,
    ureq_srctag => ureq_srctag,
    ureq_full => ureq_full,
    ureq_notag => ureq_notag,
    ureq_addr => ureq_addr,
    ureq_size => ureq_size,
    ureq_mask => ureq_mask,
    ureq_rw_n => ureq_rw_n,
    ureq_data => ureq_data,
    ureq_ts => ureq_ts,
    ureq_byte_req => ureq_byte_req
  );

  tb : PROCESS
  variable data : std_logic_vector(63 downto 0);
  variable counter : std_logic_vector(7 downto 0);
  BEGIN

    -- Wait 100 ns for global reset to finish
    wait for 100 ns;

    -- Place stimulus here

----------------------------------------------
--             writer testbench             --
----------------------------------------------


    reset_n <= '1';
    set_base_addr <= '1';
    addr_in <= B"0000000000000000000000000000000000001";
    wait for 1 ns;
    set_base_addr <= '0';
    wait for 1 ns;
    wait for 10 ns;
    valid <= '1';

    -- Test flushing of writer, write 6 elements and flush
    report "Test flushing of writer, write 6 elements and flush (1/11)";
    data := X"0000000000000000";
    while data < 6 loop
      data := data + 1;
      data_in <= data;
      wait for 1 ns;
      wait for 1 ns;
    end loop;
    valid <= '0';
    wait for 4 ns;
    flush <= '1';
    wait for 20 ns;
    flush <= '0';
```

```
wait for 2 ns;

-- Test flush, write 60 elements and flush
report "Test flush, write 60 elements and flush (2/11)";
data := X"0000000000000000";
valid <= '1';
while data < 60 loop
  data := data + 1;
  data_in <= data;
  wait for 1 ns;
  wait for 1 ns;
end loop;
valid <= '0';
wait for 2 ns;
flush <= '1';
wait for 30 ns;
flush <= '0';
wait for 2 ns;

-- Test transfer of 9 elements, so that one element is left behind after burstwrite
report "Test transfer of 9 elements, so that one element is left behind after burstwrite
     (3.1/11)";
valid <= '1';
data := X"0000000000000000";
while data < 9 loop
  data := data + 1;
  data_in <= data;
  wait for 1 ns;
  wait for 1 ns;
end loop;
valid <= '0';

-- Test transfer of 15 elements so that BRAMS are emptied
report "Test transfer of 15 elements so that BRAMS are emptied (3.2/11)";
wait for 20 ns;
data := X"0000000000000000";
valid <= '1';
while data < 15 loop
  data := data + 1;
  data_in <= data;
  wait for 1 ns;
  wait for 1 ns;
end loop;
valid <= '0';
wait for 30 ns;

-- Test transfer of 24 elements
report "Test transfer of 24 elements (4/11)";
data := X"0000000000000000";
valid <= '1';
while data < 24 loop
  data := data + 1;
  data_in <= data;
  wait for 1 ns;
  wait for 1 ns;
end loop;
valid <= '0';
wait for 40 ns;

-- Test transfer after reset, with a new base_address
report "Test transfer after reset, with a new base_address (5/11)";
reset_n <= '0';
wait for 2 ns;
reset_n <= '1';
wait for 2 ns;
set_base_addr <= '1';
addr_in <= B"00000000000000000000000000000000000010";
wait for 1 ns;
set_base_addr <= '0';
wait for 1 ns;

data := X"0000000000000000";
valid <= '1';
while data < 24 loop
  data := data + 1;
  data_in <= data;
  wait for 1 ns;
  wait for 1 ns;
end loop;
valid <= '0';
wait for 40 ns;

-- Test transfer of 64 elements
report "Test transfer of 64 elements (6/11)";
data := X"0000000000000000";
valid <= '1';
while data < 64 loop
  data := data + 1;
```

```
    data_in <= data;
    wait for 1 ns;
    wait for 1 ns;
  end loop;
  valid <= '0';
  wait for 100 ns;

  -- Test transfer of 64 elements and flush
  report "Test transfer of 64 elements and flush (7/11)";
  data := X"0000000000000000";
  valid <= '1';
  while data < 64 loop
    data := data + 1;
    data_in <= data;
    wait for 1 ns;
    wait for 1 ns;
  end loop;
  valid <= '0';
  wait for 10 ns;
  flush <= '1';
  wait for 10 ns;
  flush <= '0';
  wait for 20 ns;

  -- Test flush with 1, 2, ... , 7 elements
  report "Test flush with 1, 2, ... , 7 elements (8/11)";
  counter := B"00000000";
  while counter < 8 loop
    data := X"0000000000000000";
    valid <= '1';
    while data <= counter loop
      data := data + 1;
      data_in <= data;
      wait for 1 ns;
      wait for 1 ns;
    end loop;
    valid <= '0';
    wait for 10 ns;
    flush <= '1';
    wait for 10 ns;
    flush <= '0';
    wait for 20 ns;

    counter := counter + 1;
  end loop;

  -- Test that memory is cyclic
  report "Test that memory is cyclic (9/11)";
  data := X"0000000000000000";
  valid <= '1';
  while data < 270 loop
    data := data + 1;
    data_in <= data;
    wait for 1 ns;
    wait for 1 ns;
  end loop;
  valid <= '0';
  wait for 2 ns;
  flush <= '1';
  wait for 2 ns;
  flush <= '0';
  wait for 10 ns;

  -- Try with valid write and active flush
  report "Try with valid write and active flush (10/11)";
  data := X"0000000000000000";
  valid <= '1';
  while data < 270 loop
    data := data + 1;
    data_in <= data;
    if data = 130 then
      flush <= '1';
    end if;
    wait for 2 ns;
  end loop;
  valid <= '0';
  wait for 20 ns;
  flush <= '0';
  wait for 20 ns;


  -- Test for overflow/stall
  report "Test for overflow/stall (11/11)";
  data := X"0000000000000000";
  valid <= '1';
  while overflow = '0' loop
    data := data + 1;
    if stall = '0' then
```

```
        data_in <= data;
        valid <= '1';
      else
        valid <= '0';
      end if;
      wait for 1 ns;
      wait for 1 ns;
    end loop;
    valid <= '0';


