Kunstig utvikling:
Utvidelse av FPGA-basert SBlock-plattform

14. juni 2005

Sammendrag

Utvikling av maskinvare foregår tradisjonelt med en topp-ned designstrategi. I fremtiden kan oppgavene som skal løses bli for kompliserte for denne utviklingsmetoden. En tilnærming som er foreslått er å benytte inspirasjon fra naturen og utvikle det som kalles for bio-inspirert-maskinvare.

Utvikling av evolusjonær maskinvare (EHW) betyr at elektronikk utvikles med grunnlag i evolusjonære algoritmer. Kort beskrevet går metoden ut på at det genereres tilfeldige individer, individene evalueres ut fra egnethet og nye individer dannes. Et individ er her en representasjon av en krets som forsøker å løse en gitt oppgave.

Det blir ofte benyttet en FPGA som plattform for utvikling av EHW. Det har vist seg at denne type krets har egenskaper som er godt egnet, men også egenskaper som kan gjøre det vanskelig å komme frem til gode resultater. Datamaskingruppa ved NTNU har foreslått en "virtuell FPGA" som enkelt lar seg implementere i en kommersielt tilgjengelig brikke. Denne plattformen blir kalt en *SBlock-matrise*. Mange av ulempene ved en vanlig FPGA er løst her. En SBlock-matrise kan betraktes som en to-dimensjonal cellulær automat som styres av ett sett regler. Ved siden av matrisen og regelsettet består SBlock-konseptet av en algoritme for utvikling av en organisme. Utviklngen er inspirert av naturen der en enkelt celle utvikles til en multicellær organisme. Algoritmen opererer på to nivåer. Tilstandene for hver enkelt SBlock kan endres og typen til hver blokk kan endres. Begge disse endringene avhenger av naboblokkenes egenskaper. Alle endringer i matrisen skjer i diskrete tidsintervaller.

Systemet som har blitt utvidet i denne oppgaven har som formål å være plattform for eksperimentering med SBlock-konseptet. Utgangspunktet for oppgaven var et system delvis skrevet i VHDL og delvis i C. Systemet kan kjøre begge de omtalte nivåene i algoritmen.

Utvidelsene som er gjort har hatt som formål å øke effektiviteten for systemet. Tidligere måtte all data som skulle prosesseres transporteres ut av systemet. Etter at utvidelsene er gjort blir data lagret under kjøring slik at evaluering kan gjøres internt. Datamengden som må leses ut har dermed avtatt. Det er gjort tre utvidelser av systemet. En utvidelse innbefatter lagring av informasjon for hver tilstandsendring i matrisen. Det har blitt implementert mulighet for å evaluere tilstandsdata med en funksjon, kalt *fitnessfunksjon*. Det har også blitt implementert minne for lagring av hvilken regel i utviklingsprosessen som har forårsaket endring av typen til en SBlock under kjøring.

Resultatene fra kjøring av systemet viser at de tre utvidelsene har vært vellykket. For mange typer eksperimenter vil effektiviteten være økt. Dette gjelder eksperimenter hvor data ikke lenger må transporteres ut av systemet for evaluering, men kan evalueres internt med de nye funksjonene. Dersom data generert under kjøring ikke skal benyttes medfører endringene noe økt tidsforbruk i forhold til hva som oppnåes med det gamle systemet. Overvåkning av regelsettet gjør det lettere å holde kontroll med utviklingen.

Forord

Denne hovedoppgaven er utført ved Norges teknisk-naturvitenskapelige universitet, Institutt for datateknikk og informasjonsvitenskap, datamaskingruppen.

Hovedoppgaven teller 30 studiepoeng og er hele studiebelastningen vårsemesteret i 5. årstrinn i sivilingeniørstudiet. Oppgaven er utført våren 2005.

Takk til faglærer Gunnar Tufte for god veiledning og et godt samarbeid under arbeidet med oppgaven.

Kjetil Aamodt
14. juni 2005

Oppgavetekst

Oppgaven viderefører arbeid gjort innen biologisk-inspirert utvikling i maskinvare. Arbeidet skal være rettet mot forskingen til Datamaskingruppen innen evolusjonær maskinvare og kunstig utvikling. I tidligere arbeid er det utviklet en maskinvareplattform basert på "SBlock Virtual FPGA" for en cPCI vertsdatamaskin med et BenERA FPGA-kort.

Kandidaten skal utvide den eksisterende maskinvareplattformen slik at det er mulig å gjøre evaluering av oppførselen til organismer (SBlock-baserte digitalekretser) i SBlock-matrisen i maskinvaren. Ved å flytte evaluering til maskinvaren avgrenser en data-overføringen over cPCI-bussen. Ved å avgrense dataoverføring mellom SBlockmatrisen og vertsdatamaskinen kan mengden eksperiment som er realistiske (tidsmessig) å kjøre utvides. Det er også ønskelig med en utvidelse som lagrer hvilke regler (gen) som er aktive i utviklingen av en organisme. Ved overvåkning av aktive gen, håper man å kunne få økt innsikt i den kunstige utviklingsprosessen.

I tidligere arbeid er det utviklet en samlebåndbasert koprosessor for å kjøre utvik-lingsprosessen for "celler" i SBlockmatrisen. Det er ønskelig å beholde denne løsningen. Derfor må utvidelsene gjøres med de maskinvareressursene som er tilgjengeleg etter at en 32x32 SBlockmatrise og koprosessoren er implementert i FPGAen på BenERA-kortet.

Det er viktig at eventuelle endringer som blir gjort ikke påvirker eksisterende funksjon.

Innhold

Figurer

Tabeller

Kapittel 1

Innledning

Halvlederindustrien tilbyr stadig flere transistorer på en brikke for utvikling av elektroniske kretser. En utfordringen med denne utviklingen er å utnytte mulighetene som tilbys. Dette problemet blir kalt *designgapet* [2]. En løsning kan være å benytte andre utviklingsmetoder enn den tradisjonelle topp-ned metoden [18]. En metode kalles for evolusjonær maskinvareutvikling (EHW). Denne går ut på å benytte metoder inspirert av biologien, hovedsaklig fra Darwins utviklingslære.

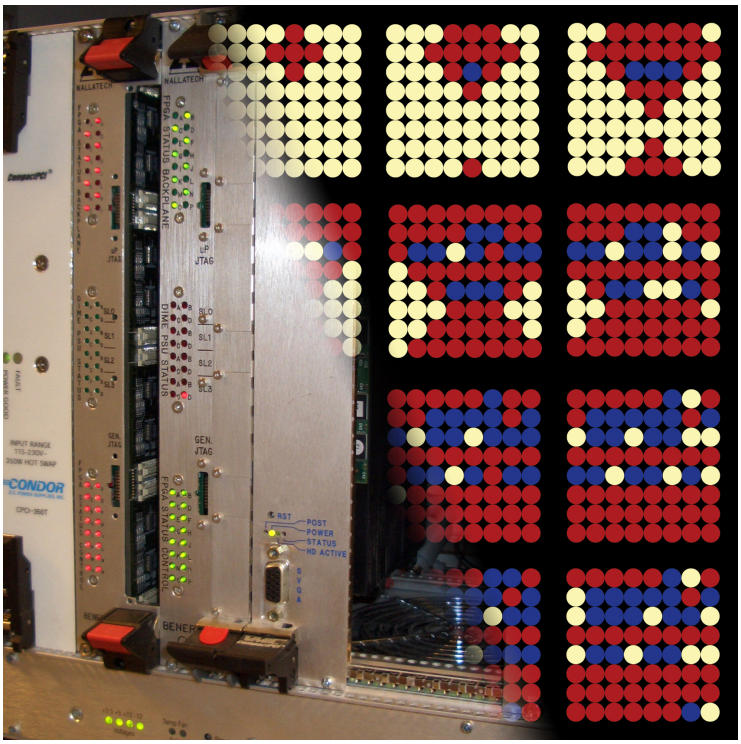
Grunnlaget for EHW er evolusjonære algoritmer (EA), genetiske algoritmer (GA) og genetisk programmering (GP).

Kanskje kan evolusjonær maskinvareutvkiling gjøre bruke av tilgjengelig transistormengde på nye og bedre måter, samtidig som kretsene kan få nye og bedre egenskaper en tradisjonelle metoder vil kunne finne frem til [6].

Som plattform for EHW blir det ofte benyttet en Field-Programmable Gate Array (FPGA), en konfigurerbar digital krets. En FPGA har mange egenskaper som gjør den godt egnet for EHW, men også egenskaper som gjør det vanskelig å oppnå gode resultater [16]. Det har derfor blitt foreslått å lage en Virtuell FPGA [33]. Denne virtuelle FPGAen er designet som en to-dimensjonal ikke-uniform cellulær automat, hvor hver celle blir kalt en SBlock. Cellen har en av to tilstander, ’0’ eller ’1’, og en bestemt type. Typen bestemmes ut fra et regelsett og angir SBlockens oppførsel. Tilstanden bestemmes fra type, samt egen og nabo blokkenes tilstander. Når flere SBlocker blir satt sammen kalles systemet for en SBlock-matrise.

En organisme i en Sblock-matrise utvikles etter gitte regler. Organismen utvikles ved bruk av to operasjoner. Trinn én kalles for development og endrer typen for hver enkelt SBlock, trinn to består av ett eller flere run-step. Development er inspirert av en organismes utviklingen slik det foregår biologien. Et run-step kan endre tilstanden for en SBlock. Ved å benytte disse to operasjonene i kombinasjon utvikles det ett individ.

Denne oppgaven tar sikte på å utvide et eksisterende system for kjøring av SBlock-baserte eksperimenter. Systemet ble opprinnelig laget som en del av en hovedoppgave våren 2003. Designet som ble videreutviklet er ett maskinvaresystemt for kjøringe av eksperimenter på en SBlock-matrise. Systemet blir styrt av en co-prosessor og kan kjøre komplette eksperimenter. Ankepunktet mot systemet er små muligheter for lagring av produserte resultater. Dette medfører at all aktuell informasjon må leses ut. Utlesing er svært tidkrevende operasjoner. Denne oppgaven har derfor bl.a. hatt som mål å lagre tilstandsinformasjon i systemet under kjøring. FPGAen som er benyttet er av typen Xilinx Virtex-E. Virtex-E serien har Block Random Access Memory (BRAM) implementert på brikken. Mulighetene for lagring av data er derfor gode.



Figur 1.1: Systemet kjører på en compact PCI-datamaskin.

Rapporten er organisert som følger:

- Kapittel 2 beskriver bakgrunn for oppgaven. Det beskrives noen relaterte arbeider som er utført. Teknologien som er benyttet i FPGA blir beskrevet og hvilke begrensninger og muligheter som tilbys i benyttet brikke.
- Maskinvaren som er benyttet i oppgaven blir beskrevet i kapittel 3.
- Designet er laget for å kjøre SBlockbaserte eksperimenter. I kapittel 4 forklares SBlockkonseptet.
- Kapittel 5 beskriver systemet før utvidelsene på ett overordnet nivå. Det gis en introduksjon både til maskinvare- og programvaredelen av systemet.
- Utvidelsene som er gjort i denne hovedoppgaven beskrives i kapittel 6.
- Detaljer om hele implementasjonen gis i kapittel 7. Alle samlebånd og tilstandsmaskiner forklares.
- Tester og testresultater blir listet og forklart i kapittel 8.
- Rapporten avsluttes med diskusjon av resultater og arkitektur i kapittel 9. Det blir gitt noen forslag til utvidelser og videre arbeid med systemet, samt at problemer under utviklingen blir nevnt.

Det er lagt ved vedlegg som gjør det lettere å forstå og bruke systemet.

Kapittel 2

Bakgrunn

Transistormengden som kan benyttes i elektroniske kretser har nå blitt så høy at det er svært vanskelig å utnytte kapasiteten [2]. For å løse denne utfordringen kan nye metoder for utvikling bli introdusert. En metode er bruk av evolusjonære metoder, metoder som er inspirert av biologien [5]. Metoden ble foreslått allerede i 1975 [11].

Med evolusjonær maskinvare (EHW) menes at elektroniske kretser utvikles ved bruk av evolusjonære teknikker [10].

Kapittelet starter med å forklare genetiske algoritmer generelt. Deretter beskrives biologisk inspirert maskinvare. To viktig begrep for systemet i denne oppgaven blir så beskrevet, disse er *cellulære automater* og *development (utvikling)*. Det beskrives så noen prosjekter som har vært utført innenfor EHW. FPGA teknologien har vist seg å være svært anvendelig for EHW og siste delen av kapittelet omhandler dette.

2.1 Genetisk programmering

Helt siden 50-tallet har det blitt forsøkt å gi datamaskiner egenskapen *å lære*. Dette blir med et samlebegrep kalt maskinlæring, ML [1]. En undergruppe av ML er genetisk programmering, GP.

Sentralt i GP er et utgangspunkt bestående av en populasjon med tilfeldig valgte individer (mulige løsninger). Disse individene blir ved hjelp av operatorer inspirert av naturlig utvelgelse tilpasset problemet som skal løses. Med GP er det selve programmet som evolveres frem.

En genetisk algoritme, GA, er en iterativ prosess som består av en populasjon av konstant størrelse. Hvert individ består av en streng, av fast lengde, fra ett gitt alfabet [13]. Individet beskriver en mulig løsning på problemet i et løsningsrom. Dette løsningrommet omtales ofte som søkerommet for løsningen. Målet for en genetisk algoritme er å finne et individ som oppfyller kravene til en løsning.

En typisk genetisk algoritme funger ved at det genereres en tilfeldig populasjon av individer. For hvert trinn i evolusjonen genereres en ny populasjon, kalt en generasjon. Hvert individ i en generasjon evalueres ut fra visse bestemte krav. Ut fra hvor godt individet er i forhold til disse kravene settes det en verdi, denne verdien er et mål for *fitness*. Den neste generasjonen genereres med utgangspunkt i foregående generasjons fitnessverdier. Det kan f.eks. brukes en funksjon der sannsynligheten for at et individ videreføres er proporsjonal med fitnessverdien. For at nye individer skal kunne oppstå er ikke en videreføring nok. Det benyttes derfor funksjoner som *kryssing* og *mutasjon*.

GENETIC ALGORITHM

```
1  g ← 0                          ▷ Generation counter
2  Initialize population P(g)
3  Evalute population P(g)        ▷ i.e., compute fitness value
4  while not done
5      do g ← g + 1
6         Select P(g) from P(g − 1)
7         Crossover P(g)
8         Mutate P(g)
9         Evalute P(g)
```

Figur 2.1: Standard genetisk algoritme

Kryssing vil si at to individer kombineres til ett nytt individ, mens mutasjon er at egenskaper ved ett individ endres. Figur 2.1 viser pseudokode for en vanlig genetisk algoritme.

Det er ingen garanti for at en genetisk algoritme finner frem til en løsning. Det angis et maksimalt antall iterasjoner som skal kjøres for at programmet avsluttes.

Hvert individ er representert som en streng tegn fra et endelig alfabet. Denne representasjonen kalles for individets genotype [1]. Genotypen kan sees på som en oppskrift for hvordan individet skal utvikle seg.

Genetiske algoritmer har blitt benyttet med gode resultatet til f.eks. optimalisering, automatisk programmering, maskinlæring, økonomi og en mengde andre områder [28].

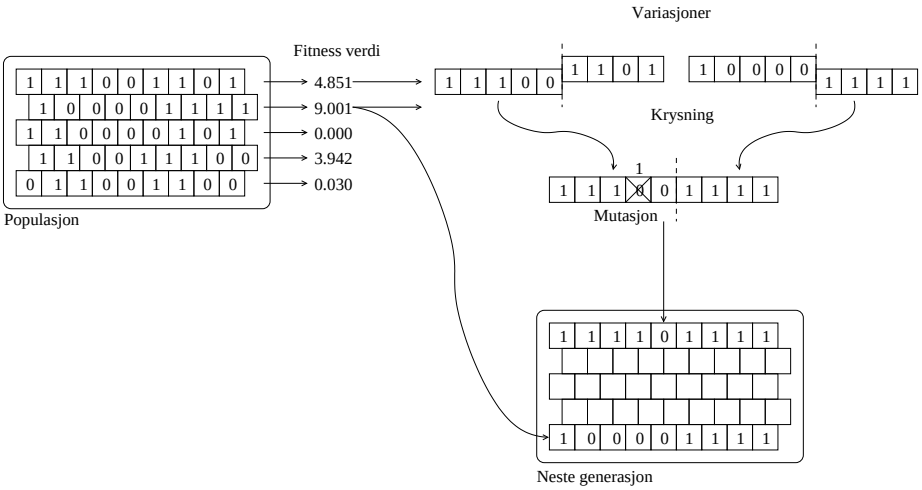
2.2 Biologisk inspirert maskinvare

Evolusjonær maskinvare omhandler kunstig evolusjon brukt i design eller optimalisering av elektroniske kretser. Prinsipielt går det ut på å benytte en evolusjonær algoritme til å kontrollere konfigureringen av en elektronisk brikke, f.eks. en FPGA. Figur 2.1 viser pseudokode for en evolusjonær algoritme. Ytelsen til hvert individ blir målt mot et fitnesskrav, og individene med høyest poengsum blir benyttet videre i prosessen. Dette blir gjentatt inntil kretsen utfører ønsket oppgave eller utviklingen blir avbrutt.

Det blir brukt ulike navn på biologisk inspirert maskinvare, bio-HW. Eksempler er *Evolutionary Electronic, EE, E-Hard* og *Evolvable Systems (ES)*. I resten av oppgaven benyttes *Evolusjonær maskinvare, EHW*.

Figur 2.2 og figur 2.3 viser skjematisk prosessen med utvikling av EHW. Her er hvert individ representert med 9 bit. Det benyttes mutasjoner, krysninger, samt at enkelte individ forblir uforandret for å skape neste populasjon. Denne prosessen gjentas inntil det blir funnet et individ som utfører ønsket funksjonalitet. Det kan ikke garanteres at det blir funnet ett individ som løser oppgaven, slik at etter en gitt tid må prosessen avbrytes.

Før en fitnessfunksjon kan anvendes på et individ må individets genotype vanligvis omformes til individets fenotype. For EHW vil genotypen være en bitstreng. Fenotypen er den elektroniske kretsen som genotypen beskriver. Genotypen sier altså ikke noe om hvordan individet løser oppgaven, derimot er genotypen *oppskriften* på hvordan individet skal se ut. Dette er vist skjematisk i figur 2.3.



Figur 2.2: Eksempel på en populasjon som omformes til neste generasjon via ulike funksjoner [16].

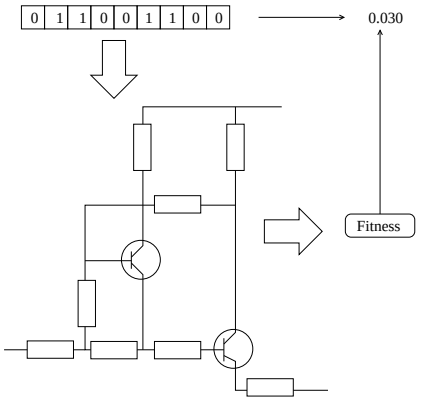
Biologisk inspirert maskinvare kan karakteriseres ut fra to egenskaper. Den ene går på hvor de genetiske operasjonene foregår. Det skilles mellom *offline-utvikling*, operasjonene foregår i programvare, og *online-utvikling*, hvor operasjonene foregår i maskinvare. To andre betegnelser som benyttes er henholdsvis *extrinsic* og *intrinsic* [4]. Den andre egenskapen er hva som er målet med utviklingen, er det definert et endelig mål eller skal organismen utvikle seg åpent. Med utgangspunkt i disse to egenskapene kan det defineres fire kategorier av evolusjonær maskinvare[29]:

- Den første kategorien blir kalt “Evolusjonært kretsdesign”. Hele prosessen foregår i programvare og resultatet lastes inn i en krets til slutt.
- Den andre kategorien omfatter prosesser der elektroniske kretser benyttes ved utvikling, men det meste foregår også her i programvare.
- I den tredje kategorien finnes systemer hvor alle genetiske operasjoner er utført i maskinvare. Hovedforskjellen fra naturlige organismer er at målet med utviklingen er bestemt. En bestemt funksjonalitet er ønskelig å realisere.
- Den siste kategorien omfatter systemer med en populasjon av organismer og mangelen på fastbestemt mål. Organismene endres ut fra hva som best er tilpasset miljøet.

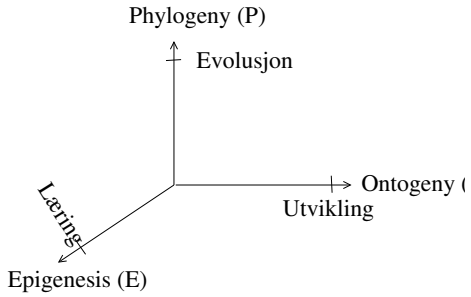
2.2.1 POE modellen

Sipper *et.al.* kategoriserer i [29] EHW etter ulike akser. De tre aksene som benyttes er (se figur 2.4):

- *Phylogeny*: Denne aksen brukes for å beskrive det som tradisjonelt menes med evolusjon, nemlig at en organisme vha. reproduksjon tilpasses miljøet den lever i.



Figur 2.3: Eksempel på at genotypen til et individ omformes til fenotypen før fitnessevaluering[16].



Figur 2.4: POE-modellen har tre akser; Phylogeny(P), Ontogeny(O) og Epigenesis (E).

- *Ontogeny*: Utvikling av en enkelt organisme beskrives på denne aksen. Deling av celler for å utvikle en flercellet organisme beskrives her. Ontogeny kan også beskrives som *utvikling*.
- *Epigenesis*: Etter en vis mengde med celledeling må organismen kunne bruke informasjon fra omverdenen for å fortsette utviklingen. Denne aksen kan også kalles for *læring*.

Eksempler på teknikker som befinner seg på hver av de tre aksene er: (P) Evolusjonære algoritmer, (O) multicellulære automater hvor en enkelt mor-celle formerer seg til flere celler og (P) kunstige nevrale nettverk.

2.2.2 Maskinvare for EHW

Plattform for utvikling av evolusjonær maskinvare må tilfredsstille en del kriterier. En plattform som oppfyller de fleste krav er FPGA.

En FPGA er en krets som kan konfigureres til ønsket funksjonalitet. Konfigurasjon kan gjøres mange ganger, enten kan hele kretsen konfigureres eller bare enkelte deler. Delviskonfigurering kan gjøres når kretsen er i bruk. Se kapittel 2.6 for mer informasjon om FPGA.

Selv om mange av egenskapene ved en FPGA gjør den egnet til EHW, er det også egenskaper som ikke er like gode [15]. For det første er en FPGA tilpasset digitale

design, de analoge egenskapene er det langt vanskeligere å utnytte. For det andre er sammenkoblingsnettverket i en FPGA laget med utgangspunkt i et tradisjonelt blokkbasert og synkront design. Ved bruk av evolusjonære metoder kan dette legge en begrensning på resultatet. Et tredje problem er de svært begrensede mulighetene en FPGA gir til å analysere et resultatet; det finnes ikke muligheter for målinger internt i kretsen [5].

