NTNU
Det skapende universitet

En parallell løsning av cellulær utvikling i maskinvare

Mats Jørgen Øyan

Master i datateknikk
Oppgaven levert: Juni 2006
Hovedveileder: Gunnar Tufte, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

For å utnytte den potensielt massive parallelle databehandlingskraften i en cellulær datamaskin foreslår datamaskingruppa på IDI å bruke cellulær utvikling.

På IDI har en parallell maskinvareimplementasjon av en cellulær datamaskin blitt implementert i en FPGA sammen med en delvis parallell utviklingsprosess. For å utvide skopet til eksperimenter som kan utføres på en slik maskin er det nødvendig med en komplett parallell implementasjon.

Kandidaten skal ta bakgrunn i funksjonalitet i den eksisterende implementasjonen, men flytte utviklingsprosessen inn i hver enkelt celle.

Oppgaven gitt: 20. januar 2006
Hovedveileder: Gunnar Tufte, IDI

Sammendrag

Dagens elektroniske kretser utvikles som regel med en top-ned designstrategi. Siden kretsene blir større og mer komplekse blir designet av kretsene også en større og vanskeligere jobb. For å takle denne økende kompleksiteten har det blitt introdusert nye designmetoder, blant annet inspirert av naturen. Denne type maskinvare kan for eksempel bruke evolusjon, cellestrukturer eller prøve å simulere intelligens.

En mye brukt platform for biologisk inspirert maskinvare er FPGA. På grunn av en del begrensninger med denne har datamaskingruppa ved IDI på NTNU introdusert en virtuell FPGA kalt Sblock. Dette er en platform som fjerner en del av begrensningene med å jobbe direkte på en FPGA og består av en matrise med celler. Hver celle har en funksjonalitet og en development-prosess som forandrer funksjonaliteten til cellen ut fra visse regler. Tidligere har det blitt laget en implementasjon av en Sblock-matrise med development-prosessen i en sentral ko-prosessor. Målet med denne oppgaven er å implementere og teste ut en Sblock-matrise hvor også development-prosessen er lagt parallellt i hver Sblock og finne ut hvor store matriser som er mulige å lage med tilgjengelig maskinvare.

Resultatene viser at Sblock-matrisen skalerer godt og at maskinvarebehovet øker lineært med antall Sblocker i matrisen. Den nye implementasjonen fungerer korrekt, men noen instruksjoner har blitt fjernet og noen lagt til i forhold til den tidligere implementasjonen.

Forord

Denne diplomoppgaven er utført ved Norges teknisk-naturvitenskaplige universitet (NTNU), Institutt for datateknikk og informasjonsvitenskap, datamaskingruppen.

Oppgaven teller 30 studiepoeng og utføres på vårsemesteret 5. året som en avslutning på sivilingeniørstudiet.

Takk til veileder Gunnar Tufte for et godt samarbeid, hjelp og en inspirerende oppgave.

Mats Jørgen Øyan
14. juni 2006

Innhold

Figurer vii

Tabeller ix

1 Innledning 1

2 Bakgrunn 3
- 2.1 Biologisk inspirert maskinvare 3
  - 2.1.1 Phylogeny 3
  - 2.1.2 Ontogeny 5
  - 2.1.3 Epigenesis 8
- 2.2 Maskinvare 9
  - 2.2.1 FPGA 9
  - 2.2.2 Xilinx Virtex-II Pro 11
- 2.3 FPGA EHW 15
  - 2.3.1 Relatert arbeid 15
  - 2.3.2 Tidligere arbeid 16

3 Development model 19
- 3.1 Beskrivelse av en sblock 19
- 3.2 Matrisen 20
- 3.3 Regler og rekonfigurering 21
- 3.4 I/O 21
- 3.5 Utvikling 22

4 Implementasjonens funksjonalitet 25
- 4.1 Instruksjonssett 25
- 4.2 Funksjonalitet 26
- 4.3 Eksempelprogram 26

5 Maskin- og programvareplatform 29
- 5.1 Nallatech BenBlue™-III 29
  - 5.1.1 FUSE og DIME 30
  - 5.1.2 Inter-kommunikasjon 31
- 5.2 Nalle 31
- 5.3 Arbeidsstasjon 32

6 Implementasjon **33**
6.1 Kompromisser . . . 33
6.1.1 Regelsett/Genom . . . 33
6.1.2 Konfigurering og lagring av matrise . . . 34
6.2 Oppbygning . . . 35
6.2.1 State LUT . . . 36
6.2.2 Type2LUTConf . . . 36
6.2.3 Development unit . . . 37
6.2.4 Rulechecker . . . 37
6.2.5 State og Type buffer . . . 37
6.3 Tilstander . . . 38
6.3.1 Reset . . . 38
6.3.2 Idle . . . 39
6.3.3 Set . . . 39
6.3.4 Run . . . 40
6.3.5 Development . . . 41
6.3.6 Configure . . . 41
6.3.7 Save . . . 42
6.4 Kontroller . . . 43
6.4.1 Instruksjonssett . . . 43
6.4.2 Kommunikasjon . . . 46

7 Resultater **49**
7.1 Syntese . . . 49
7.2 Manuell test . . . 50
7.3 Funksjonell test . . . 52
7.4 Hastighetstest . . . 53

8 Diskusjon **61**
8.1 Testresultater . . . 61
8.2 Arkitektur . . . 61
8.3 Videre arbeid . . . 62

Bibliografi **63**

A Filer og syntetisering **67**

B Instruksjonsmanual **69**

C Regelformat **77**

D Kommunikasjonsprotokoll **81**

Figurer

2.1 POE-rammeverket 4
2.2 Genetisk algoritme 5
2.3 Kommunikasjon i en cellulær automat 6
2.4 Eksempel på en cellulær automat 8
2.5 Oppbygning av en typisk FPGA 10
2.6 Standard CLB 11
2.7 FPGA island-style routing 12
2.8 Oppbygning av en Virtex-II FPGA 13
2.9 Oppbygning av en Virtex 2 Pro CLB 13
2.10 Eksempel på en sblock-matrise 15
2.11 Gammel Sblock-matrise 16

3.1 Komponentene i den cellulære development modellen [27] 20
3.2 Eksempel på en matrise av Sblocker 20
3.3 LUT-konfigurasjon for en Sblock 21
3.4 Stabil Sblockmatrise 23
3.5 Endelig developmentløkke 23

4.1 Pseudokode for kjøring av en Sblock-matrise 28

5.1 Nallatech BenBlue^{TM}-III 29
5.2 Funksjonellt diagram over BenERA [18] 30
5.3 Kommunikasjonslinjer for BenBlue [18] 31

6.1 Regel-BRAM koblet til Sblock-matrise 34
6.2 Konfigurasjon av Sblock-matrisen 35
6.3 Systemarkitektur 36
6.4 Overordnet sammenkobling i Sblock-matrisen 37
6.5 Oppbygningen av en Sblock 38
6.6 State LUT og Type2LUTConv 39
6.7 Tilstandsdiagram for en Sblock 40
6.8 Oppstart av Sblocken 41
6.9 Tilstandsdiagram for set-tilstanden 42
6.10 Setting av Sblock-tilstand og -type fra BRAM. 43
6.11 Kjøring av en Sblock-matrise 43
6.12 Eksempel på development 44

6.13 Eksempel på LUT-konfigurasjon av en Sblock 44
6.14 Lagring av matrisen 45
6.15 Tilstandsdiagram for kommunikasjon 47

7.1 Ressursforbruk for Sblock-matriser 50
7.2 Vekstregelen brukt i test-matrisen 52
7.3 Manuellt utført celleutvikling 53
7.4 Simulering av matrisen i ModelSim 57
7.5 Startmatrise for funksjonell test 58
7.6 Sblock-matrisen etter funksjonell test 59
7.7 Typene til Sblock-matrisen etter post place and route-simulering 60

D.1 Eksempel på skriving til FIFO-buffer 82
D.2 Eksempel på lesing fra FIFO-buffer 82

Tabeller

2.1 Sannhetstabell for OR-celle 7
2.2 Sammenligning mellom gammel og ny FPGA 12
2.3 Ressursforbruk for en parallell Sblock-matrise 17

3.1 Sannhetstabell for en XOR-Sblock 22

4.1 Gammelt instruksjonssett 27

7.1 Syntese av ulike Sblock-matriser 51
7.2 Regler for den funksjonelle testen 52
7.3 Instruksjoner for funksjonell test 54
7.4 Kjøretid for hastighetstest i gammel implementasjon 56

Kapittel 1

Innledning

Dagens elektroniske kretser inneholder svært mange transistorer og kan bli meget komplekse. Disse kretsene er allikevel ikke i nærheten av å bli like komplekse som systemer vi finner i naturen [28]. I tillegg til å ha en avansert struktur og oppførsel tilpasser organismer i naturen seg omgivelsene sine, reproduserer seg selv og har mulighet for selvreparasjon ved skade [15]. Med utgangspunkt i disse egenskapene har man funnet inspirasjon til å lage nye metoder og design-ideer for programvare [2] og maskinvare [23] [22].

En av metodene som har blitt utviklet med grunnlag i naturen kalles Evolusjonær maskinvare (EHW). Denne teknikken tar i bruk evolusjonære algoritmer (EA), genetiske algoritmer (GA) og genetisk programmering (GP). Dette er metoder som igjen henter inspirasjon fra Darwins utviklingslære [5]. Tanken er at ved å la maskinvaren utvikle seg ved hjelp av en form for evolusjon fjernes de begrensningene designeren legger på løsningen. Dette kan gjøre at tilgjengelige tranistorer vil kunne benyttes på en bedre måte og nye egenskaper bli funnet. Utfordringen med å klare å utnytte den økende mengden transistorer på moderne kretser har blitt kalt designgapet [13].

I tradisjonelt kretsdesign bestemmes alle funksjonelle enheter og rutingen mellom dem under designet. Med denne designmetoden vil en liten krets være enkel å lage, mens en større krets krever mer konfigurasjon og mer komplekst design jo større den blir. Kunstig utvikling [12] er en av metodene som har blitt utviklet ved å bruke naturen som inspirasjon. Organismer i naturen starter som en celle og utvikler seg til et voksent individ gjennom blant annet celle-deling og -differensiering. Reglene for hvordan cellene skal utvikle og hvilken funksjon de skal ha ligger i genomet til organismen. Dette flytter kompleksitet fra konstruksjonen av genotypen over på utviklingsprosessen. Det er vanlig å la det være en 1-1 kobling mellom genotype (et bestemt sett med regler) og phenotype (et bestemt individ) hvor et bestemt genom alltid gir det samme individet.

Det har blitt eksperimentert med EHW hvor selve konstruksjonen av en krets utvikles med evolusjonære metoder [7]. Da endres konfigurasjonen til en FPGA [20] direkte av evolusjonsalgoritmen. En annen retningen innenfor evolusjonær maskinvare bruker naturens egne byggeblokker, celler, for å lage maskinvare. Istedenfor å beskrive hele

kontruksjonen og kjøre en EA på denne brukes det et sett med regler, et genom. Ved å bruke kunstig utvikling brukes et bestemt regelsett, en genotype til å utvikle en ferdig utviklet "organisme", phenotype, fra en enkelt celle.

For å kunne lage biologisk inspirert maskinvare er man avhengig av en egnet platform. En Field Programmable Gate Array (FPGA) er en rekonfigurerbar elektronisk krets som består av en matrise av rekonfigurerbare komponenter. Komponentene kan konfigureres blant annet til å utføre alle typer logiske operasjoner. Dette gjør FPGA til en egnet platform for EHW, men den har den en del mangler. Derfor har det blitt foreslått å lage en virtuell FPGA [8] som et abstraksjonsnivå over maskinvaren. Denne virtuelle FPGAen er en to-dimensjonell ikke-uniform cellulær automat. Cellene i matrisen kalles Sblock og har to egenskaper: type og verdi. Typen bestemmes ut fra genomet, mens verdien bestemmes av naboverdiene og cellens egen verdi og type.

På grunn av Sblock-matrisens innebygde parallellitet er det hensiktsmessig å implementere denne i maskinvare framfor å simulere i programvare. Dette har tidligere blitt gjort i en hovedoppgave 2003 og utvidet i 2005. Problemet med disse implementasjonene er at utviklingsprosessen er implementert i en sekvensiell co-prosessor på grunn av begrensning i maskinvare. Det har blitt kjøpt inn en FPGA med mer ressurser enn den gamle, og det er derfor ønskelig å parallellisere mest mulig av Sblock-matrisen. Denne oppgaven beskriver hvordan dette er gjort og hva som gjenstår.

Rapporten er bygd opp som følger:

- Kapittel 2 : Bakgrunn som trengs for å forstå oppgaven. Det gis en nærmere innføring i biologisk inspirert maskinvare og teknologien bak.
- Kapittel 3 : Hvordan Sblock-matrisen og development-modellen fungerer.
- Kapittel 4 : Funksjonalitet i tidligere implementasjon og krav-spesifikasjon for denne implementasjonen.
- Kapittel 5 : Maskinvare og programvare som ble brukt under utviklingen.
- Kapittel 6 : Forklaring om hvordan implementasjonen ble gjort og hvilke kompromisser som ble tatt.
- Kapittel 7 : Tester av systemet og sammenligning med tidligere arbeid.
- Kapittel 8 : En diskusjon av resultater, forslag til videre arbeid og konklusjon fra oppgaven.
- Vedlegg : Bakerst i rapporten følger det noen vedlegg som gjør det mulig å ta implementasjonen i bruk.

Kapittel 2

Bakgrunn

Dette kapittelet starter med å forklare hva biologisk inspirert maskinvare (EHW) er og setter de ulike grenene av EHW inn i et rammeverk (POE) [21]. Deretter forklares maskinvaren som ofte brukes i EHW. Til slutt nevnes noen relaterte prosjekter og hvordan EHW brukes i praksis.

2.1 Biologisk inspirert maskinvare

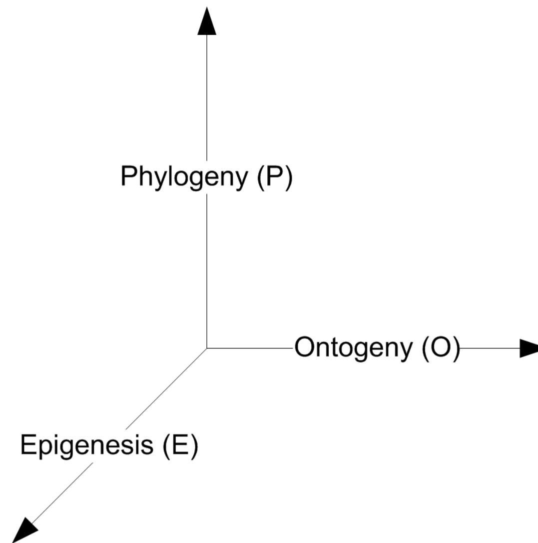
Med biologisk inpirert maskinvare menes elektroniske kretser som er designet med designmetoder og ideer fra naturen eller kretser som er designet med tradisjonelle metoder, men som har funksjonalitet inspirert fra naturen [7].

Siden begrepet er såpass vidt kan man dele inn biologisk inspirert maskinvare i tre akser [21]. Disse aksene er vist i figur 2.1. Phylogeny beskriver ulik grad av evolusjon, ontonogy tar for seg kunstig utvikling og epigenesis handler om kunstig intelligens og læring. De tre aksene vil bli forklart under med et fokus på ontogeny. Det er denne aksen som er mest sentral for oppgaven siden den tar for seg kunstig utvikling. Siden Sblock-matrisen som er implementert skal brukes til evolusjon av genom, vil phylogeny også bli forklart relativt detaljert.

POE-rammeverket gjør det mulig å klassifisere forskjellige biologisk inspirerte maskinvaresystemer. Enkle systemer er ofte et stykke ut på én av aksene, men ikke på de andre to. Svært komplekse, selvlærende systemer bygd opp av celler, under stadig utvikling og evolusjon er ytterst på alle tre aksene.

2.1.1 Phylogeny

Phylogeny beskriver hvordan systemet utvikler seg fra generasjon til generasjon. I den enkleste enden av skalaen finner man systemer som ikke har noen form for utvikling,



Figur 2.1: POE-rammeverket

men designes én gang. Den mest avanserte formen for phylogeny er også den som ligner mest på naturen, nemlig naturlig utvelgelse og evolusjon. Ved hjelp av en evolusjonær algoritme får det systemet som klarer å løse oppgaven best leve videre.