    wait; -- will wait forever
  END PROCESS;
END;
```

B.3.3 Kildekode for testbenk til system-test

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   15:27:00 05/03/2007
-- Design Name:   user_app
-- Module Name:   /home/dmlab/perandg/kode/fpwmv3/src/80-0037_vhdl/par/xc2vp50/user_app_tb.
   vhd
-- Project Name:  top
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: user_app
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY user_app_tb_vhd IS
END user_app_tb_vhd;

ARCHITECTURE behavior OF user_app_tb_vhd IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT user_app
  PORT(
    reset_n : IN std_logic;
    user_clk : IN std_logic;
    user_enable : IN std_logic;
    rt_ready : IN std_logic;
    qdr_ready : IN std_logic;
    dr_1 : IN std_logic_vector(71 downto 0);
    dr_2 : IN std_logic_vector(71 downto 0);
    dr_3 : IN std_logic_vector(71 downto 0);
    dr_4 : IN std_logic_vector(71 downto 0);
    ureq_srctag : IN std_logic_vector(4 downto 0);
    ureq_full : IN std_logic;
    ureq_notag : IN std_logic;
    fresp_valid : IN std_logic;
    fresp_ts : IN std_logic;
    fresp_size : IN std_logic_vector(2 downto 0);
    fresp_srctag : IN std_logic_vector(4 downto 0);
    fresp_data : IN std_logic_vector(63 downto 0);
    freq_addr : IN std_logic_vector(39 downto 3);
    freq_size : IN std_logic_vector(3 downto 0);
    freq_srctag : IN std_logic_vector(4 downto 0);
    freq_mask : IN std_logic_vector(7 downto 0);
    freq_rw_n : IN std_logic;
    freq_ts : IN std_logic;
    freq_valid : IN std_logic;
    freq_data : IN std_logic_vector(63 downto 0);
    uresp_full : IN std_logic;
```

```
    debug : OUT std_logic_vector(25 downto 0);
    ar_1 : OUT std_logic_vector(19 downto 0);
    aw_1 : OUT std_logic_vector(19 downto 0);
    r_n_1 : OUT std_logic;
    w_n_1 : OUT std_logic;
    bw_n_1 : OUT std_logic_vector(7 downto 0);
    dw_1 : OUT std_logic_vector(71 downto 0);
    ar_2 : OUT std_logic_vector(19 downto 0);
    aw_2 : OUT std_logic_vector(19 downto 0);
    r_n_2 : OUT std_logic;
    w_n_2 : OUT std_logic;
    bw_n_2 : OUT std_logic_vector(7 downto 0);
    dw_2 : OUT std_logic_vector(71 downto 0);
    ar_3 : OUT std_logic_vector(19 downto 0);
    aw_3 : OUT std_logic_vector(19 downto 0);
    r_n_3 : OUT std_logic;
    w_n_3 : OUT std_logic;
    bw_n_3 : OUT std_logic_vector(7 downto 0);
    dw_3 : OUT std_logic_vector(71 downto 0);
    ar_4 : OUT std_logic_vector(19 downto 0);
    aw_4 : OUT std_logic_vector(19 downto 0);
    r_n_4 : OUT std_logic;
    w_n_4 : OUT std_logic;
    bw_n_4 : OUT std_logic_vector(7 downto 0);
    dw_4 : OUT std_logic_vector(71 downto 0);
    ureq_addr : OUT std_logic_vector(39 downto 3);
    ureq_size : OUT std_logic_vector(2 downto 0);
    ureq_mask : OUT std_logic_vector(7 downto 0);
    ureq_rw_n : OUT std_logic;
    ureq_data : OUT std_logic_vector(63 downto 0);
    ureq_ts : OUT std_logic;
    ureq_byte_req : OUT std_logic;
    fresp_enable : OUT std_logic;
    freq_enable : OUT std_logic;
    uresp_ts : OUT std_logic;
    uresp_size : OUT std_logic_vector(3 downto 0);
    uresp_srctag : OUT std_logic_vector(4 downto 0);
    uresp_data : OUT std_logic_vector(63 downto 0)
    );
  END COMPONENT;

  --Inputs
  SIGNAL reset_n : std_logic := '0';
  SIGNAL user_clk : std_logic := '0';
  SIGNAL user_enable : std_logic := '0';
  SIGNAL rt_ready : std_logic := '0';
  SIGNAL qdr_ready : std_logic := '0';
  SIGNAL ureq_full : std_logic := '0';
  SIGNAL ureq_notag : std_logic := '0';
  SIGNAL fresp_valid : std_logic := '0';
  SIGNAL fresp_ts : std_logic := '0';
  SIGNAL freq_rw_n : std_logic := '0';
  SIGNAL freq_ts : std_logic := '0';
  SIGNAL freq_valid : std_logic := '0';
  SIGNAL uresp_full : std_logic := '0';
  SIGNAL dr_1 : std_logic_vector(71 downto 0) := (others=>'0');
  SIGNAL dr_2 : std_logic_vector(71 downto 0) := (others=>'0');
  SIGNAL dr_3 : std_logic_vector(71 downto 0) := (others=>'0');
  SIGNAL dr_4 : std_logic_vector(71 downto 0) := (others=>'0');
  SIGNAL ureq_srctag : std_logic_vector(4 downto 0) := (others=>'0');
  SIGNAL fresp_size : std_logic_vector(2 downto 0) := (others=>'0');
  SIGNAL fresp_srctag : std_logic_vector(4 downto 0) := (others=>'0');
  SIGNAL fresp_data : std_logic_vector(63 downto 0) := (others=>'0');
  SIGNAL freq_addr : std_logic_vector(39 downto 3) := (others=>'0');
  SIGNAL freq_size : std_logic_vector(3 downto 0) := (others=>'0');
  SIGNAL freq_srctag : std_logic_vector(4 downto 0) := (others=>'0');
  SIGNAL freq_mask : std_logic_vector(7 downto 0) := (others=>'0');
  SIGNAL freq_data : std_logic_vector(63 downto 0) := (others=>'0');

  --Outputs
  SIGNAL debug : std_logic_vector(25 downto 0);
  SIGNAL ar_1 : std_logic_vector(19 downto 0);
  SIGNAL aw_1 : std_logic_vector(19 downto 0);
  SIGNAL r_n_1 : std_logic;
  SIGNAL w_n_1 : std_logic;
  SIGNAL bw_n_1 : std_logic_vector(7 downto 0);
  SIGNAL dw_1 : std_logic_vector(71 downto 0);
  SIGNAL ar_2 : std_logic_vector(19 downto 0);
  SIGNAL aw_2 : std_logic_vector(19 downto 0);
  SIGNAL r_n_2 : std_logic;
  SIGNAL w_n_2 : std_logic;
  SIGNAL bw_n_2 : std_logic_vector(7 downto 0);
  SIGNAL dw_2 : std_logic_vector(71 downto 0);
  SIGNAL ar_3 : std_logic_vector(19 downto 0);
  SIGNAL aw_3 : std_logic_vector(19 downto 0);
  SIGNAL r_n_3 : std_logic;
  SIGNAL w_n_3 : std_logic;
  SIGNAL bw_n_3 : std_logic_vector(7 downto 0);
```

```
  SIGNAL dw_3 :  std_logic_vector(71 downto 0);
  SIGNAL ar_4 :  std_logic_vector(19 downto 0);
  SIGNAL aw_4 :  std_logic_vector(19 downto 0);
  SIGNAL r_n_4 :  std_logic;
  SIGNAL w_n_4 :  std_logic;
  SIGNAL bw_n_4 :  std_logic_vector(7 downto 0);
  SIGNAL dw_4 :  std_logic_vector(71 downto 0);
  SIGNAL ureq_addr :  std_logic_vector(39 downto 3);
  SIGNAL ureq_size :  std_logic_vector(2 downto 0);
  SIGNAL ureq_mask :  std_logic_vector(7 downto 0);
  SIGNAL ureq_rw_n :  std_logic;
  SIGNAL ureq_data :  std_logic_vector(63 downto 0);
  SIGNAL ureq_ts :  std_logic;
  SIGNAL ureq_byte_req :  std_logic;
  SIGNAL fresp_enable :  std_logic;
  SIGNAL freq_enable :  std_logic;
  SIGNAL uresp_ts :  std_logic;
  SIGNAL uresp_size :  std_logic_vector(3 downto 0);
  SIGNAL uresp_srctag :  std_logic_vector(4 downto 0);
  SIGNAL uresp_data :  std_logic_vector(63 downto 0);

  signal data : std_logic_vector(63 downto 0);

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut: user_app PORT MAP(
    debug => debug,
    reset_n => reset_n,
    user_clk => user_clk,
    user_enable => user_enable,
    rt_ready => rt_ready,
    qdr_ready => qdr_ready,
    dr_1 => dr_1,
    ar_1 => ar_1,
    aw_1 => aw_1,
    r_n_1 => r_n_1,
    w_n_1 => w_n_1,
    bw_n_1 => bw_n_1,
    dw_1 => dw_1,
    dr_2 => dr_2,
    ar_2 => ar_2,
    aw_2 => aw_2,
    r_n_2 => r_n_2,
    w_n_2 => w_n_2,
    bw_n_2 => bw_n_2,
    dw_2 => dw_2,
    dr_3 => dr_3,
    ar_3 => ar_3,
    aw_3 => aw_3,
    r_n_3 => r_n_3,
    w_n_3 => w_n_3,
    bw_n_3 => bw_n_3,
    dw_3 => dw_3,
    dr_4 => dr_4,
    ar_4 => ar_4,
    aw_4 => aw_4,
    r_n_4 => r_n_4,
    w_n_4 => w_n_4,
    bw_n_4 => bw_n_4,
    dw_4 => dw_4,
    ureq_srctag => ureq_srctag,
    ureq_full => ureq_full,
    ureq_notag => ureq_notag,
    ureq_addr => ureq_addr,
    ureq_size => ureq_size,
    ureq_mask => ureq_mask,
    ureq_rw_n => ureq_rw_n,
    ureq_data => ureq_data,
    ureq_ts => ureq_ts,
    ureq_byte_req => ureq_byte_req,
    fresp_valid => fresp_valid,
    fresp_ts => fresp_ts,
    fresp_size => fresp_size,
    fresp_srctag => fresp_srctag,
    fresp_data => fresp_data,
    fresp_enable => fresp_enable,
    freq_addr => freq_addr,
    freq_size => freq_size,
    freq_srctag => freq_srctag,
    freq_mask => freq_mask,
    freq_rw_n => freq_rw_n,
    freq_ts => freq_ts,
    freq_valid => freq_valid,
    freq_data => freq_data,
    freq_enable => freq_enable,
    uresp_full => uresp_full,
    uresp_ts => uresp_ts,
```

```
    uresp_size => uresp_size,
    uresp_srctag => uresp_srctag,
    uresp_data => uresp_data
  );

  tb : PROCESS
  variable i, j : std_logic_vector(3 downto 0);
  variable k : std_logic_vector(19 downto 0);
  BEGIN

    -- Wait 100 ns for global reset to finish
    reset_n <= '0';
    wait for 100 ns;
    reset_n <= '1';
    -- Place stimulus here
    wait for 1 ns;
    user_enable <= '1';
    wait for 1 ns;

    -- Write data to sram
    i := "0100";
    k := (others => '0');
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    data <= X"0000000000000001";
    wait for 2 ns;
    while i<8 loop
      freq_addr <= B"0000000000000" & i & k;
      freq_data <= data;
      if k=524287 then
        i := i+1;
        k := (others => '0');
      else
        k := k+1;
      end if;
        data <= data + 1;
      wait for 2 ns;
    end loop;

    -- Zero out signals
    freq_valid <= '0';
    freq_ts <= '0';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"000000000";
    freq_data <= X"0000000000000000";
    wait for 2 ns;

    -- Write pwm
    i := (others => '0');
    j := (others => '0');
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_data <= X"0000000000000001";
    while i<8 loop
      freq_addr <= B"00000000000000010000000000" & i & B"000" & j;
      if j=3 then
        i := i+1;
        j := (others => '0');
      else
        j := j+1;
      end if;
      wait for 2 ns;
    end loop;

    -- Zero out signals
    freq_valid <= '0';
    freq_ts <= '0';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"000000000";
    freq_data <= X"0000000000000000";
    wait for 2 ns;

    -- Write set_pointer
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"00000000A";
    freq_data <= X"000000000000AAAA";
    wait for 2 ns;

    -- Write threshold
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"000000004";
    freq_data <= X"0000000000000000";
```

```
    wait for 2 ns;

    -- write start_count
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"00200000F";
    freq_data <= X"0000000000FFFFFF";
    wait for 2 ns;

    -- Zero out signals
    freq_valid <= '0';
    freq_ts <= '0';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"000000000";
    freq_data <= X"0000000000000000";
    wait for 2 ns;

    wait for 100000 ns;
    reset_n <= '0';
    wait for 10 ns;
    reset_n <= '1';
    wait for 10 ns;

    -- Write set_pointer
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"00000000A";
    freq_data <= X"000000000000AAAA";
    wait for 2 ns;

    -- Write threshold
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"000000004";
    freq_data <= X"0000000000000008";
    wait for 2 ns;

    -- Write start_count
    freq_valid <= '1';
    freq_ts <= '1';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"00200000F";
    freq_data <= X"0000000000FFFFFF";
    wait for 2 ns;

    -- Zero out signals
    freq_valid <= '0';
    freq_ts <= '0';
    freq_rw_n <= '0';
    freq_addr <= '0' & X"000000000";
    freq_data <= X"0000000000000000";
    wait for 2 ns;

    wait; -- will wait forever
  END PROCESS;

END;
```