Layzell beskriver i [14] og [15] ett design som åpner for langt flere muligheter enn en tradisjonell FPGA. Plattformen som foreslås er en matrise med analoge brytere og mulighet for montering av opp til seks kort med "grunnelementene" for prosessen. Det er rundt 1500 brytere, dette tilsvarer et søkerom på 10^{420} mulige kretser. Selv om en stor mengde av disse er ulovlige, da det medfører kortslutninger i kretsen, har kortet muligheter for et svært stort søkerom. Kortet kan styres fra en datamaskin. En viktig funksjon på kortet er tilgangen for å utføre målinger direkte på komponenter. Kortet har blitt benyttet med suksess for å konstruere enkle logiske funksjoner vha. transistorer [15].

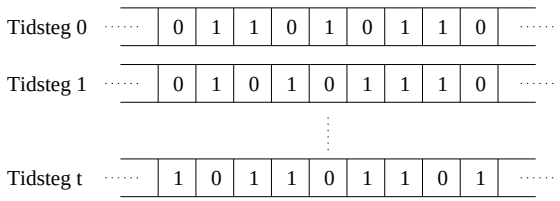
Murakawa *et. al.* beskriver i [20] en brikke for evolusjonær maskinvare for nevrale nettverk. Denne brikken er tenkt brukt i miljøer som endrer seg. Brikken kan da selv oppfatte endringen og om konfigurere seg etter en angitt funksjon. Brikken inkluderer en 32-bit RISC prosessor.

Av analoge plattformer kan nevnes:

- Analogue EHW filter chip [19].
- Analog Neural Chip [31].
- Palmo Analoge Signal Processing IC [8].

2.3 Cellulære Automater

Cellulære automater (CA) ble introdusert av von Neumann [24].



Figur 2.5: En CA i en dimensjon består av ett array med celler. Hver celle har kun forbindelse til sine to naboer, det er ingen global kommunikasjon i systemet. For hvert tidssteg endres tilstandene for cellene etter gitte regler [27].

En enkel CA består av én dimensjonal matrise av celler [27], figur 2.5, som kan være uendelig i begge retninger. Tidsintervallene er diskrete og hver celle er i en bestemt tilstand i hvert tidssteg. Tilstanden velges fra et endelig alfabet. Cellene endrer tilstand for hvert tidssteg avhengig av egen og nabocellenes tilstand. Funksjonen som beskriver endringen i tilstand er lik for alle celler. En CA har ingen ekstern kommunikasjon. Ved oppstart er CAen i en initiell tilstand, og endres deretter for hvert tidssteg. Endringene er fullstendig deterministiske.

En CA er turing komplett [27]. Turing komplett betyr at systemet kan emulere en Turing maskin (TM) [35]. TM er en ekstremt enkel datamaskin som kan simulere

en hvilken som helst annen datamaskin [1]. Altså kan en hver datamaskin som kan simulere TM også simulere andre maskiner som kan simulere TM. Dette betyr at en CA er en svært generell struktur som kan utføre alle mulige beregninger og operasjoner som andre datamaskiner kan utføre.

En CA karakteriseres av fire egenskaper [27]: geometriske egenskaper, reglene for tilstandsendring, mulige tilstander og naboskapet for hver enkelt celle.

En CA kan være enten uniform eller ikke-uniform. I en uniform CA har alle celler det samme regelsettet, mens i en ikke-uniform CA har cellene ulike regelsett.

Et eksempel på en CA er henter fra [28]. Eksempelet viser en to-dimensjonal matrise hvor hver celle har fire naboer. Funksjonaliteten som er realisert er *paritets regelen*. Hver celle blir satt til '1' ved neste tidssteg dersom summen av egen tilstand og tilstanden til de fire naboene er odde, og til '0' dersom summen er lik (funksjonaliteten kan også beskrives som modulo-2 addisjon). Regler for alle kombinasjoner av tilstander er vist i tabell 2.1. Tilstanden for matrisene ved noen tidssteg er vist i figur 2.6.

CNØSV	S_{neste}	CNØSV	S_{neste}	CNØSV	S_{neste}	CNØSV	S_{neste}
00000	0	01000	1	10000	1	11000	0
00001	1	01001	0	10001	0	11001	1
00010	1	01010	0	10010	0	11010	1
00011	0	01011	1	10011	1	11011	0
00100	1	01100	0	10100	0	11100	1
00101	0	01101	1	10101	1	11101	0
00110	0	01110	1	10110	1	11110	0
00111	1	01111	0	10111	0	11111	1

Tabell 2.1: Paritetsregler. CNØSV betyr tilstanden til cellene nord, øst, sør og vest for aktuell celle (Center). S_{neste} er tilstanden for center ved neste tidsteg.

2.4 Utvikling (development)

Utvikling av en organisme skjer langs x-aksen (ontogeny) i figur 2.4. Utvikling beskriver sammenhengen mellom genotypen og fenotypen [1]. Utvikling av en organisme foregår med utgangspunkt i genotypen. I genotypen ligger informasjon om hvordan organismen skal bygges. Ved å følge denne planen blir organismen utviklet [1].

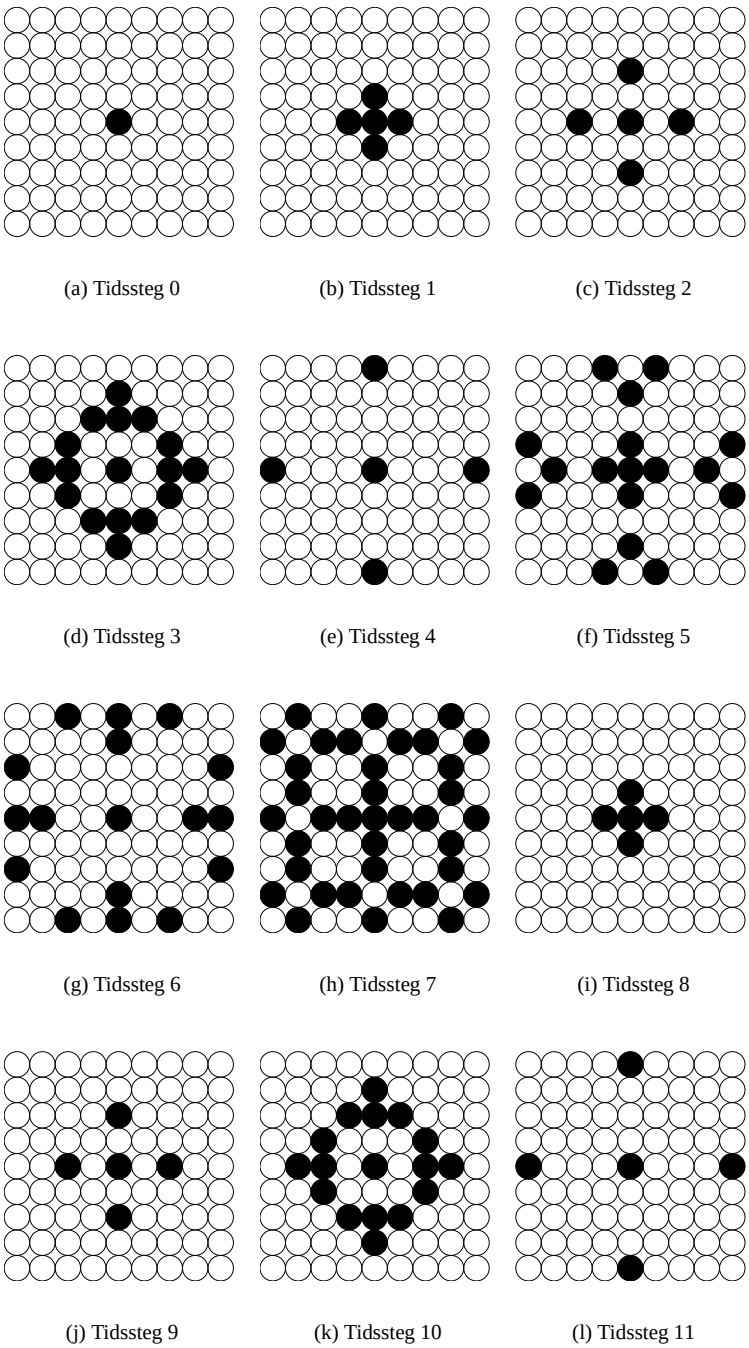
Overgangen fra genotype til fenotype kalles genotype-fenotype-omforming[1]. Det er vanlig å benytte en en-til-en omforming, dvs. at en genotype fører til en bestemt fenotype.

For GP er det ikke nødvendig å skille genotypen fra fenotypen ved å benytte en utviklingsprosess, men utvikling kan benyttes som er verktøy for å forbedre prosessene. Ved bruk av GA er det vanlig å benytte en omforming.

2.5 Relatert arbeid

I dette avsnittet beskrives noen prosjekter som har vært utført med EHW.

[1] eng. Genotype-phenotype-mapping



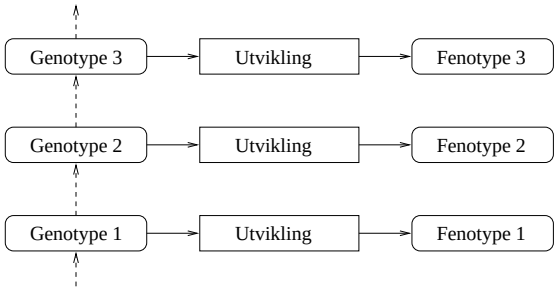
(a) Tidssteg 0 (b) Tidssteg 1 (c) Tidssteg 2

(d) Tidssteg 3 (e) Tidssteg 4 (f) Tidssteg 5

(g) Tidssteg 6 (h) Tidssteg 7 (i) Tidssteg 8

(j) Tidssteg 9 (k) Tidssteg 10 (l) Tidssteg 11

Figur 2.6: Figuren viser hvordan en CA som har implementert paritetsregelen oppdaterer sine tilstander. Svart sirkel betyr at cellen har tilstanden ’1’, hvit sirkel betyr tilstanden ’0’.



Figur 2.7: Ett individs genotype endres til individets fenotypen gjennom en utviklingsprosess [29].

2.5.1 Firefly – En online EHW

Målet med denne maskinen var å demonstrere at online evolusjon kan fungere i praksis, altså at en evolusjonær krets kan utvikles uten tilkobling til en datamaskin. Målet for kretsen var en endimensjonal synkroniserings-operasjon. Systemet er kalt for *Firefly* [28]. Med endimensjonal menes her at hver celle har to naboer. Målet for hele systemet er at ut fra en tilfeldig startsekvens skal alle cellene synkroniseres. Det er vesentlig å merke seg at alle celler kun har kontakt med sine naboer, det finnes ingen global kommunikasjon i systemet.

Maskinvaren som er konstruert består hovedsaklig av FPGAer. Det er LCD-display for utlesing av tilstander og brytere for justering av parametre. Hver celle har også en LED som viser tilstanden og en bryter for manuell justering av tilstanden for hver celle. Den eneste eksterne tilkoblingen er strømforsyning.

Målet for hver celle er å komme inn i sekvensen $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$, veksle mellom 1 og 0. Dersom dette er oppnådd settes fitness verdien til 1, ellers 0.

Firefly er en ikke-uniform synkron CA. For hvert tidssteg endres utgangsverdien ut fra aktuelt regelsett (som kan være ulikt i alle celler).

Hver celle har en fitness funksjon og to komparatorer for å sammenligne egen fitness med sine to naboer. Avhengig av hvordan nabocellene har blitt evaluert vil regelsettet som cellen benytter endres etter metoder som *krysning* og *mutasjon*.

2.5.2 Embryonics – Biodule

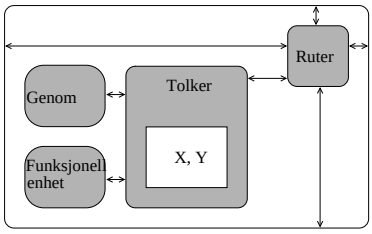
Tempesti beskriver i [32] ett design som forsøker å kopiere oppførselen til celler fra naturen vha. elektronikk. Ved å la seg inspirere av naturen har det blitt laget en todimensjonal matrise av identiske celler. Hver celle kan kjøre en spesifikk del av genomet, avhengig av hvor i matrisen celler er plassert.

Siden alle celler inneholder hele genomet, kan ødelagte celler erstattes av tilleggsceller (celler som opprinnelig ikke var i bruk). På denne måten oppnåes reparasjon av organismen.

For å lage en elektronisk celle, figur 2.8, kreves følgende:

- Minne for å lagre genomet.
- Et koordinatsystem som holder kontroll med hvor i organismen cellen befinner seg.
- En tolker som leser og kjører data kodet i genomet.

- En funksjonell enhet for data prosessering.
- Tilkobling for naboceller.



Figur 2.8: Komponenten i en kunstig celle [32].

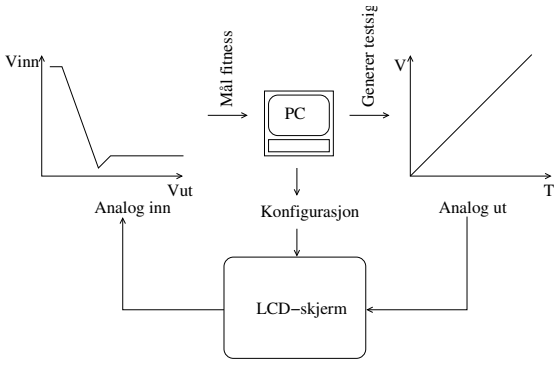
For å illustrere mulighetene ved dette designet har det blitt produsert en prototype i maskinvare, kalt *MicTree - Tree for microinstruksjons*. Vha. denne prototypen har det blitt laget applikasjoner som:

- *BioWatch*: Dette er en klokke med minutter og sekunder realisert med tre MicTree-enheter [30].
- Tallgenerator: En generator som produserer tilfeldige tall vha. fem celler [17].
- Spesialisert Turing Maskin: En turingmaskin med selvreparerende egenskaper [26].

Prosjektet har vist at det lar seg gjøre å utvikle kompleks maskinvare ved å benytte inspirasjon fra biologien.

2.5.3 Evolusjon i materialer – "Evolution in materio"

Det meste som er gjort av forskning på EHW er gjort med standard elektroniske komponenter. Det har blitt hevdet at fordi slike komponenter er menneskeskapt, er det ikke så godt egnet for oppgaven [18]. Julian Miller og Simon Harding beskriver i [9] et eksperiment som er utført med en LCD-skjerm som plattform.



Figur 2.9: En LCD-skjerm brukes for utvikling av maskinvare [9].

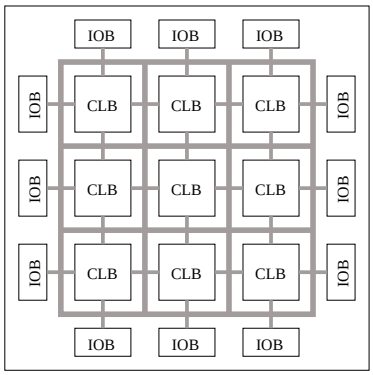
Figur 2.9 viser oppsett for å benytte en LCD-skjerm som plattform. Displayet benyttes ikke for å vise data i dette oppsettet. Det er kun sammenhengen mellom inn-

og ut-spenning som er aktuell. Når en spenning pådyttes displayet genereres det også en spenning på utgangen. Ved å måle sammenhengen mellom inn- og ut-spenning, og å evaluere denne mot en fitnessfunksjon kan konfigurasjonsdataen til skjermen endres. Denne sammenhengen avhengig av hvilken konfigurasjon som er lastet.

Eksperimentet har vist at det kan benyttes ulike materialer som plattform for EHW.

2.6 Field Programmable Gate Array

Dette avsnittet beskrives Field Programmable Gate Array (FPGA).



Figur 2.10: Hovedkomponentene i en FPGA er konfigurerbare logiske blokker (CLB), inn/ut-blokker (IOB) og sammenkoblingsnettverk [39].

Figur 2.10 viser en hvordan en FPGA er bygget opp. Hovedkomponentene er konfigurerbare logiske blokker (CLB), I/O-blokker og et konfigurerbart sammenkoblingsnettverk. CLBene er plassert ut i en matrise på brikken. Mellom nabo CLBer er det lokal sammenkobling. I ytterkantene og på tvers av brikken finnes det ruting for global kommunikasjon. I utkanten av brikken er det plassert I/O-enheter.

Innholdet i CLBene og hvilke koblinger som skal eksistere i sammenkoblingsnettverket velges ved konfigurasjon av brikken. Konfigurasjon kan gjøres så mange ganger som ønskelig.

2.6.1 Xilinx Virtex-E

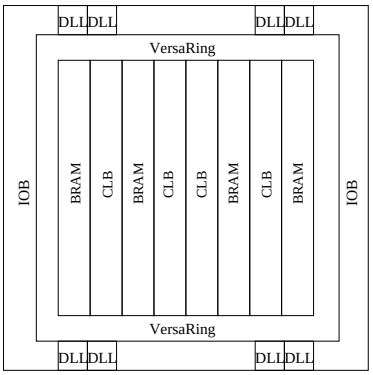
FPGA som er benyttet i denne oppgaven er av typen Xilinx Virtex-E [39]. Brikken har spesifikasjonene gitt i tabell 2.2. Figur 2.11 viser oppbyggingen til kretsen.

Konfigurerbare logiske blokker – CLB

CLBene benyttes for konstruksjon av logiske funksjoner [39]. Hver CLB består av to “slice”, se figur 2.12. En “slice” er igjen delt opp i to logisk celler (LC), se figur 2.13. En LC inneholder en 4-inngangs funksjonsgenerator, mente logikk (eng. carry logic) og et minne element. Utgangen fra funksjons-logikken driver en utgang fra en CLB og inngangen på en D-vippe. Hver CLB har også logikk for å sette sammen flere fire-inngangs funksjonsgeneratorer slik at det dannes funksjonsgeneratorer med flere innganger.

Enhet	Antall
System Gates	1 569 178
Logic Gates	331 776
CLB Array	64 x 96
Logic Cells	27 648
Differential I/O Pairs	281
User I/O	660
BlockRAM Bits	393 216
BlockRAM Modules	96
Distributed RAM Bits	393 216

Tabell 2.2: Spesifikasjoner for Xilinx-E XCV1000E [39].



Figur 2.11: Xilinx Virtex-E består av IOBer, CLBer og BRAM [39].

Funksjonsgeneratorene er implementert som oppslagstabeller[2]. Ved siden av å fungere som funksjonsgeneratorer kan en LUT også brukes som 16x1 bit synkron RAM. Funksjonsgeneratoren kan også brukes som 16 bits skiftregister.

Hver CLB har muligheter for å rute signaler. Denne muligheten utnyttes for å øke rutingskapasiteten i brikken.

Hver CLB inneholder to tre-nivå-drivere[3], disse er ikke vist i figuren.

Inn-/ ut-blokker – IOB

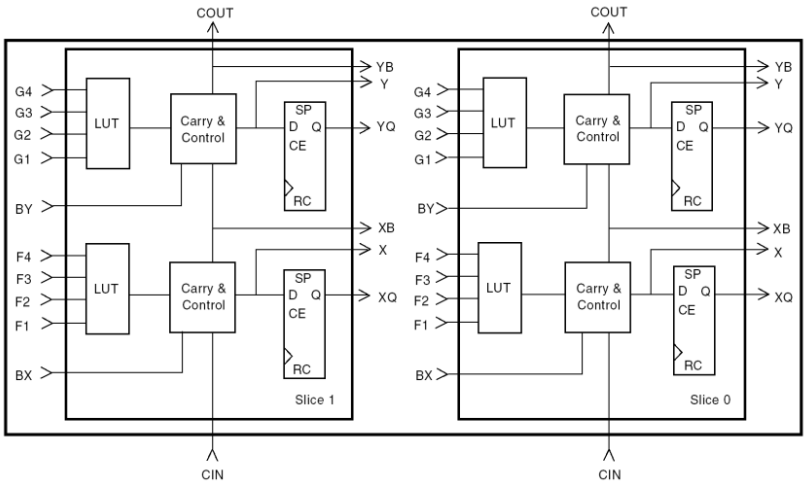
Hver IOB er knyttet opp mot en fysisk pinne på FPGAen. Det er tre minneelementer i hver IOB. Disse brukes enten for inngangs- eller utgangs-signaler.

I/O-blokkene i Virtex-E støtter en mengde standarder. For oversikt over støttede standarder henvises det til datablad [39].

Block SelectRAM – BRAM

Virtex-E inneholder spesielle strukturer for bruk som minneelementer. Disse kalles Block Select Random Access Memory (BRAM) og kan brukes i tillegg til minneelementer generert i CLB. BRAM er distribuert rundt i brikken. Se figur 2.11.

[2] eng. look-up tables (LUT)
[3] eng. 3-state-drivers



Figur 2.12: En CLB består av to "slicer" [39].

BRAM er ekte *to-port synkron RAM*. En BRAM blokk kan altså både leses og skrives til samtidig. Men samme adresse kan ikke både leses og skrives i samme periode.

Digital Delay-Locked Loops – DLL

DLL benyttes for å gjøre klokkesignalene stabile. Det finnes åtte DLLer i Xilinx Virtex-E. Modulen har funksjonalitet for å multiplisere eller dividere klokkehastigheten med 1.5, 2, 2.5, 3, 4, 5, 8 eller 16.

VersaRing

Langs ytterkanten av brikken er det en komponent for sammenkobling av blokker. Spesielt brukes komponenten for I/O-ruting.

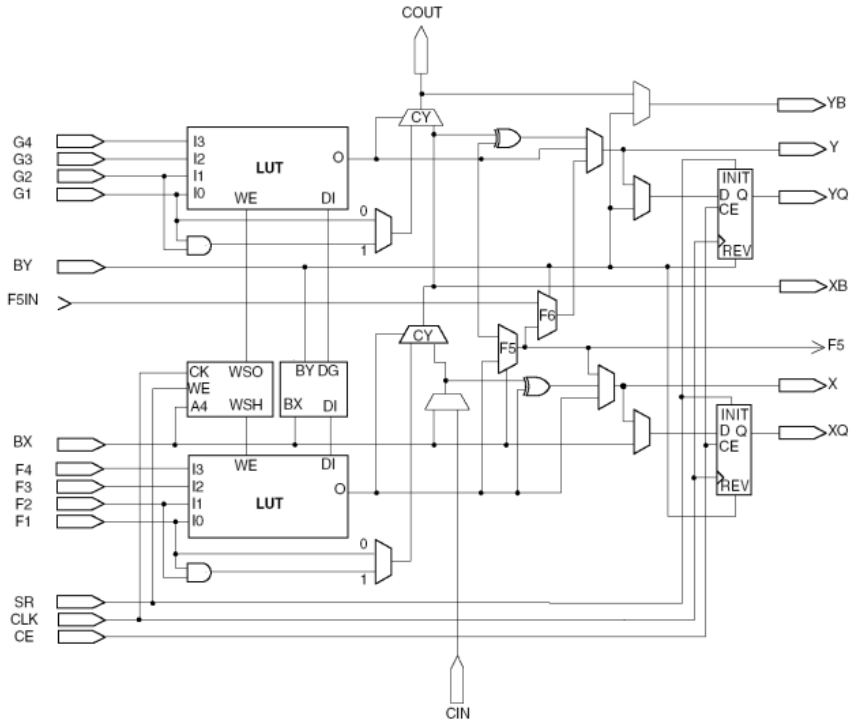
2.6.2 Konfigurering

Konfigurering av en FPGA skjer ved at det lastes en bitstrøm til brikken [38]. Denne bitstrømmen er ofte generert ved implementasjon av et design skrevet i et maskinvarebeskrivende språk. Bitstrømmen kan lastes gjennom JTAG grensesnittet til Virtex-E. Men andre konfigurasjonsmodi støttes også.