En genetisk algoritme (GA) [10] er en type evolusjonær algoritme mye brukt for å finne et program eller maskinvare som løser et bestemt problem. I en GA opererer man med ulike generasjoner med individer. Et individ er en bitsekvens, program-del eller lignende som er bygget opp av lovlige tegn. En generasjon er en samling av slike individer som lever samtidig i algoritmen og dermed sammenlignes med hverandre. Sammenligningen foregår ved utregning av en fitness-verdi. Dette er et mål på hvor godt et individ klarer å utføre en bestemt oppgave. Når alle individene har fått en fitness-verdi, sammenlignes disse og individene blir valg ut ved hjelp av en utvelgelsesmetode, for eksempel basert på at individene har en viss sannsynlighet for å bli med å danne neste generasjon. Når en ny generasjon skal lages kan ulike teknikker brukes. Noen ganger får en viss andel av de utvalgte individene fra forrige generasjon bli med uendret. Crossover blander arvestoffet til individer fra forrige generasjon og skaper et “barn”. Mutasjon brukes for å innføre nytt arvestoff i populasjonen, da byttes deler av arvestoffet ut eller nytt arvestoff legges til ett av individene. GA er en iterativ prosess og med utvelgelse, krysning og mutasjon blir nye generasjoner til.

En instans av en GA begynner med en start-populasjon som er bygget opp av lovlige tegn. Denne start-populasjonen er som regel bestemt tilfeldig, men kan også inneholde hint om løsninger gitt av designeren.

Eksempel på pseudokode til en GA er gitt i figur 2.2.

GA garanterer ingen løsning, og finner algoritmen en løsning er det ikke sikkert at dette er den beste. Allikevel har metoden vist seg å gi gode resultater i blant annet

```
1   g = 0
2   Initialize P(g)
3   Evaluate P(g)
4   while not finished
5           g = g + 1
6           Select P(g) from P(g-1)
7           Crossover P(g)
8           Mutate P(g)
9           Evaluate P(g)
```

Figur 2.2: Genetisk algoritme

optimalisering, automatisk programmering, maskinlæring, økonomi og flere områder [7].

2.1.2 Ontogeny

Dette begrepet tar for seg utviklingen av en enkelt organisme. I den ene enden av skalaen har man systemer som er beskrevet i detalj på forhånd. I den andre, mer avanserte enden av skalaen har man organismer som utvikler seg fra én celle til en hel organisme ved hjelp av prosesser som beskrives senere. Denne organismen bygges opp ved hjelp av regler i form av et genom istedenfor en komplett beskrivelse av det ferdige systemet, slik som i naturen hvor oppbygningen av en organisme beskrives ved hjelp av DNA. Dette er ikke en beskrivelse av oppførselen til organismen, men heller en oppskrift på hvordan denne skal settes sammen.

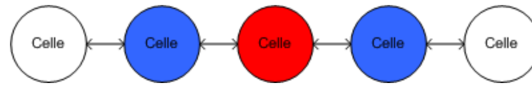
Ved å innføre denne tankegangen i maskinvarekonstruksjon har man skapt kunstig utvikling [12]. Dette er i praksis en prosess hvor et sett med regler (genotype) brukes til å konstruere en ferdig utviklet "organisme" (phenotype). Siden regelsettet er av konstant størrelse og likt for alle cellene gjør dette at organismen skalerer godt. En større organisme trenger ikke nødvendigvis en større genotype, slik tilfellet er ved en direkte mapping mellom genotype og phenotype i tradisjonelt design. Man lar det allikevel vanligvis være en 1-1 mappping mellom genotype og phenotype, slik en bestemt genotype alltid gir den samme phenotypen.

Denne måten å konstruere maskinvare på flytter kompleksiteten fra genotypen over på mapping-prosessen som bruker genotypen til å bygge opp phenotypen.

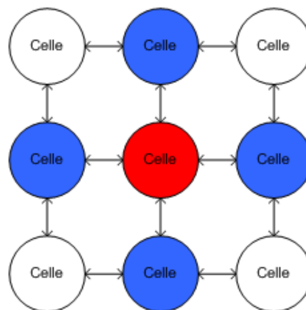
Cellulære automater

I tradisjonell maskinvare ser man ofte på funksjonaliteten til en enhet som verdien på en eller flere ledere ut fra enheten, gitt en bestemt inn-verdi. For at inn-verdien skal ha effekt på ut-verdien må disse være koblet sammen gjennom logiske enheter eller ledere. Når man lager maskinvare med evolusjons-algoritmer er det ingen garanti for dette. For å unngå dette problemet kan man velge å lage maskinvaren som en cellulær automat.

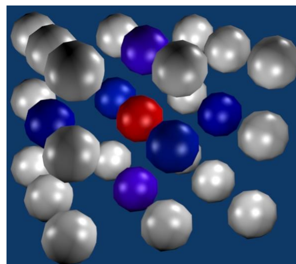
En cellulær automat (CA) består av en n-dimensjonal struktur med celler. Det er ingen global kommunikasjon utover en klokke, så cellene kan bare kommunisere med nabo-cellen sine. Eksempler på kommunikasjon fra en celle til naboene i cellulære automater med 1, 2 og 3 dimensjoner er gitt i figur 2.3. Alle celler har en bestemt tilstand fra et sett med gyldige tilstander som blir oppdatert hvert tidssteg i henhold til et sett regler. Regelsettet kan være felles for alle cellene, eller det kan være flere regelsett.



(a) 1-dimensjonal cellulær automat



(b) 2-dimensjonal cellulær automat



(c) 3-dimensjonal cellulær automat

Figur 2.3: Kommunikasjon i en cellulær automat: Grønn celle er sentrum, blå er naboene dens.

En CA har fire viktige egenskaper [3]:

- Geometri - Hvordan cellene er plassert, hvor mange dimensjoner CAen har
- Regelsettet - Hvordan cellene skal oppdatere tilstanden sin
- Lovlige tilstander - Hvilke tilstander som er tillatt i CAen

Senter	Venstre	Høyre	Ny senter
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabell 2.1: Sannhetstabell for OR-celle

- Naboskapet til cellene - Hvilke celler som er koblet sammen og om CAen “wrapper” i endene

I figur 2.4 er oppførselen til en en-dimensjonal cellulær automat vist grafisk. Den eneste regelen som brukes er OR, det vil si at hver celle sjekker begge nabo-celler og endrer sin egen verdi til 1 hvis en eller flere av naboene har verdien 1. Cellene i eksempelet har to lovlige tilstander: 0 og 1. Cellene er koblet sammen med naboene, og de to cellene ytterst er også koblet sammen. Sannhetstabellen til hver celle med naboer er gitt i tabell 2.1.

Figur 2.4(a) viser start-tilstanden til den cellulære automaten, her er en av cellenes tilstand satt til 1 manuellt. Figur 2.4(b) viser CA’en etter et tidssteg. Her har naboene til den cellen som startet med tilstand 1 endret seg. Figur 2.4(c) viser CA’en etter to tidssteg. Ved neste tidssteg vil CA’en endre seg tilbake til 2.4(b) og skifte fram og tilbake mellom mønsteret i figur 2.4(b) og 2.4(c).

Celler

I naturen utvikler en organisme seg fra én befruktet celle til en mangecellet organisme ved hjelp av flere prosesser. Disse er ikke adskilt fra hverandre, men kan godt være igang samtidig. Prosessene er igang helt til organismen dør og er [28]:

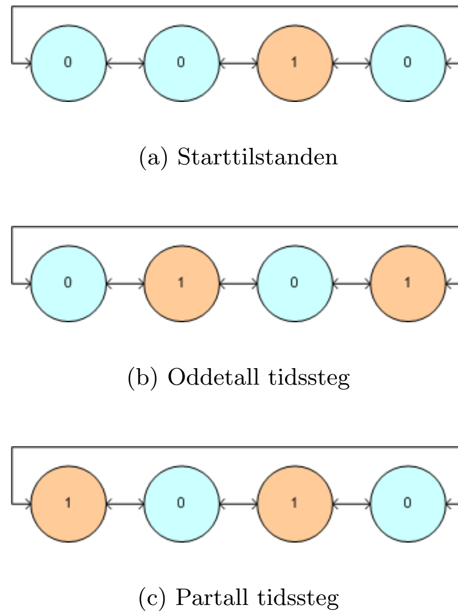
- Gruppering

  Celler organiseres i mønstre i forhold til celletype. Når like celler samles på denne måten dannes blant annet organer.

- Morphogenesis

  Celler endrer form og påvirker på denne måten også andre celler. Celler som vokser dytter andre celler unna, celler endrer form og utstrekning.

- Differensiering



(a) Starttilstanden

(b) Oddetall tidssteg

(c) Partall tidssteg

Figur 2.4: Eksempel på en cellulær automat

I denne fasen får cellene sin funksjon og struktur. Cellen utvikler seg til å bli blodceller som transporterer oksygen, nerveceller som overfører signaler eller andre spesialiseringer.

- Vekst

  Ved celledeling og celledød kan en nesten ferdig organisme lage bestemte strukturer som fingre. Når alle organene er ferdig utviklet og fungerende er det denne prosessen som lar organismen øke størrelsen sin.

Maskinvare som klarer å implementere flere av disse eller alle prosessene vil ligge langt ute på ontogeny-aksen.

2.1.3 Epigenesis

Denne aksen beskriver læring. Den enkleste typen system innholder all informasjon det trenger fra det blir bygd. Tradisjonelle kretser hvor en viss inn-verdi alltid gir samme ut-verdi faller i denne kategorien.

I den andre enden av skalaen finner man selvlærende systemer. Dette kan for eksempel være nevrale nett som evaluerer seg selv og trener seg selv opp under hele levetiden [14].

2.2 Maskinvare

For å lage biologisk inspirert maskinvare er man avhengig av å ha maskinvare som passer til dette formålet. Blant ulike teknologier er FPGA en mye brukt platform for dette. Rekonfigurerbarheten og den innebygde parallelliteten i en FPGA gjør den til et godt verktøy for EHW.

2.2.1 FPGA

En Field Programmable Gate Array (FPGA) er en rekonfigurerbar maskinvareenhet. Sammenlignet med en mikroprosessor kan den designes for å løse mer spesifikke problemer raskere, er naturlig parallell og kan støtte fler funksjoner enn det begrensede instruksjonssettet til en mikroprosessor [19]. Med tanke på hurtighet og fleksibilitet er en FPGA mellom mikroprosessorer og en Application Spesific Integrated Circuit (ASIC) som er maskinvare designet for å utføre én bestemt oppgave. I forhold til ASIC er en FPGA svært mye mer fleksibel og nesten like rask [19].

FPGA ble opprinnelig laget for å forenkle prototyping av blant annet ASIC. Etter å ha testet ut forskjellig design i en FPGA lager man den ferdige ASIC som settes i produksjon. Etter hvert har FPGA allikevel fått flere arbeidsoppgaver, blant annet som co-prosessor [4]. Ved å laste inn en konfigurasjon spesialdesignet for å løse en oppgave kan en del oppgaver løses mye raskere enn på en mikroprosessor [4].

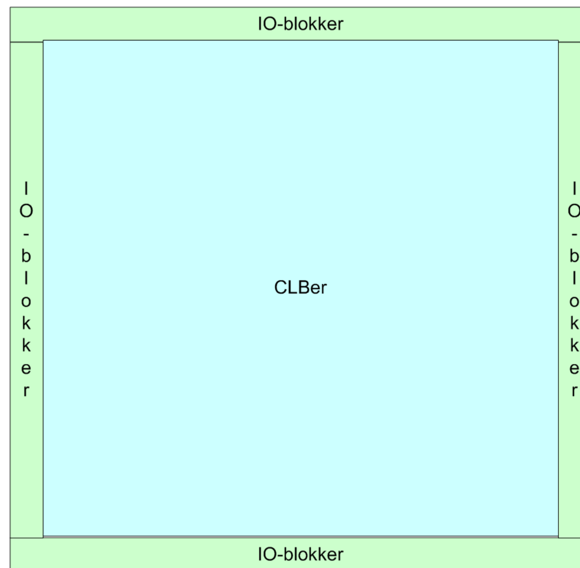
FPGA er et begrep som omfatter en del ulike arkitekturer. Noen grunnleggende egenskaper og enheter for mange FPGAer vil bli beskrevet under.

Oppbygning

En FPGA er satt sammen av mange "byggeklosser" som kan konfigureres. Disse byggeklossene er vanligvis arrangert i en matrise som vist i figur 2.5. En vanlig byggekloss i en FPGA er Configurable Logic Block (CLB).

En figur som viser en typisk CLB er gitt i figur 2.6 Denne komponenten kan konfigureres til å utføre alle typer logiske operasjoner ved hjelp av en lookup-table (LUT). CLBen består også av en flip-flop (DFF) som kan ta vare på verdier og logikk for mente. Som vist i figuren er det en MUX ved utgangen til CLBen som velger enten resultatet fra LUT eller flip-flopen.

En FPGA er ikke brukbar til mye hvis man ikke kan kommunisere med den. Derfor er som regel alle kantene på brikken koblet til fysiske pinner som igjen styres av Input/Output-blokker. IOB er plassert langs alle kantene på de fleste FPGAer. Hver blokk er knyttet til en fysisk IO-pin og kan fungere som input, output eller begge deler.



Figur 2.5: Oppbygning av en typisk FPGA

Arkitektur

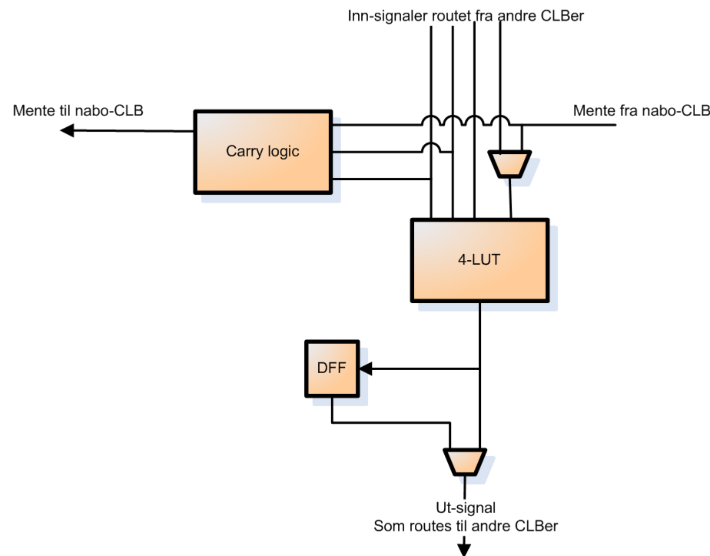
Man pleier å skille mellom homogene og heterogene FPGAer. I en homogen FPGA er alle de funksjonelle enhetene brikken er bygget opp av like, med unntak av IO-blokkene. En heterogen FPGA har ofte spesialiserte enheter på brikken som er optimaliserte for bestemte oppgaver. Vanlige komponenter som sitter på en heterogen FPGA kan være minneblokker, gange-enheter, eller hele mikroprosessorer. Disse enhetene utfører ofte oppgaver som er vanskelige å implementere effektivt i vanlige CLBer.

Sammenkoblingen mellom de ulike enhetene i en FPGA kan gjøres på flere forskjellige måter. To vanlige metoder er segmentering og hierarki. I segmentert routing er all kommunikasjon i korte ledere, for å få kommunikasjon over lange strekninger på brikken kobles flere slike ledere sammen. I hierarkisk routing er det flere nivåer med ledere, noen korte for lokal kommunikasjon og noen lange for kommunikasjon over lengre avstander.

Routing på en FPGA er vanligvis 2-dimensjonal, men det finnes også brikker som bare bruker en mer 1-dimensjonal type routing [9]. En vanlig 2-dimensjonal routinteknikk kalles island-style routing, og er bygget opp som vist i figur 2.7.

Konfigurasjon

En FPGA konfigureres ved at en bitstrøm lastes ned på den. Denne bitstrømmen er konfigurasjonsdata for alle enheter og koblingen mellom dem og har som regel blitt syntetisert fra Very High Speed Integrated Circuit Hardware Description Language (VHDL) eller lignende kode.



Figur 2.6: Standard CLB

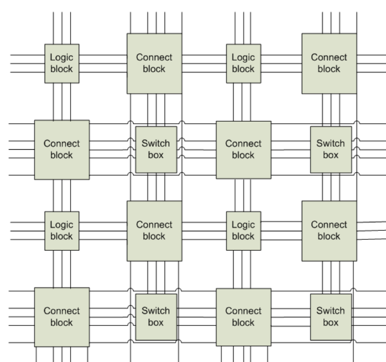
Det finnes flere måter å laste inn bitstrømen til brikken. Blant annet kan man ha konfigurasjonen lagret på en Programmable Read Only Memory (PROM) [31] eller på en datamaskin og overføre den ved hjelp av boundry-scan (JTAG) [11].