B.4 Kildekode for `user_app.vhd`

```
--------------------------------------------------------------------------------
-- FPWM Mark III
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use ieee.numeric_std.ALL;
use work.user_pkg.all;

entity user_app is
  port (
  debug : out std_logic_vector(25 downto 0);

    -- Global signals
    reset_n         : in  std_logic;
    user_clk        : in  std_logic;
    user_enable     : in  std_logic;
    rt_ready        : in  std_logic;
    qdr_ready       : in  std_logic;
    -- QDR II RAM Interface
```

```
    -- RAM 1
    dr_1            : in   std_logic_vector(71 downto 0);  -- read data
    ar_1            : out  std_logic_vector(19 downto 0);  -- read address
    aw_1            : out  std_logic_vector(19 downto 0);  -- write address
    r_n_1           : out  std_logic;       -- Read strobe
    w_n_1           : out  std_logic;       -- Write strobe
    bw_n_1          : out  std_logic_vector(7 downto 0);   -- byte write
    dw_1            : out  std_logic_vector(71 downto 0);  -- write data
    -- RAM 2
    dr_2            : in   std_logic_vector(71 downto 0);  -- read data
    ar_2            : out  std_logic_vector(19 downto 0);  -- read address
    aw_2            : out  std_logic_vector(19 downto 0);  -- write address
    r_n_2           : out  std_logic;       -- Read strobe
    w_n_2           : out  std_logic;       -- Write strobe
    bw_n_2          : out  std_logic_vector(7 downto 0);   -- byte write
    dw_2            : out  std_logic_vector(71 downto 0);  -- write data
    -- RAM 3
    dr_3            : in   std_logic_vector(71 downto 0);  -- read data
    ar_3            : out  std_logic_vector(19 downto 0);  -- read address
    aw_3            : out  std_logic_vector(19 downto 0);  -- write address
    r_n_3           : out  std_logic;       -- Read strobe
    w_n_3           : out  std_logic;       -- Write strobe
    bw_n_3          : out  std_logic_vector(7 downto 0);   -- byte write
    dw_3            : out  std_logic_vector(71 downto 0);  -- write data
    -- RAM 4
    dr_4            : in   std_logic_vector(71 downto 0);  -- read data
    ar_4            : out  std_logic_vector(19 downto 0);  -- read address
    aw_4            : out  std_logic_vector(19 downto 0);  -- write address
    r_n_4           : out  std_logic;       -- Read strobe
    w_n_4           : out  std_logic;       -- Write strobe
    bw_n_4          : out  std_logic_vector(7 downto 0);   -- byte write
    dw_4            : out  std_logic_vector(71 downto 0);  -- write data
    -- RT Interface
    -- User Request Interface
    ureq_srctag    : in   std_logic_vector(4 downto 0);    -- request srctag
    ureq_full      : in   std_logic;       -- request buffer full
    ureq_notag     : in   std_logic;       -- request out of source tags
    ureq_addr      : out  std_logic_vector(39 downto 3);  -- request address
    ureq_size      : out  std_logic_vector(2 downto 0);    -- request size
    ureq_mask      : out  std_logic_vector(7 downto 0);    -- request byte mask
    ureq_rw_n      : out  std_logic;       -- request read/write
    ureq_data      : out  std_logic_vector(63 downto 0);  -- request write data
    ureq_ts        : out  std_logic;       -- request transfer start
    ureq_byte_req  : out  std_logic;       -- request type (byte or double word)
    -- Fabric Response Interface
    fresp_valid    : in   std_logic;       -- response signals valid
    fresp_ts       : in   std_logic;       -- response transfer start
    fresp_size     : in   std_logic_vector(2 downto 0);    -- response size
    fresp_srctag   : in   std_logic_vector(4 downto 0);    -- response srctag
    fresp_data     : in   std_logic_vector(63 downto 0);  -- response data
    fresp_enable   : out  std_logic;       -- enable responses
    -- Fabric Request Interface
    freq_addr      : in   std_logic_vector(39 downto 3);  -- request address
    freq_size      : in   std_logic_vector(3 downto 0);    -- request size
    freq_srctag    : in   std_logic_vector(4 downto 0);    -- request srctag
    freq_mask      : in   std_logic_vector(7 downto 0);    -- request byte mask
    freq_rw_n      : in   std_logic;       -- request read/write
    freq_ts        : in   std_logic;       -- request transfer start
    freq_valid     : in   std_logic;       -- request valid
    freq_data      : in   std_logic_vector(63 downto 0);  -- request write data
    freq_enable    : out  std_logic;       -- enable request interface
    -- User Response Interface
    uresp_full     : in   std_logic;       -- response buffer full
    uresp_ts       : out  std_logic;       -- response transfer start
    uresp_size     : out  std_logic_vector(3 downto 0);    -- response size
    uresp_srctag   : out  std_logic_vector(4 downto 0);    -- response srctag
    uresp_data     : out  std_logic_vector(63 downto 0)    -- response data
    );

end entity user_app;


architecture rtl of user_app is
  signal ftr_pointer : std_logic_vector(39 downto 3);
  signal ftr_pointer_is_set : std_logic; -- separated as a signal due to difficulties meeting
       timing

  signal start_count, count_gen: std_logic_vector(23 downto 0);
  signal start_count_is_set : std_logic;

  signal index_counter : std_logic_vector(25 downto 0);
  signal count_cooldown : std_logic_vector(7 downto 0);

  signal writing, stall: std_logic;

  signal burst_size : std_logic_vector(3 downto 0);
  signal burst_addr : std_logic_vector(24 downto 3);
```

```
signal pwm_elem_in : std_logic_vector(7 downto 0);
signal pwm_data_out: std_logic_vector(37 downto 0);
signal pwm_index_in, pwm_index_out: std_logic_vector(25 downto 0);
signal pwm_valid, pwm_data_out_valid : std_logic;

signal write_addr : std_logic_vector(39 downto 3);
signal write_data : std_logic_vector(63 downto 0);
signal write_valid, write_bypass, write_flush, write_set_base_addr, write_empty,
     write_overflow : std_logic;

signal pwm_table: lu_table;

signal reader_active, reader_out_valid : std_logic;
signal reader_data_offset : std_logic_vector(25 downto 3);
signal reader_data_out : std_logic_vector(7 downto 0);
signal reader_index_out : std_logic_vector(25 downto 0);

signal sram_write_data : std_logic_vector(63 downto 0);
signal sram_write_addr : std_logic_vector(22 downto 3);
signal sram_write_sel : std_logic_vector(1 downto 0);
signal sram_write_valid : std_logic;

signal pwm_write_data : std_logic_vector(31 downto 0);
signal pwm_write_row : std_logic_vector(6 downto 0);
signal pwm_write_col : std_logic_vector(13 downto 0);
signal pwm_write_valid : std_logic;

-- Added signals for filter
signal filter_data_in : std_logic_vector(37 downto 0);
signal filter_data_out : std_logic_vector(63 downto 0);
signal filter_index_in : std_logic_vector(25 downto 0);
signal filter_valid_in : std_logic;
signal filter_valid_out : std_logic;

signal thresh : std_logic_vector(37 downto 0);

-- timing is fun
 signal tmp_freq_data      : std_logic_vector(63 downto 0);
 signal tmp_freq_addr      : std_logic_vector(39 downto 3);
 signal tmp_freq_size      : std_logic_vector(3 downto 0);
 signal tmp_freq_srctag    : std_logic_vector(4 downto 0);
 signal tmp_freq_mask      : std_logic_vector(7 downto 0);
 signal tmp_freq_rw_n      : std_logic;
 signal tmp_freq_ts        : std_logic;
 signal tmp_freq_valid     : std_logic;

component reader is
 port(
  reset_n     : in  std_logic;
  user_clk    : in  std_logic;
  stall     : in  std_logic;

  active    : in  std_logic;
  data_offset : in  std_logic_vector(25 downto 3);

  out_valid : out std_logic;
  data_out    : out std_logic_vector(7 downto 0);
  index_out : out std_logic_vector(25 downto 0);

  -- sram interface
  dr_1        : in  std_logic_vector(71 downto 0);  -- read data
  ar_1        : out std_logic_vector(19 downto 0);  -- read address
  r_n_1       : out std_logic;      -- Read strobe

  dr_2        : in  std_logic_vector(71 downto 0);  -- read data
  ar_2        : out std_logic_vector(19 downto 0);  -- read address
  r_n_2       : out std_logic;      -- Read strobe

  dr_3        : in  std_logic_vector(71 downto 0);  -- read data
  ar_3        : out std_logic_vector(19 downto 0);  -- read address
  r_n_3       : out std_logic;      -- Read strobe

  dr_4        : in  std_logic_vector(71 downto 0);  -- read data
  ar_4        : out std_logic_vector(19 downto 0);  -- read address
  r_n_4       : out std_logic      -- Read strobe
 );
end component reader;


component pwm is
 port(
   reset_n     : in  std_logic;
  user_clk    : in  std_logic;
  stall     : in  std_logic;
  valid     : in  std_logic;

  -- Input lines
  elem_in    : in  std_logic_vector(7 downto 0);
```

```
      index_in     : in   std_logic_vector(25 downto 0);
      pwm_table    : in   lu_table;

      -- Output lines
      data_out       : out std_logic_vector(37 downto 0);
      index_out    : out std_logic_vector(25 downto 0);
      data_out_valid : out std_logic
    );
  end component pwm;

  component filter is
    port (
      reset_n      : in   std_logic;
      user_clk     : in   std_logic;
      valid        : in   std_logic;
      stall      : in   std_logic;
      data_in      : in   std_logic_vector(37 downto 0);
      index_in     : in   std_logic_vector(25 downto 0);
      threshold    : in   std_logic_vector(37 downto 0);

      data_valid     : out std_logic;
      data_out     : out std_logic_vector(63 downto 0)
    );
  end component filter;

  component writer is
    port (
      reset_n      : in   std_logic;
      user_clk       : in   std_logic;
      valid        : in   std_logic;
      bypass       : in   std_logic;
      flush        : in   std_logic;
      set_base_addr : in   std_logic;

      data_in      : in   std_logic_vector(63 downto 0);
      addr_in      : in   std_logic_vector(39 downto 3);

      stall        : out std_logic;
      empty        : out std_logic;
      overflow       : out std_logic;

      ureq_srctag      : in   std_logic_vector(4 downto 0); -- unused
      ureq_full    : in   std_logic;
      ureq_notag     : in   std_logic; -- unused
      ureq_addr    : out std_logic_vector(39 downto 3);
      ureq_size    : out std_logic_vector(2 downto 0);
      ureq_mask    : out std_logic_vector(7 downto 0);
      ureq_rw_n    : out std_logic;
      ureq_data    : out std_logic_vector(63 downto 0);
      ureq_ts      : out std_logic;
      ureq_byte_req : out std_logic
    );
  end component writer;

begin  -- architecture rtl

  -- Stub unused parts of SRAM interface (byte filter)
  bw_n_1          <= (others => '0');
   bw_n_2          <= (others => '0');
   bw_n_3          <= (others => '0');
   bw_n_4          <= (others => '0');

  -- Stub unused parts of RT interface (fabric response -- no reads issued from FPGA)
   fresp_enable  <= '0';

  rt_xfer : process (reset_n, user_clk)
  begin
    if reset_n='0' then
      ftr_pointer <= (others => '0');
      ftr_pointer_is_set <= '0';