Bitstrømmen består av en kombinasjon av kontrollkommandoer og konfigurasjonsdata. Dersom brikken forsøkes konfigureres med en bittstrøm som ikke inneholder korrekt data kan brikken ødelegges.

Virtex konfigurasjonsminnet er delt opp i vertikale rammer, der hver ramme er den minste enheten som kan konfigureres. Flere rammer er igjen samlet i en kolonne. I praksis betyr dette at større deler av systemet må programmeres samtidig.

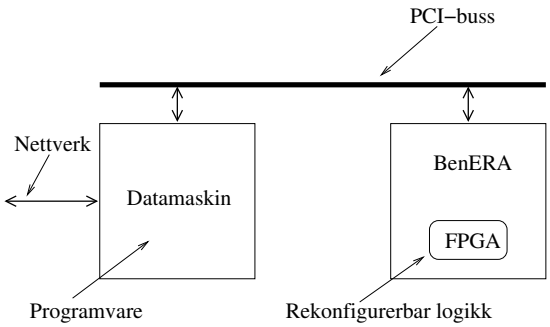
Datamengden for konfigurasjon av Virtex-1000E er på 6 587 520 bit [39].



Figur 2.13: En slice består av to LUTer og to vipper [39].

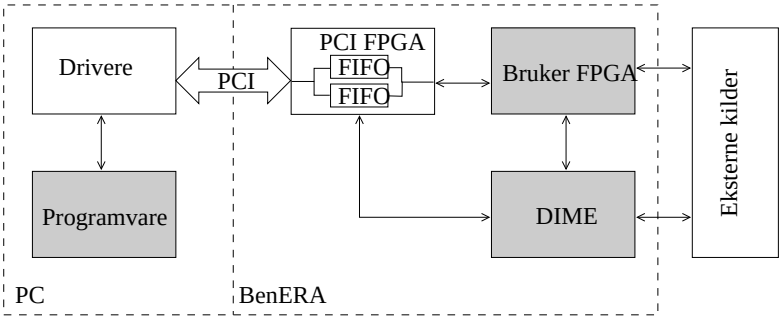
Kapittel 3

Maskinvareplattform



Figur 3.1: FPGAen som benyttes er montert på ett PCI-kort, PCI-kortet er koblet mot datamaskinen over PCI-bussen.

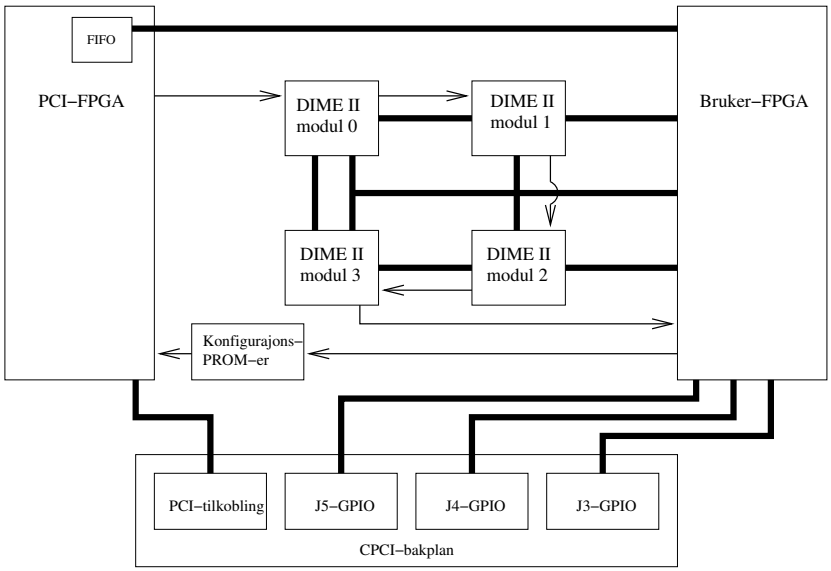
FPGAen som er benyttet i denne opppgaven er montert på et kort fra BenERA [21]. Dette kortet er montert i en datamaskin med nettverkstilkobling. Kommunikasjon mellom BenERA kortet og datamaskinen foregår over PCI-bussen slik figur 3.1 viser. Ingen fysisk tilgang til FPGA brikken er nødvendig for kjøring av systemet, all kontroll skjer over nettverket.



Figur 3.2: En applikasjon som skal kjøres på BenERA-kortet er partisjonert i en maskinvare- og en programvaredel [21]. De skraverte blokkene kan konfigureres av bruker.

Et system som skal kjøre på BenERA kortet består av to deler; en maskinvaredel og en programvaredel. Figure 3.2 viser hvordan systemet er partisjonert og hvordan de ulike delene kommuniserer. Kun de skraverte modulene skal konfigureres av brukeren. Modulen merker *Programvare* er et kontrollprogram som styrer systemet. Modulen *Drivere* sørger for å kommunisere med kortet. *PCI-FPGA* er transparent for brukeren, men benyttes for kommunikasjon med *Bruker-FPGA* hvor selve designet kjøres. *DIME* kan benyttes for å utvide kapasiteten til BenERA kortet. *Eksterne kilder* kan være ulike komponenter som skal sammarbeide med systemet.

3.1 BenERA FPGA-kort



Figur 3.3: Forenklet figur over BenERA FPGA-kortet [3].

BenERA cPCI-kortet gir tilgang til en Xilinx 1000E FPGA. Det er plassert to FPGAer på kortet, men kun en av dem benyttes for eget design. Figur 3.3 viser en noe forenklet oversikt over komponentene på kortet.

PCI-FPGAen er prekonfigurert og sørger for konfigurasjon og kontroll av resten av kortet. PCI-FPGAen er koblet til PCI-bussen i vertsmaskinen og dermed ansvarlig for kommunikasjon ut og inn på kortet. Klokken som benyttes for kommunikasjon mellom PCI FPGA og bruke FPGA skal være på 33-40 MHz (40 MHz er anbefalt) [21].

DIME-modulene [1] gjør kortet utvidbart dersom dette er ønskelig. Modulene som kan settes inn er f.eks. ekstra FPGAer.

Kortet er utstyrt med tre separate klokkegeneratorer som alle kan benyttes til eget design. Dette muliggjør bruk av flere klokkedomener.

BenERA kortet er montert i en datamaskin. Datamaskinen som er benyttet er en CompactPCI maskin. Maskinen kjører Debian GNU/ Linux. Kontroll av maskinen blir gjort via SSH. De viktigste spesifikasjonene til denne maskinene er:

[1] DSP and Image Processing Modules for Enhanced FPGAs

- To Pentium III-prosessorer montert på et hovedkort av typen VMICPCI-7760 [37].
- I/O-kort av typen VMIACC-0320 [36].
- 512 MB RAM
- To 10/ 100 MBit ethernetkort

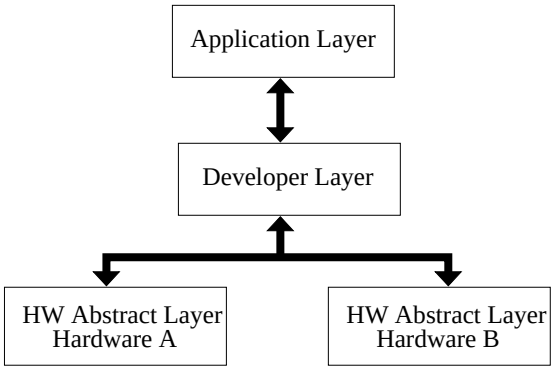
Maskinen har montert to BenERA FPGA-kort.

Når en egenprodusert bitfil (se kapittel 2.6.2) lastes opp er det bruker-FPGAen som blir konfigurert. Konfigurasjonen skjer via JTAG [12].

Som figur 3.3 viser har ikke bruker-FPGAen tilgang til PCI-bussen direkte. Derfor sender og mottar den data gjennom PCI-FPGAen. PCI-FPGAen inneholder to FIFO buffer som all data går gjennom. Mellom de to FPGAene er det en egen databuss på 40 bit, 32 av disse benyttes til data. Denne bussen er nærmere forklart i vedlegg H.

3.2 FUSE API

Kontroll av FPGA-kortet gjøres gjennom en egen API[2], FUSE (Field Upgradeable Systems Enviroment) [23]. FUSE er et programvarebibliotek som gjør det mulig å utvikle programvare som benytter BenERA.



Figur 3.4: FUSE API Layer [22]

Figure 3.4 viser programvare lagene i FUSE API. Maskinvareabstraksjonslaget er koblet direkte mot maskinvaren. Det gis ikke tilgang direkte til dette laget for utvikling av programvare. Tilgang til maskinvaren gis via et utviklingslag. Utviklingslaget består av et bibliotek kalt DIMESDL[3]. DIMESDL inneholder funksjoner for kontroll av maskinvaren.

Eksempler på funksjoner som tilbys gjennom DIME er lokalisering av FPGA-kort, åpning og lukking av tilgang til kort, kontroll av klokkefrekvens, reset av kort, kontroll med interrupt, overføring av data, konfigurasjon, utlesing av systeminformasjon og I/O kontroll [22].

[2] Application Program Interface
[3] DIME Software Development Library

Kapittel 4

SBlock

SBlock-begrepet ble introdusert i [5] som en arkitektur som er bedre egnet for evolusjonær maskinvare (EHW) enn en vanlig FPGA.

Det er nødvendig å ha en lett tilgjengelig plattform for EHW for at det skal gjøres fremskritt innenfor forskningsområdet [7]. Det er lite trolig at en slik plattform vil komme på markedet med det første. Det er derfor nødvendig å utvikle en plattform som enkelt kan realiseres på tilgjengelig teknologi.

En av hovedutfordringene med EHW for komplekse kretser er valg av representasjon [7], altså valg av genotype. Generelt antas det en en-til-en overgang mellom genotypen og fenotypen, se kapittel 2.4. Dette medfører en genotypen må inneholde all informasjon som er nødvendig for utvikling av fenotypen. For FPGA betyr dette både informasjon om logikken og rutingen. For komplekse elektroniske kretser fører dette til en kompleks genotype.

Med økende genotype øker også regne- og minneforbruket ved bruke av evolusjonære metoder [33, 34]. Ved å overføre kompleksiteten fra genotype-representasjonen til genotype-fenotype-omformingen kan genotypen reduseres. Reduksjon i genotype kan oppnåes ved å gi en beskrivelse av hvordan organismen skal utvikles sammen med genotypen.

4.1 Virtuell FPGA

Den virtuelle FPGA en bygget opp av blokker lagt ut i en to-dimensjonale matrise. Hver enkelt blokk kalles en SBlock. Det er ingen global kommunikasjon i matrisen, kun lokal ruting mellom naboblokker.

Den virtuelle FPGAen kan konfigureres med en langt mindre datastrøm enn en vanlig FPGA. Dette er oppnådd bl.a. ved at all ruting er fast. I tillegg til å redusere datamengden er også mulighetene for kortslutninger ved konfigurering fjernet [7].

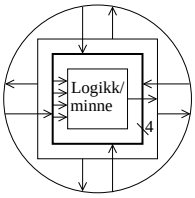
4.1.1 SBlock

Funksjonalitet til en SBlock er svært enkel. For hvert tidssteg endres utgangen i forhold til en funksjon av de fem inngangsverdiene. Funksjonen er bestemt av SBlockens type. En SBlock omtales også som en celle.

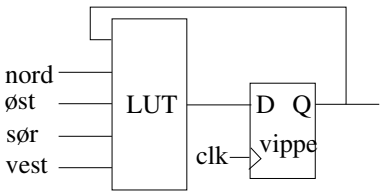
En SBlock er vist i figur 4.1(a). SBlocken består av logikk/ minneelement og ruting. Det er en inngang og en utgang på hver side av elementet. Det er samme utgangs-

signalet på alle fire sider. Funksjonen kan realiseres som en oppslagstabell[1] og en vippe, vist i figur 4.1(b). En n-inngangs LUT er en funksjonsgenerator som kan realisere alle n-inngangs boolske operasjoner. De fem inngangene er fra fire naboer og egen vippe.

En SBlock kan konfigureres som en logiske funksjon for alle eller noen av inngangen. Ved å konfigurere LUTen som en kopieringsfunksjon for et av inngangsignalene vil SBlocken fungere som ett rutingselement.



(a) En sblock har en inngang fra hver naboblokk og en utgang i hver retning.



(b) En sblock kan realiseres som en LUT og en vippe

Figur 4.1: SBlock [7].

En SBlock kan realiseres som en fem-inngangs LUT og en D-vippe. Ved å kombinere to fire-inngangs LUT lages det en LUT med fem-innganger. Figur 2.13 viser en "slice" i en Xilinx Virtex-E som inneholder to fire-inngangs LUT og to D-vipper. Det lar seg derfor gjøre å realisere én SBlock per "slice".

Hver oppslagstabell konfigureres med data avhengig av SBlockens type. Datainnholdet i hver LUT er $2^5 = 32$ bit. Et eksempel på data er hentet fra [7]. Eksempelet realiserer funksjonaliteten $NorS$, verdiene er vist i tabell 4.1. Signalene fra Ø og V har her ingen betydning, det viser at selv om rutingen er fast kan det velges hvilke signaler som skal påvirke utgangen.

CNØSV	Data	CNØSV	Data	CNØSV	Data	CNØSV	Data
00000	0	01000	1	10000	0	11000	1
00001	0	01001	1	10001	0	11001	1
00010	1	01010	1	10010	1	11010	1
00011	1	01011	1	10011	1	11011	1
00100	0	01100	1	10100	0	11100	1
00101	0	01101	1	10101	0	11101	1
00110	1	01110	1	10110	1	11110	1
00111	1	01111	1	10111	1	11111	1

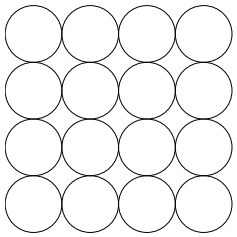
Tabell 4.1: LUT-verdier som realiserer $NorS$.

4.1.2 SBlock-matrise

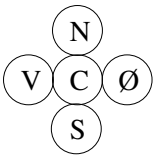
En SBlock-matrise er satt sammen av flere SBlocker. Sammenkoblingsnettverket er fast definert. En SBlock er koblet til alle sine naboblokker. Figur 4.2(a) viser ett sammen-

[1] eng. look-up table, LUT

koblingsnettverk for en *fire ganger fire* matrise. Kolonnen til høyre er koblet til kolonnen til venstre og motsatt, samme gjelder for øvre og nedre rad.



(a) En SBlock-matrise er laget av SBlocker satt sammen i en 2-dimensjonal matrise.



(b) Hver SBlock har en nabo i hver retning.

Figur 4.2: SBlock-matrise.

En SBlock-matrise kan sees på som en cellulær automat [7]. CAen kan enten være uniform (ved at alle SBlocker har samme type) eller ikke-uniform (ved at ulike SBlocker har ulik type).

4.2 Utvikling av organisme

En organisme utvikles fra ett sett regler. Disse reglene beskriver hvordan de ulike SBlockene utvikles og endrer type.

Algoritmen som benyttes i systemet er todelt. Etter at initialisering er gjort kan det kjøres en kombinasjon av development- og run-step. Denne prosessen er vist med pseudokode i figur 4.3. Antall run- og delvelopment-step som skal kjøres er inngangsverdier til algoritmen.

```
SBLOCK(dev, run)
1  init                          ▷ Init matrix and load rules
2  d ← dev
3  while d > 0
4       do d ← d − 1
5          r ← run
6          while r > 0
7               do r ← r − 1
8                  RUN           ▷ One run-step
9          DEVELOPMENT           ▷ One developement-step
```

Figur 4.3: Algoritme for kjøring av SBlock- matrisen

Development-step Et development-step endrer typen til en SBlock. Dette vil i praksis si at innholdet i oppslagstabellen endres. Endringen i type skjer avhengig av hvilke

regler som er aktuelle, se kapittel 4.2.1.

Run-step I løpet av et run-step kan tilstanden for hver av SBlockene endres. Hver blokk kan ha en av to tilstander, '0' eller '1'. Alle SBlockene endrer tilstand synkront. Hvilken tilstand en SBlock får bestemmes ut fra type og tilstandene til naboblokkene og egen tilstand.

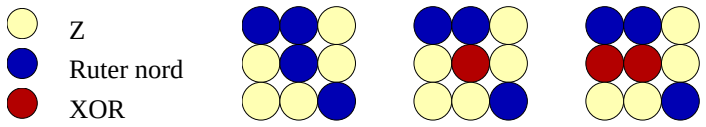
4.2.1 Regler

Et development-step endrer SBlock-matrisen ut fra aktuelle regler. En regel består av en resultat- og en betingelse-del. Resultatdelen beskriver hva som skal skje med en SBlock som aktiverer regelen. Betingelsedelen angir hva som må være oppfylt for at regelen skal aktiveres.

Det finnes to typer regler, henholdsvis regler som medfører endring av type og endringer som medfører at en tom blokk gis en funksjon. Disse kalles *change* og *growth* i [33].

Growth Denne regeltypen gjør at en ubrukt SBlock tillegges en bestemt type. Det finnes fire typer av denne regelen, henholdsvis G_N, G_S, $G_\oslash$ og G_V. Hver av regeltypene medfører vekst, fra gitt retning, dersom den blir aktivert. Vekst betyr at SBlocktypen blir kopiert over fra gitt naboblokk.

Change Regler av denne typen fører til at en SBlock endrer type. Regelen kan ikke benyttes av en ubrukt (tom) SBlock.



(a) Tegnforklaring (b) Utgangstilstand. (c) Change (endring) (d) Growth (vekst)

Figur 4.4: Ut fra en starttilstanden aktiveres det en endrings-regel (4.4(c)), deretter aktiveres en vekst-regel (4.4(d)) [33].

Reglene er rangert slik at regel én har høyeste prioritet. Dersom flere regler aktiveres er det altså regelen med lavest nummer som blir benyttet. [2]

4.2.2 Eksempel på utvikling av en organisme

Dette eksempelet er hentet fra [33].

Tabell 8.2 viser aktuelle regler. I tabellen betyr *Gx* vekst fra x-retning, celletypen som er i x-retning for aktuell type kopieres altså over til aktuell celle. $T > 0$ betyr at cellen får funksjonen *"sett utgangen høy dersom minst en inngang er høy"*. Denne typen regel kan finnes med ulike verdier for *T*. *ØR* står for Øst Ruter og har funksjonen *"overfør signalet fra øst til utgangen"*.

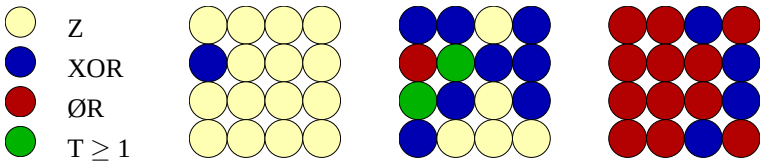
[2] Dette er i motsetning til hvordan implementasjonen av systemet er gjort, her har regler med høyest nummer også høyest prioritet.

	Resultat	C	S	Ø	N	V
Regel 1	Gs	Z	XOR	DC	DC	DC
Regel 2	Gø	Z	DC	XOR	DC	DC
Regel 3	Gv	Z	DC	DC	XOR	DC
Regel 4	Gn	Z	DC	DC	DC	XOR
Regel 5	T>0	DC	Z	Z	DC	DC
Regel 6	ØR	DC	DC	XOR	DC	DC

Tabell 4.2: Regler brukt i eksempelet [33].

Typen som er angitt i kolonne C (C benyttes for senter for å skille fra S for sør) tilsvarer typen aktuell blokk må ha for at regelen skal benyttes, S - Ø - N - V tilsvarer type for blokker i tilsvarende retning.

Figur 4.5 viser utviklingen til en organisme gitt reglene i tabell 8.2 er implementert. Målet med denne organismen er å vise at en symmetrisk organisme lar seg utvikle. Denne organismen er ikke utviklet med tanke på å realisere en bestemt funksjon. Som vist utvikles organismen fra en enkelt celle (aksiom). Etter to developmentstep har 11 SBlocker blitt aktivert og alle tre typene av celler eksisterer. Etter fem developmentstep har organismen endret seg til en symmetrisk organisme.



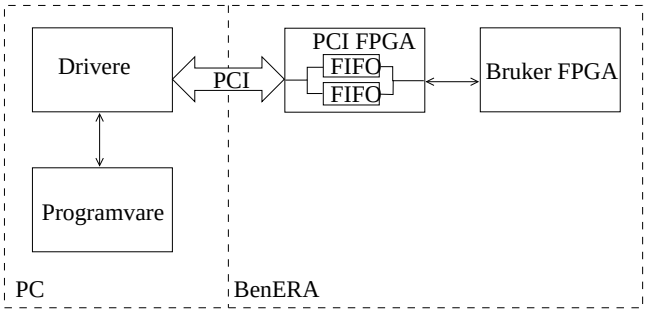
(a) Tegnforklaring (b) Start tilstand (c) Development steg 2 (d) Development steg 5

Figur 4.5: Eksempel på utvikling til en symmetrisk organisme [33].

Kapittel 5

Systembeskrivelse

Systemet er en maskinvarerealisering av en SBlock-matrise. SBlock konseptet er beskrevet i kapittel 4. Systemet består av to deler; en maskinvaredel og en programvaredel.



Figur 5.1: Systemet består av to deler; en programvaredel og en maskinvaredel. Programvaren kjører på vertsmaskinen og maskinvaren kjører i *Bruker FPGA*. Drivere og *PCI FPGA* er ferdig programmert fra produsent.

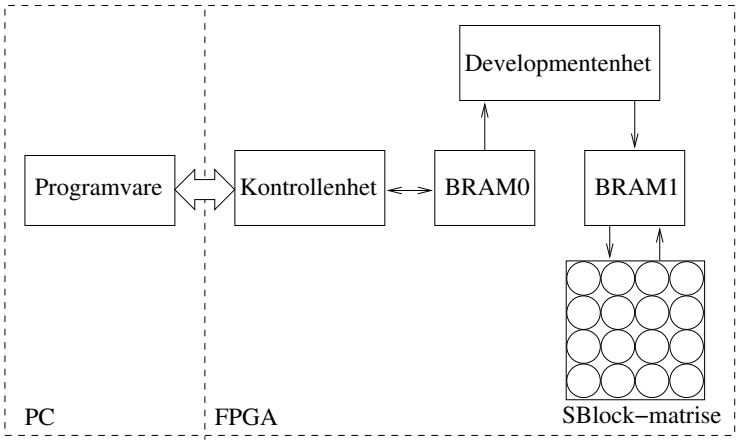
Maskinvaredelen av designet er implementert i en FPGA på ett BenERA cPCI-kort. Kontrollen av dette kortet utføres over PCI bussen fra en datamaskin. Figur 5.1 viser hvilke deler av systemet som kommuniserer. *Drivere* og *PCI FPGA* er programmert av produsenten og blir derfor ikke beskrevet nærmere i dette kapittelet. Dersom ikke annet er beskrevet kan begge disse modulene sees som transparente for systemet.

5.1 Maskinvare

Figur 5.2 viser en overordnet tegning av maskinvaredelen av designet. Det er kun dataflyten i systemet som er vist. Modulen merket *Programvare* er programvare skrevet i C. Programmet kontrollerer kjøringen av maskinvaren. Alle modulene innenfor rammen merket *FPGA* er skrevet i VHDL. Designet kjører i modulen merket *Bruker-FPGA* på figur 5.1.