Konfigurasjonen av en FPGA har tre steg: Sletting av tidligere konfigurasjon, opplasting av ny konfigurasjon og oppstart. Noen FPGAer har også mulighet for å foreta en delvis rekonfigurasjon av brikken. Da laster man inn konfigurasjon på et bestemt område mens resten av FPGAen kjører.

2.2.2 Xilinx Virtex-II Pro

FPGAer har som alle kretser blitt mer kompakte og mer avanserte. Dette avsnittet beskriver hvordan FPGAen som skal brukes i dette prosjektet skiller seg fra en standard FPGA. Som sammenligning brukes en Xilinx Virtex-E som tidligere har blitt brukt ved IDI, blant annet til den tidligere implementasjonen som senere blir beskrevet.

I denne oppgaven skal en Xilinx Virtex-II Pro (XC2VP70) benyttes. Den største forskjellen mellom Virtex-II og Virtex-II Pro er introduksjonen av RocketIO og PowerPC på brikken. I forhold til Virtex-E er Virtex-II Pro mye større: 77448 logikk-celler i forhold til 27648. I tabell 2.2 er FPGAen i dette prosjektet sammenlignet med den som tidligere var i bruk.



Figur 2.7: FPGA island-style routing

	Virtex-E (modellnavn)	Virtex-II Pro (XC2VP70)
RocketIO Tranciever Blocks	0	16/20
PowerPC Processor Blocks	0	2
Logic Cells [a]	27 648	74 448
Max Distr RAM (Kb)	48	1034
Multiplier Blocks	Ukjent (0?)	328
BRAM Blocks	96	328
Max BRAM (Kb)	48	5 904
User I/O Pads	660	996

[a]XC2VP70: 4 Slices/CLB, 2 LC/Slice;
XCV1000E: 2 Slices/CLB, 2 LC/Slice

Tabell 2.2: Sammenligning mellom gammel og ny FPGA

Oppbygning

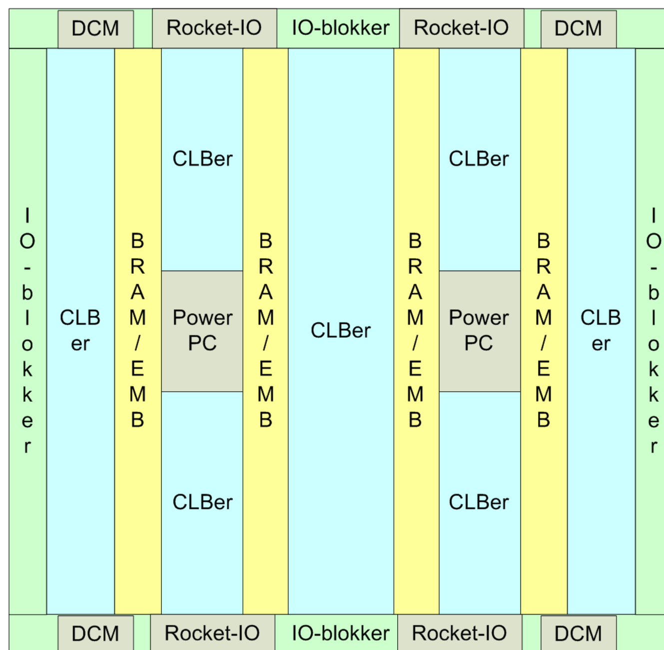
Oppbygningen av en Virtex-II Pro vises i figur 2.8. I forhold til en standard FPGA har den en del utvidelser og annerledes løsninger. De ulike komponentene blir gått gjennom relativt kortfattet her. For mer detaljert informasjon, se databladet [30].

RocketIO

Disse enhetene er spesielle IO-enheter som støtter mange ulike kommunikasjons-modi, blant andre FibreChannel, Gigabit Ethernet og Infiniband. RocketIO kan blant annet brukes for å koble to eller flere FPGAer sammen. Fordelen med RocketIO er at overføringen bruker en annen klokke enn resten av FPGAen og støtter høye overføringshastigheter, over 3Gb/s.

PowerPC 405 Processor Block

Den FPGA som skal brukes har 2 PowerPC-enheter på brikken. Dette er komplette mikroprosessorer med minnekontrollere, klokke og et CPU-FPGA-grensesnitt. Proses-

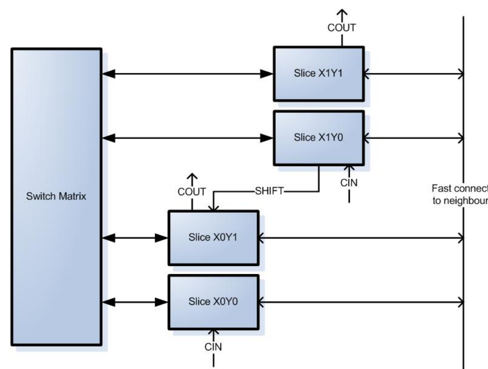


Figur 2.8: Oppbygning av en Virtex-II FPGA

soren kjører på 300+ MHz. Minnet er separert i et instruksjonsminne og dataminne, samtidig som prosessoren kan adressere BRAM-minnet i FPGAen. Det er lesing og skriving av BRAM og en viss kommunikasjon med FPGAen som er mest relevant for dette prosjektet.

Configurable Logic Blocks (CLB)

På Virtex-II Pro består hver CLB av en Switch Matrix og 4 Slices. Disse er koblet sammen som vist i figur 2.9.



Figur 2.9: Oppbygning av en Virtex 2 Pro CLB

Hver slice tilsvarer en utvidet CLB i en standard FPGA som vist i figur 2.6 og består av

4 funksjonsgeneratorer. Det er disse funksjonsgeneratorene som blant annet kan konfigureres til å fungere som en LUT. De ulike konfigurasjonene hver funksjonsgenerator kan ha er:

- 16 bits RAM
- 16 bits Shift register
- 16 bits LUT

I tillegg til funksjonsgeneratorer har hver slice mente-logikk, aritmetisk-logiske porter, multipleksere og to registere/latcher.

Block SelectRAM (BRAM)

Dette er dedikerte minne-enheter. Brikken som skal brukes har 328 blokker med BRAM som hver holder 18 Kb. Minnet er er synkront, dvs at det kan leses fra og skrives til samtidig. Blokkene er konfigurerbare, og man kan blant annet selv bestemme bredden på ordene som lagres og rekkefølgen av lesing og skriving hver klokke.

Klokke

Virtex-II Pro har støtte for å ta inn 16 klokkesignal. Disse kan brukes direkte eller gjennom en Digital Clock Manager (DCM). Denne oppgaven bruker bare to vanlige klokkesignaler.

Routing

Routingen på FPGAen er hierarkisk og er delt inn i følgende kategorier:

- Lange linjer: Sprer signaler over hele brikken, horisontalt og vertikalt
- Hex-linjer: Kobler hver 3. eller hver 6. blokk sammen, horisontalt og vertikalt
- Dobbelt-linjer: Kobler til første eller andre blokk, horisontalt og vertikalt
- Direkte-linjer: Kobling til alle naboer, horisontalt, vertikalt og skrått
- Rask-kobling: Linjene internt i CLB mellom LUTene

I tillegg til disse ledningene finnes det en del dedikerte nett f.eks. klokke, rad-busser og mente-kjede.

Den hierarkiske oppbygningen tillater både global og lokal kommunikasjon. Dette er svært nyttig for dette prosjektet hvor styringen kommer til å være sentral mens utregninger og kommunikasjon kommer til å være lokal og parallell.

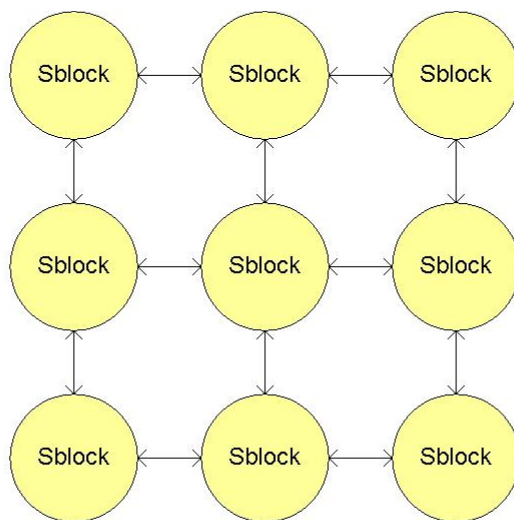
Embedded Multiplier Blocks (EMB)

FPGAen har innebygde gange-enheter. Disse multipliserer 18-bit med 18-bit og er bedre med tanke på hastighet og strøm i forhold til en gange-enhet implementert vha. slicer.

2.3 FPGA EHW

Selv om en FPGA har egenskaper som gjør den egnet for EHW er det allikevel en del problemer med den. En FPGA er avhengig av å bli konfigurert med en gyldig konfigurasjon og hvis man jobber direkte med konfigurasjons-bitstrømmen er det krevende å validere denne. En ugyldig konfigurasjon kan i verste fall skade maskinvaren og gjøre FPGAen ubrukelig. Å generere en lovlig konfigurasjon med en algoritme som manipulerer konfigurasjonsbitstrømen direkte med en tilfeldig-algoritme er lite sannsynlig. Dette gjør at man enten krever en evolusjonsalgoritme som tar høyde for maskinvarespesifikke problemer eller forkaster mange av de genererte konfigurasjonene.

En foreslått løsning på problemene med å drive evolusjon på en FPGA er å lage en virtuell EHW FPGA, en Sblock. En Sblock er en abstraksjon vekk fra den underliggende maskinvaren og er en utvidet cellulær automat. En matrise med Sblocker er vist i figur 2.10. Hver Sblock har en tilstand '0' eller '1' og en type. Typen til en Sblock er en funksjon som tar Sblockens egen tilstand og naboenes tilstand som inn-verdi og gir Sblockens nye tilstand som ut-verdi. Sblockens type er gitt fra et sett med regler som kalles Sblockens genom. Det er størrelsen på matrisen, genomet og start-tilstanden til alle Sblockene som konfigureres, deretter endres typene ved kunstig utvikling og tilstandene oppdateres fortløpende. Detaljer om oppbygningen og funksjonen til en Sblock er gitt i kapittel 3.



Figur 2.10: Eksempel på en sblock-matrise

2.3.1 Relatert arbeid

Det finnes andre som har laget biologisk inspirert maskinvare. For eksempel har det blitt laget en cellulær datamaskin kalt "Firefly" [23] som er et system som bruker evolusjon for å oppnå sin funksjonalitet. Dette systemet er implementert på en FPGA og er

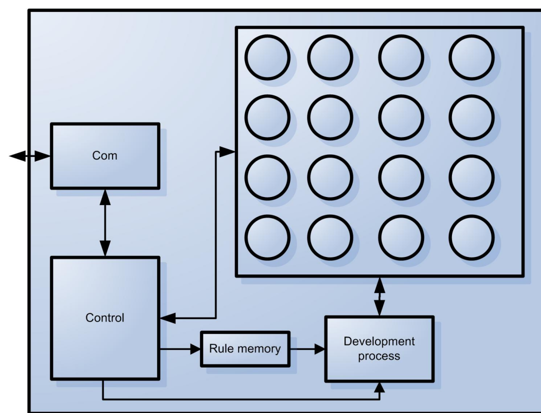
fullstendig autonomt med evolusjonsprosessen ombord på kortet.

“Biowatch” [24] er en biologisk inspirert klokke som har selvreparerende egenskaper. Dette systemet er implementert på det rekonfigurerbare materialet BioWall [25] som er bygget opp av mange FPGA og lysdioder.

2.3.2 Tidligere arbeid

Det er allerede laget en realisering av en Sblock-matrise [6] som også har blitt forbedret [1]. Funksjonaliteten til denne implementasjonen er beskrevet i kapittel 4. Da systemet ble laget måtte man ha en avveining av hvor stor del av Sblocken som skulle kunne kjøre parallellt. Jo mer som er parallellt, jo mer maskinvare krever det. På grunn av begrensningene til FPGAen som ble brukt ble det bestemt å legge genomet og utviklingsprosessen sentralt som en co-prosessor, mens Sblockens funksjonalitet ble spredt som en Sblock-matrise. En forenklet figur av systemet er gitt i figur 2.11.

Implementasjonen laget av Djupdal i 2003 støttet konfigurajon av matrisen og genom, kjøring av Sblockene, utviklingsprosess og utlesing av resultater. Utvidelsen laget av Aamodt i 2005 flyttet evalueringen av genomet, altså utregning av fitness-verdien inn på FPGAen og laget støtte for bedre utlesning av blant annet brukte regler.



Figur 2.11: Gammel Sblock-matrise

FPGAer har, som nevnt i avsnitt 2.2.1, blitt større og mer avanserte siden disse implementasjonene. Dette gjør det mulig å parallellisere enda mer av Sblock-matrisen.

For en fullstendig parallell Sblock-matrise kreves ressursene vist i tabell 2.3. Her er den “gamle” FPGAen sammenlignet med den som skal brukes i denne oppgaven. Tallene i tabellen er optimistiske og baserte på en delvis implementasjon av en parallell Sblock-matrise blant annet uten logikk for lagring av tilstand og brukte regler. En ferdig matrise vil sannsynligvis kreve mer logikk og ha plass til en litt mindre matrise. Tabellen viser også at man kan fjerne den største begrensningen ved å lagre regelsettet sentralt. Mer om dette i kapittel 6.1.1.

	Enkel Sblock	Virtex-E	Virtex-II Pro
Blockram	14080 bit	393216 bit	5904 Kb
LUT	114	24576	66176
FF	92	24576	66176
Sblocker mht BRAM	-	27,9	419,3
Sblocker mht LUT	-	215,5	580,5
Sblocker mht FF	-	261,4	704
Største matrise	-	5x5	20x20
Største matrise m/sentrale regler	-	14x14	24x24

Tabell 2.3: Ressursforbruk for en parallell Sblock-matrise

Kapittel 3

Development model

For å kunne forske på evolusjonær maskinvare er man avhengig av å ha en brukbar plattform til dette. Det er ikke sannsynlig at en slik maskinvareplattform vil bli tilgjengelig i nærmeste framtid, noe som har motivert utviklingen av en virtuell FPGA som kan realiseres på eksisterende maskinvare.

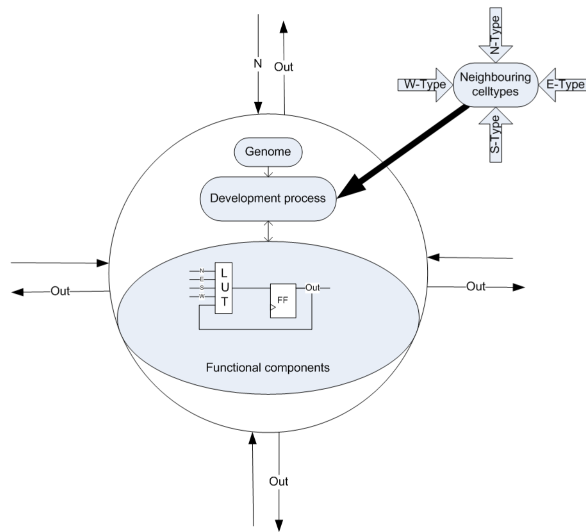
Sblocker er en meget sentral del av denne oppgaven hvor det skal implementeres en matrise med disse i maskinvare. For å gjøre dette er det viktig å få på det rene hva en Sblock er og hvilke egenskaper en slik matrise skal ha. Dette beskrives i detalj i dette kapittelet.

3.1 Beskrivelse av en sblock

Som nevnt i avsnitt 2.3 er en Sblock en abstraksjon bort fra underliggende maskinvare. Sblock ble introdusert i [26] som en arkitektur designet for og egnet for evolusjonær utvikling. Hver Sblock består av en enkel logikk/minne-enhet og routing-ressurser i tillegg til en utviklingsdel. En Sblock er koblet til sine fire naboer med inn- og utporter. I tillegg får Sblocken inn sin egen ut-verdi som en inn-verdi. Komponentene i en Sblock er vist i figur 3.1

Routingen fungerer slik at en inn-port kun kan være koblet til en ut-port og motsatt. Dette gjør at ingen ulovlige og farlige konfigurasjoner for maskinvaren er mulige. Hver Sblock er ansvarlig for å prosessere inn-signalene og legge verdier på ut-porten sin. Siden hver Sblock kommuniserer med sine nærmeste naboer gir dette en symmetrisk og skalerbar struktur [26].