      start_count <= (others => '0');
      start_count_is_set <= '0';

      uresp_ts      <= '0';
      uresp_size    <= (others => '0');
      uresp_srctag  <= (others => '0');
      uresp_data    <= (others => '0');

      freq_enable   <= '0';

      sram_write_valid <= '0';
      sram_write_data <= (others => '0');
      sram_write_addr <= (others => '0');
      sram_write_sel <= (others => '0');

      burst_size <= (others => '0');
      burst_addr <= (others => '0');
```

```
      writing <= '1';

      thresh <= (others => '0');

      tmp_freq_data <= (others => '0');
      tmp_freq_addr <= (others => '0');
      tmp_freq_size <= (others => '0');
      tmp_freq_srctag <= (others => '0');
      tmp_freq_mask <= (others => '0');
      tmp_freq_rw_n <= '0';
      tmp_freq_ts <= '0';
      tmp_freq_valid <= '0';
    elsif user_clk='1' and user_clk'event then
      -- SMP-side reads and writes

      tmp_freq_data <= freq_data;
      tmp_freq_addr <= freq_addr;
      tmp_freq_size <= freq_size;
      tmp_freq_srctag <= freq_srctag;
      tmp_freq_mask <= freq_mask;
      tmp_freq_rw_n <= freq_rw_n;
      tmp_freq_ts <= freq_ts;
      tmp_freq_valid <= freq_valid;


      if user_enable='1' then
        -- determine sram valid signal
        if or_reduce(burst_size)='1' or (tmp_freq_valid='1' and tmp_freq_ts='1' and
            tmp_freq_rw_n='0' and tmp_freq_addr(25)='1') then
          sram_write_valid <= '1';
        else
          sram_write_valid <= '0';
        end if;

        -- determine pwm valid signal
        if tmp_freq_valid='1' and tmp_freq_ts='1' and tmp_freq_rw_n='0' and tmp_freq_addr(25)
            ='0' and tmp_freq_addr(24)='1' then
          pwm_write_valid <= '1';
        else
          pwm_write_valid <= '0';
        end if;

        -- write request
        if or_reduce(burst_size)='1' then
          burst_size <= burst_size - 1;
          burst_addr <= burst_addr + 1;

          sram_write_addr <= burst_addr(22 downto 3);
          sram_write_sel <= burst_addr(24 downto 23);
          sram_write_data <= tmp_freq_data;
        elsif writing='1' then
          freq_enable <= '1';
          uresp_ts <= '0';
          uresp_srctag <= (others => '0');
          uresp_data <= (others => '0');
          writing <= '0';
        elsif tmp_freq_valid='1' and tmp_freq_ts='1' then
          -- the start of some kind of access...
          if tmp_freq_rw_n='1' then
            -- .. which is a read..
            case tmp_freq_addr(6 downto 3) is
              when X"A" =>
                uresp_data(39 downto 3) <= ftr_pointer;
              -- perandg's add for threshold transfer
              when X"4" =>
                uresp_data(37 downto 0) <= thresh;
              when X"F" =>
                uresp_data(23 downto 0) <= start_count;
              when others =>
                uresp_data(26 downto 3) <= tmp_freq_addr(26 downto 3);
                uresp_data(63 downto 27) <= (others => '1');
            end case;

            uresp_ts <= '1';
            uresp_srctag <= tmp_freq_srctag;
            freq_enable <= '0';
            writing <= '1';
          elsif tmp_freq_rw_n='0' then
            -- .. which is a write...
            if tmp_freq_addr(25)='1' then
              -- .. to the SRAM..
              if tmp_freq_size>X"0" then
                -- .. and will continue as a burst.
                burst_size <= tmp_freq_size - 1;
                burst_addr <= tmp_freq_addr(24 downto 3) + 1;
              end if;
```

```
        -- either way, read and store this quadword.
        sram_write_addr <= tmp_freq_addr(22 downto 3);
        sram_write_sel <= tmp_freq_addr(24 downto 23);
        sram_write_data <= tmp_freq_data;
      elsif tmp_freq_addr(24)='1' then
        -- .. to the PWM lookup table (warning: there is no burst support for this)
        pwm_write_data <= tmp_freq_data(31 downto 0);
        pwm_write_col <= tmp_freq_addr(23 downto 10);
        pwm_write_row <= tmp_freq_addr(9 downto 3);
      else
        -- .. to a variable (these are assumed to be set using fpga_wrt_appif_val
        case tmp_freq_addr(6 downto 3) is
          when X"A" =>
            -- sets writeback address (FTR)
            report "set ftr_pointer";
            ftr_pointer <= tmp_freq_data(39 downto 3);
            ftr_pointer_is_set <= '1';
          when X"4" =>
            ---Sets threshold value for filter
            report "set threshold value";
            thresh <= tmp_freq_data(37 downto 0);
            -- thresh <= "00000000000000000000000000000000000101";
          when X"F" =>
            -- sets number of cycles to run
            -- FTR mem *must* hold the required amount of data, or data corruption may
                occur!
            -- THERE IS NO MEMORY PROTECTION, AND CHECKING SMP'S BUFFER FROM FPGA IS
               IMPOSSIBLE!
            report "set start_count";
            start_count <= tmp_freq_data(23 downto 0);
            start_count_is_set <= '1';
          when others =>
            null;
        end case;
      end if;
    end if;
  else
    start_count <= (others => '0');
    start_count_is_set <= '0';
  end if;
 end if;
end if;
end process rt_xfer;

sram_write : process (reset_n, user_clk)
begin
  if reset_n='0' then
     aw_1          <= (others => '0');
     w_n_1         <= '1';
     dw_1          <= (others => '0');

     aw_2          <= (others => '0');
     w_n_2         <= '1';
     dw_2          <= (others => '0');

     aw_3          <= (others => '0');
     w_n_3         <= '1';
     dw_3          <= (others => '0');

     aw_4          <= (others => '0');
     w_n_4         <= '1';
     dw_4          <= (others => '0');
  elsif user_clk='1' and user_clk'event then
    -- sram write stuff
    if qdr_ready='1' then
      if sram_write_valid='1' then
        if sram_write_sel="00" then
          aw_1 <= sram_write_addr;
          dw_1 <= insert_parity(sram_write_data);
          w_n_1 <= '0';
        else
          w_n_1 <= '1';
        end if;

        if sram_write_sel="01" then
          aw_2 <= sram_write_addr;
          dw_2 <= insert_parity(sram_write_data);
          w_n_2 <= '0';
        else
          w_n_2 <= '1';
        end if;

        if sram_write_sel="10" then
          aw_3 <= sram_write_addr;
          dw_3 <= insert_parity(sram_write_data);
          w_n_3 <= '0';
        else
          w_n_3 <= '1';
```

```
          end if;

          if sram_write_sel="11" then
            aw_4 <= sram_write_addr;
            dw_4 <= insert_parity(sram_write_data);
            w_n_4 <= '0';
          else
            w_n_4 <= '1';
          end if;
        else
          w_n_1 <= '1';
          w_n_2 <= '1';
          w_n_3 <= '1';
          w_n_4 <= '1';
        end if;
      end if;
    end if;
  end process sram_write;

  pwm_write : process (reset_n, user_clk)
  begin
    if reset_n='0' then
      pwm_table <= (others => (others => '0'));
    elsif user_clk='1' and user_clk'event then
      if pwm_write_valid='1' then
        pwm_set(pwm_write_col(2 downto 0), pwm_write_row, pwm_write_data, pwm_table);
      end if;
    end if;
  end process pwm_write;


  data_push : process (reset_n, user_clk)
  begin
    if reset_n='0' then
      -- sets initial values
      count_gen <= (others => '0');
      count_cooldown <= (others => '0');
      index_counter <= (others => '0');

      pwm_elem_in <= (others => '1');
      pwm_index_in <= (others => '1');

      pwm_valid <= '0';
      write_data <= (others => '0');
      write_valid <= '0';
      write_bypass <= '0';
      write_flush <= '0';
      write_set_base_addr <= '0';

      filter_data_in <= (others => '0');
      filter_index_in <= (others => '0');
      filter_valid_in <= '0';

      debug <= (others => '0');
    elsif user_clk='1' and user_clk'event then
        if user_enable='1' then
        if or_reduce(count_gen)='1' and ftr_pointer_is_set='1' then -- count_gen > 0 resulted
             in a carry chain with too much delay to work
          reader_active <= '1';
          write_set_base_addr <= '0';
          write_flush <= '0';
          count_cooldown <= X"10";

          if stall = '0' then
            count_gen <= count_gen -1;
          end if;

        elsif count_cooldown > 0 then
          pwm_valid <= '0';
          count_cooldown <= count_cooldown -1;
          debug(25 downto 8) <= (others => '1');
          debug(7 downto 0) <= count_cooldown;
        else
          if start_count_is_set='1' then
            reader_data_offset <= (others => '0'); -- fixme: make offset-able
            count_gen <= start_count;
          else
            reader_data_offset <= (others => '0');
            count_gen <= (others => '0');
          end if;

          reader_active <= '0';
          write_addr <= ftr_pointer;
          write_set_base_addr <= '1';
          pwm_valid <= '0';
          write_flush <= '1';
          debug <= (others => '0');
        end if;
```

```
      end if;
    end if;
  end process data_push;

  reader_inst : reader
  port map(
    -- Global control signal
    reset_n => reset_n,
    user_clk => user_clk,
    stall => stall,

    active      =>  reader_active,
    data_offset => reader_data_offset,

    out_valid => reader_out_valid,
    data_out      => reader_data_out,
    index_out => reader_index_out,

    dr_1   => dr_1,
    ar_1   => ar_1,
    r_n_1 =>  r_n_1,

    dr_2   => dr_2,
    ar_2   =>  ar_2,
    r_n_2 =>  r_n_2,

    dr_3   => dr_3,
    ar_3   => ar_3,
    r_n_3 =>  r_n_3,

    dr_4   => dr_4,
    ar_4   =>  ar_4,
    r_n_4 =>  r_n_4
  );

  pwm_inst : pwm
  port map(
    -- Global control signal
    reset_n => reset_n,
    user_clk => user_clk,
    valid => reader_out_valid,
    stall => stall,

    -- Input signals
    elem_in => reader_data_out,
    index_in => reader_index_out,
    pwm_table => pwm_table,

    -- Output signals
    data_out => pwm_data_out,
    index_out => pwm_index_out,
    data_out_valid => pwm_data_out_valid
  );

  filter_inst : filter
  port map(
    reset_n => reset_n,
    user_clk => user_clk,
    valid => pwm_data_out_valid,
    stall => stall,
    data_in => pwm_data_out,
    index_in => pwm_index_out,
    threshold => thresh,

    data_valid => filter_valid_out,
    data_out => filter_data_out
  );

  write_inst : writer
  port map(
    reset_n => reset_n,
    user_clk => user_clk,
    valid => filter_valid_out,
    bypass => write_bypass,
    flush => write_flush,
    set_base_addr => write_set_base_addr,

    data_in => filter_data_out,
    addr_in => write_addr,

    stall => stall,
    empty => write_empty,
    overflow => write_overflow,

    ureq_srctag => ureq_srctag,
    ureq_full => ureq_full,
    ureq_notag => ureq_notag,
    ureq_addr => ureq_addr,
```

```
    ureq_size => ureq_size,
    ureq_mask => ureq_mask,
    ureq_rw_n => ureq_rw_n,
    ureq_data => ureq_data,
    ureq_ts => ureq_ts,
    ureq_byte_req => ureq_byte_req
  );

end architecture rtl;
```