Modulene i dataveien i systemet består av følgende deler:

- *SBlock-matrise*: Denne modulen implementerer en kjørbar SBlock-matrise. Størrelsen på matrisen kan varieres (før syntese). Matrisen konfigureres med verdier



Figur 5.2: Toppnivå av maskinvaren

fra BRAM-1.

- *Developmentenhet*: Denne enheten utfører development på SBlock-matrisen. Enheten benytter data som ligger i BRAM-0 og resultatet skrives til BRAM-1.
- *Kontrollenhet*: Modulen fungerer som systemets grensesnitt mot programvaren. Enheten finnes ikke direkte igjen i den faktiske implementasjonen. I implementasjonen er det flere moduler som sammen danner kontrollenheten
- *BRAM-0 og BRAM-1*: I disse minneblokkene blir tilstander og typer for matrisen lagret. BRAM-0 og BRAM-1 er to identiske moduler. Blokkene brukes ved development-step og ved konfigurering. Navnet *BRAM* kan være litt misvisende i og med at alle lagringsmoduler er består av BRAM, men navnet er beholdt for at det skal være konsistent med VHDL-koden som var utgangspunkt for denne oppgaven.

Kontrollenheten i systemet er designet som en co-prosessor med eget instruksjonssett. Dette gjør at systemet kan kjøre egne program. Co-prosessoren er beskrevet i kapittel 5.3.

5.2 Programvare

Programvaredelen av systemet kjører på vertsmaskinen og styrer BenERA-kortet.

Programmet konfigurerer FPGAen før systemet kan benyttes og sørger for å avslutte kjøringen når eksperimentet er ferdig. I tillegg sørger programvaren for å lese ut data fra maskinvaren under kjøring.

Programmet i figur 5.3 viser et typisk program som kjører på vertsmaskinen. Det første som gjøres er å åpne og konfigurere kortet. Alle operasjoner som kontrollerer BenERA kortet går gjennom FUSE API, se kapittel 3.2. Konfigurasjon av FPGA gjøres ved å laste en bitfil. Deretter lastes programmet som skal kjøre i co-prosessoren, et eksempel på et slikt program er vist i figur 5.4. Så startes co-prosessoren som kjører i en evig løkke. Oppgaven til C-programmet blir da å lese ut data som skrives til PCI-bussen og evt. skrive data til fil eller skjerm. Før programmet avslutter må koblingen

C-PROGRAM

```
1  OpenCard
2  ConfigureCard
3  StoreProgram              ▷ i.e. RUN-AND-DEVELOP, fig. 5.4
4  StartCoProcessor
5  for 0 to n                ▷ n set at startup
6       do ReadData
7          PrintData
8  CloseCard
```

Figur 5.3: Eksempel på program som styrer systemet

til BenERA kortet lukkes. Det er vesentlig at programmet i co-prosessoren og vertsmaskinen leser og skriver data i samme rekkefølge for at data som leses ut skal bli tolket riktig.

5.2.1 SBlocklib

SBlocklib er hentet fra [3] og utvidet i denne oppgaven. Biblioteket er et abstraksjonslag over FUSE API, kapittel 3.2. Det finnes funksjoner for kommunikasjon mot FPGA-kortet og funksjoner for å skrive instruksjoner til co-prosessoren.

Funksjoner i sblocklib er listet i tabell 5.1.

Funksjon	Arg.	Beskrivelse
openCard	n	Åpner FPGA-kort nummer *n*
closeCard		Lukker FPGA-kort
configureCard	bitfile	Konfigurere FPGA-kort med bitfil
insertDMA	data	Skriver 32 bit data til et buffer
flushDMA		Skriver bufferet til FPGA-kortet
readDMA	n	Leser *n* ord fra FPGA-kortet
instruksjoner		Alle instruksjoner som co-prossoren kan utføre [a]
saveSendBuffer	file	Skriver alle data fra buffer som Modelsim-script [b]

[a] Instruksjonene er gitt i tabell A.28

[b] Scriptet kan brukes for å utføre simuleringer i Modelsim og er meget nyttig under utvikling.

Tabell 5.1: Metoder i SBlocklib

5.3 Co-prosessor

Kretsen er designet som en co-prosessor. Den har et instruksjonsminne som gjør at den fungerer autonomt når et program er lastet. Dette frigjør PCI bussen og vertsmaskin til andre oppgaver. Alle instruksjonene som kan brukes er listet i tabell 5.2.

Et typisk program vil bestå av en løkke som kjører en mengde run-step for hvert develeopment-step. Figur 5.4 viser et eksempelprogram som kan kjøre på prosessoren.

Det første som må gjøres er initialisering, laste regler og sette opp en utgangstilstand for matrisen. Deretter kan det kjøres et antall run-step. Co-prosessoren fungerer

```
RUN-AND-DEVELOP(n)
1  LoadRules
2  InitMatrix
3  while true                    ▷ Until C-program finish
4      do Run(n)                 ▷ n number of steps
5         ReadData
6         DevelopmentStep
```

Figur 5.4: Typisk program som kan kjøres på co-prosessoren består av en løkke som kjører et antall iterasjoner.

slik at den kjører frem til vertsmaskinen avslutter programmet. Programmet som er valgt her skriver ut data under kjøring, det er ikke nødvendig. Data som skrives ut kan være typer og tilstander for hver SBlock. Det bli så kjørt et development-step før løkken gjentas.

5.3.1 Instruksjonssett

En oversikt over instruksjonene er vist i tabell 5.2. Mer informasjon om instruksjonene finnes i vedlegg A

5.4 Funksjonalitet

Co-prosessorkoden i figur 5.4 introduserer noen av operasjonene som systemet kan utføre. I dette avsnittet blir disse beskrevet.

$LoadRules$ Alle regler (se kapittel 4.2.1) som skal brukes under ett eksperiment må skrives til systemet før de skal benyttes. Det må også spesifiseres hvor mange regler som er skrevet.

$InitMatrix$ Systemet kan skrive initialverdier til SBlock-matrisen. Det kan skrives ett komplett sett verdier eller bare verdier for enkelte blokker. Det kan skrives både tilstand og type for hver av blokkene.

$Run(n)$ Det kan kjøres ett eller flere run-step for hvert development-step. Et run-step endrer kun tilstandene i SBlock-matrisen, ikke i BRAM. Derfor finnes det en egen instruksjon som skriver tilstandene tilbake til BRAM-1, denne må brukes før ett nytt development-step skal kjøres (dersom reglene tar hensyn til tilstand for SBlockene).

$DevelopmentStep$ Når et development-step kjøres blir data som er lagret i BRAM-0 sjekket mot regelsettet. Resultatet blir lagrer i BRAM-1. BRAM-1 inneholder de nye tilstandene for systemet etter development.

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

Tabell 5.2: Oversikt over instruksjonssett.

5.5 Lesing av data

Data leses ut via PCI-bussen. For å utnytte bussens kapasitet blir data først pakket. Denne pakkingen gjør at mer informasjon overføres pr. transaksjon. Det er viktig å være klar over hvordan pakkingen er gjort for å kunne tolke dataene korrekt ved mottak. Alle disse operasjonene startes med en instruksjon og vertsmaskinen må lese dataene eksplisitt fra PCI-bussen.

5.5.1 Tilstander

Tilstandene sendes i blokker på 32 bit. Hver tilstand representeres som et bit, dermed overføres 32 tilstander pr. transaksjon. Figur 5.5 viser hvordan dataene er formatert.

Bit 31	Bit 30	***	Bit 0
SBlock (32n+31)	SBlock (32n+30)	***	SBlock (32n)

Figur 5.5: Formatering for utskrift av matrisens tilstander, $0 \leq n < (\#SBlocker/32)$

Det er også mulighet for å lese ut tilstanden til en spesifikk SBlock, da må blokkens nummer angis i instruksjonen.

5.5.2 Typer

Hver SBlock har en gitt type. Ved utskrift sendes det fire typer pr. transaksjon. Dataene er formatert som vist i figur 5.6.

Bit 20-16	Bit 15-11	Bit 10-6	Bit 5-0
SBlock (4n+3)	SBlock (4n+2)	SBlock (4n+1)	SBlock (4n)

Figur 5.6: Formatering for utskrift av typer, $0 \leq n < (\#SBlocker/4)$

Typen til én enkelt SBlock kan leses ut ved å angi SBlocknummeret i en instruksjon.

5.5.3 Motivasjon for utvidelser

Ett eksperiment beskrevet i [33] viser hvordan utgangstilstand, antall run-step og development påvirker SBlock-matrisen. Eksperimentet er utført ved at antall satte SBlocker blir telt opp for hvert run-step. Dersom dette eksperimentet skulle vært utført på det opprinnelige maskinvaresystemet måtte all tilstandsdata bli lest ut for hvert step. Dette er tidkrevende operasjoner.

For å bedre effektiviteten ville det vært hensiktsmessig å lagre data som beskriver tilstanden i matrisen for hvert run-step. Tilstanden kan transformeres til en verdi som beskriver visse aspekter ved tilstanden, f.eks. antall satte SBlocker.

Etter at data er lagret kan de produserte verdien evalueres. Eksperimentet beskrevet over søkte etter etterfølgende sekvenser av stigende verdier, en slik funksjon er lett realiserbar i maskinvare.

Det kan være vanskelig å forutsi hvilke regler som påvirker en utviklingsprosess. Enkelte regler kan aldri bli benyttet, mens andre kan være svært dominerende. Ved å overvåke hvilke regler som aktiveres kan uønskede effekter ved ett regelsett avdekkes.

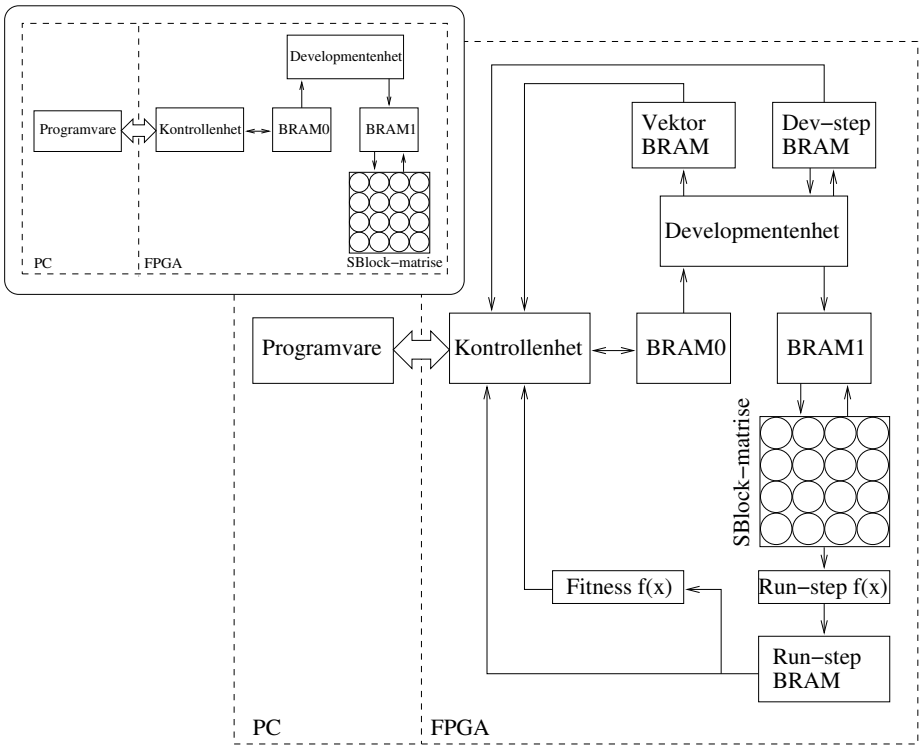
Neste kapittel beskriver nettopp utvidelser som implementerer de tre nevnte funksjonene.

Kapittel 6

Utvidelser

Dette kapittelet beskriver utvidelsene av systemet som har blitt gjort i denne oppgaven.

Systemet har blitt utvidet både på programvare- og maskinvare-siden. For bruker av systemet vil endringene som er gjort innbære tilgang til flere instruksjoner, alle nye instruksjoner er vist i tabell 6.1.



Figur 6.1: Systemet etter at utvidelsene er implementert. Systemet før utvidelsene er vist øverst til høyre.

Modulene som er lagt til er (se figur 6.1):

- *Run-step f(x)*: Funksjonen leser ut tilstanden fra SBlock-matrisen og transformerer

data gjennom en funksjon.

- *Run-step BRAM*: Minnemodulen lagrer data som har blitt generert av *Run-step f(x)*.
- *Fitness f(x)*: Dette er en funksjon som evaluerer data generert ved run-step.
- *Dev-step BRAM*: Lagrer hvilket regelnummer som har blitt kjørt på hver enkelt SBlock ved forrige development-step.
- *Vektor BRAM*: Har samme funksjon som *Dev-step BRAM*, men lagrer aktiverte regler på en mer kompakt måte. Har kapasitet til å lagre regler som er benyttet for flere etterfølgende development-step.

Det er også lagt til funksjonalitet for å lese ut lagrede/ genererte data.

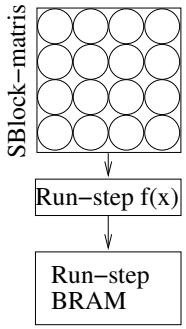
6.1 Utvidet instruksjonssett

For å benytte utvidelsene har det blitt laget nye instruksjoner. De fleste nye instruksjonene har med å utlesing av generert data å gjøre. Alle nye instruksjoner er vist i tabell 6.1. Komplett oversikt over alle instruksjoner som kan benyttes i systemet er vist i vedlegg A.28.

Alle moduler som er lagt til blir beskrevet i etterfølgende avsnitt.

6.2 Lagring av data ved run-step

Hvert run-step kan endre tilstandene i SBlock-matrisen. Dersom det er ønskelig å ha kontroll med tilstanden for hvert enkelt step kan data leses ut på PCI-bussen. Dette er en svært ressurskrevende operasjon. For å forbedre kjøretiden har det derfor blitt implementert ett eget minne (BRAM) for å lagre data generert ved run-step.



Figur 6.2: Dataflyt for prosessering av run-step data

Figur 6.2 viser dataflyten for enheten. Enheten merket *Run-step f(x)* leser 16 bit pr. klokke fra matrisen. Enheten har 16 bit databuss for utdata. Utgangen skal reflektere hele matrisen, ikke kun en gruppe av blokker. Dermed må resultatet akkumuleres opp over tid. Enheten *Run-step BRAM* er ett område med BRAM som resultatene skrives til. Utlesning av lagrede tilstander gjøres med egen instruksjon. Ved lesing overføres

Navn	Type	Argumenter	Beskrivelse
doFitness	Kort		Kjører en fitnessfunksjon og lagrer resultatet i et buffer
jumpEqual	Kort	verdi, address	Hopper til addresse dersom antall dev-step er kjørt
readFitness	Kort		Sender 32 bit data fra fitness register, flere etterfølgende kall gir mer data
readRuleVector	Kort	antall	Skriver ut et gitt antall vektorer som beskriver kjørte regler
readSums	Kort	antall	Les tilbake lagrede summer
resetDevCounter	Kort		Nullstiller register som teller antall kjørte dev-step
readUsedRules	Kort		Skriver ut hvilke regler som er kjørt på hvilke blokker

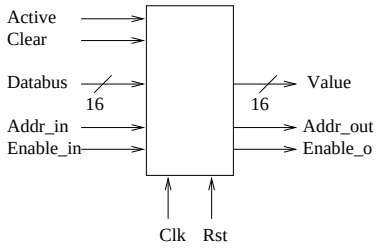
Tabell 6.1: Oversikt over alle nye instruksjoner.

Bit 31-16	Bit 15-0
RunStep (2n+1)	RunStep (2n)

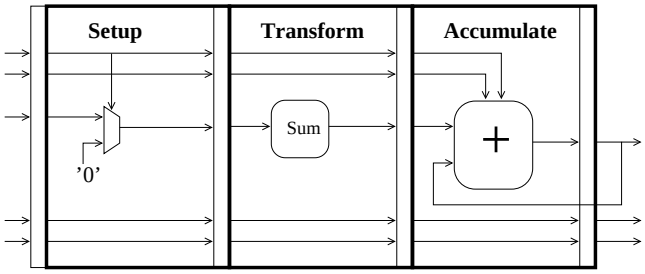
Figur 6.3: Formatering for utskrift for data generert ved run-step, $0 \leq n < (angitt\ i\ instruksjon)$

det to 16 bits verdier pr. transaksjon. Hvor mange verdier som skal overføres må angis i instruksjonen. Figur 6.3 viser hvordan data er formatert.

Antall blokker i matrisen vil variere avhengig av hvilket eksperiment som skal utføres. Kretsen støtter, slik den fremstår nå, fra 8^2 til 32^2 antall blokker. Dersom det skulle vært implementert logikk for å evaluere alle SBlocker uten å forsinke tilstandsmaskinen måtte altså 1024 bit vært behandlet i parallell. For å begrense logikkbehovet er det valgt å tillate en forsinkelse.



(a) Grensesnittet for funksjonen.



(b) Run-step funksjonen er implementert som et samlebånd, inn- og utsignaler er tegnet i samme rekkefølge som på figur 6.4(a).

Figur 6.4: Run-step funksjonen.

Figur 6.4(a) viser grensesnittet for modulen som transformerer data fra SBlock-matrisen til en 16 bits verdi. Funksjonen er implementert med registere mellom hvert trinn slik figur 6.4(b) viser.

Utgangen fra SBlock-matrisen er 16 bit. Figur 6.5 viser hvilken blokk som er koblet til hvilken posisjon i utlesningsvektoren. Figuren viser en matrise med 16 ganger 4 blokker. I figuren er 16 og 16 bit gruppert. For hver blokk i en gruppe er det skrevet ett nummer. Dette nummeret tilsvarer bitposisjonen i utlesningsvektoren.

Designet er laget slik at funksjonene som transformerer tilstanden i matrisen lett skal kunne byttes ut.

1	0	3	2	9	8	11	10	1	0	3	2	9	8	11	10
5	4	7	6	13	12	15	14	5	4	7	6	13	12	15	14
1	0	3	2	9	8	11	10	1	0	3	2	9	8	11	10
5	4	7	6	13	12	15	14	5	4	7	6	13	12	15	14

Figur 6.5: Data overføres ut fra SBlockmatrisen (her vist med dimensjonene 16 ganger 4 blokker) i grupper på 16 bit. Hver SBlock overfører sin tilstand på bitposisjonen som er angitt i figuren. Rekkefølgen på utsending av vektorer er fra venstre mot høyre og ovenfra og ned.

6.2.1 Minne

Verdiene blir lagret som 16 bit i en egen BRAM. Verdiene lagres fra adresse 0 og oppover etter hvert development-step. Det vil i si at kun verdier mellom to development-step lagres. Antall minneplasser settes før syntese (indirekte via bredden på adressebussen). Ut fra oppnådde resultater (kapittel 8) ved syntes er adressebussen satt til 13 bit, slik at det er 8192 minneadresser.

6.2.2 Valgt funksjonalitet

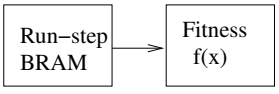
I denne oppgaven ble det valgt å implementere en funksjon som teller antall enere i SBlock-matrisen.

6.2.3 Ytelsespåvirkning

Det har blitt valgt å gjøre summeringen sekvensielt i grupper på 16 bit. Dette har to fordeler; logikkmengden og rutingen begrenses. Ulempen er en økning i tidsforbruket. Denne økningen i tid bør sees i sammenheng med tidsforbruket ved evt. utlesning av data over PCI-bussen. For resultater se kapittel 8.

6.3 Fitness-funksjon

Data som blir generert gjennom en serie med run-step kan evalueres av en fitness-funksjon. Hvilke fitnessfunksjoner som kan implementeres avhenger av hvilken funksjon som er benyttet for å konvertere tilstandsdata fra matrisen. Figur 6.6 viser fitness-funksjonen som leser data fra minnet og sender resultatet ut av modulene.

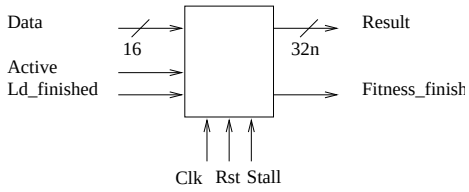


Figur 6.6: Dataflyt gjennom fitnessfunksjonen

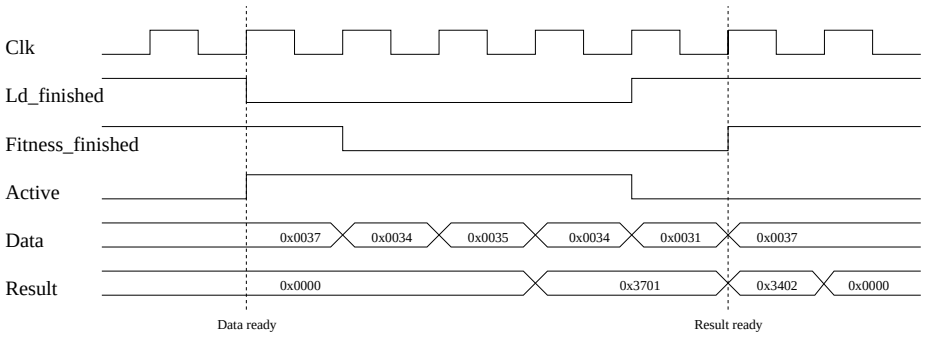
Etter at en fitnessfunksjon er kjørt kan data som ble produsert leses ut. Fitness-funksjonen er laget slik at størrelsen på resultatdata kan varieres. Dersom datamengden overstiger 32 bit kan funksjonen som leser ut data kalles flere ganger etter hverandre og etterfølgende data leses ut. Formateringen for dataene er ikke bestemt av designet, men velges ut fra hva aktuell funksjon produserer av verdier (dette må gjøres før syntese).

Funksjonen er implementert som et samlebånd. Rammeverket for funksjonen sørger for å lese ut data fra minnet og kjøre verdiene gjennom evalueringsfunksjonen. Evalueringsfunksjonen kan også være implementert som et samlebånd med valgfri lengde. Lengden på samlebåndet vil på virke ytelsene for funksjonen i mindre grad. Resultater fra funksjonen skrives til et register som deretter kan leses med en egen instruksjon.

Funksjonen er satt inn i siste steg av samlebåndet som en egen modul. Grensesnittet for denne modulen er vist i figur 6.7(a). Figur 6.7(b) viser et tidsforløp for funksjonen. Modulen mottar *Data* sekvensielt fra resten av samlebåndet. For at funksjonen skal kunne implementeres som et samlebånd må den forsinke signalet *Ld_Finished* (Load er forrige trinn i samlebåndet) like mange perioder som den bruker på å evaluere data. Figuren viser en operasjon med en periode forsinkelse. Signalet *Active* er høyt når data kan prosesseres. *Result* verdien er klar samtidig med at *Fitness_finished* går høyt. Bredden på *Result* avhenger av funksjonen som er implementert, men må være et multiplum av 32. *Stall* sørger for at dataavhengigheter mellom de ulike samlebåndenen i systemet ikke oppstår.



(a) Grensesnitter for funksjonen.



(b) Tidsdiagram som viser flyten av data gjennom fitnessfunksjonen, verdiene i figuren viser et tilfeldig gjennomløp. Det evalueres her 5 dataverdier. Antall dataverdier som skal evalueres spesifiseres i instruksjonen.