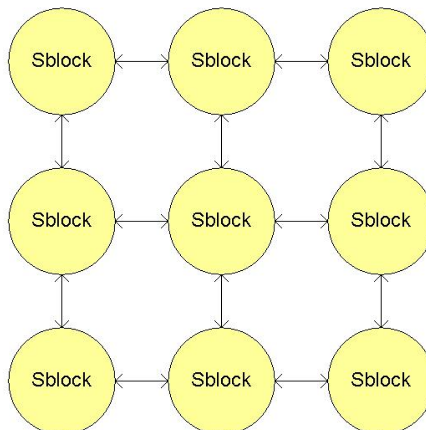
I tradisjonelle FPGAer konfigureres de logiske blokkene og routing separat. Spesielt konfigurasjonen for routing krever svært stor plass selv om mange av ledningene ikke brukes. I Sblocken skilles det ikke mellom logikk, minne og routingkonfigurasjon, all data om dette ligger i konfigurasjonsstrømmen til hver Sblock.



Figur 3.1: Komponentene i den cellulære development modellen [27]

3.2 Matrisen

Sblocker arrangeres logisk i en matrise lignende det CLBer gjør i en FPGA. En tegning av en typisk Sblock-matrise vises i figur 3.2. Det er kommunikasjon mellom alle naboene og hver Sblock har en konfigurasjon som forteller den om den skal oppføre seg som logisk enhet, minne eller router.



Figur 3.2: Eksempel på en matrise av Sblocker

3.3 Regler og rekonfigurering

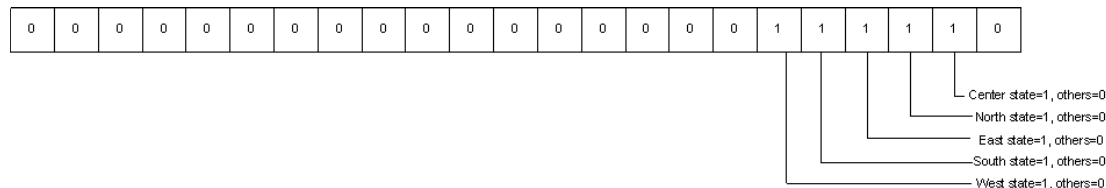
Som beskrevet i avsnitt 2.1.2 er det ønskelig å flytte kompleksitet fra genotypen til utviklingsprosessen. Dette gjøres i Sblocken ved rekonfigurerbarhet. Målet er at Sblock-matrisen skal ha en oppførsel lignende en ekte celle som beskrevet i 2.1.2. Dette er løst ved at hver Sblock har et genom som beskriver oppbygningen av organismen. Genomet består av to typer regler: endring og vekst [26].

Vekst-regler krever at organismen har plass til å gro. Dvs at cellen det skal gros til må være tom. I tillegg må kravene til naboer og inn-verdier i regelen være oppfylt. Hvis dette er tilstede vil den tomme Sblocken bli fylt med en kopi fra en av naboene som beskrevet i regelen.

Endrings-reglene krever at cellen ikke er tom og at tilstanden og typene til naboene er i henhold til regelen. En endrings-regel kan få cellen til å endre type, dø eller få en annen celle til å gro inn over seg.

Til enhver tid kan kun én av reglene i genomet aktiveres i hver celle. Det betyr at hvis en celle har et genom med 32 regler og regel nummer 4 aktiveres vil ikke de resterende 28 sjekkes.

En Sblocks type er enten bestemt manuelt eller er et resultat av utviklingsprosessen. Sblockens tilstand er igjen bestemt av typen til Sblocken i kombinasjon med naboenes tilstander og Sblockens tidligere tilstand. Dette kan implementeres med en 5-inputs LUT og en D-vippe. Konfigurasjonen i lookup-tabellen er gitt av Sblockens type. Hvis Sblocken har en type hvor tilstanden en 1 hvis nøyaktig én av inputene til LUTen er 1 og blocken selv har tilstanden 0, vil konfigurasjonsstrengen være som figur 3.3 og sannhetstabellen for tilstanden som tabell 3.1.



Figur 3.3: LUT-konfigurasjon for en Sblock

3.4 I/O

En mulig I/O-metode kunne vært å koble alle kant-nodene til I/O-blokker på FPGAen. Ved å drive inn-verdien på disse blokkene ble regler i Sblockene på kanten aktivert, og systemet endret seg. Som beskrevet i avsnitt 2.1.2 kan det bli vanskelig å se noen resultater i ut-verdien på I/O-blokkene. Dette løses ved å se på Sblock-matrisen som en Cellulær automat. Som forklart i avsnitt 2.1.2 betyr dette at systemets ut-verdi ses

CNØSV	Data	CNØSV	Data	CNØSV	Data
00000	0	01000	1	11000	0
00001	1	01001	0	11001	0
00010	1	01010	0	11010	0
00011	0	01011	0	11011	0
00100	1	01100	0	11100	0
00101	0	01101	0	11101	0
00110	0	01110	0	11110	0
00111	0	01111	0	11111	0

Tabell 3.1: Sannhetstabell for en XOR-Sblock

på som tilstanden til hele systemet. Det betyr at enhver tilstandsendring i en Sblock i matrisen er en endring i ut-verdien.

Istedenfor å koble kant-nodene i Sblock-matrisen til I/O-blokker er det nå routing mellom de ytterste nodene. Det vil si at i hver rad er den siste Sblocken koblet til den første i samme rad. Tilsvarende er det i hver kolonne. Dette gjør matrisen enda mer symmetrisk, og plassering i forhold til "kanten" er irrelevant.

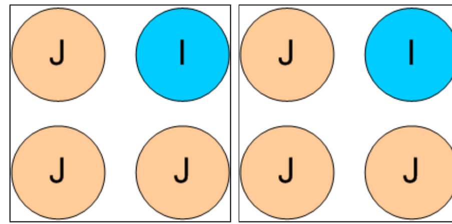
Som inn-verdi har man valgt å sette en eller flere Sblocker manuellt. Siden Sblocken er helt symmetrisk har det ingenting å si hvor i matrisen disse Sblockene er, kun plassering i forhold til de andre Sblockene.

3.5 Utvikling

Ettersom Sblock-matrisen aktiverer regler kan det utvikle seg en flercellet organisme. Uansett vil man ende opp i én av tre tilstander [26].

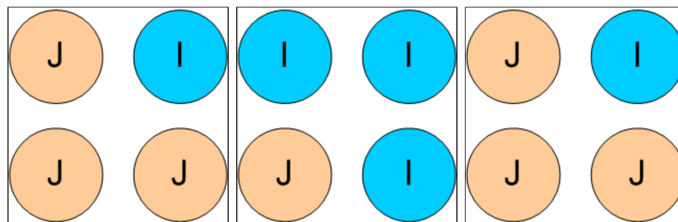
- Utviklingen kan tilsynelatende stoppe opp. Dette skjer hvis ingen celler kan aktivere regler. Da vil celle-typene være konstante og vi sier at organismen er stabil. En stabil Sblock-matrise vises i figur 3.4. Det er viktig å merke at utviklingsprosessen fortsatt er aktiv. Hvis en ytre påvirkning endrer matrisen, ved for eksempel å drepe en celle, kan reglene starte å aktiveres igjen.
- Utviklingen går i en endelig løkke. Hvis matrisen ved et tidspunkt kommer i nøyaktig samme tilstand som den har vært i tidligere er matrisen i en løkke. Siden utviklingen er deterministisk vil neste tilstand bli den samme som tidligere og mønsteret gjentar seg. Dette vises grafisk i figur 3.5
- Utviklingen kommer i en uendelig løkke. Dette kan skje hvis organismen får vokse fritt. Siden matrisen er begrenset av maskinvare kan dette aldri bli tilfellet her.

Når Sblocken har nådd stabilitet eller en endelig løkke har organismen modnet.



(a) Devstep 0 (b) Devstep 1

Figur 3.4: Stabil Sblockmatrise



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

Figur 3.5: Endelig developmentløkke

Kapittel 4

Implementasjonens funksjonalitet

Dette kapittelet beskriver funksjonaliteten introdusert i [6] og [1]. Et krav til arbeidet i denne oppgaven er å bevare funksjonaliteten og mulighetene fra det tidligere arbeidet. Kapittelet starter derfor med en beskrivelse av instruksjonssettet og fortsetter med funksjonaliteten til co-prosessoren som ble utviklet i de tidligere prosjektene. Til slutt gis et eksempel på programkoden til et eksperiment.

4.1 Instruksjonssett

I implementasjonen til [6] er det co-prosessoren som tar seg av all kontroll av Sblock-matrisen og kommunikasjon med datamaskinen. Dette gjøres ved at instruksjoner sendes gjennom PCI-bussen. En komplett liste over instruksjonene fra [6] og [1] er gitt i tabell 4.1. Det er ønskelig å bevare all funksjonalitet fra dette systemet. Det må med andre ord være mulig å:

- Sette opp Sblock-matrisen
- Sette tilstanden til en Sblock
- Sette typen til en Sblock
- Nullstille tilstanden til alle Sblocker
- Kjøre et visst antall tilstands-steg
- Kjøre et utviklings-steg
- Lese ut tilstanden til en Sblock
- Lese ut tilstanden til alle Sblocker
- Lese ut typen til én Sblock

- Lese ut typen til alle Sblocker
- Konfigurere genomet
- Definere betydningen av genomet (mapping til LUT)
- Kjøre en fitnessfunksjon
- Lese ut fitness
- Lese ut antall kjørte utviklings-steg
- Lese ut hvilke regler som er kjørt på hvilke blokker

Under implementasjon av det nye systemet viste det seg allikevel å være hensiktsmessig å fjerne noen av instruksjonene, endre betydning av noen instruksjoner og legge til et par nye. Instruksjoner som er fjernet i den nåværende løsningen er merket med et utropstegn, instruksjoner som er endret er merket med spørsmålstegn, mens nyinnførte instruksjoner har en stjerne ved navnet sitt.

Noen av instruksjonene som er fjernet eller har endret funksjon har blitt forandret på grunn av tidspress. Det er fortsatt ønskelig at disse skal være med eller fungere som tidligere. Dette diskuteres mer i kapittel 8.3.

4.2 Funksjonalitet

Det er ønskelig at den nye implementasjonen skal ha minst like god funksjonalitet som det tidligere systemet. I tillegg til å ha tilsvarende instruksjonssett er det noen fler faktorer som er viktige. Disse vil bli diskutert her.

Den nåværende implementasjonen støtter fra 2^2 til 32^2 antall Sblocker. Den nye implementasjonen skal helst ikke støtte færre, eller hvertfall ha mulighet til utvidelse.

Hver gang en regel aktiveres for en Sblock skal dette lagres. Det er støtte for aktivering av opptil 256 regler for hver Sblock i den nåværende implementasjonen, det argumenteres i [1] at dette er mer enn nok. Det er også mulig å hente ut en total for hvilke regler som har blitt aktivert i systemet.

4.3 Eksempelprogram

Et typisk program for kjøring av en Sblock-matrise er gitt i figur 4.1. Programmet starter med å konfigurere Sblocken med de nødvendige data som regelsettet, hvilke funksjoner Sblock-typene skal ha og til slutt starttilstanden til matrisen. Så starter en standard programløkke hvor sblockene kommuniserer og oppdaterer tilstanden sin n-antall ganger, etterfulgt av et utviklingssteg hvor typene oppdateres og utlesning av

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		(Tidligere: Konfigurer SBlockmatrise fra BRAM-1)
			Nå: Konfigurere Sblocken med riktig LUT-verdi
* customClear	Kort		Setter telleren til 0
* customInc	Kort		Øker telleren med 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, type, address	Hopper til addresse dersom antall dev-step er kjørt
			eller teller har nådd verdi
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, må kjøre storeSums først
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
* set	Kort		Setter alle Sblocker i matrisen med data i minnet
setNumberOfLastRule	Kort	prioritet	Angir regel med høyest prioritet
store	Kort	startadresse	Lagre etterfølgende instruksjoner i instruksjonslager
* storeSum	Kort		Lagre summen av 1-ere for alle Sblocker
! 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 4.1: Gammelt instruksjonssett

dataene. Til slutt i programmet sjekkes hvilke regler som har blitt brukt fra genomet i kjøringen.

```
1  LoadGenome
2  LoadTypes
3  InitMatrix
4  while(!finished)        // Til programmet er ferdig
5          Run(n)          // Kjør n antall run-steps
6          Develop()       // Kjør development-prosessen
7          ReadMatrix()    // Leser ut tilstand og type til alle Sblocker
8  ReadUsedRules()         // Leser ut alle brukte regler i genomet
```

Figur 4.1: Pseudokode for kjøring av en Sblock-matrise

Kapittel 5

Maskin- og programvareplatform

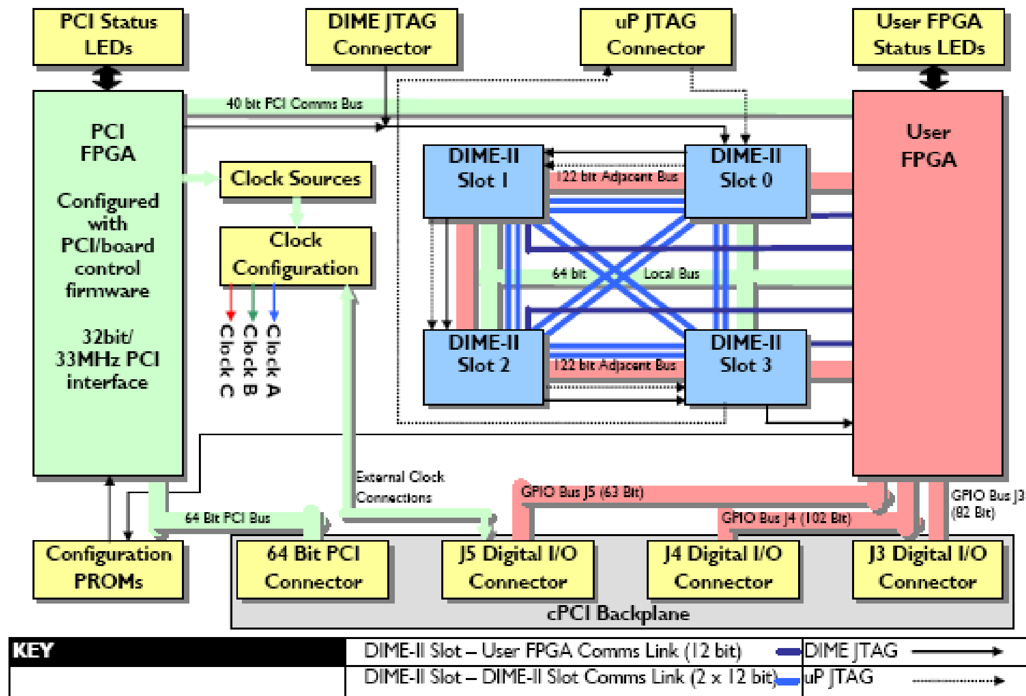
Konfigurering av en FPGA krever en eller annen form for kommunikasjon for å laste inn bit-filen. I tillegg trengs det i denne oppgaven kommunikasjon for å sette opp Sblock-matrisen og lese ut resultater. Det finnes en del leverandører som tilbyr ferdiglagde kort med en eller flere FPGAer på. Da følger det også med drivere og gjerne et API for å konfigurere og kommunisere med kortet og de FPGAene som sitter på det.

Dette kapittelet beskriver maskinvareplatformen Sblockmatrisen er implementert for. Først beskrives kortet hvor FPGAene sitter, så utviklingsplatformen og til slutt maskinene som er brukt.