B.5 Kildekode for c-rammeverk

```
/* Original file made by Lars Krutaadal, edited by perandg.

*  File takes dna file, pwm file and optionally threshold as input parameters.
*  If no threshold is given summation is used.

*/

#define _XOPEN_SOURCE 600
#include "ufplib.h"
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <argp.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <math.h>

// DEFINE THE FPGA WRITE TYPES (ADDRESS OR DATA VALUE)
#define TYPE_VAL  0x0UL
#define TYPE_ADDR 0x1UL

// DEFINE THE ADDRESS OFFSETS FOR THE MINCE FPGA REGISTERS
#define APP_ID_REG        0x00UL
#define APP_CFG_REG       0x08UL
#define APP_LATCH_REG     0x10UL

#define SCALE 24
// DECLARE A TYPE FOR A 64 BIT UNSIGNED INTEGER.
typedef unsigned long u_64;
//typedef int fixedPoint;

// DECODE AND PRINT OUT ANY DRIVER ERRORS.
int print_err (err_e e)
{
  switch (e) {
    case NOERR:
      printf("Success.\n");
      break;
    case FILEOPRERR:
      printf("File operation system call failed.\n");
      break;
    case INVALIDOP:
      printf("Invalid API operation requested.\n");
      break;
    case INVALIDVAL:
      printf("Invalid value passed to the API call.\n");
      break;
    case INVALIDARGS:
      printf("Invalid argument passed to the API call.\n");
      break;
    case INVALIDINP:
      printf("Invalid input given to the API call.\n");
      break;
    case DEVOPRERR:
      printf("FPGA device operation error.\n");
      break;
    case UNKNOWNERR:
      printf("Unknown error.\n");
      break;
    default:
      break;
    }
    return 0;
}

// LOAD CONTENT FROM FILE AND COPY TO MEMORY
char * openDNAFile(char *filename) {
  FILE * pFile;
  unsigned long size;
  size_t result;
  char *buffer;
```

```
  // Open file
  pFile = fopen(filename, "r");
  if (pFile == NULL) {fputs("File error", stderr); exit(1);}

  // Obtain file size
  fseek(pFile, 0, SEEK_END);
  size = ftell(pFile);
  rewind(pFile);

  // Allocate memory to contain the whole file
  buffer = (char*) malloc(sizeof(char)*size);
  if (buffer == NULL) {fputs("Memory error", stderr); exit(2);}

  // Copy the file into the buffer
  result = fread(buffer, 1, size, pFile);
  if (result != size) {fputs("Reading error", stderr); exit(3);}

  // close file
  fclose(pFile);

  return buffer;
}

int fpEncode(float x) {
  return x * 16777216; //roundf()
}

float fpDecode(long x) {
  return (float) ((x) / 16777216.0);
}

float fpDecodeI(int x) {
  return (float) ((x) / 16777216.0);
}

// CREATE PWM-VALUE
int calcPWMvalue(float val, float divisor) {
  float floatTemp;
  double pwmTemp;

  // Safety, because log(0) == error.
  if (val == 0.0) val = 0.01; // redundant because all number + 0.01 on load

  pwmTemp = (double) ((val / divisor) / 0.25);
  floatTemp = (float) log(pwmTemp); // Need to go via double because of log(double x)-
       function

  return fpEncode(floatTemp);
}

// CREATE PWM-MATRIX
int *createPWM(float *prepwm) {
  int i, j;
  int *pwm;

  pwm = (int *) malloc(sizeof(int)*8*4);
  for(i=0; i<8; i++) {
    double divisor = (prepwm[i*4+0]+prepwm[i*4+1]+prepwm[i*4+2]+prepwm[i*4+3]);
    for(j=0; j<4; j++) {
      pwm[i*4+j]= calcPWMvalue(prepwm[i*4+j], divisor);
    }
  }
  return pwm;
}

// MAIN
int main (int argc, char **argv) {
  int fp_id;
  err_e e;

  float threshold = 0;
  int filter = 0;
  char *filename = "";
  char *pwmName = "";
  char *dnaName = "";

  switch (argc) {
    case 3: // summation
      dnaName = argv[1];
      pwmName = argv[2];
      filter = 1;
      filename = "top.bin.ufp.sum";
      printf("Execution with summation filter starting with dna %s and pwm %s.\n", dnaName,
           pwmName);
      break;
    case 4: // threshold
      dnaName = argv[1];
```

```
      pwmName = argv[2];
      threshold = atof(argv[3]);
      filter = 2;
      filename = "top.bin.ufp.thresh";
      printf("Execution with threshold filter starting, with dna %s, pwm %s and threshold is
          %f\n", dnaName, pwmName, threshold);
      break;
    default:
      printf("Parameters: ./fpwm [file containing dna] [file containing PWM] [integer
          threshold], if threshold is not given summation filter will be assumed\n");
      printf("Examples:\n [./fpwm file.dna file.pwm 8] executes file.dna and file.pwm with a
          threshold of 8\n [./fpwm file.dna file.pwm] executes file.dna and file.pwm with
          summation filter\n");
      break;
  }

  if (filter > 0) {

    // Read dna and pwm from file
    FILE * pFile;
    unsigned long dnaSize;
    char *dna;
    float *prepwm;
    int * pwm;

    // Load DNA
    dna = openDNAFile( dnaName );
    dnaSize = strlen(dna);

    // Load PWM
    prepwm = (float *) malloc(sizeof(float)*8*4);
    pFile = fopen (pwmName, "r");
    int i, j, num;
    float val;
    for (i=0; i<8; i++) {
      for (j=0; j<4; j++) {
        num = fscanf (pFile, "%f", &val);
        prepwm[i*4+j] = val + 0.01;
      }
    }
    fclose(pFile);

    pwm = createPWM(prepwm);

    // Open file for saving results
    pFile = fopen("output-fpwm.txt", "w");
    if (pFile==NULL) {fputs("File error", stderr); exit(1);}

    // Start FPGA
    if((fp_id = fpga_open ("/dev/ufp0", O_RDWR|O_SYNC, &e)) > 0) {

      printf("\nLoading FPGA..\n");

      // Load FPGA
      if(fpga_load(fp_id, filename, &e) > 0) {

        printf("FPGA successfully loaded with status 0x%02X\n\n", fpga_status(fp_id, &e));

        // Map FPGA memory to a local address

        u_64 *memmap = (u_64*) fpga_memmap(fp_id, 1<<27, PROT_READ|PROT_WRITE, MAP_SHARED, 0,
            &e);

        // Allocate an FPGA transfer region

        u_64 *ftrmem;

        // Make sure the needed mem-area fits in memory page size exactly (because of
            register_ftrmem)
        unsigned int ftrSize = 0;
        if ((16777216*8) % getpagesize() != 0) {
          ftrSize = (((16777216*8) % getpagesize()) + 1 ) * getpagesize();
        } else {
          ftrSize = 16777216 * 8;
        }

        // Align memory
        int status = posix_memalign((void **)&ftrmem, getpagesize(), ftrSize);

        if (status != 0) {
          printf("Error memalign, errorcode %d\n", status);
        } else {

          // Register shared memory
          status = fpga_register_ftrmem(fp_id, ftrmem, ftrSize, &e);
          if (status != 0) {
            printf("Allocation of shared memory failed with status %d\n", status);
          } else {
```

```
printf("Allocated an FPGA transfer region of %d at 0x%lX ", ftrSize, (u_64)ftrmem
    ); print_err(e);
unsigned long i = 0;
int j = 0;
long DNAiterator = 0;
int run = 0;
float sum = 0.0;
int nrOfResults = 0;
unsigned long allTransferred = 0;


// Loop to transfer the whole dna, in several parts if needed
while (DNAiterator < dnaSize) {

  int k = 8;
  unsigned long other = 0;
  unsigned long transferred_elem = 0;

  // Rewind data for new cycle so that the search is complete
  // (this is crude and does not, like the program, handle "NNNN" in input)
  if (DNAiterator > 7) DNAiterator = DNAiterator - 7;

  run++;
  printf("\nTransferring to sram.");
  u_64 qw = 0x0000000000000000UL;