Figur 6.7: Fitness funksjon

6.3.1 Valgt funksjonalitet

Funksjonen som er implementert finner lengste etterfølgende sekvens med stigende verdier. Resultatet fra funksjonen lagres i en 128 bits vektor. Formateringen på vektoren er som følger:

Bit 127-24	Bit 23-4	Bit 7-0
Ubrukt	Startverdi i lengste sekvens	Lengde på lengste sekvens

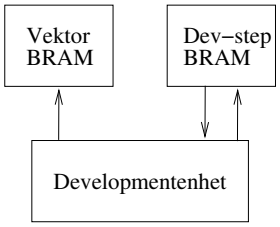
Resultatet vist i figur 6.7(b) er $0x3402$. Altså finnes det en sekvens på 2 etterfølgende tall som starter på $0x34$, dette er data som kan finnes igjen i figuren.

6.3.2 Ytelsespåvirkning

Fitnessfunksjonen påvirker ikke systemet når den ikke benyttes. Ved kjøring vil datamengden som skal evalueres være den bestemmende størrelsen for kjøretiden. Det evalueres tilnærmet en verdi pr. klokke.

6.4 Aktiverte regler ved development

Hvilke regler som blir aktivert under et development-step varierer med eksperimentet som kjøres. For å ha kontroll med hvilke regler som har blitt kjørt blir det lagret hvilke regelnummer som har blitt benyttet. Figur 6.8 viser developmentenheten og de to minnemodulene som lagrer resultatene.



Figur 6.8: Dataflyt gjennom enhetene som lagrer informasjon om kjørte regler

Dersom det ikke blir aktivert noen regel for en SBlock i løpet av et develeopment-step blir verdien $0xFF$ (255) benyttet for denne SBlocken. $0xFF$ er altså en kode som beskriver hendelsen "ingen regel aktivert". Det er lite sannsynlig at et eksperiment vil benytte 256 regler. Derfor vill ikke dette bety en reduksjon i funksjonalitet for de fleste eksperimenter. Denne kodingen gjelder for begge de implementerte minnemodulene.

6.4.1 Et development-step

For hvert development-step blir det lagret hvilken regel som har blitt aktivert for hver enkelt SBlock. Enheten som utfører development-step jobber på to og to SBlocker. Det er mulig å benytte maksimalt 256 regler i et eksperiment [3], derfor kan regelnummeret representeres med 8 bit. Minnet som benyttes har databredde på 16 bit, plass til å lagre aktiverte regler for to SBlocker pr. adresse. Verdiene må leses ut av systemet før neste development-step kjøres, om de ikke skal bli skrevet over.

Ved utskrift av aktiverte regler for ett enkelt development-step blir data skrevet ut over PCI-bussen slik som figur 6.9(a) viser. Antall transaksjoner er $\#SBlocker/2$. Dersom det ikke har blitt kjørt noen regel på en SBlock vil verdien $0xFFFF$ skrives ut.

Bit 31-16	Bit 15-0
SBlock (2n+1)	SBlock (2n)

(a) Formatering for aktiverte regler pr. SBlock, $0 \leq n < \#SBlocker$

PCI-transaksjon	Regel
0	31-0
1	63-32
2	95-64
3	127-96
4	159-128
5	191-160
6	223-192
7	255-224

(b) Formatering av data for aktiverte regler ved et development-step

Figur 6.9: Formatering av data for utskrift av aktiverte regler

6.4.2 Flere etterfølgende development-step

Dersom det er ønskelig, av ytelsesmessige årsaker, å ikke lese ut alle kjørte regler for hvert development-step kan det i stedet leses en vektor som viser hvilke regler som har blitt aktivert. Vektoren beskriver ikke hvilken blokk som har aktivert hvilken regel. Denne vektoren har en bredde 256 bit. Dersom et bit er satt betyr det at regelen med tilsvarende nummer har blitt kjørt minst en gang for aktuelt development-step. Bitposisjonene er nummerert fra høyre mot venstre. Det har blitt implementert minne for 256 vektorer.

Når aktiverte regler for ett eller flere development-step skal skrives ut er data formatert som vist i figur 6.9(b). Dersom ett bit er satt i vektoren betyr det at tilsvarende regel har blitt kjørt minst en gang. Bit 255 vil være satt dersom minst en av SBlockene ikke har aktivert en regel. Antall regler er begrenset i maskinvaren til maksimalt 256. Antall regler som faktisk er aktive settes ved kjøretid, men for at systemet skal benytte 256 regler blir det alltid skrevet ut en 256 bits lang vektor.

6.4.3 Ytelsepåvirkning

Funksjonaliteten påvirker ikke ytelsen for systemet før data leses ut.

Kapittel 7

Implementasjon

Dette kapittelet beskriver implementasjonen av systemet. Først beskrives data- og kontroll-flyten i systemet. Deretter gis en detaljert beskrivelse av viktige moduler i systemet.

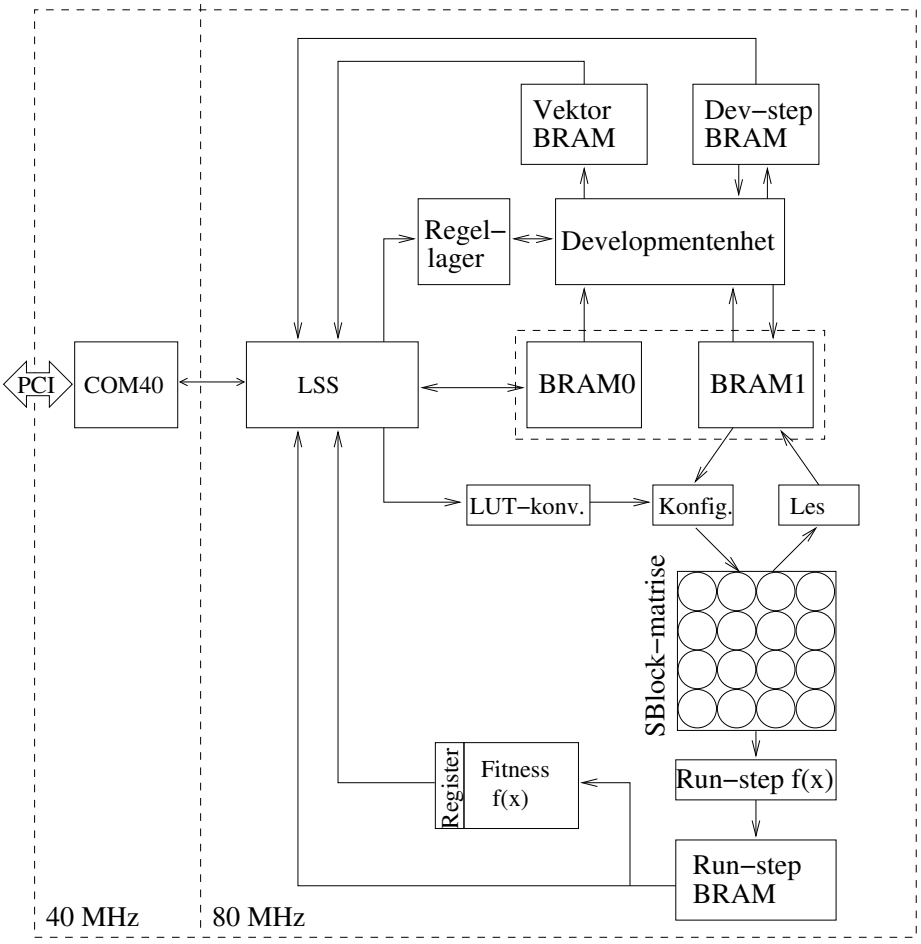
7.1 Dataflyt

Figur 7.1 viser dataflyten og de viktigste koblingene mellom modulene.

Modulene i systemet kan deles opp i to grupper, moduler som prosesserer data og moduler som lagrer data.

Moduler for prosessering:

- *COM40*: Denne modulen kjører på 40 MHz og er eneste delen av designet som benytter lavere klokke. Den er koblet mot PCI-bussen og sørger for mottak og sending av data.
- *Konfig. og Les enhetene i fbm. SBlock-matrisen*: Enhetene sørger for å lese og skrive data mellom BRAM1 og SBlock-matrisen.
- *Developmentenhet*: Developmentenheten utfører development delen av algoritmen gitt i kapittel 4.2. Enheten leser data fra BRAM 0 og skriver data til BRAM 1. Dataene transformeres gjennom enheten avhengig av hvilke regler som er lastet og hvordan de blir aktivert til.
- *SBlock-matrise*: SBlock-matrisen representerer den kjørbare matrisen.
- *Run-step f(x)*: Transformerer tilstanden til matrisen over til en verdi som lagres i et eget minne.
- *LSS (Load, send, store)*: Denne enheten leser og skriver data til og fra PCI-bussen. Den er koblet mot minnemodulene.
- *Fitness f(x)*: Modulen leser ut data generert ved run-step og produserer et resultat avhengig av implementert fitnessfunksjon.



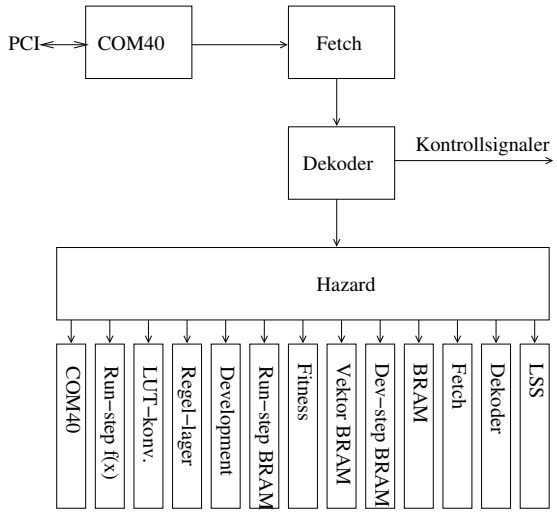
Figur 7.1: Dataflyten i systemet.

Moduler for datalagring:

- *BRAM (0 og 1)*: Dette er minnet som developmentenheten leser og skriver til/ fra.
- *Vektor BRAM*: Dette minnet lagrer en kompakt representasjon av hvilke regler som har vært aktivert i løpet av flere etterfølgende development-step.
- *Dev-step BRAM*: Minnet lagrer hvilken regel som ble aktivert for hver enkelt SBlock ved et enkelt development-step.
- *Regel-lager*: Ved oppstart av systemet skrives alle regler som skal brukes i eksperimentet til dette minnet.
- *LUT-konv*: Dette er en tabell som gir rask tilgang på data for å konvertere en gitt SBlock-type til innholdet i blokkens oppslagstabell.
- *Run-step BRAM*: For hvert enkelt run-step blir data som produseres lagret her.

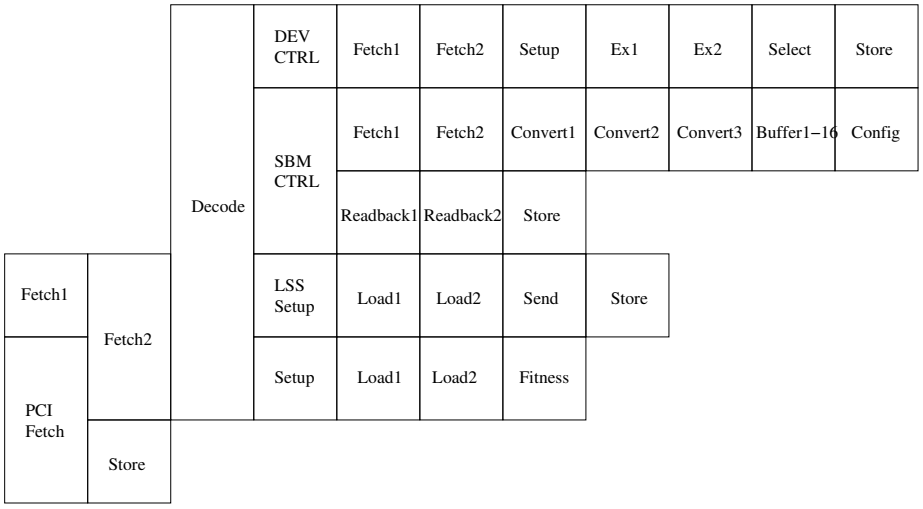
Lagringsmodulene er implementert som BRAM i FPGAen.

7.2 Kontrollflyt



Figur 7.2: Kontrollflyt

Kontrollflyten i systemet er vist i figur 7.2. Under kjøring av systemet mottas instruksjoner fra PCI-bussen via COM40 modulen. Instruksjoner kan deretter lagres i instruksjonsminnet eller de utføres direkte. Når en instruksjon skal utføres blir den dekodet i dekodingsenheten. Denne enheten styrer alle andre moduler. Modulen merket *Hazard* sørger for at ingen samlebånd kjører samtidig slik at dataavhengigheter kan oppstå.



Figur 7.3: De fire delsamlebåndene i systemet

Det er fire delsamlebånd i systemet. Alle implementert som tilstandsmaskiner[1].

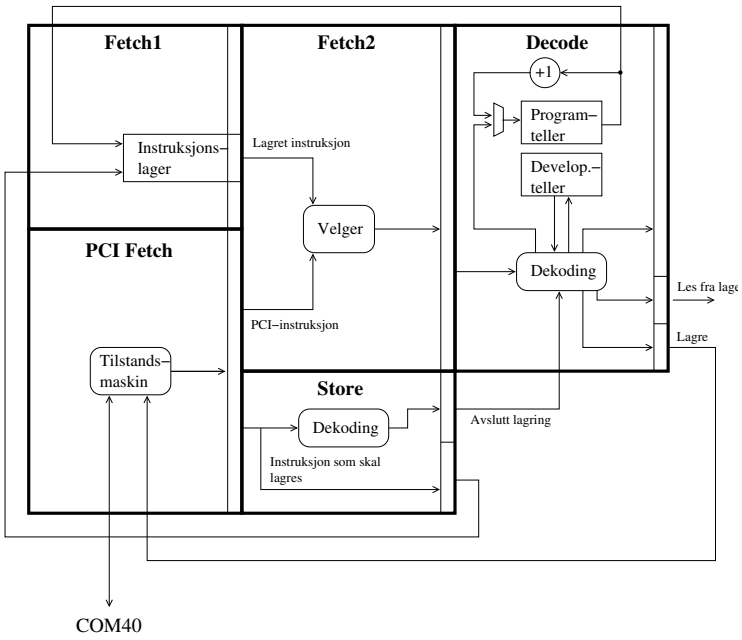
[1] eng.: Final State Machines (FSM)

Figur 7.3 viser stegene i samlebåndene. Det første trinnet til venstre på figuren leser instruksjoner, enten fra det interne instruksjonslagerert eller fra PCI-bussen. Avhengig av instruksjonstypen blir instruksjonene lagret eller hentet i neste trinn av samlebåndet. Tredje steg dekoder instruksjoner som skal utføres. Disse tre trinnene er forklart nærmere i kapittel 7.2. Avhengig av instruksjonstype sendes instruksjonene videre fra dekoder-trinnet til et av de fire delsamlebåndene. Disse fire er (ovenfra og ned i figuren):

- *Development*: Utfører development
- *SBlock-matrise*: Utfører både initialisering og tilbakelesning av matrisens tilstand.
- *LSS (Load, send, store)*: Sender og mottar data til/ fra systemet.
- *Fitness*: Kjører en fitnessfunksjon på genererte data.

Alle disse samlebåndene blir nærmere forklart i kapittel 7.6.

7.2.1 Instruksjonsutføring



Figur 7.4: Instruksjonshenting, dekoding og lagring i instruksjonslager [3].

Figur 7.4 viser samlebåndet for instruksjonshenting, dekoding og lagring. Alle instruksjoner er internt representert med 64-bit. Dette er valgt for å forenkle kontroll-logikken [3]. Alle instruksjoner mottas i *PCI Fetch* delen av samlebåndet. Trinnet merket *Store* dekoder instruksjonene for å finne ut om den skal lagres eller utføres direkte. Dekoderdelen sjekker også om lagring skal avsluttes. Dersom data skal lagres skrives instruksjonene til et instruksjonslager. *Fetch 2* mottar instruksjoner enten fra instruksjonslageret eller direkte fra PCI-bussen, via *PCI Fetch*.

Decode dekoder alle instruksjoner som skal utføres. Trinnet inneholder en program-teller og en teller for antall utførte development-step.

Hele instruksjonssettet (bortsett fra instruksjoner som kontrollerer skriving til instruksjonslageret) er tilgjengelige fra begge kildene.

7.2.2 Hazard

Dataavhengigheter unngås ved at hazard enheten sørger for at kun et samlebånd er aktivt om gangen.

Hvert delsamlebånd (DEV, SBM, LSS, Fitness) garanterer at de ikke har noen instruksjoner i sitt samlebånd så lenge de er i tilstanden *Ledig*. Tilstandsmaskinene er derfor i en ventetilstand frem til samlebåndet er helt tømt.

Dersom ett av delsamlebåndene ikke er i *Ledig*-tilstanden, vil samlebåndstrinnene *PCI Fetch*, *Fetch1*, *Fetch2* og *Decode* stoppes og vente uten å forandre noen av sine registre. Ingen nye instruksjoner vil forlate *Decode* før alle delsamlebåndene er klare. Derfor vil det aldri befinne seg to instruksjoner samtidig i samlebåndene etter *Decode*-trinnene, og ingen dataavhengigheter vil ha mulighet til å oppstå.

De fleste instruksjoner bruker flere sykler på å fullføre en operasjon og vil holde delsamlebåndet aktivt i lang tid. Et opphold med tømming og fylling av samlebåndet mellom hver instruksjon vil derfor påvirke kjøretiden i liten grad. Metoden som er valgt er ikke mest effektiv, men den er enkel og krever lite logikk [3].

Utlesningen av instruksjoner blir gjort flere sykler før dekodingen. Når en instruksjon dekodes (i *Decode*-trinnet) vil det finnes instruksjoner etter denne i samlebåndstrinnene *Fetch1* og *Fetch2*. Ved bruk av hopp-instruksjoner vil allerede dekodede instruksjoner være ugyldige, og må fjernes. Dette utføres i *Decode*-trinnet hver gang den mottar en kontrollflytinstruksjon, dette kalles for “flushing”. Ved flushing vil instruksjonene i *Fetch1* og *Fetch2* fjernes.

7.3 Frekvensdomener

Designet kjører på to ulike frekvenser. For at kommunikasjon på PCI-bussen skal være mulig kjører den delen av designet som er koblet mot bussen med en klokke på 40 MHz, se kapittel 3.1. Resten av designet er det ønskelig at kjører på så høy klokke som mulig for å oppnå god ytelse.

Vedlegg F beskriver hvordan konverteringen mellom de to klokkedomenene er gjort.

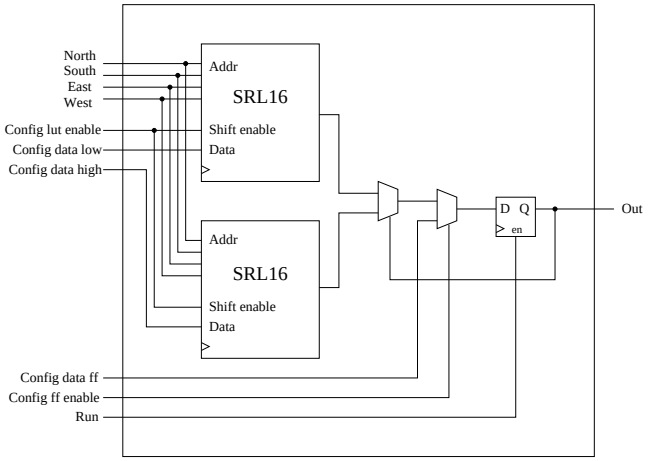
7.4 SBlock-matrise

Hver enkelt SBlock er designet slik at den lar seg implementere i en “slice” (se kapittel 2.6.1) i FPGAen. Ved å bruke to SRL16 moduler, noen mulitpleksere og en vippe oppnås ønsket funksjonalitet. Figur 7.5 viser hvordan dette er gjort. SRL16 er et 16 bits skiftregister.

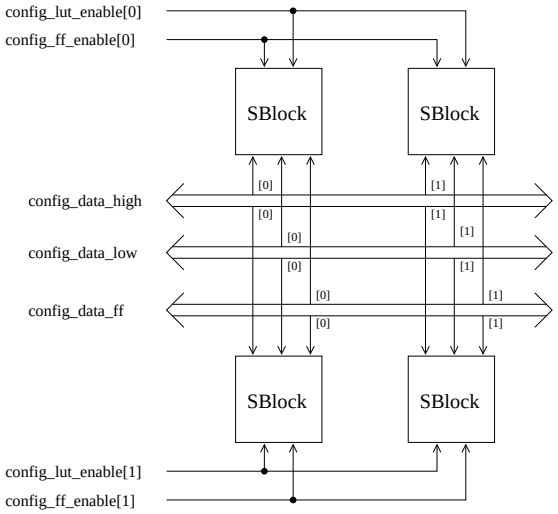
SBlock-matrisen er laget ved å koble sammen flere SBlocker etter mønsteret gitt i figur 4.2(a). Alle blokkene har ett felles *run* signal som garanterer synkron endring av tilstander.

SBlockene er laget slik at systemet kan konfigurere innholdet i LUTene uten ekstern kontroll. Det er altså mulig å endre innholdet i registrene når SBlocktypen skal endres.

SBlock matrisen har en databuss på 16 bit. Denne bussen benyttes når data leses ut. Bredden på denne bussen avgjør ytelsen for lesing av data i systemet. Alle SBlockene



Figur 7.5: En SBlock er implementert vha. to 16 bits skiftregister [3].



Figur 7.6: Konfigurasjon av matrisen foregår i grupper, konfigurasjonsdata overføres via tre separate databusser [3].

er koblet til denne bussen via ett tre-nivå-buffer. Dette gjør det mulig å kontrollerer hvilke SBlocker som skal drive bussen.

Konfigurering av SBlock-matrisen foregår i grupper. Størrelsen på gruppene er bestemt fra bredden på databussene. En SBlock har tre komponenter som skal konfigureres, to LUTer og en vippe. Disse tre komponentene har hver sin databus og kan derfor konfigureres samtidig, figur 7.6.

7.4.1 LUT konverteringstabell

BRAM inneholder en 5-bits verdi som beskriver hver enkelt SBlock type, som igjen spesifiserer innholdet i LUTen. Hver LUT er på 32 bit. Ved å angi typen med fem bit blir du mulig å ha $2^5 = 32$ ulike typer representert i systemet. Konverteringstabellen gjør at denne 5-bits typeverdien lett kan konverteres til 32 bits LUT-innhold. To uavhengige LUT-verdier kan leses ut pr. klokkesykel.

7.5 Bytte av BRAM-0 og BRAM-1

BRAM-0 og BRAM-1 har ulike oppgaver, men begge inneholder informasjon som beskriver tilstanden til SBlock-matrisen. BRAM-0 mottar data fra PCI-bussen og er utgangspunkt for et development-step. BRAM-1 inneholder resultatet fra development-step samt inneholder data som er utgangspunkt for run-step og data skrives til BRAM-1 etter ett run-step. For å unngå å kopiere data mellom disse to modulene avhengig av oppgaven som skal utføres er det laget funksjonalitet for å bytte om på minneområdene. Funksjonen utføres med en egen instruksjon.

7.6 Tilstandsmaskiner og samlebånd

I dette kapittelet beskrives de fire delsamlebåndene som er vist i figur 7.3. De tre første stegene i figuren regnes som en del av kontrollflyten og blir ikke beskrevet her.