5.1 Nallatech BenBlue™-III

Figur 5.1: Nallatech BenBlue™-III

Dette er et DIME-kort som består av to Virtex-II Pro brikker, minne og en del logikk. Figur 5.1 viser hvordan kortet ser ut. BenBlue-kortet sitter på et BenERA-kort i en datamaskin. Denne maskinen styrer all kommunikasjon med BenERA over PCI-bussen. BenBlue er koblet til nabo-kortene sine med en 122bits buss og UserFPGA med en 64bits buss som vist i figur 5.2

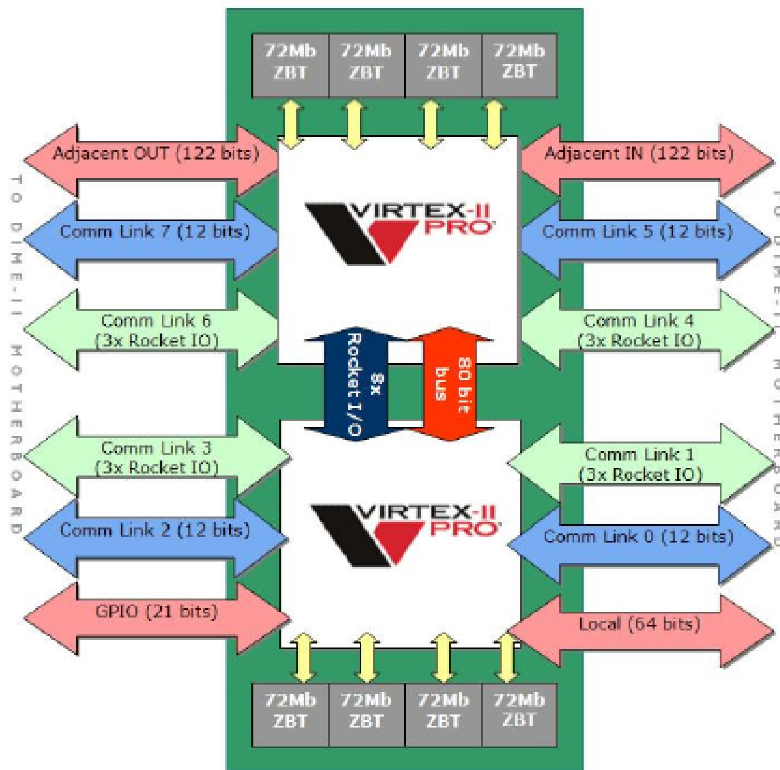


Figur 5.2: Funksjonellt diagram over BenERA [18]

BenBlue er koblet til BenERA kortet som vist i figur 5.3. Denne figuren viser også sammenkoblingen mellom FPGAene på kortet.

5.1.1 FUSE og DIME

FPGAen konfigureres ved hjelp av programvarebiblioteket FUSE (Field Upgradeable Systems Environment) [17] på datamaskinen PCI-kortet sitter i. Dette biblioteket gir utviklere mulighet for å skrive programvare som bruker BenERA. Utvikleren benytter seg også av biblioteket DIME som tilbyr funksjonalitet som lokalisering av FPGA, åpning og lukking av kort, styring av klokkefrekvens, konfigurasjon av FPGA, utlesing av informasjon og kontroll over I/O [16].



Figur 5.3: Kommunikasjonslinjer for BenBlue [18]

5.1.2 Inter-kommunikasjon

Det er mulig å konfigurere FPGAene på BenBlue til å kommunisere med hverandre. Ved å gjøre dette kan man dele opp Sblockmatrisen på to FPGAer, med litt overhead får man doblet antallet CLBer.

Kommunikasjon mellom de to FPGAene kan gjøres ved å koble sammen IO-pinner på vanlig måte, eller ved hjelp av RocketIO.

5.2 Nalle

Maskinen BenEra kortet er montert i er en CompactPCI-maskin og har disse spesifikasjonene:

- To pentium III-prosessorer

- Hovedkort: VMICPCI-7760
- I/O kort: VMIACC-0320
- RAM: 512 MB
- Nettverk: 2 stykk 10/100 MBit ethernetkort

5.3 Arbeidsstasjon

Utviklingen ble utført på en vanlig arbeidsstasjon:

- Pentium 4
  2,40 GHz
  512 MB RAM
- Windows XP
  Servicepack 2
- Xilinx ISE
  Project Navigator 6.1.03i
  Project Navigator 7.1i
- ModelSim SE PLUS 6.0a

Kapittel 6

Implementasjon

I denne delen beskrives utfordringene under implementasjonen og hvordan disse har blitt løst. Først beskrives kompromisser mellom maskinvarebehov og parallellitet, deretter oppbygning av systemet med tilstander Sblockmatrisen kan være i. Til slutt beskrives kontrolleren som styrer systemet.

6.1 Kompromisser

Målet med denne oppgaven var blant annet å parallellisere Sblock-matrisen så mye som mulig. På grunn av begrensninger i maskinvare ble det allikevel gjort noen kompromisser. Denne delen forklarer dette i detalj.

6.1.1 Regelsett/Genom

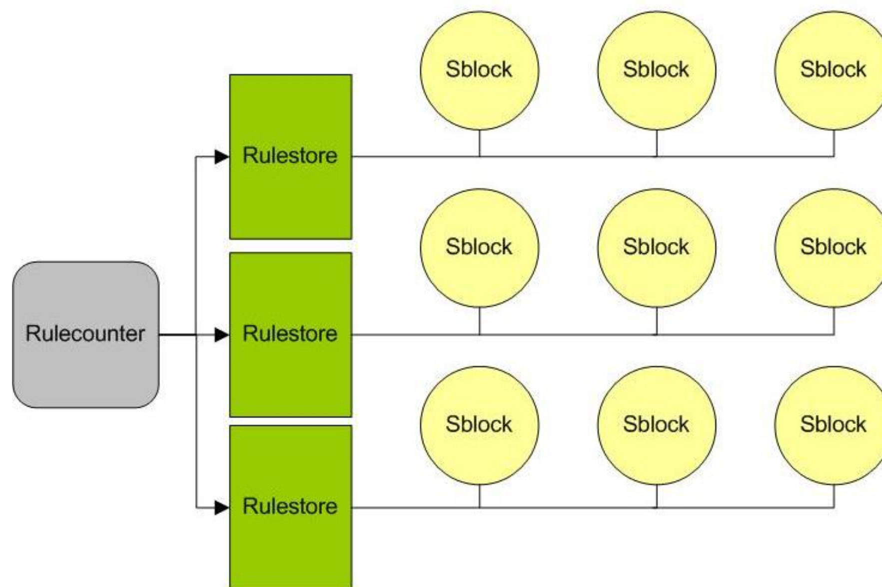
Xilinx Virtex-II Pro XC2VP70 FPGA som ble brukt i denne oppgaven har 328 blokker med 18Kb RAM hver. Det er mulig å konfigurere bredden på denne til å være hele regelstørrelsen som er på 49 bit pr regel. Utlesing av en regel tar dermed én klokkesykel, og hele genomet som består av 256 regler får plass i én BRAM-blokk.

Hvis alle BRAM-blokker er ledige og man distribuerer genomet betyr det at det er plass til 328 Sblocker. Dette tilsvarer en matrise på 18x18 Sblocker.

Ved å distribuere regelsettet til alle Sblocker blir antall BRAM-blokker en fysisk begrensning for antallet Sblocker i matrisen. Når utviklingssteget starter leser alle Sblocker ut første regel, sjekker denne mot sin egen og naboenes verdier, så sjekkes andre regel, så tredje og slik fortsetter det til alle reglene er sjekket, eller en av reglene aktiveres. Siden alle Sblocker sjekker samme regel samtidig er det mulig å desentralisere lageret med regelsettet uten å miste parallellitet. Ved å la flere Sblocker dele samme lagringsplass for regler fjernes begrensningen fra antall BRAM uten å miste parallellitet.

Den valgte løsningen er å la hver rad dele BRAM-blokk for å lese ut regler som vist i figur 6.1. Ved å fordele minnet slik, unngåes det å bruke for mange BRAM-blokker samtidig som rutingen holdes mindre kompleks enn hvis alle Sblocker deler samme BRAM-blokk. Alle "rulestore" i figuren representerer en BRAM-blokk med samme innhold og bruker en felles teller for å iterere over reglene.

Denne løsningen skalerer godt helt til antall rader nærmer seg antall BRAM-blokker. Hvis man har en matrise med svært mange rader krever det liten endring i VHDL-koden for å la to rader dele regel-lager. Dette vil frigjøre BRAM-blokker, men lage mer kompleks ruting.

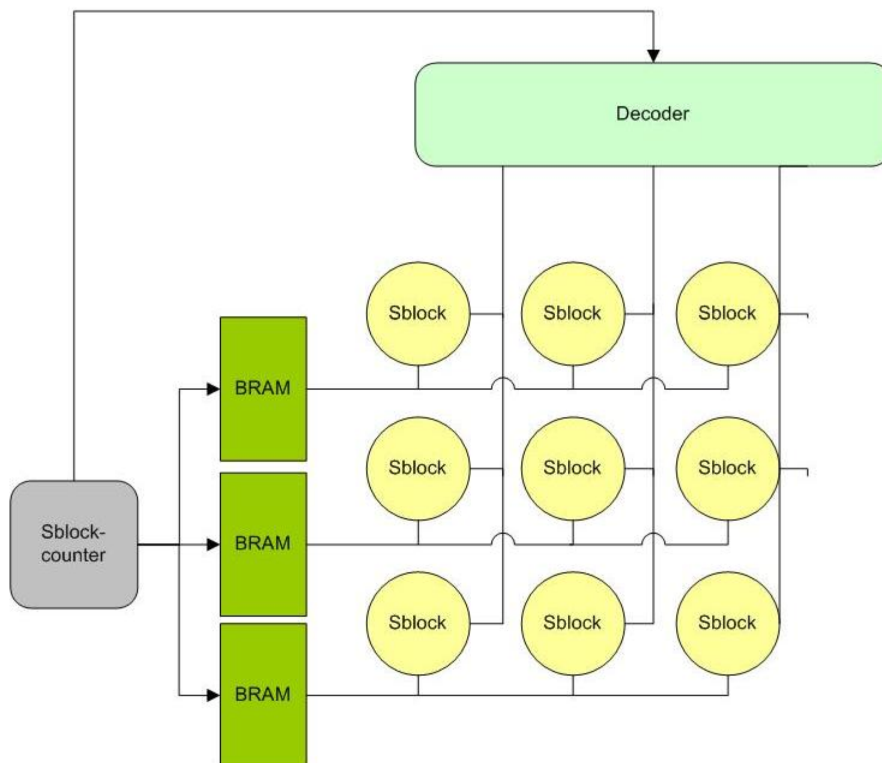


Figur 6.1: Regel-BRAM koblet til Sblock-matrise

6.1.2 Konfigurering og lagring av matrise

Et av kravene til Sblock-matrisen er at hver Sblock skal kunne konfigureres med type og tilstand. Dessuten skal det være mulig å lese ut alle typene og tilstandene til Sblockene i matrisen til enhver tid. Dette er løst ved å la hver rad ha en dedikert BRAM-blokk på samme måte som regel-lageret er løst.

Når Sblock-matrisen skal konfigureres settes adressetelleren til 0. Dataene i BRAM-blokkene må være en gyldig konfigurasjon av Sblock-matrisen. Sblock-dekoderen får inn verdien til addressen og dekoder dette til en "one-hot"-signal. Det betyr at kun ett av signalene fra dekoderen er høy. Dette er koblet sammen som på figur 6.2 og gjør at én minnelokasjon i hver BRAM er koblet til én Sblock. Sblock-matrisen kan konfigureres på like mange klokkesykler som det er antall rader i matrisen.



Figur 6.2: Konfigurasjon av Sblock-matrisen

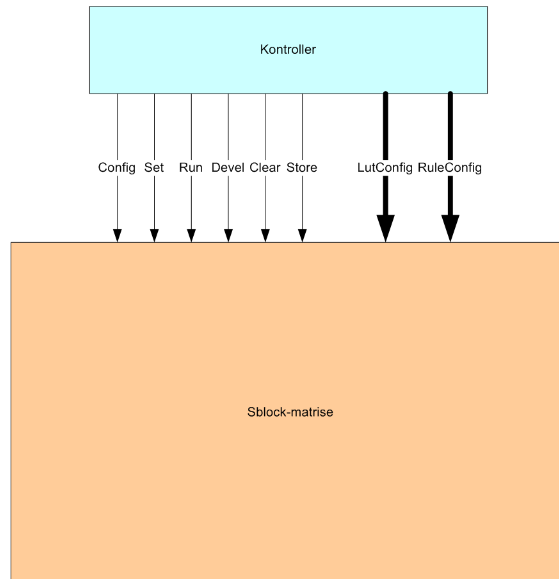
Utlesing av data fra Sblockene gjøres på tilsvarende måte som konfigurering. Adressetelleren bestemmer minnelokasjon og hvilken Sblock som skal få legge data ut på indatabussen til minnet.

6.2 Oppbygning

Systemet har en overordnet arkitektur som vist i figur 6.3. Oppdelingen i en kontroller som styrer matrisen ved hjelp av kontrollsignaler er laget for å enkelt kunne bytte ut kontrolleren med for eksempel PowerPC-prosessoren som sitter på FPGAen. Dette kan gjøres ved å innføre registre mellom kontrolleren og matrisen og blir nærmere diskutert i avsnitt 8.3

I matrisen er Sblockene koblet til hverandre, kontrolleren og de globale signalene clk og rst som vist i figur 6.4.

Hver Sblock er bygd opp som vist i figur 6.5. Dette avsnittet beskriver de ulike komponentene.



Figur 6.3: Systemarkitektur

6.2.1 State LUT

Når Sblock-matrisen kjører bestemmes ut-verdien til hver Sblock av en lookup-tabell. Denne tabellen konfigureres når matrisen settes opp i starten, når brukeren setter typen manuelt eller når cellen utvikler seg.

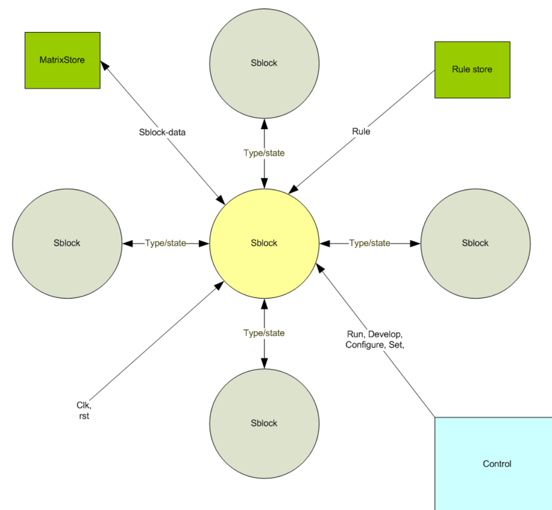
Lookup-tabellen tar verdien til alle naboene og Sblockens egen ut-verdi som input. Ut-verdien fra tabellen er gitt fra konfigurasjonen.

Ved hjelp av forskjellige konfigurasjoner kan man implementere alle typer Sblocker. Enheten er koblet til Type2LUTConv og naboene som vist i figur 6.6. Når LUT-en skal konfigureres settes alle verdiene på input-portene til '1', write_enable-signalet settes høyt og konfigurasjonen klokkes inn gjennom D-porten. Når tabellen brukes til å finne tilstanden til Sblocken settes input-verdien til å være tilstanden til naboene. En 5-input LUT er implementert ved å bruke to 4-input LUT og Sblockens egen tilstand for å velge mellom disse.

6.2.2 Type2LUTConf

Dette er et distribuert minne hvor koblingen mellom Sblock-type og LUT-konfigurasjon er lagret. Når Sblocken endrer sin type må den også slå opp i dette minnet for å finne riktig konfigurasjon for lookup-tabellen.

Enheten konfigureres med gyldige Sblock-typer og LUT-konfigurasjonene deres før Sblockmatrisen kjøres. Dette skjer parallellt i alle Sblocker samtidig.



Figur 6.4: Overordnet sammenkobling i Sblock-matrisen

6.2.3 Development unit

Development-enheten tar seg av utvikling av Sblocken. Den bruker regelsjekkeren som beskrevet i neste avsnitt til å finne ut om en regel er trigget. Hvis dette er tilfellet endres Sblockens tilstand og type i henhold til regelen. I tillegg konfigureres State LUT med ny konfigurasjon fra Type2LUTConf-minnet.

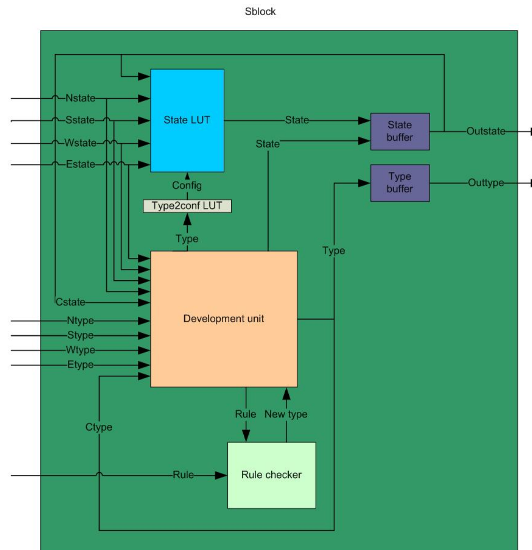
Informasjon om hvilken regel som ble brukt sendes ut av Sblocken og lagres sentralt hvis det er ønske om det.

6.2.4 Rulechecker

Regelsjekkeren er hentet fra development-steget i prosessoren laget i [6]. Denne får inn regler fra Sblocken som sjekkes mot tilstandene og typene til nabo-Sblockene og Sblockens egen type og tilstand. Deretter gir den beskjed til development-enheten om eventuell ny tilstand og type.