  // Add on-the-fly conversion here

  // Iterate through DNA until 16MB is transferred or end of DNA has occured
  while ((transferred_elem < 16777215) && (DNAiterator < dnaSize)) {
    // If vector contains 8 elements (64 bit) then transfer it
    if (k == 0) {
      k = 8;
      u_64 addr = ((1<<25) | (transferred_elem-8));
      fpga_wrt_appif_val(fp_id, qw, addr, 0, &e); // write using single writes
      qw = 0;
    }
    // Convert "a" to 0, "c" to 1, "g" to 2 and "t" to 3
    // Also construct 64 bit vector with 8 elements
    switch(dna[DNAiterator]) {
      case ('a'):
        k--;
        qw = (qw<<8) + (0x00);
        transferred_elem++;
        break;
      case ('c'):
        k--;
        qw = (qw<<8) + (0x01);
        transferred_elem++;
        break;
      case ('g'):
        k--;
        qw = (qw<<8) + (0x02);
        transferred_elem++;
        break;
      case ('t'):
        k--;
        qw = (qw<<8) + (0x03);
        transferred_elem++;
        break;
      default:
        // The start of a crude scheme to account for unmapped regions of the DNA
        // But nothing more has been implemented yet
        other++;
        break;
    }
    DNAiterator++;
  }

  // Write the last values, as a quadword with the values in more signifacant
      plositions.
  if (k != 8) {
    u_64 addr = ((1<<25) | (transferred_elem-(8-(k))));
    while (k != 0) {
      k--;
      qw = qw<<8 | (0x00);
    }
    fpga_wrt_appif_val(fp_id, qw, addr, 0, &e); // write using single writes
  }


  // If end of DNA is encountered (transferred_elem == 0) dont execute
  if (transferred_elem > 0) {
    printf(" Done\n");

    fpga_mem_sync(fp_id, &e);
```

```
    fflush(NULL);

    u_64 read_ftrmem, read_count, read_threshold;

    // Write PWM to FPGA
    if (run == 1) {
      printf("Wrote  PWM:\n");
      fprintf(pFile,"Wrote  PWM:\n");
    }

    // Iterate through pwm-matrix
    for(i=0; i<8; i++) {
      for(j=0; j<4; j++) {
        int data = pwm[i*4+j];
        u_64 addr = 1<<24 | i<<10 | j<<3;
        fpga_wrt_appif_val(fp_id, data, addr, 0, &e);
        if (run == 1) {
          printf("0x%08X\t", data);
          fprintf(pFile, "0x%08X\t", data);
        }
      }
      if (run == 1) {
        printf("\n");
        fprintf(pFile, "\n");
      }
    }

    // Calculate remaining runs and project it to user
    int remainingRuns = ((dnaSize-DNAiterator) / 16777215) + run;

    if (DNAiterator < dnaSize) printf("This is run %d of %d runs (roughly
        predicted), due to DNA size.\n", run, remainingRuns);
    else if (run == 1) printf("This is the only run.\n");
    else printf("This is the last run of %d runs\n", run);

    // Write values for execution to FPGA:
    // First memoryaddress
    fpga_wrt_appif_val(fp_id, (u_64)ftrmem, 0x50UL, 1, &e); // NOTE: type = 1
        meaning address not value
    if (run == 1) {printf("Wrote ftr_pointer=%lX: ", (u_64)ftrmem); print_err(e);
         fflush(NULL);}

    fpga_rd_appif_val(fp_id, &read_ftrmem, 0x50UL, &e);
    if (run == 1) {printf("Read ftr_pointer=%lX: ", read_ftrmem); print_err(e);
        fflush(NULL);}

 // Added by perandg, write threshold value to FPGA
    if (filter == 2) {
      fpga_wrt_appif_val(fp_id, fpEncode(threshold), 0xA0UL, 0, &e);
      if (run == 1) {printf("Wrote threshold=%f: ", threshold); print_err(e);
          fflush(NULL);}
      // Read value back
      fpga_rd_appif_val(fp_id, &read_threshold, 0xA0UL, &e);
      if (run == 1) {printf("Read threshold=%f: ",fpDecode(read_threshold));
          print_err(e); fflush(NULL);}
    }
// End add

    // Last number of elements in SRAM, this start computation on FPGA
    fpga_wrt_appif_val(fp_id, transferred_elem, 0x78UL, 0, &e);
    if (run == 1 || transferred_elem != 16777215) {printf("Wrote count=%lu: ",
        transferred_elem); print_err(e); fflush(NULL);}

    fpga_rd_appif_val(fp_id, &read_count, 0x78UL, &e);
    if (run == 1 || transferred_elem != 16777215) {printf("Read count=%lX: ",
        read_count); print_err(e); fflush(NULL);}

    usleep(200000); // 200.000us is about twice of the expected worst case time
        use of the FPWM when processing 16MB of data

    //printf("FPGA said:\n");

    int nulldata = 0;
    // Read results from shared memory
    for(i=0; i<transferred_elem; i++) {
      long data = *(ftrmem+i);
      long index = data >> 38;
      float result = fpDecodeI(data & 0xFFFFFFFF); // Use bitmask to get the
          relevant bits

      if (data != 0) {
        // Format according to filter
        if (filter == 1) {
          fprintf(pFile,"Sum: %ld\t(0x%016lX)\n", data, data);
          sum = sum + fpDecode(data);
        } else if (filter == 2) {
          fprintf(pFile,"Index: %ld\tResult: %f\t(0x%016lX)\n", index, result,
              data);
```

```
                      nrOfResults++;
                    }
                  } else {
                    nulldata++;
                  }
                }

                if (nulldata != 0) printf("Results equal to 0 have not been printed.\n");

                // If another round is required to finish computation then reset FPGA
                if (DNAiterator < dnaSize) {
                  status = fpga_reset(fp_id, &e);
                  printf("Reset: "); print_err(e);
                  usleep(1);
                  status = fpga_start(fp_id, &e);
                  printf("Restart: "); print_err(e); fflush(NULL);
                  usleep(1);
                }
                allTransferred = allTransferred + transferred_elem;
              } else { // If there were no elements to process
                printf("\nThe run was not completed, because there were no elements to
                    process\n");
              }
            } // End while, the whole DNA file has been processed

            // Deregister shared memory
            status = fpga_dereg_ftrmem(fp_id, ftrmem, &e);
            printf("\nDeregistering memory, status %d, ",status); print_err(e); fflush(NULL);

            // Print final stats according to filter
            if (filter == 1) {
              fprintf(pFile,"Final sum: %f, sum/8: %f\n", sum, sum/8);
              printf("Final sum: %f, sum/8: %f\n", sum, sum/8);
            } else if (filter == 2) {
              fprintf(pFile,"Nr of results printed: %d\n", nrOfResults);
              printf("Nr of results printed: %d\n", nrOfResults);
            }
            fprintf(pFile, "Nr of transferred elements: %lu\n", allTransferred);
            printf("Nr of transferred elements: %lu\n", allTransferred);
          } // fpga_register_ftrmem() was successful;
        } // Mem_align() was successful;
        free(dna);
        free(pwm);
        free(prepwm);
        fclose(pFile);
        printf("Results from computation is saved to file output-fpwm.txt, and the file is
            closed.\n");
      } else {
        printf("Failed to load FPGA.\n");
      }

      // Close the FPGA device
      fpga_close (fp_id, &e);

    } else {
      printf("Failed to open FPGA.\n");
    }
  } // No filter was loaded
  return 0;
}
```