7.6.1 SBlock-matrise

SBlockmatrise (SBM) samlebåndet har tre oppgaver:

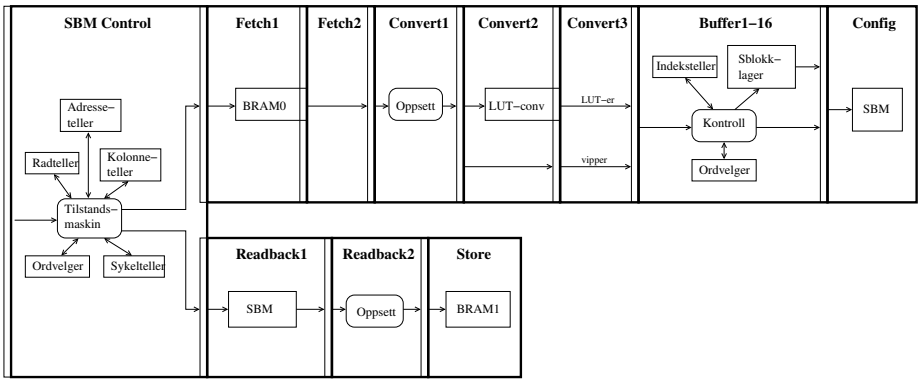
- Konfigurering av matrisen før kjøring
- Tilbakelesing av data fra matrisen etter kjøring
- Kjøre run-step i matrisen

Figur 7.8 viser FSM for SBlock-kontroll enheten. Samlebåndet er vist i figur 7.7.

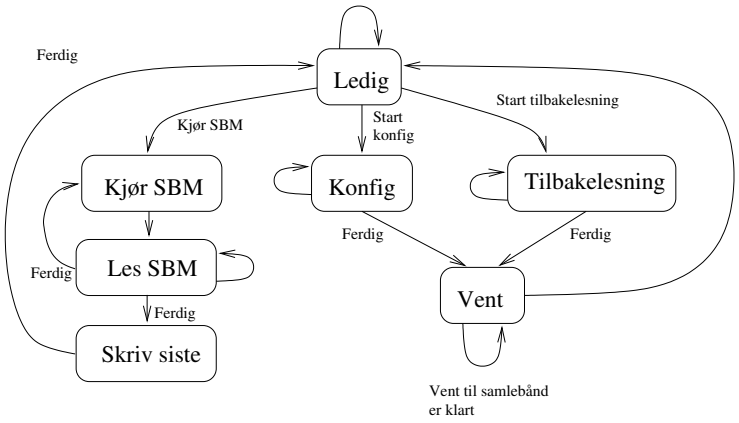
Konfigurering Samlebåndet i figur 7.8 er todelt, konfigurering skjer i den øverste delen av figuren. Det er alltid to SBlocker som blir konfigurert i parallell.

Tilstandsmaskinen i *SBM Control* styrer denne prosessen som foregår slik (henter fra [3]):

- *Fetch1* og *Fetch2* leser ut SBlockdata. For hver sykel leses det ut to nye blokker. En radteller og en kolonneteller brukes til å gå gjennom SBlockmatrisen systematisk. Adressen til SBlockene i BRAM-1 genereres av verdiene fra rad- og kolonnetellerene.



Figur 7.7: Samlebånd for kjøring og konfigurering av SBlock matrise [3].



Figur 7.8: Tilstandsmaskin for SBlock kontroll

- *Convert1*, *Convert2* og *Convert3* konverterer SBlocktype til 32-bits data for hver enkelt LUT. Dette gjøres ved oppslag i en LUT-konverteringstabellen.
- *Buffer1-16* bufrer 32 SBlocker. Det tar 16 sykler å konfigurere én enkelt SBlock-LUT. For å være i stand til å ha en gjennomstrømming på to SBlocker pr. sykel må det konfigureres 32 blokker i parallell. Dette trinnet samler opp data for 32 blokker før de sendes videre til *Config*-trinnet. Det er med andre ord 16 syklers forsinkelse i dette trinnet, men gjør at *Config* alltid har data til 32 SBlocker klare.
- *Config* konfigurerer to bits av 32 SBlocker i parallell. Hver 16. sykel vil i tillegg SBlockenes tilstand konfigureres.

Tilbakelesning Tilbakelesning styres av tilstandsmaskinen i SBM Control. Kun tilstanden leses ut og skrives til BRAM-1. Tilstandsmaskinen fungerer slik:

- *Readback1* og *Readback2* leser data ut fra SBlock-matrisen.
- *Store* skriver tilstandene til BRAM-1.

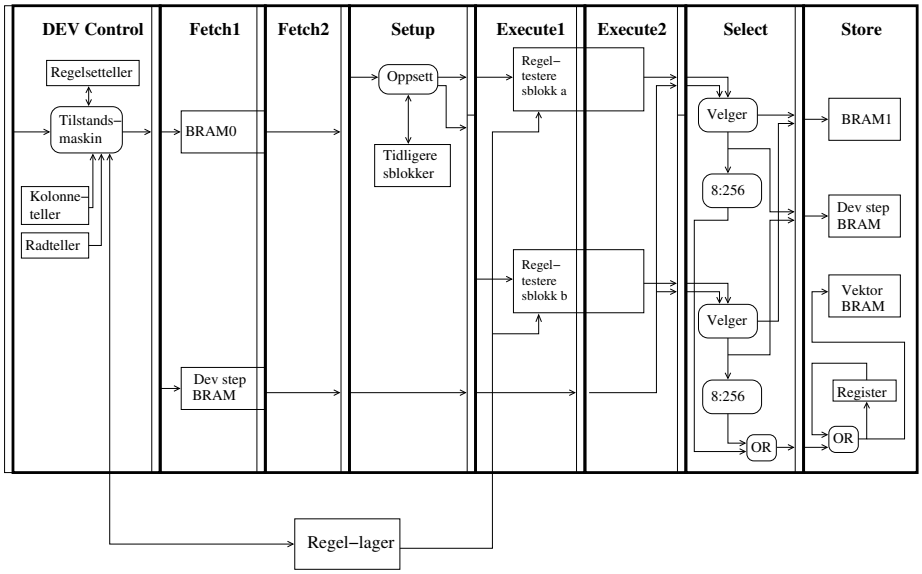
Kjøring Kjøring av matrisen foregår uten bruk av samlebånd. Et run-step tar kun en klokkesykkel for å endre tilstandene for alle SBlockene. Men når data fra et run-step skal akkumuleres (se kapittel 6.2) går det flere sykler. Antallet avhenger direkte av antall blokker i matrisen. Designet muliggjør kun utlesing av tilstandene til 16 SBlocker pr. klokke. Tidsforbruket pr. run-step kan dermed beregnes ut fra formel 7.2.

$$T_{run-step} = T_{kjor} + T_{utlesing} \tag{7.1}$$

$$T_{run-step} = (1 + \frac{\#SBlocker}{16}) * \frac{1}{f_{clk}} \tag{7.2}$$

7.6.2 Development

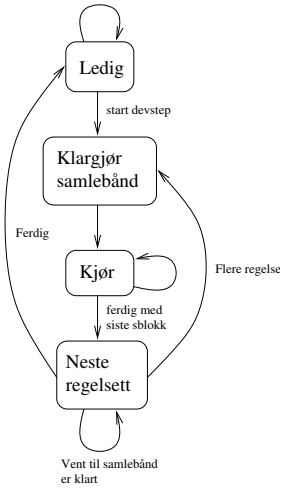
Developmentenheten kjører developmentprosessen (se kapittel 4.2)på dataene i BRAM-0. Enheten jobber med to SBlocker i parallell, og arbeider systematisk fra venstre mot høyre, rad for rad nedover SBlock-matrisen.



Figur 7.9: Samlebånd for development

Tilstandsmaskinen, figur 7.10, har følgene funksjoner:

- Tilstandsmaskinen venter på en instruksjon i tilstanden *Ledig*. Etter å ha mottatt instruksjonen *start devstep* går den inn i tilstand *Klargjør samlebånd*
- I *Klargjør samlebånd* settes samlebåndet opp riktig før kjøring.
- Tilstandsmaskinen går umiddelbart videre til *Kjør*-tilstanden der den er til den har kommet til den siste SBlocken. Når siste blokk er prosessert, går tilstands-maskinen videre til *Neste regelsett*.
- I tilstand *Neste regelsett* vil først tilstandsmaskinen vente til samlebåndet er helt ferdig (tomt). Deretter vil enten et nytt regelsett hentes inn fra regellageret, og prosessen gjentas, eller kjøringen avsluttes dersom det ikke er flere regelsett igjen.



Figur 7.10: Tilstandsmaskin for development [3].

Developmentenheten jobber etter regler som ligger lagret i et regellager. Det er viktig å merke seg at regellageret bare har 8 regler (ett regelsett) aktive. Er det flere regler enn dette totalt, må developmentenheten bytte ut de aktive reglene med nye regler og kjøre prosessen om igjen. Det vil altså bli én iterasjon for hvert sett med 8 regler som finnes før developmentsteget er ferdig. Regellageret kan inneholde 256 regler lagret.

Etter første iterasjon vil BRAM-0 fremdeles være uforandret, men BRAM-1 vil inneholde alle endringer bestemt av de første 8 reglene i regellageret. Alle iterasjoner deretter må i tillegg til å lese fra BRAM-0 for å sjekke betingelsesdelen av reglene, også lese ut verdier fra BRAM-1 for ikke å miste data som ble produsert i forrige iterasjon. Developmentenheten må både lese og skrive til BRAM-1 (se figur 7.1). Av samme grunn må også minnet som inneholder aktiverte regler leses for hvert regelsett. Det vil alltid være den regelen med høyest regelnummer som har prioritet om flere regler blir aktivert.

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

- *Fetch1* og *Fetch2* leser ut alle nye SBlocker fra BRAM-ene som er nødvendige for å behandle to nye SBlocker i parallell. Ved første regelsett settes det at alle blokker har fyret regel 255 (0xFF). Dersom ingen av reglene i noen av regelsettene aktiveres vil 255 bli stående som eneste aktiverte regel.

- *Setup* samler SBlockdata som er lest ut med tidligere utlest data som er lagret i registre. Siden SBlockdata kan komme fra forskjellige kilder (både BRAM og registre), setter dette samlebåndstrinnet opp alle data slik at de er klare for neste trinn.

- *Execute1* og *Execute2* inneholder 16 enheter (regeltestere) som hver tar seg av å teste betingelsen til en gitt regel på en bestemt SBlock med naboskap. Disse regner også ut resultatet dersom betingelsen slår til. Det er 16 enheter fordi 8 regler må testes ut på 2 SBlocker.

- Dersom noen av reglene slår til, velger *Select* ut resultatet fra én av disse ut. Regelen med høyste nummer har også høyest prioritet. Slår ingen regler til velges

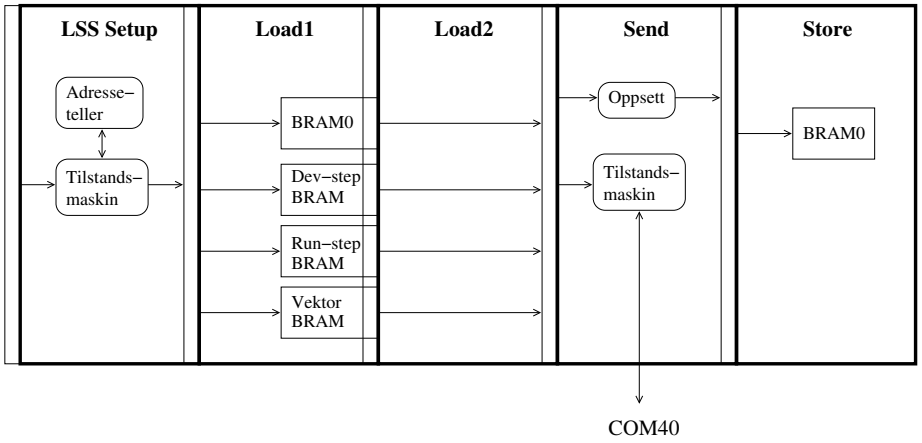
den gamle verdien til SBlocken (ingen forandring). Det sendes også ut nummeret på regelen som ble valgt. Denne verdien blir dekodet til en 256 bit lang vektor (8-til-256 dekoder). Det behandles to SBlocker i parallel slik at disse to vektoren blir satt sammen til en (bitvis "OR").

- *Store* lagrer valgt verdi til BRAM-1. For hver SBlock blir aktivert regel lagret i en BRAM. Dette skjer for hvert regelsett som blir kjørt, slik at den sist aktiverte regelen for hver blokk blir stående igjen når alle regelsett er kjørt. Den 256 bits brede vektoren skal først skrives til BRAM når alle regler har blitt kjørt på alle blokker. Derfor akkumuleres verdien opp i et eget register før det skriver til minne.

7.6.3 LSS (Load, send, store)

LSS samlebåndet gjør det mulig å lese ut og skrive verdier til/ fra minneblokkene.

Figur 7.12 viser tilstandsmaskinen til LSS samlebåndet. Samlebåndet er vist i figur 7.11.



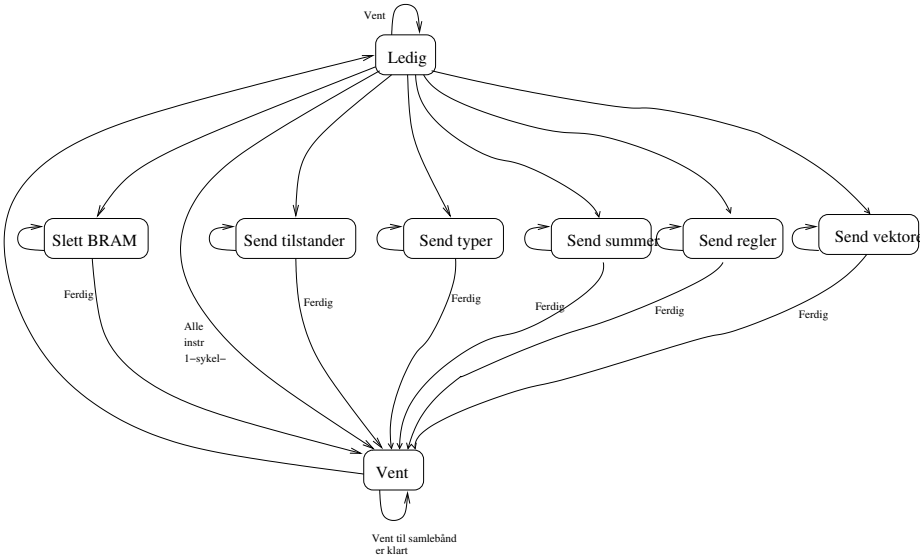
Figur 7.11: Samlebånd for LSS

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

Lesing

Når programvaren på vertsmaskinen leser ut data, skjer det på denne måten:

- En instruksjon aktiverer LSS
- Data leses ut fra aktuelt minne i *Load1* og *Load2*
- Data sendes over PCI-bussen til programvare på vertsmaskin i *Send*. Dette samlebåndssteget inneholder en tilstandsmaskin som kommuniserer med COM40.



Figur 7.12: Tilstander for LSS

Data som kan leses ut er:

- Tilstander for alle SBlockene i matrisen.
- Typer for alle SBlocker i matrisen.
- Data generert ved run-step.
- Regler som er aktivert for hver enkelt SBlock ved et development-step.
- Regler som er aktivert ved flere etterfølgende development-step. Dataene er komprimert til en 256 bits vektor.

For å utnytte kapasiteten på PCI-bussen pakkes det flere verdier, dersom det er mulig, til ett ord slik at det totale antallet ord som sendes minimeres. Dette styres av tilstandsmaskinen i *LSS Setup*.

Skriving

Når programvaren på vertsmaskinen skriver SBlockdata til BRAM-0, skjer det på denne måten:

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

Databussbredden inneholder data for to SBlocker. Det må derfor alltid skrives minimum to verdier til BRAM. Er det kun én SBlock som skal skrives må det først lese ut eksisterende data som kombineres med nye data før det skrives til BRAM.

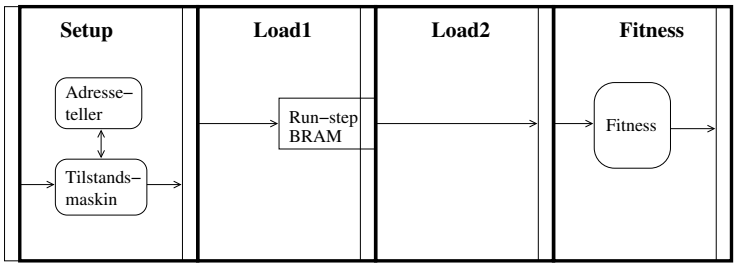
Det er mulig å sette alle SBlocker i BRAM-0 til en gitt verdi. Dette håndteres av tilstanden *Slett BRAM*.

7.6.4 Fitness

Fitnessfunksjonen startes med en egen instruksjon. All data som skal behandles blir lest ut sekvensielt fra minnemodulen.

Figur 7.14 viser tilstandsmaskinen til fitness samlebåndet. Samlebåndet er vist i figur 7.13.

- En instruksjon aktiverer samlebåndet
- Data leses ut i *Load1* og *Load2*
- Verdiene som leses ut blir evaluert ut fra fitnessfunksjonen som er implementert. Resultatet blir lagret i et register som kan leses ut via LSS-samlebåndet.



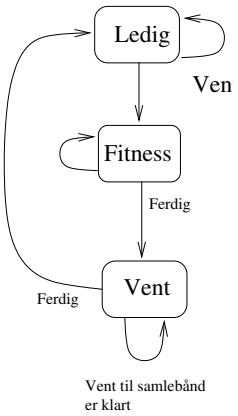
Figur 7.13: Samlebånd for fitness

Fitnessfunksjonen er implementer slik at den lett lar seg endre. Modulen som utfører funksjonen er satt inn i slutten av samlebåndet. Funksjonen kan enten være rent kombinatorisk eller den kan implementeres som ett eget samlebånd. Se kapittel 6.3 for detaljer om implementasjon av modulen.

Tilstandsmaskinen har følgene funksjoner:

- Tilstandsmaskinen venter på en instruksjon i tilstanden *Ledig*. I denne tilstanden er det også mulig å lese ut resultatet som har blitt generert ved tidligere kjøringer.
- I *Fitness* leses data fra minnet og evalueres.
- I tilstand *Vent* tømmes samlebåndet for data etter kjøring og resultatet genereres.

Utlesing Avhengig av hvilken fitnessfunksjon som benyttes kan datamengden som skal leses ut varieres. For å kunne variere hvor mye data som skal genereres av funksjonen er utlesningen implementert som et skiftregister. For hver leseoperasjon leses det 32 bit. Ved å ha flere etterfølgende instruksjoner kan dermed all generert data leses ut.



Figur 7.14: Tilstander for fitness

7.7 Organisering av BRAM for SBlockdata

Minnemodulene som benyttes for lagring av SBlockmatrisens tilstander er organisert som vedlegg G viser. Organiseringen av minnet er tilpasset development-enhetens bruk av dataene og er ikke triviell.

Kapittel 8

Resultater

Dette kapittelet oppsummerer resultater fra syntese og fra tester som er kjørt på systemet.

8.1 Syntese

De har blitt kjørt syntese med de samme verdiene som ble benyttet i [3]. Resultatene er vist i tabell 8.1. *BRAM*, *slicer* og *TBUF* viser ressursforbruk. *TBUF* er antall Tri-State-Buffer som er benyttet.

X-størrelse	Y-størrelse	Hastighet	BRAM	slicer	TBUF
8	8	80 MHz	76 (79 %)	4036 (32 %)	286 (2 %)
8	16	80 MHz	76 (79 %)	4221 (34 %)	368 (2 %)
8	32	80 MHz	76 (79 %)	4460 (36 %)	514 (4 %)
16	8	80 MHz	76 (79 %)	4218 (34 %)	368 (2 %)
16	16	80 MHz	76 (79 %)	4458 (36 %)	514 (4 %)
16	32	80 MHz	76 (79 %)	5024 (40 %)	788 (6 %)
32	8	80 MHz	76 (79 %)	4455 (36 %)	514 (4 %)
32	16	80 MHz	76 (79 %)	5018 (40 %)	788 (6 %)
32	32	80 MHz	76 (79 %)	6101 (49 %)	1318 (10 %)

Tabell 8.1: Synteseresultater

Matriser større enn 32x32 lar seg ikke syntetisere. Dimensjonene må være angitt som 2^n, hvor $3 \leq n \leq 5$.

8.2 Test

Alle tester har blitt kjørt på alle matrisestørrelsene oppgitt i tabell 8.1. Testene er nærmere forklart i etterfølgende avsnitt. Resultater generert ved testkjøring ligger på vedlagt CD.

Reglene som benyttes i etterfølgende eksperimenter er angitt i tabell 8.2. Reglene som benyttes her tar kun hensys til type, ikke tilstand for hver av SBlockene.

Det benyttes fire mulige typer for hver av SBlockene, typene er vist i figur 8.3.

Regel	Type	Senter	Sør	Øst	Nord	Vest
0	Growth fra sør	0	DC	DC	DC	DC
1	Growth fra øst	0	DC	DC	DC	DC
2	Growth fra nord	0	DC	DC	DC	DC
3	Growth fra vest	0	DC	DC	DC	DC
4	Change til 1	DC	0	DC	2	DC
5	Change til 0	DC	2	DC	DC	1

Tabell 8.2: Regler benyttet i testene. *DC* betyr «Don't care».

SBlocksymbol	Nummer	Beskrivelse
	0	Tom, SBlocken kan ikke endre tilstand
	1	XOR av SBlockens naboer
	2	OR av SBlockens naboer
	3	AND av SBlockens naboer

Tabell 8.3: Tabellen viser symbolene som er benyttet for ulike typer i resten av kapittelet.

To av testene som ble benyttet i [3] ble kjørt på nytt med den nye kretsen. *Test A: Test av konfigurasjonshastighet* fra [3] ble ikke kjørt siden modulen som styrer konfigurasjonen av FPGAen ikke er endret.

8.2.1 Funksjonell test

For å kontrollere at kretsen ikke har endre oppførsel har *Test B: Funksjonell test* fra [3] blitt benyttet.

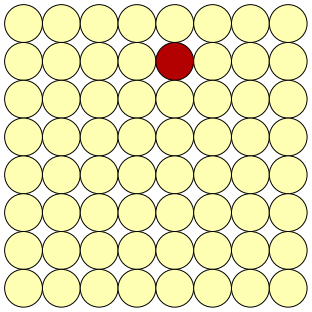
FUNCTIONAL TEST

```
1  writeRules
2  init
3  while true                    ▷ Until C-program exit
4      do config
5         run(77)
6         readback
7         switch
8         readTypes
9         readStates
10        devstep
```

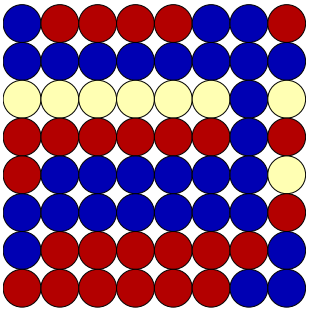
Figur 8.1: Program for funksjonell test

Figur 8.1 viser pseudokode for programmet som ble benyttet. Antall ganger **while** løkken kjøres avhenger av C-programmet på vertsmaskinen, i dette eksperimentet ble løkken totalt kjørt 150 ganger. Figur 8.2 viser utgangstilstand for matrisen som ble benyttet, 8x8 SBlocker, og tilstand etter kjøring. Resultatet stemmer overens med [3].