6.2.5 State og Type buffer

Dette er et buffer som tar vare på tilstanden og typen til Sblocken. Dette settes som utgangsverdier hver klokkesykel.



Figur 6.5: Oppbygningen av en Sblock

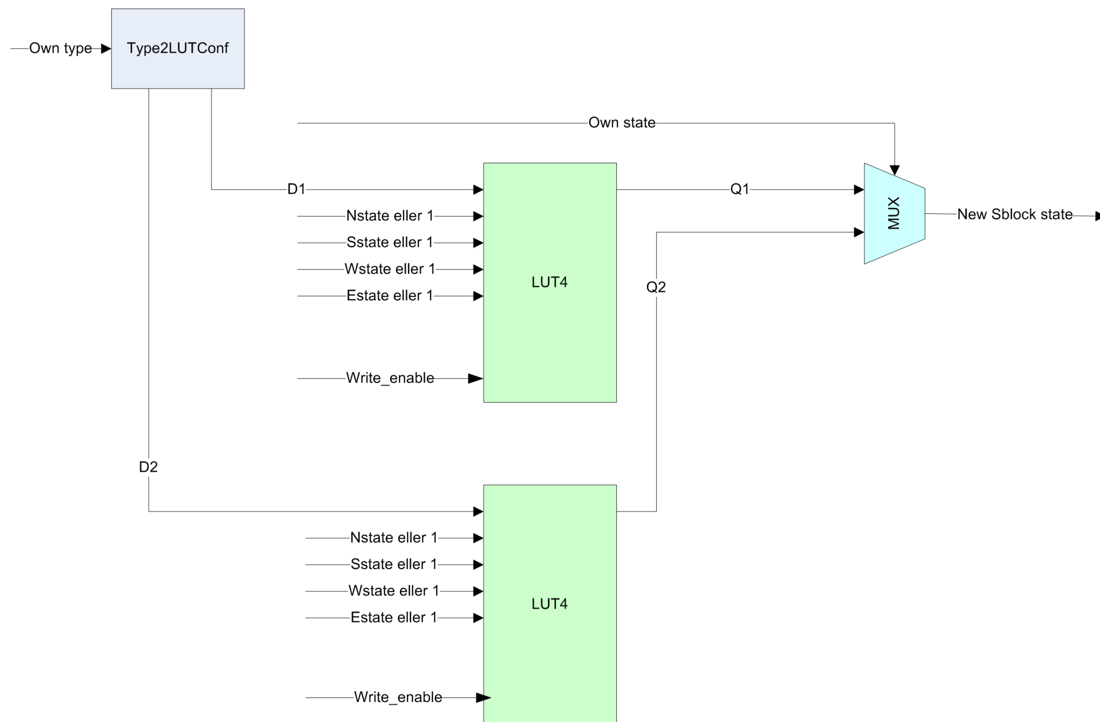
6.3 Tilstander

Sblock-matrisen kan være i flere forskjellige tilstander. Kontroll-enheten styrer tilstanden til alle Sblockene i matrisen. Sblockene er alltid i samme tilstand. De ulike tilstandene til Sblockene er vist i figur 6.7 og blir beskrevet i dette avsnittet. I tillegg til tilstandene Sblockene kan ha, har kontrolleren et par tilstander som Sblockene ikke påvirkes av. Skriving til regel-minnet tas derfor ikke med i denne oversikten, heller ikke når kontrolleren skriver direkte til matrise-minnet. Skriving til Type2LUTConf-minnet regnes heller ikke som en tilstand og tas derfor heller ikke med.

6.3.1 Reset

I denne tilstanden blir hele systemet startet opp. Alle start-verdier blir satt og Sblockene mister all informasjon om tilstand og type. Det eneste som ikke resettes er informasjonen i BRAM-blokkene og i Type2LUTConf-minnet.

Når reset-signalet på FPGAen settes lavt går hele matrisen inn i denne tilstanden. Som vist i figur 6.8 blir da Sblocken satt opp med starttype “00000” og tilstand “0” når reset-signalet går høyt igjen. Mange andre signaler i systemet blir også resatt på samme måte.



Figur 6.6: State LUT og Type2LUTConv

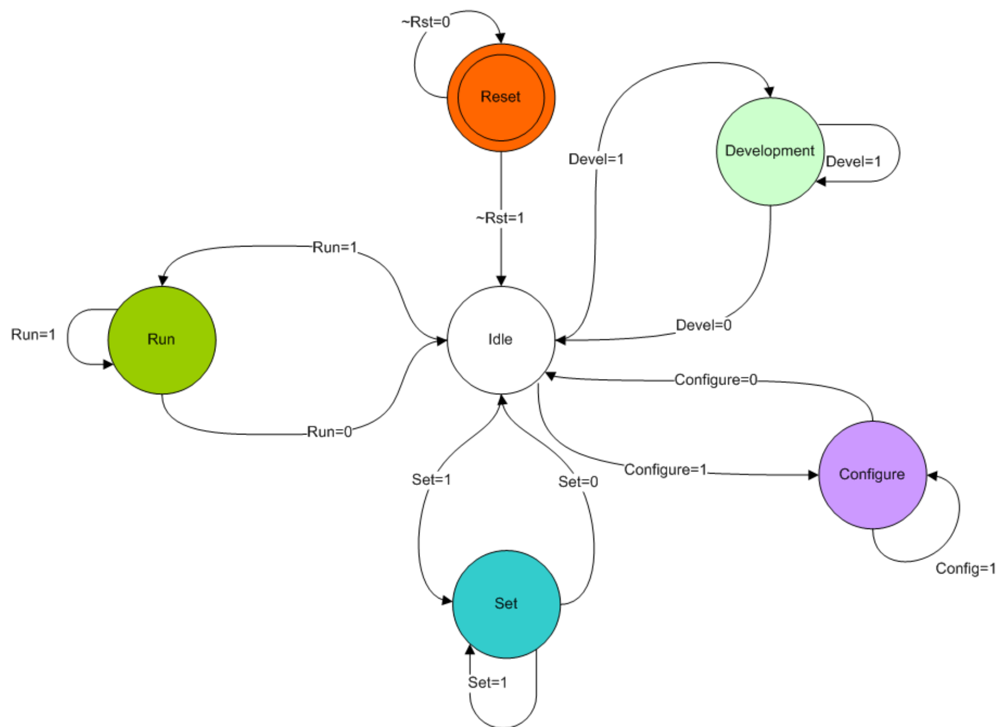
6.3.2 Idle

Dette er en hviletilstand hvor systemet ikke gjør noen beregninger eller forandrer seg. Når Sblockene er i denne tilstanden kan kontrolleren lese ut tilstander og typer fra matrisen og legge det i minnet. Kontrolleren kan også være i Idle-tilstanden, noe som kan brukes hvis f.eks PowerPC-prosessoren skal kjøre en fitness-funksjon eller kjøre en genetisk algoritme.

6.3.3 Set

Før simulering eller som en del av et eksperiment er det ønskelig å konfigurere én eller flere Sblocker til en bestemt tilstand og type. Dette gjøres delvis sekvensielt med "set"-tilstanden som beskrevet i avsnitt 6.1.2, noe som betyr at systemets set-tilstand består av like mange undertilstander som det er kolonner i matrisen. Dette er vist i tilstandsdiagram 6.9.

Systemet klarer å konfigurere en hel kolonne i Sblock-matrisen av gangen. For at alle Sblockene skal konfigureres korrekt må derfor kontrollsignalet fra kontrolleren holdes høyt i like mange klokkesykler som det er kolonner i matrisen. Figur 6.10 viser hvordan en av raden i en 2x2-matrise settes. Først settes tilstanden og typen til Sblocken i posisjon (0,0). Deretter settes "set_enable"-signalet høyt for Sblocken i posisjon (0,1).



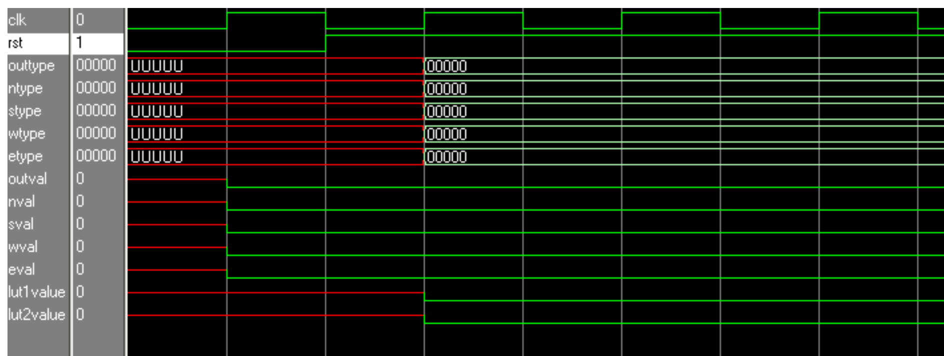
Figur 6.7: Tilstandsdiagram for en Sblock

Tilsvarende skjer for den andre raden.

6.3.4 Run

Run-tilstanden er den "normale" tilstanden til Sblock-matrisen. Det er her hver enkelt Sblock bruker sin egen ut-verdi sammen med naboene sine for å finne sin nye ut-verdi. I Run-tilstanden forandres aldri typen til Sblockene, bare verdien.

Det er når Sblocken er i denne tilstanden at verdien fra StateLUT klokkes ut som Outstate. Dette skjer som vist i figur 6.11. Her vises tilstanden til 4 Sblocker oppdatert gjennom 77 run-step. Ved hver stigende klokkeflanke legges ny verdi ut som Outstate i henhold til LUT-konfigurasjonen til Sblocken, som igjen er gitt fra Sblockens type.



Figur 6.8: Oppstart av Sblocken

6.3.5 Development

Development-tilstanden er en av de mest komplekse tilstandene i implementasjonen. I denne tilstanden sjekkes alle reglene i genomet med Sblockenes naboer. Dette starter med at kontroll-enheten forteller Sblockene om at de skal i Development-tilstanden. Så setter kontrolleren regelminneadressen til 0 og første regel legges ut på regelbussen. Sblockene sjekker én regel hver klokkesykel.

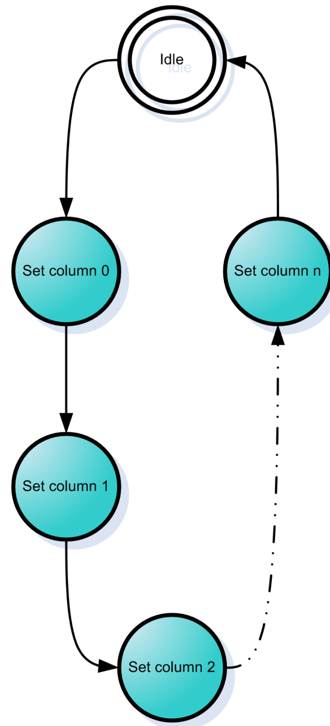
Hvis en Sblock aktiverer en regel stoppes development-prosessen i denne, og Sblocken sjekker ikke flere regler denne runden. Siden resultatet av development ikke skal være synlig før etter hele prosessen er ferdig, mellomlagres resultatet. Sblocken som aktiverte en regel sier fra til kontrollenheten om at regelen ble brukt.

Development-tilstanden tar like mange klokkesykler å utføre som det er regler i genomet. Kontrollsignalet til denne tilstanden må altså holdes høyt i tilsvarende antall klokkesykler.

Figur 6.12 viser et development-steg med 6 regler kjørt på en av Sblock. Som waveformen viser aktiveres regel 3 av Sblocken. Ifølge regelen skal denne vokse fra fra Sblocken nord for seg og få typen "00001". Dette mellomlagres som "owntype" og settes som ut-type når development-steget er ferdig.

6.3.6 Configure

Etter Set- eller Development-tilstandene har som regel en eller flere Sblocker fått ny type og tilstand. For at Sblocken skal oppføre seg i henhold til typen sin må lookup-tabellen konfigureres riktig. Dette gjøres ved at Sblocken slår opp i sitt lokale Type2LUTConfig-minne etter sin type. Så konfigureres LUT'en med dataene ut fra dette. Konfigurasjonen som er på 32 bit klokkes inn en etter en, noe som tar tar 32 klokkesykler. Det betyr at hver gang matrisen er i Configure-tilstanden må kontrollsignalet holdes høyt i 32 klokkesykler.



Figur 6.9: Tilstandsdiagram for set-tilstanden

Etter Configure er Sblocken konfigurert med funksjonalitet i henhold til typen som ble satt under Set- eller Development-tilstanden.

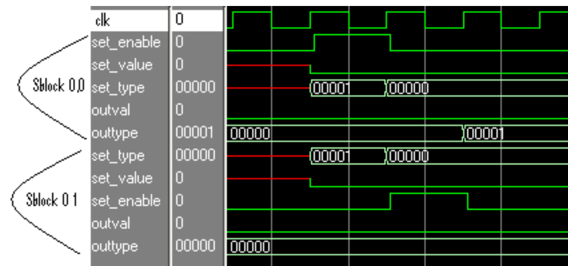
I figur 6.13 vises hvordan en Sblock med typen “00001” blir konfigurert. I dette eksempelet har LUT-konfigurasjonen verdien “1”. config_data-signalet er dataene som klokkes inn i den LUT-en som har enable-signalet satt høyt. Hvilket av bit-ene i newlutconfig som skal legges på config_data-linjen bestemmes av count-signalet.

6.3.7 Save

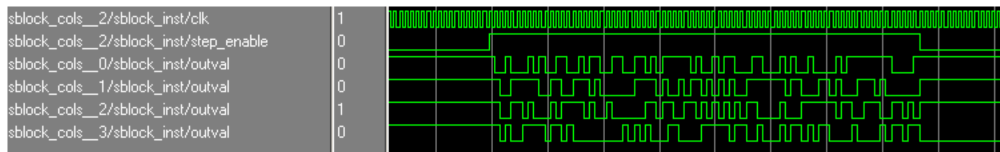
Under simulering vil det være ønskelig å hente ut informasjon om matrisen ved ulike tidspunkt. Noen ganger ønskes det data etter hvert Development-steg, andre ganger mellom hvert Run-steg eller kanskje trengs bare sluttresultatet. Dette er muliggjort ved en egen Save-tilstand. I denne tilstanden er alle Sblocker i Idle-tilstanden, mens matrisens kontroll-enhet lagrer all informasjon om tilstander og typer.

Save-tilstanden ligner en del på Set-tilstanden, men kontrolleren må passe på at riktig Sblock er koblet til input-porten til minnet.

Kontrollsignalet til Save må holdes høyt i like mange klokkesykler som det er kolonner i Sblock-matrisen.



Figur 6.10: Setting av Sblock-tilstand og -type fra BRAM.



Figur 6.11: Kjøring av en Sblock-matrise

Et eksempel på lagring av en 2x2 matrise er gitt i figur 6.14. Når store-signalet blir høyt legges adressen til de første Sblockene på adressebussen til BRAMen, write-enable satt høyt og type og tilstand til Sblocken legges på din-bussen. Dette skjer parallellt for begge rader i matrisen. Neste gang klokkesignalet går høyt skjer det tilsvarende for andre kolonne, og nå er hele matrisen lagret.

6.4 Kontroller

For å ta imot og tolke instruksjonssettet til systemet har det blitt laget en kontroller. Dette er en enkel flersykel-prosessor som ut fra instruksjonene setter verdien til de ulike kontrollsignalene og holder dem der i riktig antall klokkesykler. Instruksjonene mottas enten over PCI-bussen som beskrevet i 6.4.2 eller fra et dedikert BRAM-instruksjonsminne. For å fortsatt kunne bruke adresseringen i instruksjonssettet er antall instruksjoner begrenset til 256, men dette går enkelt ann å utvide.

6.4.1 Instruksjonssett

Instruksjonssettet fra [1] har blitt litt forandret. Så langt det har vært mulig har instruksjonene beholdt funksjonaliteten sin. Allikevel har noen instruksjoner blitt fjernet, noen endret litt og et par nye lagt til.