Tillegg C

Utskrift fra kjøring

C.1 Utskrift fra kjøring med terskelfilter

Terskelverdi ← 0

```
Execution with threshold filter starting, count is 6000000 and threshold is 0
Because of issues with memory allocation the largest count possible is 262144 (2MB).
     Therefore count has been reset to 262144.

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
Wrote PWM data: 1 to 1000000
Wrote PWM data: 2 to 1000008
Wrote PWM data: 3 to 1000010
Wrote PWM data: 4 to 1000018
Wrote PWM data: 1 to 1000400
[...]
Wrote PWM data: 4 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=8D000000: Success.
Wrote threshold=0: Success.
Read threshold=0: Success.
Wrote count=262144: Success.
Read count=0: Success.

FPGA said:
Index: 0  Result: 11  (0x000000000000000B)
[...]
Index: 262132 Result: 13  (0x00FFFD000000000D)
Index: 262133 Result: 13  (0x00FFFD400000000D)
Index: 262134 Result: 13  (0x00FFFD800000000D)
Index: 262135 Result: 13  (0x00FFFDC00000000D)
Results equal to 0 have not been printed.
Nr of results printed: 262136
```

Terskelverdi ← 11

```
Execution with threshold filter starting, count is 6000000 and threshold is 11
Because of issues with memory allocation the largest count possible is 262144 (2MB).
     Therefore count has been reset to 262144.

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
```

```
Wrote PWM data: 1 to 1000000
Wrote PWM data: 2 to 1000008
Wrote PWM data: 3 to 1000010
Wrote PWM data: 4 to 1000018
Wrote PWM data: 1 to 1000400
[...]
Wrote PWM data: 4 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=8CE00000: Success.
Wrote threshold=11: Success.
Read threshold=B: Success.
Wrote count=262144: Success.
Read count=0: Success.

FPGA said:
Index: 1  Result: 14  (0x000000400000000E)
[...]
Index: 262139 Result: 14  (0x00FFFEC00000000E)
Index: 262140 Result: 14  (0x00FFFF000000000E)
Index: 262141 Result: 15  (0x00FFFF400000000F)
Index: 262142 Result: 15  (0x00FFFF800000000F)
Results equal to 0 have not been printed.
Nr of results printed: 196606
```

Terskelverdi ← 15

```
Execution with threshold filter starting, count is 6000000 and threshold is 15
Because of issues with memory allocation the largest count possible is 262144 (2MB).
     Therefore count has been reset to 262144.

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
Wrote PWM data: 1 to 1000000
Wrote PWM data: 2 to 1000008
Wrote PWM data: 3 to 1000010
Wrote PWM data: 4 to 1000018
Wrote PWM data: 1 to 1000400
[...]
Wrote PWM data: 4 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=8D200000: Success.
Wrote threshold=15: Success.
Read threshold=F: Success.
Wrote count=262144: Success.
Read count=0: Success.

FPGA said:
Index: 89 Result: 17  (0x0000164000000011)
[...]
Index: 261980 Result: 17  (0x00FFD70000000011)
Index: 262105 Result: 17  (0x00FFF64000000011)
Index: 262106 Result: 17  (0x00FFF68000000011)
Index: 262107 Result: 17  (0x00FFF6C000000011)
Index: 262108 Result: 17  (0x00FFF70000000011)
Results equal to 0 have not been printed.
Nr of results printed: 8192
```

C.2 Utskrift fra kjøring med summasjonsfilter

Antall elementer = 100

```
Execution with summation filter starting, count is 100

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
Wrote PWM data: 1 to 1000000
Wrote PWM data: 2 to 1000008
Wrote PWM data: 3 to 1000010
```

```
Wrote PWM data: 4 to 1000018
Wrote PWM data: 1 to 1000400
[...]
Wrote PWM data: 4 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=80800000: Success.
Wrote count=100: Success.
Read count=0: Success.

FPGA said:
Sum: 1270 (0x00000000000004F6)
Results equal to 0 have not been printed.
Final sum: 1270
```

Antall elementer = 1100

```
Execution with summation filter starting, count is 1100

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
Wrote PWM data: 1 to 1000000
Wrote PWM data: 2 to 1000008
Wrote PWM data: 3 to 1000010
Wrote PWM data: 4 to 1000018
Wrote PWM data: 1 to 1000400
[...]
Wrote PWM data: 4 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=80600000: Success.
Wrote count=1100: Success.
Read count=0: Success.

FPGA said:
Sum: 14474  (0x000000000000388A)
Results equal to 0 have not been printed.
Final sum: 14474
```

Antall elementer = 262144

```
Execution with summation filter starting, count is 848484848
Because of issues with memory allocation the largest count possible is 262144 (2MB).
    Therefore count has been reset to 262144.

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
Wrote PWM data: 1 to 1000000
Wrote PWM data: 2 to 1000008
Wrote PWM data: 3 to 1000010
Wrote PWM data: 4 to 1000018
Wrote PWM data: 1 to 1000400
[...]
Wrote PWM data: 4 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=80400000: Success.
Wrote count=262144: Success.
Read count=0: Success.

FPGA said:
Sum: 3473347  (0x000000000034FFC3)
Results equal to 0 have not been printed.
Final sum: 3473347
```

Antall elementer = 262144 og høye verdier i vektingsmatrise

```
Execution with summation filter starting, count is 848484848
Because of issues with memory allocation the largest count possible is 262144 (2MB).
    Therefore count has been reset to 262144.

Loading FPGA..
```

```
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region: 0x2A9DAA9000 Success.
Writing data for SRAM0... done!
Writing data for SRAM1... done!
Writing data for SRAM2... done!
Writing data for SRAM3... done!
Wrote PWM data: 10000000000 to 1000000
Wrote PWM data: 10000000001 to 1000008
Wrote PWM data: 10000000002 to 1000010
Wrote PWM data: 10000000003 to 1000018
Wrote PWM data: 10000000000 to 1000400
Wrote PWM data: 10000000001 to 1000408
[...]
Wrote PWM data: 10000000003 to 1001C18
Wrote ftr_pointer=2A9DAA9000: Success.
Read ftr_pointer=81000000: Success.
Wrote count=262144: Success.
Read count=0: Success.

FPGA said:
Sum: 140747088830369  (0x000080023C3B9FA1)
Sum: 140747088830487  (0x000080023C3BA017)
Sum: 140747088830387  (0x000080023C3B9FB3)
Sum: 140747088830479  (0x000080023C3BA00F)
Sum: 140747088830399  (0x000080023C3B9FBF)
Sum: 140747088830467  (0x000080023C3BA003)
Sum: 140747088830411  (0x000080023C3B9FCB)
Sum: 140747088830455  (0x000080023C3B9FF7)
Sum: 140747088830423  (0x000080023C3B9FD7)
Sum: 140747088830443  (0x000080023C3B9FEB)
Sum: 140747088830435  (0x000080023C3B9FE3)
Sum: 140747088830425  (0x000080023C3B9FD9)
Sum: 140747088830455  (0x000080023C3B9FF7)
Sum: 140747088830393  (0x000080023C3B9FB9)
Sum: 140747088830467  (0x000080023C3BA003)
Sum: 140747088830387  (0x000080023C3B9FB3)
Sum: 140747088830473  (0x000080023C3BA009)
Sum: 140747088830381  (0x000080023C3B9FAD)
Sum: 140747088830479  (0x000080023C3BA00F)
Sum: 140747088830375  (0x000080023C3B9FA7)
Sum: 140747088830485  (0x000080023C3BA015)
Sum: 1387504362112  (0x000001430DB46280)
Results equal to 0 have not been printed.
Final sum: 2957076369801187
```

C.3 Utskrift fra kjøring med rammeverk

Enkelt DNA og terskel 3

```
Execution with threshold filter starting, with dna chr21.fa, pwm chr21.pwm and threshold is
    3.000000

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region of 134217728 at 0x2AA09AC000 Success.

Transferring to sram. Done
Wrote PWM:
0x005FA1A9  0x005FA1A9  0x00163295  0xFA612D68
0xFFAED2CC  0x0098A0E8  0x005FA1A9  0xFA612D68
0x00163295  0xFFAED2CC  0x005FA1A9  0xFFAED2CC
0xFFAED2CC  0xFA612D68  0x012EDFEC  0xFA612D68
0xFA612D68  0x016231DA  0xFA612D68  0xFA612D68
0x016231DA  0xFA612D68  0xFA612D68  0xFA612D68
0xFA612D68  0xFA612D68  0x016231DA  0xFA612D68
0xFEFEA5F2  0xFFAED2CC  0x0110C1FE  0xFA612D68
This is the only run.
Wrote ftr_pointer=2AA09AC000: Success.
Read ftr_pointer=D8000000: Success.
Wrote threshold=3.000000: Success.
Read threshold=3.000000: Success.
Wrote count=15756742: Success.
Read count=0: Success.
Results equal to 0 have not been printed.

Deregistering memory, status 0, Success.
Nr of results printed: 57120
Nr of transferred elements: 15756742
Results from computation is saved to file output-fpwm.txt, and the file is closed.
```

Firedobbelt DNA og terskel 3

```
Execution with threshold filter starting, with dna quadchr21.fa, pwm chr21.pwm and threshold
    is 3.000000

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region of 134217728 at 0x2AA92AA000 Success.

Transferring to sram. Done
Wrote PWM:
0x005FA1A9  0x005FA1A9  0x00163295  0xFA612D68
0xFFAED2CC  0x0098A0E8  0x005FA1A9  0xFA612D68
0x00163295  0xFFAED2CC  0x005FA1A9  0xFFAED2CC
0xFFAED2CC  0xFA612D68  0x012EDFEC  0xFA612D68
0xFA612D68  0x016231DA  0xFA612D68  0xFA612D68
0x016231DA  0xFA612D68  0xFA612D68  0xFA612D68
0xFA612D68  0xFA612D68  0x016231DA  0xFA612D68
0xFEFEA5F2  0xFFAED2CC  0x0110C1FE  0xFA612D68
This is run 1 of 8 runs (roughly predicted), due to DNA size.
Wrote ftr_pointer=2AA92AA000: Success.
Read ftr_pointer=80000000: Success.
Wrote threshold=3.000000: Success.
Read threshold=3.000000: Success.
Wrote count=16777215: Success.
Read count=0: Success.
Results equal to 0 have not been printed.
Reset: Success.
Usleep() status: 0
Restart: Success.
Usleep() status: 0

Transferring to sram. Done
This is run 2 of 6 runs (roughly predicted), due to DNA size.
Results equal to 0 have not been printed.
Reset: Success.
Usleep() status: 0
Restart: Success.
Usleep() status: 0

Transferring to sram. Done
This is run 3 of 4 runs (roughly predicted), due to DNA size.
Results equal to 0 have not been printed.
Reset: Success.
Usleep() status: 0
Restart: Success.
Usleep() status: 0

Transferring to sram. Done
This is the last run of 4 runs
Wrote count=12695344: Success.
Read count=0: Success.
Results equal to 0 have not been printed.

Deregistering memory, status 0, Success.
Nr of results printed: 242704
Nr of transferred elements: 63026989
Results from computation is saved to file output-fpwm.txt, and the file is closed.
```