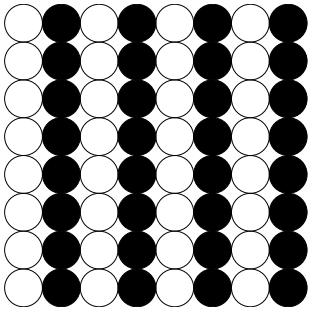
Det konkluderes dermed med at systemet har beholdt korrekt funksjonalitet. Samme test har blitt kjørt på de andre matrisestørrelsen, angitt i tabell 8.1. Alle med "fornuftige" resultat, det finnes ingen tidligere resultater å kontrollere mot.



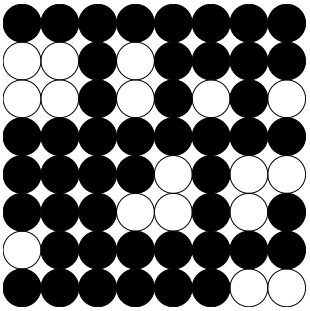
(a) SBlock-matrisene typer ved oppstart.



(b) SBlock-matrisens typer etter 150 development-step.



(c) Initiell tilstand.



(d) Tilstand ved eksperimentets slutt.

Figur 8.2: Tilstander og typer for *Funksjonell test*.

8.2.2 Hastighetstest

Test C: Hastighetstest fra [3] ble også kjørt på den nye kretsen. Denne testen er nesten identisk med testen i kapittel 8.2.1. Forskjellen består i at det er kjørt 50 000 run-step pr. development-step og 10 000 development-step. Tabell 8.4 viser resultatene av testen.

8x8	8x16	8x32	16x8	16x16	16x32	32x8	32x16	32x32
31.3s	56.4s	106.5s	56.4s	106.5s	206.7s	106.5s	206.7s	412.9s

Tabell 8.4: Kjøretiden for hastighetstest avhenger av matrisestørrelsen.

I [3] oppgis verdiene under ved kjøring av hastighetstest på en 8x8 matrise:

- Simulator: 18 minutter og 6 sekunder
- Co-prosessor i FPGA (gammelt design): 6,2 sekunder

Ved kjøring med nytt design har tidsforbruket økt til 31,3 sekunder for en 8x8 matrise. Dette er en markant tidsforskjell. Dette kan forklares ut fra formel 7.2. Det skal bemerkes at det er lagret informasjon om hvert enkelt run-step som kan leses ut. Dersom disse dataene ellers skulle vært funnet ved simulering er det betydelig raskere å kjøre eksperimentet med det nye designet.

8.2.3 Regeltest

Test av funksjonalitet som skriver ut regler som har blitt aktivert har blitt gjort med programmet vist i figur 8.3.

```
RULE TEST
 1  writeRules
 2  init
 3  while true                    ▷ Until C-program exit
 4      do config
 5         readback
 6         switch
 7         readTypes
 8         devstep                ▷ Increment counter
 9         readUsedRules
10         jumpEqual(20, PRINTVECTOR)

PRINTVECTOR
1  readRuleVector
2  resetDevCounter
3  return                         ▷ Return to while loop
```

Figur 8.3: Test av utskrift av aktiverte regler

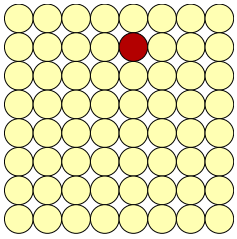
Programmet kjører først 20 development-step hvor aktiverte regler for hvert step skrives ut. Når 20 step er telt opp hopper programmet til PRINTVECTOR og 20 regelvektorer leses ut.

Når denne testen blir kjørt på en 8x8 matrise blir resultatet som vist i figur 8.4, her er de 16 første development-stepene vist. Under de fire første figurene er det skrevet på hvilke SBlocker som har aktivert regler. Notasjonen er (SBlock nr. - Regel nr.). Aktiverte regler er listet i tabell 8.5 for de 16 første development-step. Det er de samme reglene som fremkommer både i figur 8.4 og i tabell 8.5. Testen ble også kjørt på andre matrisedimensjoner med riktige resultat.

8.2.4 Fitnesstest

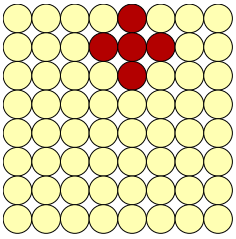
Test av funksjonen som lagrer data generert ved run-step ble testet sammen med fitnessfunksjonen. Pseudokoden for testen er gitt i figur 8.5.

Ved kjøring av systemet blir det lagret en verdi som beskriver antall satte SBlocker for hvert eneste run-step. Fitnessfunksjonen som er implementert finner lengste etter-



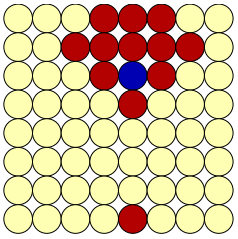
(4-0) (11-1) (13-3) (20-2)
-
-
-
-

(a) Dev-step 0



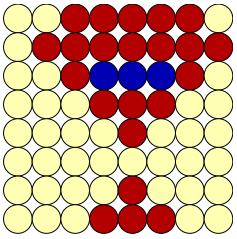
(3-1) (5-3) (10-1) (14-3)
(19-2) (20-4) (21-3) (28-2)
(60-0)
-
-

(b) Dev-step 1



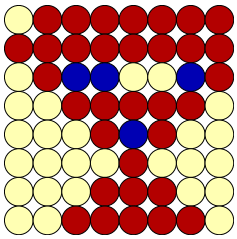
(2-1) (6-3) (9-1) (15-3)
(18-2) (19-4) (21-4) (22-3)
(27-2) (29-3) (36-2) (52-0)
(59-1) (61-3)
-

(c) Dev-step 2

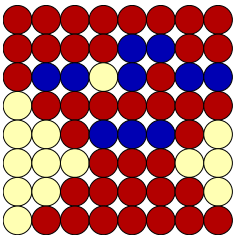


(1-1) (7-3) (8-3) (17-2)
(18-4) (20-5) (21-5) (22-4)
(23-3) (26-2) (30-3) (35-2)
(36-4) (37-3) (44-2) (51-1)
(53-3) (58-1) (62-3)

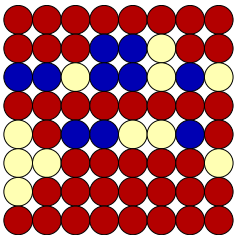
(d) Dev-step 3



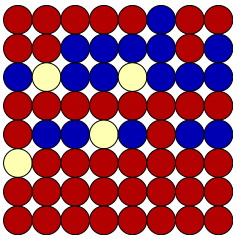
(e) Dev-step 4



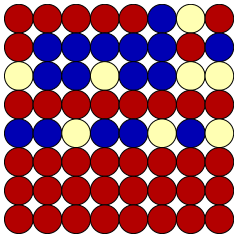
(f) Dev-step 5



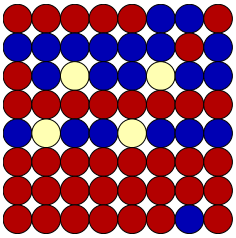
(g) Dev-step 6



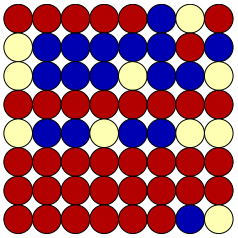
(h) Dev-step 7



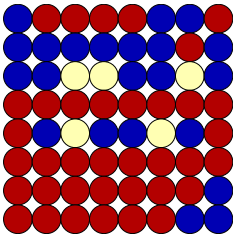
(i) Dev-step 8



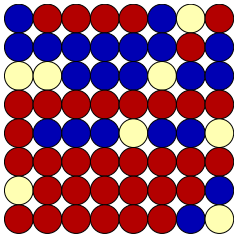
(j) Dev-step 9



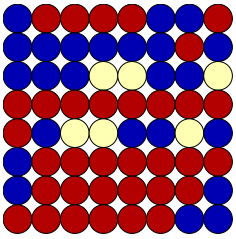
(k) Dev-step 10



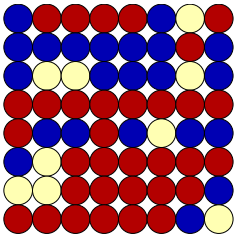
(l) Dev-step 11



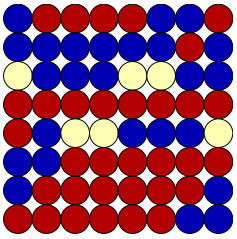
(m) Dev-step 12



(n) Dev-step 13



(o) Dev-step 14



(p) Dev-step 15

Figur 8.4: Tilstander for matrisen under kjøring av *Regel test*.

Development-step	Aktiverte regler	Development-step	Aktiverte regler
0	0, 1, 2, 3, 255.	8	2, 3, 4, 5, 255.
1	0, 1, 2, 3, 4, 255.	9	3, 4, 5, 255.
2	0, 1, 2, 3, 4, 255.	10	1, 2, 3, 4, 5, 255.
3	1, 2, 3, 4, 5, 255.	11	2, 3, 4, 5, 255.
4	1, 2, 3, 4, 5, 255.	12	2, 3, 4, 5, 255.
5	1, 2, 3, 4, 5, 255.	13	2, 3, 4, 5, 255.
6	2, 3, 4, 5, 255.	14	1, 2, 3, 4, 5, 255.
7	3, 4, 5, 255.	15	2, 3, 4, 5, 255.

Tabell 8.5: Regler som ble benyttet skrevet ut som en sammenhengende vektor (Regel 255 betyr at minst en SBlock ikke aktiverte noen regel, se kapittel 6.4).

FITNESS TEST

```
1  writeRules
2  init
3  while true                    ▷ Until C-program exit
4      do config
5          run(100)
6          readback
7          switch
8          readStates
9          readSums(100)
10         doFitness(100)
11         readFitness           ▷ Repeated sufficient times
12         devstep
```

Figur 8.5: Fitness test

følgende sekvens av tall i data som har blitt lagret. Det skal altså søkes etter den lengste sekvensen som følger mønsteret $n, n+1, n+2, ..., n+i$.

Eksempel på funn er fra kjøring med 32x32 matrise, etter 5 development-step. Da ble det rapportert om en sekvens på 3 verdier, med startverdien $0x210$. Denne sekvensen kan finnes igjen som:

- Run-step 2 - 528 (0x0210)
- Run-step 3 - 529 (0x0211)
- Run-step 4 - 530 (0x0212)

Denne sekvensen ble gjennomløpt flere ganger i kjøringen.

8.2.5 Hastighetstest av fitnessfunksjon

For å sammenligne hastighetene ved å benytte den interne fitnessfunksjonene fremfor å lese ut all tilstandsdata for hvert run-step har det blitt skrevet to program.

Det ene programmet skriver ut tilstandene for hele matrisen for hvert run-step (figur 8.6), det andre evaluerer data internt og skriver kun ut resultatet (figur 8.7). Programmet som leser ut data utfører ingen evaluering av verdiene. Lesing av resultat-data og evaluering forventes gjort uavhengig av eksperimentet, tiden for denne operasjonen er derfor ikke medregnet i resultatenen vist i tabell 8.6. Resultatene viser en markant hastighetsøkning ved bruk av den interne funksjonen.

Testene er utført med 1000 run-step for hvert development-step. Det er kjørt 1000 development-step.

Matrise	8x8	16x16	32x32
Utlesing	44.53s	259.10s	492.63s
Intern funksjon	0.08s	0.43s	0.85s
SpeedUp	556.6	602.6	579.6

Tabell 8.6: Kjøretiden for fitness- evaluering avhenger av matrisestørrelsen. Resultatenen viser en markant hastighetsøkning.

FITNESS TEST OLD

```
writeRules
init
while true
    do config
        i ← 0
        while i < 1000
            do run(1)
                readback
                readStates
                i ← i + 1
        devstep
```

Figur 8.6: Fitness evaluering ved utlesing av data.

FITNESS TEST NEW

```
writeRules
init
while true
    do config
        run(1000)
        readback
        switch
        doFitness(1000)
        readFitness
        devstep
```

Figur 8.7: Fitness evaluering ved bruk av intern funksjon.

Kapittel 9

Diskusjon

9.1 Testresultater

Alle tester som er utført viser at designet er korrekt implementert. Det finnes få resultater fra tidligere arbeider å kontrollere mot. Både simuleringer utført under utvikling og test har vist forventede resultat. Det finnes ingen kjente feil i systemet.

Funksjonell test gir samme resultat som rapportert i [3]. Det konkluderes ut fra dette med at systemet ikke har endre oppførsel. Det har heller ikke blitt gjort endringer som gjør det sannsynlig at feil har oppstått i fbm. med kjøring av matrisen.

Hastighetstesten som har blitt kjørt viser at ytelsen på systemet har avtatt. Kjøretiden er fremdeles betydelig raskere enn ved simulering. Kjøretiden har økt grunnet en økning i antall klokkesykler som kjøres for hvert run-step. Det blir nå lagret data for hvert run-step, slik at utlesing av data ikke er nødvendig for utføring av eksperimenter i samme grad som før.

Test av modulene som lagrer hvilke regler som har blitt aktivert viser forventede resultat på alle matrisestørrelse.

Det har blitt implementert en enkel fitness-funksjon i systemet. Tester viser at funksjonen funger og rapporterer korrekte verdier.

Ved å bruke den interne fitnessfunksjonen er det vist at det oppnåes en betydelig hastighetsøkning. Det kan derfor forventes at total kjøretid for mange eksperimenter vil avta tiltross for økt tidsforbruk ved enkelte tester.

9.2 Arkitektur

Systemet styres av en co-prosessor med eget instruksjonminne. Dette gjør designet anvendelig for kjøring av ulike eksperimenter. Instruksjonsettet er enkelt å bruke og det krever liten innsats for å endre eksisterende program. Development-algoritmen er hardkodet i VHDL og lar seg ikke endre fra kjøring til kjøring.

Co-prosessoren har relativt få og spesifikke instruksjoner. Den er på ingen måte turing-komplett og kan ikke utføre beregninger. Derfor må all behandling av resultater utføres i vertsmaskinen eller annen datamaskin.

Arkitekturen er tett knyttet mot BenERA FPGA-kort. Dette har mange fordeler, men gjør også designet mindre generelt.

Det er mulig å begrense logikkbehovet i designet. Det har i utvidelsene, gjort i denne oppgaven, blitt valgt å beholde kodestilen i designet. Som beskrevet i [3] er

mange valg gjort for å øke oversiktligheten i koden fremfor reduksjon i nødvendig logikk. FPGAen som benyttes er uansett mer enn stor nok for designet.

9.3 Videre arbeid

Det er store muligheter for optimalisering av designet. F. eks. kan mange av registrene relativt lett gjennbrukes. Enkelte av tellerne som er benyttet kan brukes på flere steder slik at noen kan tas ut av designet.

Videre arbeid som nevnes i [3] er:

- Utvide prosessoren med et turing-komplett instruksjonsett.
- Størrelsen på SBlock-matrisen må nå settes som en toer-potens. Ved å endre kontroll-logikken kan denne begrensningen fjernes.

Det er spesielt to moduler i designet som er designet slik at de er enkel å endre. Det er modulen som transformerer SBlock-matrisens tilstand til en 16 bits verdi og modulen som utfører fitness-funksjonen. Utvikling av moduler med andre egenskaper kan gjøres for å tilpasse systemet til andre eksperimenter som skal utføres.

Ved å lage en egen instruksjon som kan skru av og på modulen som evaluerer tilstanden for matrisen, kan effektiviteten for systemet økes når slik data ikke er interessant.

Ved syntese rapporteres det om timing-problemer. Selv om det blir meldt om feil her har alle tester av systemet gitt korrekte resultat. En enkel måte å løse dette problemet er å endre klokkefrekvensen som systemet syntetiseres til, men det går på bekostning av ytelse.

Ved å spesifisere, før syntese, hvor ulike moduler skal plasseres fysisk i brikken kan klokkehastigheten muligens økes.

Systemet er utviklet for å være en plattform for forskning på EHW. Hva som kan være nyttige utvidelser avhenger i stor grad av hva som kommer frem av resultat under videre forskningsarbeid.

9.4 Problemer under utviklingen

Det viste seg vanskelig å kjøre Xilinx ISE på den versjonen av Debian Linux som ble benyttet. Mye tyder på at ISE versjonen som var til rådighet var for gammel. Løsningen på problemet var å skrive et skript som inkluderte et bibliotek når syntese skulle gjøres og ekskludere det etterpå.

Løsningen på problemet var å veksle mellom å ha linjen:
`/usr/local/xilinx/bin/lin` inkludert i filen `/etc/ld.so.conf`. Og deretter kjøre kommandoen `sudo ldconfig`.

Bibliografi

[1] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller og Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[2] M. Rodgers D. Edenfeld A. B. Kahng og Y. Zorian. 2003 technology roadmap for semiconductors. 2003.

[3] Asbjørn Djupdal. Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter. Hovedfagsoppgave, juni 2003.

[4] Hugo de Garis. Cam-brain - growing an artificial brain with a million neural net modules inside a trillion cell cellular automata machine, 1993.

[5] Pauline Haddow og Gunnar Tufte. An evolvable hardware fpga for adaptiv hardware. I *Congress on Evolutionary Computation, CEC2000*, side 553–560, 2000.

[6] Pauline C. Haddow og Piet von Remortel. From here to there: Future robust ehw technologies for large digital designs.

[7] Pauline C. Haddow og Gunnar Tufte. Bridging the genotype-phenotype mapping for digital fpgas. I *Evolvable Hardware*, side 109–115, 2001.

[8] Alister Hamilton, Kostis Papathanasiou, Morgan Tamplin og Thomas Brandtner. Palmo: Field programmable analogue and mixed-signal vlsi for evolvable hardware. I *ICES*, side 335–344, 1998.

[9] S. Harding og J. F. Miller. Evolution in materio: Initial experiments with liquid crystal.

[10] T. Higuchi, H. Iba og B. Manderick. *Massively Parallel Artificial Intelligence*, kapittel 12. Evolvable Hardware, side 398–421. MIT Press, 1994.

[11] J. H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

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

[13] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, mai 1994.

[14] P. Layzell. The 'evolvable motherboard'. a test platform for the research of intrinsic hardware evolution. Cognitive Science Research Paper 479, School of Cognitive and Computing Sciences, University of Sussex, U.K., 1998.

[15] P. Layzell. A new research tool for intrinsic hardware evolution. I *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware (ICES98)*, 1998.

[16] P. Layzell. *Hardware Evolution:On the Nature of Artificially Evolved Electronic Circuits*. 2001.

[17] Daniel Mange, Maxime Goeke, Dominik Madon, André Stauffer, Gianluca Tempesti og Serge Durand. Embryonics: A new family of coarse-grained field programmable gate array with self-repair and self-reproducing properties. I *Papers from an international workshop on Towards Evolvable Hardware, The Evolutionary Engineering Approach*, side 197–220, London, UK, 1996. Springer-Verlag.

[18] Julian F. Miller og Keith L. Downing. Evolution in materio: Looking beyond the silicon box. I *Evolvable Hardware*, side 167–176, 2002.

[19] Masahiro Murakawa, Shuji Yoshizawa, Toshio Adachi, Shiro Suzuki, Kaoru Takasuka, Masaya Iwata og Tetsuya Higuchi. Analogue ehw chip for intermediate frequency filters. I *ICES '98: Proceedings of the Second International Conference on Evolvable Systems*, side 134–143, London, UK, 1998. Springer-Verlag.

[20] Masahiro Murakawa, Shuji Yoshizawa, Isamu Kajitani, Xin Yao, Nobuki Kajihara, Masaya Iwata og Tetsuya Higuchi. The grd chip: Genetic reconfiguration of dsps for neural network processing. *IEEE Trans. Computers*, 48(6):628–639, 1999.

[21] NallaTech. *BenERA Users Guide, NT107-0072 - Issue 3*.

[22] NallaTech. *FUSE C/C++ API Developers Guide, NT107-0068 - Issue 3*.

[23] NallaTech. *FUSE System Software Users Guide, NT107-0068V2 - Issue 2*.

[24] J von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966.

[25] PCI Industrial Computer Manufacturers Group (PICMG). *CompactPCI Specification*.

[26] Héctor Fabio Restrepo og Daniel Mange. An embryonics implementation of a self-replicating universal turing machine. I *ICES*, side 74–87, 2001.

[27] Palash Sarkar. A brief history of cellular automata. *ACM Comput. Surv.*, 32(1):80–107, 2000.

[28] Moshe Sipper. *Evolution of Parallel Cellular Machines*. 2004.

[29] Moshe Sipper, Eduardo Sanchez, Daniel Mange, Marco Tomassini, Andrés Pérez-Uribe og André Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Trans. Evolutionary Computation*, 1(1):83–97, 1997.

[30] Andre Stauffer, Daniel Mange, Gianluca Tempesti og Christof Teuscher. Biowatch: A giant electronic bio-inspired watch. I *EH '01: Proceedings of the The 3rd NASA/DoD Workshop on Evolvable Hardware*, side 185, Washington, DC, USA, 2001. IEEE Computer Society.

[31] Adrian Stoica, Alex S. Fukunaga, Ken Hayworth og Carlos Salazar-Lazaro. Evolvable hardware for space applications. I *ICES*, side 166–173, 1998.

[32] Gianluca Tempesti. A self-repairing multiplexer-based fpga inspired by biological processes, 1998.

[33] Gunnar Tufte og Pauline Haddow. Identification of functionality during development on a virtual sblock. 2003.

[34] Gunnar Tufte og Pauline C. Haddow. Building knowledge into developmental rules for circuit design. I *ICES*, side 69–80, 2003.

[35] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[36] Vmiacc-0320 product overview.

[37] Vmicpci-7760 product overview.

[38] Xilinx. *Virtex Series Configuration Architecture User Guide, versjon 1.5*.

[39] Xilinx. *VirtexE Complete Datasheet, versjon 1.7*.

Tillegg A

Instruksjonsmanual

Dette vedlegget er hentet fra [3] og oppdatert med nye instruksjoner som er kommet til under arbeidet med denne oppgaven. Alle nye instruksjoner er merket med '*'.

Alle instruksjoner har følgende generelle format:

operander	størrelse	opkode
n–6	5	4–0

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

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

A.1 break

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

ubrukt	0	01101
31–6	5	4–0

A.2 clearBRAM

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

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

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

A.3 config

Konfigurer SBlockmatrise med data fra BRAM 1.

ubrukt	0	00111
31–6	5	4–0

A.4 devstep

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

ubrukt	0	01010
31–6	5	4–0

A.5 doFitness*

Kjører en implementert fitnessfunksjon på generert data.

ubrukt	n	0	011010
31–6	n–8	5	4–0

- *n*; Antall genererte verdier funksjonene skal kjøres på. Alltid fra adress 0 og opp til verdien.

A.6 end

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

ubrukt	0	01111
31–6	5	4–0

A.7 jump

Start utføring av instruksjoner som er lagret i instruksjonslageret.