Figur 6.12: Eksempel på development

Figur 6.13: Eksempel på LUT-konfigurasjon av en Sblock

clearBRAM

Denne instruksjonen har ikke blitt implementert på grunn av tidspress. Den bør inkluderes i videre arbeid.

config

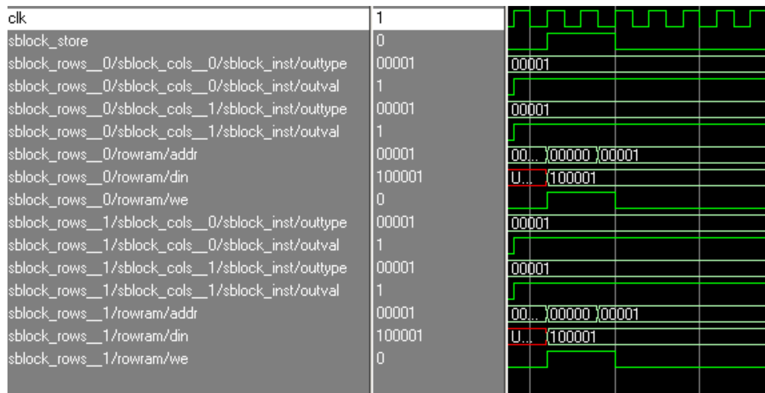
Tidligere konfigurerte denne instruksjonen Sblockene fra BRAM. Den nye betydningen av instruksjonen er å konfigurere LUT i hver Sblock. Denne instruksjonen vil som regel alltid følge "set" og "devstep".

customClear

Etter ønske fra Gunnar Tufte har det blitt introdusert en teller i tillegg til development-telleren. Denne instruksjonen er ny og setter den nye telleren til 0.

customInc

Ny instruksjon som øker den nye telleren med 1.



Figur 6.14: Lagring av matrisen

doFitness

På grunn av tidspress har ikke den innebygde fitness-funksjonaliteten blitt implementert. Dette er en relativt viktig funksjon som bør prioriteres i videre arbeid.

jumpEqual

Siden implementasjonen har inkludert en ny teller, har jumpEqual-funksjonen et nytt bit som velger hvilken teller som skal sammenlignes med.

readFitness

Er ikke implementert, men bør implementeres samtidig som "doFitness".

readRuleVector

Er ikke implementert.

run

Run krever to etterfølgende "nop" før development-steg. Dette for at development-enheten skal konfigureres med data om Sblocken før utviklingen starter.

Set

Dette er en ny instruksjon. Leser ut Sblock-data fra BRAM og setter hele matrisen med typer og tilstand fra denne.

Switch

Denne instruksjonen er ikke nødvendig siden kontrolleren jobber med samme BRAM som Sblockene i denne implementasjonen og er derfor fjernet fra instruksjonssettet.

writeState

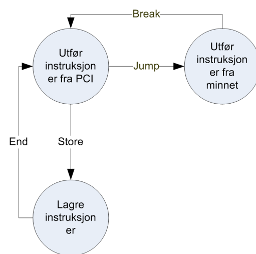
På grunn av tidspress har denne instruksjonen ikke blitt implementert. Funksjonaliteten er flyttet til writeType som lagrer både tilstand og type. Dette begrenser mulighetene noe, men implementasjonen gjør det ikke umulig å utvide designet til å støtte denne instruksjonen senere.

writeType

Denne instruksjonen har fått utvidet funksjonalitet og setter nå både tilstand og type på den angitte Sblocken.

6.4.2 Kommunikasjon

Kontrolleren tar også hånd om kommunikasjon over PCI-bussen. Dette skjer gjennom en modul utviklet i [6] nærmere beskrevet i vedlegg D. Etter oppstart av kontrolleren på FPGAen starter denne å vente på instruksjoner over PCI-bussen. Disse mottas og utføres fortløpende helt til en Store-instruksjon mottas. Da lagres de følgende instruksjonene i instruksjonsminnet helt til en end-instruksjon mottas. Når dette skjer begynner kontrolleren å utføre instruksjoner fra PCI-bussen igjen helt til en jump-instruksjon mottas. Da begynner kontrolleren å utføre instruksjoner fra minnet på minneadressen oppgitt i jump-instruksjonen. Dette utføres helt til instruksjonen i minnet er en Break-instruksjon, da begynner kontrolleren å vente på instruksjoner fra PCI igjen. Figur 6.15 viser tilstandsdiagrammet for kommunikasjon for kontrolleren.



Figur 6.15: Tilstandsdiagram for kommunikasjon

Kapittel 7

Resultater

I tillegg til kontinuerlig testing av systemet under utvikling har det blitt kjørt noen større tester. Dette kapittelet starter med en oversikt over syntetisering av systemet, så følger en manuell og en funksjonell test. Til slutt har det blitt utført noen utregninger med tanke på hastigheten til systemet.

7.1 Syntese

Systemet har blitt syntetisert med ulike dimensjoner på matrisen. Resultatet er i tabell 7.1. Designet støtter i prinsippet syntese av alle matrisen med dimensjoner 2 eller større i hver retning. Det som begrenser størrelsen er fysisk plass på FPGAen, og som tabell 7.1 viser er det antall slice som setter en stopper. Den største mulige matrisen som får plass på en FPGA er 16x16 = 256 Sblocker. Dette er mindre enn de estimerte 24x24 i kapittel 2.3.2. Grunnen til det er at hver Sblock brukte mer ressurser enn det tidlige estimatet regnet med, spesielt med tanke på logikk for lagring og konfigurasjon.

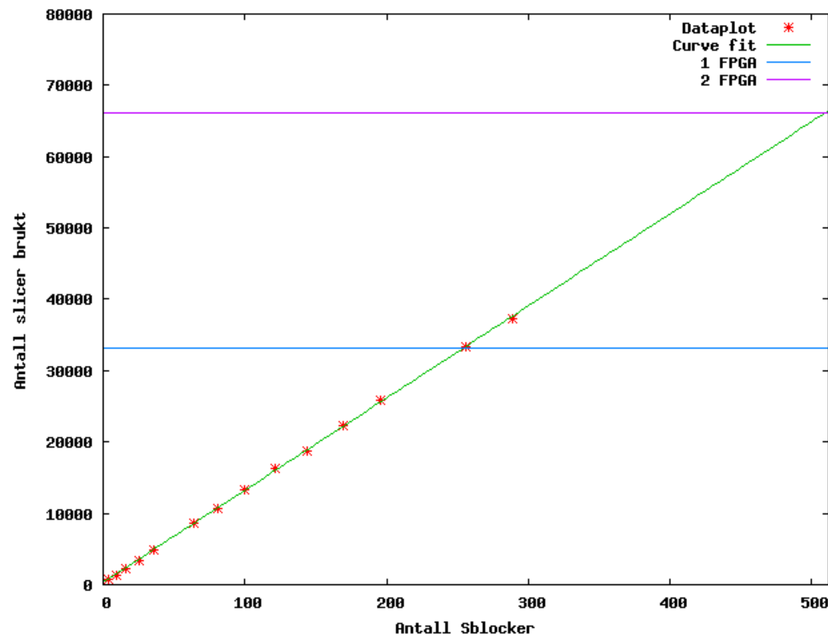
En graf som viser ressursforbruk er gitt i figur 7.1. I denne grafen er de symmetriske Sblockene plottet inn, og man ser at antallet slice per Sblock er vokser lineært. Resultatene i tabell 7.1 antyder at denne funksjonen er tilnærmet ligning 7.1. I tillegg til resultatene er en lineær graftilpasning og antall Slice for en og to FPGAer er også tegnet inn i figur 7.1. Antall Sblocker per FPGA blir tilnærmet som ligning 7.2.

$$N_{Slice} = 130,831479 * Sblock \tag{7.1}$$

$$N_{\frac{Sblock}{FPGA}} = \frac{Slices_{FPGA}}{Slices_{Sblock}} = 252,9 Sblocker \tag{7.2}$$

Dette betyr at de to tilgjengelige FPGAene vil ha plass til 512 Sblocker, noe som vil si en matrise på 22x22 Sblocker. I tillegg kommer logikk for kommunikasjon mellom de

to FPGAene.



Figur 7.1: Ressursforbruk for Sblock-matriser

7.2 Manuell test

For å sjekke at implementasjonen er gjort riktig har det blant annet blitt kjørt en svært enkel manuell test. Denne testen består av en Sblock-matrise med 2x2 Sblocker, en regel for vekst, en regel for utvikling og en celletype.

Celletypen som brukes er NOT. Det vil si at cellen har ut-verdi 1 hvis alle inn-verdiene er 0. Ellers er ut-verdien 0.

Vekstregelen er som følger: Hvis en celle er tom og cellen på vest-siden er av NOT-typen vokser cellen fra vest inn over den tomme cellen. Dette skjer som i figur 7.2.

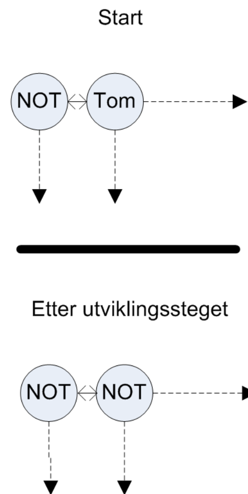
Regelen for utvikling er egenlig en regel for celledød. Hvis en celle er av typen NOT og nabocellen til øst for denne er tom skal cellen dø og bli tom. Dette blir egentlig prosessen i figur 7.2 baklengs.

Utviklingen gjennom 3 development-steg med 4 run-steg hver for de fire cellene i matrisen er vist i figur 7.3.

I figur 7.4 er samme matrise med samme typer og regler simulert i Modelsim. Ut fra waveformen kan man se at resultatet stemmer med den manuelle simuleringen.

Bredde	Høyde	Hastighet	BRAM	BRAM%	Slice	Slice%
2	2	176 MHz	5	1%	727	2%
2	4	174 MHz	9	2%	1224	3%
2	8	173 MHz	17	5%	2156	6%
2	16	175 MHz	33	10%	3992	12%
2	32	173 MHz	65	19%	7739	23%
3	3	173 MHz	7	2%	1391	4%
4	2	174 MHz	5	1%	1264	3%
4	4	173 MHz	9	2%	2269	6%
4	8	175 MHz	17	5%	4277	12%
4	16	173 MHz	33	10%	8298	25%
4	32	173 MHz	65	19%	16168	48%
5	5	147 MHz	11	3%	3455	10%
6	6	146 MHz	13	3%	4893	14%
7	7	139 MHz	15	4%	6523	19%
8	2	123 MHz	5	1%	2348	7%
8	4	116 MHz	9	2%	4434	13%
8	8	113 MHz	17	5%	8578	25%
8	16	94 MHz	33	10%	16803	50%
8	32	92 MHz	65	19%	33506	101%
9	9	108 MHz	19	5%	10743	32%
10	10	102 MHz	21	6%	13215	39%
11	11	92 MHz	23	7%	16344	49%
12	12	88 MHz	25	7%	18765	56%
13	13	100 MHz	27	8%	22366	67%
14	14	98 MHz	29	8%	25799	77%
15	15	90 MHz	31	9%	29347	88%
16	2	122 MHz	5	1%	4489	13%
16	4	118 MHz	9	2%	8644	26%
16	8	98 MHz	17	5%	16700	50%
16	16	82 MHz	33	10%	33423	101%
16	32	Syntetiserer ikke				
17	17	92 MHz	35	10%	37294	112%
32	2	91 MHz	5	1%	8795	26%
32	4	89 MHz	9	2%	16957	51%
32	8	74 MHz	17	5%	33413	101%
32	16	Syntetiserer ikke				
32	32	Syntetiserer ikke				

Tabell 7.1: Syntese av ulike Sblock-matriser



Figur 7.2: Vekstregelen brukt i test-matrisen

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

Tabell 7.2: Regler for den funksjonelle testen

7.3 Funksjonell test

For å sammenligne med tidligere resultater ble det kjørt en test med tilsvarende regler som i [6] og [1]. I testen konfigureres en 8x8-matrise med Sblocker. Testen bruker tre forskjellige Sblock-typer:

- Z; En tom eller død Sblock. Denne endrer ikke tilstand. Kodes med "00000".
- J; Logisk "eksklusiv eller" av de fire naboene til Sblocken. Kodes med "00001".
- I; Logisk "eller" av Sblockens fire naboer. Kodes med "00010".

Siden den nye implementasjonen sjekker reglene i motsatt rekkefølge av den gamle legges reglene inn som i tabell 7.2. Typen "D" betyr "Don't care", og angir at typen kan være hva som helst. De seks reglene bryr seg bare om Sblockenes type og ikke om tilstander. Sblockenes tilstand har derfor ikke noe å si for development-stegene.

Før simuleringen kjøres konfigureres starttilstanden til Sblock-matrisen. Tilstandene til Sblockene konfigureres som i figur 7.5(a) og kun en av Sblockene får en starttype som

Figur 7.3: Manuellt utført celleutvikling

ikke er tom, som illustrert i figur 7.5(b).

Psaudokoden for programmet som skal kjøres er:

```
for i = 0 to 150
    kjør 77 tilstandssteg
    kjør 1 developmentsteg
```

For denne implementasjonen av Sblock-matrisen brukes maskinkoden i tabell 7.3. For mer informasjon om instruksjonssettet, se vedlegg B.

Forsøket ble kjørt før FPGAen var tilgjengelig. For å få et mest mulig realistisk resultat ble det laget en testbenk av typen "Post place and route" for ModelSim. Dette simulerer alle komponentene i FPGAen med forsinkelser så detaljert det er mulig med disse verktøyene. Figur 7.7 viser typene til alle Sblockene i matrisen mot slutten av simuleringen. Figur 7.6(a) viser en grafisk framstilling av tilstandene til Sblockmatrisen etter kjøring av den funksjonelle testen. Figur 7.6(b) viser typene til matrisen etter kjøring. Resultatene er de samme som i [6] og [1], noe som tyder på at implementasjonen har korrekt funksjonalitet.

7.4 Hastighetstest

Maskinvaren som skal brukes var ikke tilgjengelig da denne rapporten ble skrevet. Derfor var det ikke mulig å kjøre en hastighetstest, hverken for konfigurasjon, kjøring eller utlesing av resultater.

Instr.nr	Maskinkode	Forklaring
0	00000000000000000000000000101001	Skriv til LUTposisjon 0
1	11111111111111110000000000000000	LUT-konfigurasjonsdata for Z
2	00000000000000000000000100101001	Skriv til LUTposisjon 1
3	01101001100101100110100110010110	LUT-konfigurasjonsdata for XOR
4	00000000000000000000001000101001	Skriv til LUTposisjon 2
5	11111111111111101111111111111110	LUT-konfigurasjonstada for OR
6	00100000001000001000001010101011	Skriv til regel 5: Growth fra sør
7	11101000001010000010100000101000	Starten av regel 5
8	00100000001000010000001000101011	Skriv til regel 4: Growth fra øst
9	11101000001010000010100000101000	Starten av regel 4
10	00100000001000000000000110101011	Skriv til regel 3: Growth fra nord
11	11101000001010000010100000101000	Starten av regel 3
12	00100000001000011000000100101011	Skriv til regel 2: Growth fra vest
13	11101000001010000010100000101000	Starten av regel 2
14	00101000001000001000000010101011	Skriv til regel 1: Change til J
15	10100000101000000010100000101000	Starten av regel 1
16	01101000001000000000000000101011	Skriv til regel 0: Change til Z
17	10101000001000001010100000100000	Starten av regel 0
18	00000010000000010000010000000001	Sett Sblock(4,1): Tilstand=0, Type=I
19	00000000000000000000000000011010	Set matrise-verdier fra minnet
20	00000000000000000000010100010010	Siste regel er nr 5
21	00000000000000000000000000000111	Konfigurer Sblock-LUT iht. type
22	00000000000000000000000000000000	NOP
23	00000000000000000000000000000000	NOP
24	00000000000000000100111000001001	Kjør 77 run-steps
25	00000000000000000000000000000000	NOP
26	00000000000000000000000000000000	NOP
27	00000000000000000000000000001010	Development
28	00011111000000001001011000010110	Hvis development-teller = 149, hopp til 31
29	00000000000000000000000000000000	NOP
30	00000000000000000001010100001100	Hopp til 21
31	00000000000000000000000000000000	NOP

Tabell 7.3: Instruksjoner for funksjonell test

Siden antall klokkesykler for hver instruksjon er kjent, og maks klokkefrekvens er gitt av syntetiseringsverktøyet er det allikevel mulig å regne ut en tidsforbruket for ulike simuleringer.

Kommunikasjon over PCI er avhengig av vertsmaskinen og programmet som kommuniserer med FPGAen. Disse operasjonene vil sannsynligvis ta tilsvarende tid som implementasjonene i [6] og [1], men vil ikke bli tatt hensyn til i denne utregningen.

Den høyeste klokkefrekvensen matrisen kan kjøre med er gitt fra forsinkelser i ledninger og komponenter på FPGAen. Etter syntetisering av den største matrisen som får plass på brikken, en 16x16-matrise, gir verktøyet XST [29] at maks frekvens er 82MHz

Antall klokkesykler for et development-step er avhengig av antall regler i genomet og er gitt ved: $klokkesykler_{dev} = regler + 3$. I tillegg må LUTene konfigureres, noe som tar 32 klokkesykler. Hvert run-step tar hver bare en klokkesykel.