Enkelt DNA og terskel 5.1

```
Execution with threshold filter starting, with dna chr21.fa, pwm chr21.pwm and threshold is
    5.100000

Loading FPGA..
FPGA successfully loaded with status 0x00

Allocated an FPGA transfer region of 134217728 at 0x2AA09AC000 Success.

Transferring to sram. Done
Wrote PWM:
0x005FA1A9  0x005FA1A9  0x00163295  0xFA612D68
0xFFAED2CC  0x0098A0E8  0x005FA1A9  0xFA612D68
0x00163295  0xFFAED2CC  0x005FA1A9  0xFFAED2CC
0xFFAED2CC  0xFA612D68  0x012EDFEC  0xFA612D68
0xFA612D68  0x016231DA  0xFA612D68  0xFA612D68
0x016231DA  0xFA612D68  0xFA612D68  0xFA612D68
0xFA612D68  0xFA612D68  0x016231DA  0xFA612D68
0xFEFEA5F2  0xFFAED2CC  0x0110C1FE  0xFA612D68
This is the only run.
Wrote ftr_pointer=2AA09AC000: Success.
Read ftr_pointer=88000000: Success.
Wrote threshold=5.100000: Success.
Read threshold=5.100000: Success.
```

```
Wrote count=15756742: Success.
Read count=0: Success.
Results equal to 0 have not been printed.

Deregistering memory, status 0, Success.
Nr of results printed: 2767
Nr of transferred elements: 15756742
Results from computation is saved to file output-fpwm.txt, and the file is closed.
```

Tillegg D

Veiledning til FPWM-systemet

D.1 Kort veiledning til FPWM-systemet

Denne filen gir en kort introduksjon til FPWM-systemet for fremtidige utviklere i prosjektet. Den forsøker å gi en lett forklaring av systemets moduler og hvordan de henger sammen for å kompensere for manglende dokumentasjon. Denne er skrevet på engelsk og i vanlig tekstformat. Filen plassering: prosjektkatalog/about-fpwm.txt (vanligvis fpwmv3/about-fpwm.txt).

```
FPGA−PWM

This introduction seeks to give the reader a basic understanding of the different parts of
the system and its purpose is to make up for missing documentation for the system. It is not
 written by the author of the system, but by one involved in extending the system. Therefore
 most of what is written here are conclutions made by me after reading code and observing
the system. They need not be true or correct, but are my reflections after working with the
system.

Filesystem/hierarki:

All hdl−files are located in src/80−0037_vhdl/hdl/user_app/. The only hdl−file not located
there is the top−level file (src/80−0037_vhdl/hdl/top.vhd), but this file can largely be
ignored as it represents a layer of abstraction. What in function is the top−level file is
user_app.vhd. It is in this file all components are instantiated and connected.
The main project directory is src/80−0037_vhdl/par/xc2vp50/. It is in this directory all
Xilinx Ise project files are located. This is also the location of the makefile for
synthesis and generation of a bit−file.
The src/ directory holds the file fpwm.c. This c−file controls the operation of the fpga,
and it loads the bitfile and data to the fpga, and reads the data back. Using the make−
command in this directory compiles the c−file and copies it to bin/.
The bin/ directory is home to the bitfile and an executable fpwm−file. One must attach a
header to the bit−file before attempting to load it to the fpga, this is done by the program
 fcu. Furter reading: "Introduksjon til bruk av FPGA i vitenskapelige beregninger" by Lars
Krutådal, or type fcu −h.

The system:

User_app:
As mentioned before there is a top−level file called top.vhd, but it can largely be
disregarded. The file that is the most interesting is user_app.vhd located in src/80−0037
_vhdl/hdl/user_app/. This file instatiates and connects all components, and handles the CPU−
initiated initial communication, such as the transfer of PWM−matrix and the string in which
one wants to search. The file is slightly commented. There are four processes in the file.
One that concerns data−transfers from CPU, one that transfers data to the SRAM, one that
stores the PWM, and one that controls the execution of the chip. This latter process is the
most interesting, although its contents is limited. Basically it decrements the count_gen−
signal once every cycle as long as stall is 0. Once count_gen has reached 0 (the entire
amount of data has been searched) it starts a cooldown−phase. The pipelined nature of the
chip demands a cooldown phase to complete its execution. The process concerning transfers
initated from the CPU is the most complex. Especially addressing in user_app.vhd versus
addressing in the c−file fpwm.c (which initiates the transfers). In the c−file all bits are
coded, while in the user_app−file the 3 least significant bits are omitted. Therefore
hexvalues will change dramatically when encoding/decoding. An example from the system. This
is the transfer of the frtmem−pointer (the shared memory of the CPU and the FPGA):
```

```
in fpwm.c:
fpga_wrt_appif_val(fp_id, (u_64)ftrmem, 0x50UL, 1, &e);
The corresponding code in user_app.vhd:
case tmp_freq_addr(6 downto 3) is
  when X"A" =>
    -- sets writeback address (FTR)
    report "set ftr_pointer";
    ftr_pointer <= tmp_freq_data(39 downto 3);
    ftr_pointer_is_set <= '1';

Here we see a great difference in addressing. The bits are as follows (the 8 least
signinficant bits):
0x50:01010000 (fpwm.c)
0xA : 1010     (user_app.vhd)

In addition the adress must be increased by 40 for each access, since the signal is 40 bits
wide (39 downto 3).

Other examples are:
0x78:01111000
0xF : 1111

0xA0:10100000
0x4 : 0100

There exist several user_app files, with extensions .vhd, .vhd.ter and .vhd.sum. The .vhd
file is the one that is used under simulation or sythesis. The two others belong to each
filter-implementation, and when changing filter one must also copy the corresponding file to
 user_app.vhd.


Reader:
The file reader.vhd is responsible for reading data from SRAM and sending it on to the pwm-
module. This is the module I know the least about. But I suspect that there is som
pipelining used to extract the data from the SRAM. Also, I am not shure about how the data
is packed, but I think that each element is representet by 8 bits, and that 8 elements are
packed together in a quadword. The module emits one element per cycle.

PWM:
The pwm-module applies pipelining in order to add together the score for each index. It sums
 two and two numbers from the eight original sum each cycle, and ends up with a single sum
after 3 cycles. I suspect that is would be too time-consuming to sum all eight numbers
together in one cycle, and that that is the reason for pipelining the summation.

Filter:
The filter postprocesses the data in order to make the amount written back to the CPU
smaller. There are two different implementations, one that enforces a threshold value and
discards results not compliant with this value. The other sums all results and writes the
sum back to the CPU for every thousand cycles.

There are two filter implementation, each in its own file filter.vhd.[sum|ter]. The file
used in synthesis or simulation is the filter.vhd file, so the correct file must be copied
to filter.vhd.

Writer:
This module is responisble for storing results temporarily, and write them back to CPU using
 burst-writes. It instatiates 4 BRAM-block and uses two pointers as read address and write
address. It can read and write simultaniously, and writes data back to the CPU as soon as
possible (using burstwrite). The file is well commented so I will abstain from discussing
its functionality in detail.
```

Bibliografi

[1] Lars Krutådal. Introduksjon til bruk av fpga i vitenskapelige beregninger. Avhandling tdt4720, IDI, NTNU, 2005.

[2] Field programmable gate arrays on wikipedia. [http://en.wikipedia.org/wiki/Field-programmable_gate_array]. Utgave av 22. April 2007.

[3] Xilinx. Using the virtex block selectram+ features. Application Note xapp130, Xilinx, 2000.

[4] Cray Inc. Cray xd1 system overview. Techincal data release 1.2, Cray Inc, 2005.

[5] Lars Krutådal. Weighted pattern matching with pwms on fpgas. Diplomoppgave, IDI, NTNU, 2005.

[6] Cray Inc. Cray xd1 datasheet. Techincal data release 1.3, Cray Inc, 2005.

[7] Theo Ungerer, Borut Robic;, and Jurij Silc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[8] Pauline Haddow. Reliable reconfigurable parallel computing. Nfr-frinat application, IDI, NTNU, 2006.

[9] Øyvind Bø Syrstad Lars Andreas Eidsheim Osman Abul og Finn Drabløs Geir Kjetil Sandve, Magnar Nedland. Accelerationg motif doscovery: Motif matching on parallel hardware. Prosjektoppgave, IDI NTNU, MEDISIN NTNU og Interagon, 2006.

[10] Fpwm. [http://motdisc.idi.ntnu.no/index.php/FPWM]. Passordbeskyttet wiki for forskning innen mønstergjenkjenning. Utgave av 26. Jamuar 2007.

[11] Xilinx ise foundation. [http://www.xilinx.com/ise/logic_design_prod/foundation.htm].

[12] Modeltech internettside. [http://www.model.com/].

[13] Xilinx Inc. Synthesis and simulation design guide. Techincal data release 1.2, Cray Inc, 2005.

[14] Cray Inc. Cray xd1 programming. Techincal data release 1.3, Cray Inc, 2005.

[15] Cray Inc. Cray xd1 fpga development. Techincal data release 1.2, Cray Inc, 2005.

[16] How xilinx began. [http://www.xilinx.com/company/history.htm#begin].

[17] Xilinx Inc. Virtex-ii pro and virtex-ii pro x platform fpgas: Complete data sheet. Techincal data release 4.5, Xilinx Inc, 2005.

[18] Roger Staden. Computer methods to locate signals in nucleic acid sequenses. *IRL Press Limited*, 1983.

[19] Geir Kjetil Sandve og Finn Drabløs. A survey of motif discovery methods in an intergrated framework. Unpublished, NTNU, 2005.

[20] Interagon internettside. [http://www.interagon.com].

[21] Progeniq internettside. [http://www.progeniq.com].

[22] Progeniq. Acclerated data processing for life sciences. Technical report, Progeniq, 2005.

[23] Mitrionics internettside. [http://www.mitrionics.com].

[24] Impulse c internettside. [http://www.impulsec.com/].

[25] Anany Levitin. *Introduction to the design and analysis of algorithms*. Addison-Wesley, 2003.