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

- *adresse*; Hopp til denne adressen i instruksjonslageret

A.8 jumpEqual*

Hopper til angitt posisjon når antall kjørte development-step tilsvarer gitt verdi.

adresse	ubrukt	verdi	ubrukt	0	10110
31–24	23–17	16–8	7–6	5	4–0

- *verdi*; Antall development-step som skal ha blitt kjørt før instruksjonen hopper.
- *adresse*; Hopp til denne adressen i instruksjonslageret

A.9 nop

Ingen operasjon.

ubrukt	0	00000
31–6	5	4–0

A.10 readback

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

ubrukt	0	01000
31–6	5	4–0

A.11 readFitness*

Les tilbake resultat produser av fitnessfunksjon. Leser alltid 32 bit, består resultatet som skal leses av flere bit kalles funksjonene flere ganger.

ubrukt	0	11001
31–6	5	4–0

A.12 readRuleVector*

Les antall vektorer som beskriver hvilke regler som har blitt kjørt for hvert enkelt development-step.

ubrukt	n	ubrukt	0	11000
31–24	23–8	7–6	5	4–0

- *x*; Antall vektorer som skal leses ut

A.13 readState

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

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

- *x*; SBlockens x-posisjon
- *y*; SBlockens y-posisjon

A.14 readStates

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

ubrukt	0	10001
31–6	5	4–0

A.15 readType

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

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

- *x*; SBlockens x-posisjon
- *y*; SBlockens y-posisjon

A.16 readTypes

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

ubrukt	0	10000
31–6	5	4–0

A.17 readSums*

Leser ut verdier lagret ved run-step.

ubrukt	antall	0	10100
31–11	11–8l	5	4–0

- *antall*; Antall verdier som skal leses

A.18 resetDevCounter*

Nullstiller registeret som teller antall development som har blitt kjørt.

ubrukt	0	10111
31–6	5	4–0

A.19 run

Kjør SBlockmatrise.

sykler	ubrukt	0	01001
31–8	7–6	5	4–0

- *sykler*; Antall sykler SBlockmatrisen skal kjøres

A.20 readUsedRules*

Skriver ut hvilke regler som har blitt kjørt for siste development-step.

ubrukt	0	10101
31–6	5	4–0

A.21 setNumberOfLastRule

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

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

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

A.22 store

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

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

- *adresse*; Adressen hvor programmet skal lagres fra

A.23 switch

Bytt om på de to BRAM-ene.

ubrukt	0	00011
31–6	5	4–0

A.24 writeLUTConv

Skriv til LUT konverteringstabell. Denne tabellen benyttes for å oversette et gitt SBlock-typenummer til en 32-bits LUT når SBlockmatrisen konfigureres.

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

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

A.25 writeRule

Skriv regel til regellageret. Disse benyttes av developmentenheten.

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

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

A.26 writeState

Skriv en enkelt SBlocks tilstand til BRAM 0.

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

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

A.27 writeType

Skriv en enkelt SBlocks type til BRAM 0.

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

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

A.28 Komplett instruksjonssett

Navn	Type	Argumenter	Beskrivelse
break	Kort		Avslutt kjøring fra instruksjonslager
clearBRAM	Kort	type, tilstand	Setter alle SBlocker i BRAM-0 til gitt verdi
config	Kort		Konfigurer SBlockmatrise fra BRAM-1
devstep	Kort		Kjør developmentsteg fra BRAM-0 til BRAM-1
doFitness	Kort		Kjører en fitnessfunksjon og lagrer resultatet i et buffer
end	Kort		Stopp lagring av instruksjoner
jump	Kort	adresse	Hopp til adresse i instruksjonslager
jumpEqual	Kort	verdi, address	Hopper til addresse dersom antall dev-step er kjørt
nop	Kort		Ingen handling
readback	Kort		Les tilbake fra SBlockmatrise til BRAM-1
readFitnes	Kort		Sender 32 bit data fra fitness register, flere etterfølgende kall gir mer data
readRuleVector	Kort	antall	Skriver ut et gitt antall vektorer som beskriver kjørte regler
readState	Kort	x, y	Send tilstand fra BRAM-0 over PCI
readStates	Kort		Send alle tilstander fra BRAM-0 over PCI
readSums	Kort	antall	Les tilbake lagrede summer
readType	Kort	x, y	Send type fra BRAM-0 over PCI
readTypes	Kort		Send alle typer fra BRAM-0 over PCI
resetDevCounter	Kort		Nullstiller register som teller antall kjørte dev-step
run	Kort	sykler	Kjør SBlockmatrise gitt antall sykler
readUsedRules	Kort		Skriver ut hvilke regler som er kjørt på hvilke blokker
setNumberOfLastRule	Kort	prioritet	Angir regel med høyest prioritet
store	Kort	startadresse	Lagre etterfølgende instruksjoner i instruksjonslager
switch	Kort		Bytt om på BRAM-ene
writeLUTConv	Lang	lut, type	Skriv til LUT konverteringstabell
writeRule	Lang	regel, prioritet	Skriv til regellager
writeState	Kort	tilstand, x, y	Skriv tilstand til BRAM-0
writeType	Kort	type, x, y	Skriv type til BRAM-0

Tabell A.1: Oversikt over det komplette instruksjonssettet etter utvidelsene.

Tillegg B

Regelformat

Vedlegget er hentet fra [3].

Regler beskrives internt på samme måte som de kodes i *writeRule*-instruksjonen (se tillegg A.25).

Hver regel består av følgende felt:

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

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

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

B.1 Betingelse

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

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

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

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

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

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

B.2 Resultat

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

Change For *Change* kodes resultatet slikt:

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

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

Growth For *Growth* kodes resultatet slikt:

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

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

Tillegg C

Filer, syntetisering og kjøring

C.1 Filhierarki

Filene som er benyttet i prosjektet ligger på vedlagt CD. Figur C.1 viser hvordan filene er organisert. *Hardware* er alt som har med maskinvaren å gjøre (vedlegg C.2.2), *software* er testprogrammet (vedlegg C.3.2), *report* er rapporten, samt kildekoden og figurer. *Sblock_visual* er programmet beskrevet i vedlegg E for visualisering av matriser.

```
.
|-- hardware
|   |-- output
|   `-- vhdl
|-- report
|-- sblock_visual
|   `-- output
`-- software
    |-- c-code
    `-- output

9 directories
```

Figur C.1: Filene på vedlagt CD er ordnet som figuren viser.

C.2 Maskinvare

C.2.1 Syntese

Noen egenskaper ved designet spesifiseres ved syntese. Dette gjøres i *Makefile*.

Det er tre verdier som spessifiseres:

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

- COORD_SIZE_X: Setter størrelse i x-retning på SBlock-matrisen. Velges f.eks 5 vil SBlockmatrisen være $2^5 = 32$ SBlocker i x-retning.
- COORD_SIZE_Y: Setter størrelse i y-retning på SBlock-matrisen.

Minimal SBlockmatrisestørrelse er $8 \cdot 8$ og maskinvarebegrensninger i VirtexE-1000 begrenser størrelsen til maksimalt $32 \cdot 32$.

Syntese forutsetter et GNU-miljø med Xilinx ISE installert. Syntese utføres med kommandoen:

```
make
```

Dette produserer en bitfil «sblock_matrix.bit» i samme katalog. Denne bitfilen brukes til å konfigurere FPGAen. Det blir også produsert en mengde informasjonsfiler.

C.2.2 VHDL-filer

Avsnittet beskriver alle VHDL-filer som benyttes til syntetisering. Filer merket med * er skrevet som en del av denne oppgaven. Enkelte av de eksisterende filene er også blitt endret.

All kode som er skrevet, eller endret vesentlig, som en del av denne oppgaven er markert slik:

```
--Kaa
  *Ny VHDL-kode*
--Kaa
```

Toppnivå

- *package.vhd.in*: Pakke med konstanter og komponentdeklarasjoner. Denne blir preprosessert med GNU-verktøyet m4 for å sett konstanter fra Makefile. *Filen package.vhd skal ikke redigeres, den blir generert fra package.vhd.in.*
- *funct_package.vhd*: Inneholder noen funksjoner.
- *toplevel.vhd*: Toppnivå med sammenkobling av komponenter.

SBlockmatrise Dette er den kjørbare SBlockmatrisen:

- *sblock_matrix.vhd*: SBlock-matrise, satt sammen av SBlocker.
- *sblock.vhd*: En enkelt SBlock.

Samlebånd Dette er alle samlebåndsstegene:

- *fetch.vhd*: Samlebåndsstegene *Fetch1*, *Fetch2*, *PCI Fetch* og *Store*.
- *decode.vhd*: *Decode*-samlebåndssteg.
- *dev.vhd*: DEV-samlebånd (Development).
- *sbm_pipe.vhd*: SBM-samlebånd (SBlock-matrise).
- *lss.vhd*: LSS-samlebånd (Load, Send, Store).
- *fitness.vhd**: Fitness-samlebånd.

Fitness

- *fitness_funk.vhd**: Fitness-funksjonen som er satt inn i siste trinn i fitness-samlebåndet.

Minnemoduler Modulene som lagrer data er implementert med BRAM.

- *rulevector_mem**: Lagrer den 256 bit brede vektoren som beskriver kjørte regler.
- *runstep_mem**: Minne som lagrer data generert ved run-step, har 8192 minnelokasjoner.
- *usedrules_mem**: Lagrer hvilke regel som har blitt kjørt på hver SBlock under et development-step.

Run-step funksjon

- *run_step_funk.vhd**: Rammeverk for implementasjon av funksjon som transformerer SBlock-matrisens tilstand til en 16 bits verdi.

Summeringsmoduler Summeringsmodulen er satt inn i *run_step_funk.vhd*.

- *sum_modul**: Sekvensiell adderer.
- *bitcounter16.vhd*, bitcounter8.vhd*, bitcounter4.vhd**: Modulene teller et gitt antall inngangssignaler.

Andre enheter

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

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

- *addr_gen.vhd*: Setter sammen en BRAM-adresse gitt x- og y-posisjon til SBlock.
- *counter.vhd*: Generell teller.
- *rule_exec.vhd*: Tester og kalkulerer resultat for en gitt regel på en gitt SBlock (med naboskap). Del av DEV-samlebåndet.
- *rule_select.vhd*: Velger ut ett resultat fra et sett med regler som har testet mot samme SBlock. Del av DEV-samlebåndet.

- *sbm_bram.vhd*: BRAM brukt til lagring av SBlockmatrisedata, del av SBM BRAM MGR.
- *word_select.vhd*: Et skiftregister som benyttes for å aktivere et sett med enable-signaler, en etter en.
- *decoder_8to256.vhd**: En 8 til 256 decoder.

Andre filer

- *sblock_matrix.ucf*: Fil som beskriver hvordan designet skal fysisk kobles mot pinnene på brikken, oppgir også tidskrav.
- *Makefile*: Fil som benytter GNU make for å styre syntese av designet.

C.3 Programvare

All programvare er skrevet i ANSI C.

C.3.1 Testprogram

Ved kompilering må følgende verdier settes i *Makefile*:

- COORD_SIZE_X: Setter størrelse i x-retning på SBlock-matrisen. Velges f.eks 5 vil SBlockmatrisen være $2^5 = 32$ SBlocker i x-retning.
- COORD_SIZE_Y: Setter størrelse i y-retning på SBlock-matrisen.

Disse verdiene må settes til det samme som tilsvarende verdier var ved syntes av maskinvaren (kapittel C.2.1).

Testprogrammet kompileres med kommandoen:

```
make
```

Dette produserer et program i sblocktest-katalogen. Sblocktest tar følgende argumenter:

```
./sblocktest <bitfil> <utdatafil> [[[<test>] <card>] <modelsim>]
```

- `bitfil`: filnavn til bitfilen som skal konfigurere bruker-FPGAen.
- `utdatafil`: navn på fil som utdata skrives til. Resultatene fra eksperimentet skrives hit.
- `test`: nummer til testen som skal kjøres. Det er implementert følgende tester:
  - *0*: Funksjonell test.
  - *1*: Hastighet test.
  - *2*: Regel test.
  - *3*: Fitness test.
  - *4*: Fitness test med gammelt instruksjonssett.
  - *5*: Fitness test med nytt instruksjonssett.

Dersom testnummer utelates kjøres test 0.

- `kort`: nummer på BenERA-kort som skal benyttes. Hvilke nummer som kan benyttes avhenger av maskinen kortene er montert. Ved kjøringe på nalle.idi.ntnu.no er valgene 1 eller 2. Dersom kortnummer utelates benyttes kort 1.
- `modelsim`: filen modelsim script skal skrives til. Dette scriptet kan brukes dersom samme program skal simuleres i ModelSim. Brukes ved utvikling av design. Dersom filnavn utelates skrives det ikke modelsim script.

C.3.2 C-filer

Alle C-filer som benyttes i testprogrammet er listet her.

- *types.h*: Generelle typer og konstanter
- *sblocklib.h*: Header-fil for sblocklib
- *read_print.h*: Header-fil for read_print
- *rules.h*: Header-fil for rules
- *sblocktest.c*: C-fil for testprogram
- *sblocklib.c*: C-fil for programvarebiblioteket sblocklib
- *read_print.c*: C-fil for funksjoner for lesing av data over PCI-bussen og utskrift av formatert data til fil.
- *rules.c*: C-fil for generering av alle regler som har blitt benyttet under testkjøringen.
- *Makefile*: Fil som benytter GNU make for å styre kompilering av programmet.

Sblocklib er et abstraksjonslag over FUSE API som tilbyr funksjoner for styring av koprosessoren. Funksjonene er dokumenterte i *sblocklib.h*. Se evt. kildekoden til testprogrammet for bruk av biblioteket.

Tillegg D

Programvare

Utviklingen har blitt utført på en standard datamaskin med Debian GNU/ Linux operativ-system. Kompilering av programvare for cPCI-maskinene (vertsmaskinen for BenERA kortet) har blitt gjort på vertsmaskinen.

D.1 ISE - Xilinx

Syntese har blitt gjort med Xilinx ISE versjon 6.1e for Linux. Oppsettet for syntesen er skrevet i en *Makefile* og krever derfor GNU verktøyet Make er installert.

D.2 Modelsim

Modelsim versjon 5.7g ble benyttet for simulering av systemet. For å gjøre simulering enklere har det blitt benyttet et c-program som har skrevet scriptfiler for lesing i Modelsim.

D.3 GNU C compiler

Kompilering av C-program er gjort med Gcc versjon 2.95.

Tillegg E

SBlock-visualisering

For å enkelt kunne visualisere resultatene fra kjøring har det blitt skrevet ett program om visualiserer matrisene vha. Post Script. Programmet leser en tekstfil på formatet som vist i figur E.1. Filen kan ha vilkårlig mengde matriser i tilfeldig rekkefølge. Det er vesentlig at alle matriser som skal visualiseres starter med linjen: `"Printing types:"` eller `"Printing states:"`. Typer visualiseres etter typenummer (0 til 3) og tilstander i sort ('X') eller hvit ('.').

```
Printing types:
0 0 2 2 2 2 2 0
0 2 2 2 2 2 2 2
0 0 2 1 1 1 2 0
0 0 0 2 2 2 0 0
0 0 0 0 2 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0
0 0 0 2 2 2 0 0
```

(a) Typer.

```
Printing states:
 . X X X X X X X
 . X X X X X X X
 . X X X . X X X
 . X . X X X . X
 . X . X X X . X
 . X . X . X . X
 . X . X X X . X
 . X . X X X . X
```

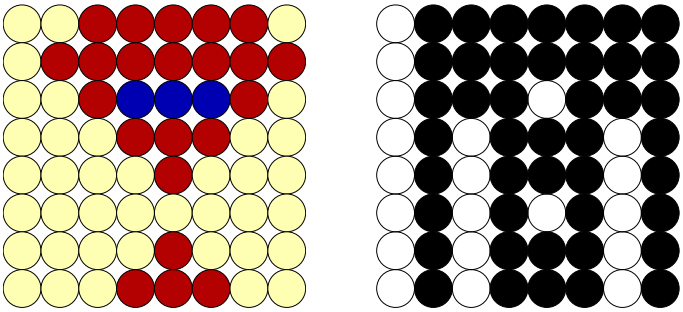
(b) Tilstander.

Figur E.1: Format på tekst som skal konverteres til EPS.

Programmet kjøres med:

```
./sblock [<x-dim> <y-dim>]
```

`x-dim` og `y-dim` er dimensjoner for matrisene i X og Y retning. Oppgis ikke dimensjonene kjøres programmet med 8 i begge retninger. Resultatfilene i EPS-format skrives til en undermappe av der programmet som heter `output`. Filene nummereres etter hvor i tekstfilen de leses. Figur E.2 viser resultatet når programmet er kjørt med teksten i figur E.1.



(a) Eksempel på visualisering av typer.

(b) Eksempel på visualisering av tilstander.

Figur E.2: Visualisering av tilstander og typer.

Tillegg F

Frekvensdomenekonvertering

Vedlegget er hentet fra [3].

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

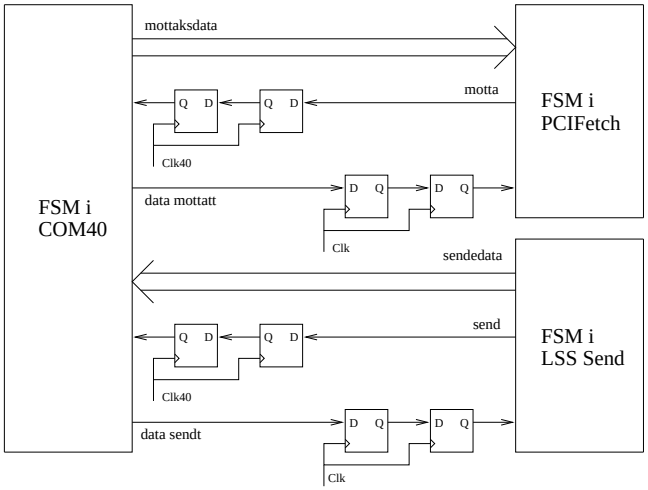
Datautveksling mellom de to frekvensdomenene skjer ved hjelp av to busser; én for sending av data fra FPGA-en til PCI-bussen, og én for mottaking av data fra PCI-bussen til FPGA-en. Kommunikasjonen er asynkron og bruker en enkel *Request–Acknowledge*-protokoll. Kretsen i det raske frekvensdomenet setter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

Koblingen mellom de to frekvensdomenene vises i figur F.1. For å hindre metastabilitet blir alle kontrollsignaler synkronisert til frekvensdomenet de leses i ved hjelp av to etterfølgende vipper.

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

Synkroniseringsprotokollen for lesing av data fra PCI-bussen vises i tabell F.1. PCI Fetch starter en leseoperasjon ved å sette «motta»-signalet. Når dette oppdages av COM40 setter COM40 igang med å hente data fra PCI-FPGA. Når dette er gjort, legger COM40 data på databussen og setter signalet «data mottatt». PCI Fetch leser data fra databussen, og resetter «motta» for å signalisere at databussen ikke er i bruk lengre. COM40 svarer med å resette «data mottatt», og dataoverføringen er avsluttet.

Synkroniseringsprotokollen for skriving av data til PCI-bussen vises i tabell F.2. Denne fungerer på samme måte som for lesing, bortsett fra at dataoverføringen går motsatt vei.



Figur F.1: Kobling mellom frekvensdomener. Clk40 = 40MHz, Clk = Maksfrekvens

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

Tabell F.1: Synkronisering av leseoperasjoner

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

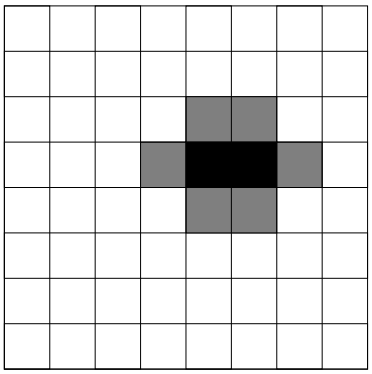
Tabell F.2: Synkronisering av skriveoperasjoner

Tillegg G

BRAM-organisering

Vedlegget er hentet fra [3].

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



Figur G.1: SBlocker som leses ut fra BRAM

Developmentenheten er avhengig av å ha klar data for 8 forskjellige SBlocker for hver eneste sykel. Dette er vist i figur G.1. Det er de svarte SBlockene som skal prosesseres, og da kreves det at data for disse, i tillegg til alle nabo-SBlocker (grå SBlocker), må være tilgjengelige for hver sykel. Da det kun finnes toports RAM i Xilinx VirtexE, vil en naiv implementasjon bare klare å lese ut to SBlocker.

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

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

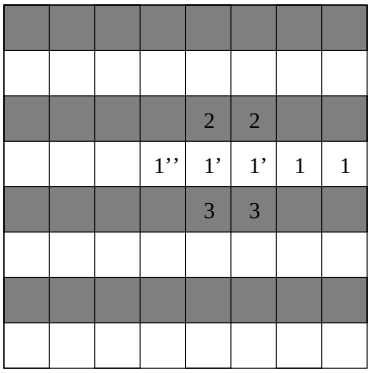
Første observasjon er at alle SBlocker som er nødvendige ligger samlet i ett område (riktignok spredd utover tre forskjellige rader). Dette medfører at leser man ut én gitt SBlock er det stor sannsynlighet for at man også trenger SBlocken som kommer

umiddelbart etterpå. En ting man kan gjøre for å minske antall RAM-oppslag er da å sette databussbredden til BRAM-ene til 2 · SBlockstørrelse. Da vil hvert RAM-oppslag alltid gi to etterfølgende SBlocker.

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

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

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



Figur G.2: Organisering av BRAM

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

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

Tillegg H

Grensesnitt PCI- og bruker-FPGA

Vedlegget er hentet fra [3].

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

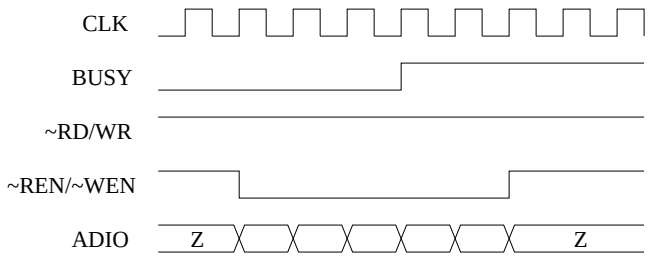
- *AS/˜DS*: Høy: adresse sendes på ADIO, lav: data sendes på ADIO. Dette signalet sier hvilken dataordtype som sendes over ADIO. To typer finnes: Adresse og data. Bruker-FPGA-en og programvaren som sender disse bestemmer hvordan disse skal tolkes.
- *EMPTY*: FIFO-buffer er tomt
- *BUSY*: FIFO-buffer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

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

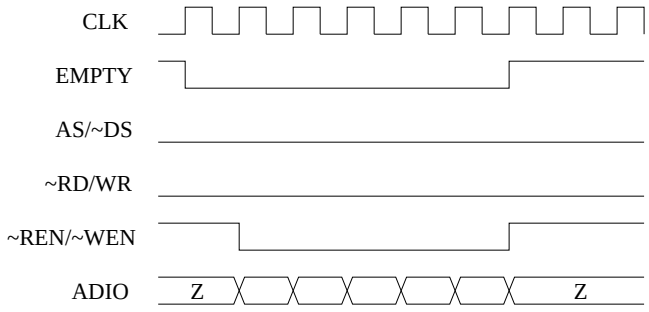
Klokken må gå på 33MHz–40MHz. Dette betyr at kommunikasjonshastigheten begrenses av hastigheten på CPCI-bussen som har en øvre teoretisk grense ved 132MB pr. sekund [25].

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

Se brukermanualen til BenERA [21] for mer informasjon.



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



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