Det er verdt å merke seg at tiden for development, configure og run ikke er avhengig av antall Sblocker i matrisen. Kun initialisering og utlesing av data påvirkes av dette. Formel 7.3 viser utregning av kjøretiden til en simulering med 10 000 developmentsteps og 50 000 run-steps for hvert development-step. $T_{utlesning}$ er ukjent og avhenging av PCI-kommunikasjonen, derfor regnes det med at denne bare kjøres på slutten av kjøringen og kan ses bort fra. Utregningen bruker "worst-case" med 256 regler.

$$\begin{aligned} T_{eksperiment} &= (T_{run-step} * 50000 + T_{dev-step} + T_{config}) * 10000 \qquad (7.3) \\ T_{eksperiment} &= ((50000 + 256 + 3 + 32) * \frac{1}{T_{clk}}) * 10000 \\ T_{eksperiment} &= 50291 * 10000 * \frac{1}{T_clk} \\ T_{eksperiment} &= 502910000 * \frac{1}{82000000} \\ T_{eksperiment} &= 6,13s \end{aligned}$$

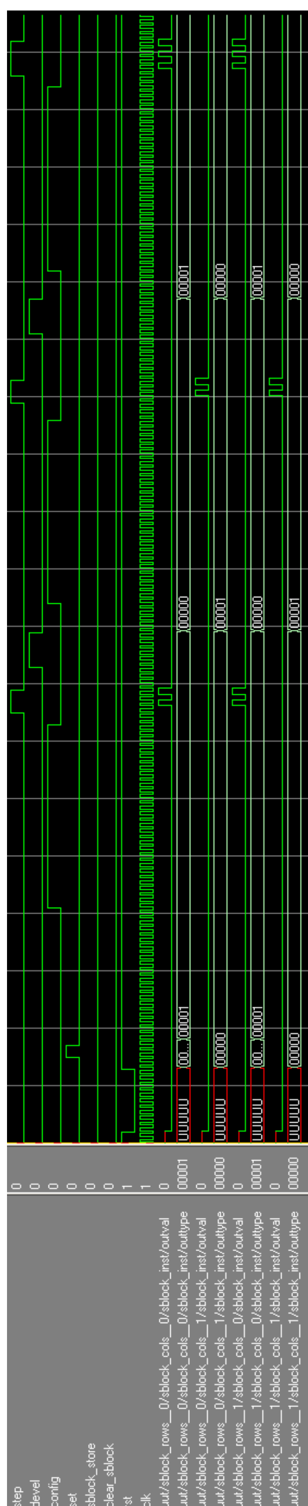
Til sammenligning er tabell 7.4 med kjøretider hentet fra [1] med ulik matrisestørrelse. Tidene er ikke direkte sammenlignbare med kjøretiden i formel 7.3, siden den tidligere kjøringen har med tid for å lese ut resultater etter hvert development-steg. Men, siden den nye implementasjonen har konstant kjøretid i forhold til dimensjonene på Sblock-matrisen vil det kun være utlesning av data som tar lenger tid ved økning av antall Sblocker. Utregning av forholdene i formel 7.4 viser at en 8x8 matrise er 5,1 ganger raskere på den nye implementasjonen, mens en 16x16 matrise er nesten 18 ganger raskere.

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

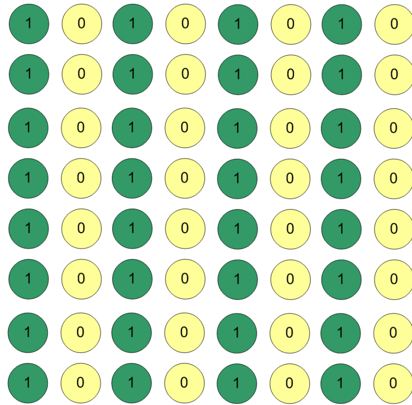
Tabell 7.4: Kjøretid for hastighetstest i gammel implementasjon

$$\begin{aligned} Forhold(8x8) &= \frac{T_{gammel}}{T_{ny}} \\ Forhold(8x8) &= \frac{31,3s}{6,13s} \\ Forhold(8x8) &= 5,1 \end{aligned} \tag{7.4}$$

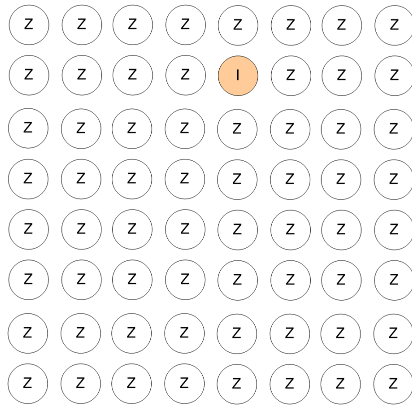
$$\begin{aligned} Forhold(16x16) &= \frac{T_{gammel}}{T_{ny}} \\ Forhold(16x16) &= \frac{106,5s}{6,13s} \\ Forhold(16x16) &= 17,37 \end{aligned}$$



Figur 7.4: Simulering av matrisen i ModelSim

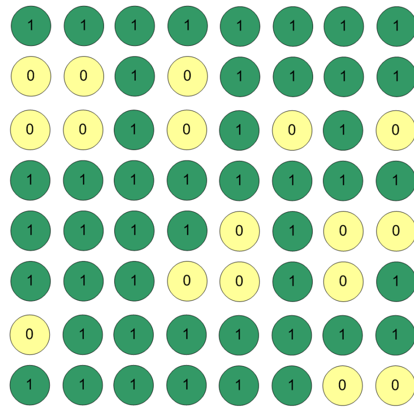


(a) Starttilstander

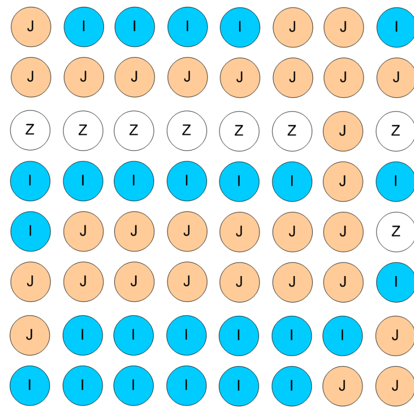


(b) Starttype

Figur 7.5: Startmatrise for funksjonell test

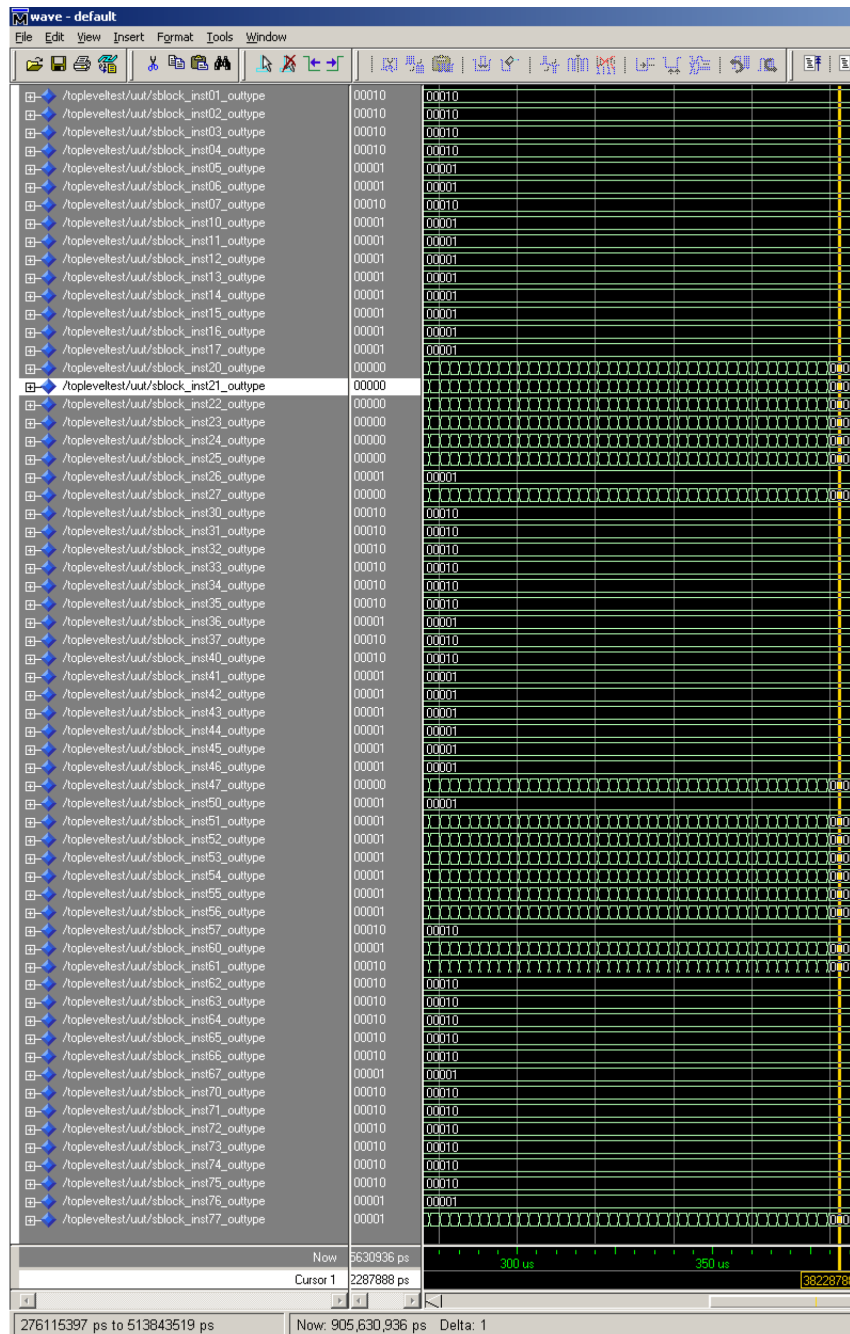


(a) Tilstander etter kjøring



(b) Typer etter kjøring

Figur 7.6: Sblock-matrisen etter funksjonell test



Figur 7.7: Typene til Sblock-matrisen etter post place and route-simulering

Kapittel 8

Diskusjon

I dette kapittelet vurderes resultatene av testene og arkitekturen. I tillegg diskuteres hva som gjenstår å gjøre med implementasjonen og mulige veier videre.

8.1 Testresultater

Testene som er utført tilsier at systemet fungerer som det skal. Det er ikke så mange tidligere resultater å sjekke mot, men at den funksjonelle testen fungerer uten at implementasjonen har samme funksjonalitet som den tidligere implementasjonen er lite sannsynlig.

Syntetisering viser at størrelsen på Sblock-matrisen har litt å si for klokkefrekvensen. Dette kommer sannsynligvis av at selv om alt er implementert så parallellt som mulig er det blant annet en del kontrollsignaler og busser som er globale. Hastighetsberegningen viser også at en størrelsen på matrisen ikke har noen innvirkning på tiden det tar for et utviklings-steg eller kjøring av matrisen, noe som betyr at matrisen skalerer godt.

Siden maskinvaren ikke ble tilgjengelig iløpet av arbeidet med denne oppgaven var det ikke mulig å kjøre tester med denne. Det er derfor mulig at systemet innholder feil som ikke har blitt oppdaget.

8.2 Arkitektur

Denne oppgaven har hatt som fokus å parallellisere Sblock-matrisen. Hver Sblock har fått sin egen development-prosess i tillegg til de funksjonelle komponentene. Selv om alt ble forsøkt parallellisert ble det gjort en vurdering som førte til at genomet ble plassert sentralt og delt av hver kolonne i matrisen.

Systemet har beholdt en del av instruksjonene fra [1], men noen instruksjoner har blitt

endret, lagt til og fjernet. Siden fokuset ikke lå på implementasjon av en ko-prosessor, men utprøving av en parallell developmentmodell, blir instruksjonene utført av en enkel flersykel-prosessor. Arkitekturen med internt instruksjonsminne i tillegg til utføring av instruksjoner fra PCI har blitt beholdt.

VHDL-koden er skrevet med tanke på lesbarhet. Det er mulig å optimalisere og redusere logikk i designet.

8.3 Videre arbeid

Det første som må gjøres er å teste ut designet på ekte maskinvare når denne blir tilgjengelig. Først da kan kommunikasjonen og utlesing av data fra matrisen over PCI-bussen testes i virkeligheten. Dette vil kreve en enkel modul for routing gjennom FPGAen på BenERA-kortet.

Det er flere områder som kan forbedres i designet. 1-er-telleren kan gjøres raskere, LUT'ene kan konfigureres parallellt og flere deler kan sannsynligvis optimaliseres. Den største mangelen i forhold til den tidligere implementasjonen er fitness-funksjonen. Det bør prioriteres å få en fungerende fitness-funksjon inn i systemet for å få tilsvarende funksjonalitet som i [1]. Det er heller ikke en veldig stor jobb å få de resterenede instruksjonene til å fungere.

BenBlue som er brikken systemet er laget for har 2 like FPGAer som kan brukes. Dette systemet benytter seg pr. idag bare av den ene FPGAen. Siden disse er koblet sammen med RocketIO bør det gå ann å utvide systemet for å bruke begge FPGAer uten å miste særlig ytelse. Siden dette fører til dobbelt så mange slicer vil Sblock-matrisen kunne bli nesten dobbelt så stor, med litt logikk for kommunikasjon mellom FPGAene.

Xilinx Virtex II-PRO FPGAen som brukes har to innebygde PowerPC-prosessorer. Dette er fullverdige mikroprosessorer med et turing-komplett instruksjonssett. Disse har lese- og skrive-tilgang til alt BRAM-minne på FPGAen og har mulighet for å styre kontrollsignalene til Sblock-matrisen. Dette kan i praksis gjøres ved å lage et "system on chip" ved hjelp av designverktøyet Embedded Development Kit (EDK) fra Xilinx. Denne prosessoren vil kunne kjøre fitness-funksjoner og også ha mulighet til å kjøre en GA hvis det er ønskelig.

Hvis det ikke er ønskelig å støtte bruk av PowerPC er det strengt talt ikke nødvendig å lagre all informasjon i BRAM. Da kan kontrolleren lese informasjon rett ut av Sblock-matrisen. Dette vil sannsynligis spare en del logikk, men krever redesign store deler av systemet.

Bibliografi

[1] Kjetil Aamodt. Kunstig utvikling: Utvidelse av fpga-basert sblock-plattform. Hovedoppgave, Norges teknisk-naturvitenskaplige universitet, Institutt for datateknikk og informasjonsvitenskap, juni 2005.

[2] P. J. Bentley, editor. *Evolutionary Design by Computers.* Morgan Kaufmann Publishers,Inc, San Francisco, USA, 1999.

[3] A. W. Burks. *Essays On Cellular Automata.* University of Illinois Press, 1970.

[4] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systemst and software. *ACM Computing surveys*, 34(2), june 2002.

[5] Charles Darwin. *On the origin of species.* John Murray, 1859.

[6] Asbjørn Djupdal. Konstruksjon av maskinvare for kjøring av sblokkbaserte eksperimenter. Hovedoppgave, Norges teknisk-naturvitenskaplige universitet, Institutt for datateknikk og informasjonsvitenskap, juni 2003.

[7] D.E. Goldberg. *GENETIC ALGORITHMS in search optimization & machine learning.* Addison Wesley, 1989.

[8] P.C. Haddow and G. Tufte. An evolvable hardware FPGA for adaptive hardware. In *Congress on Evolutionary Computation(CEC00)*, pages 553–560. IEEE, 2000.

[9] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

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

[11] IEEE. Standard 1149.1, boundry scan (jtag).

[12] H. Kitano. Designing neural networks using genetic algorithems with graph generation systems. *Complex Systems*, 4(4):461–476, 1990.

[13] A. B. Kahng og Y. Zorian M. Rodgers, D. Edenfeld. 2003 technology roadmap for semiconductors. *Computer*, 37(1):47–56, January 2004.

[14] K. Mainzer. *Thinking in Complexity The Computational Dynamics of Matter, mind, and Mankind, Fourth Edition*. Springer-Verlag, 2003.

[15] Daniel Mange, Moshe Sipper, André Stauffer, and Gianluca Tempesti. Toward self-repairing and self-replicating hardware : the embryonics approach. In *The 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 205–214, 2000.

[16] Nallatech. Dimetalk product brief. [http://www.nallatech.com/mediaLibrary/images/english/4063.pdf].

[17] Nallatech. Fuse product brief. [http://www.nallatech.com/mediaLibrary/images/english/2343.pdf].

[18] Nallatech. Benera user guide. Document number: NT107-0072, 2002.

[19] D. Wentzlaff og A. Agarwal. A quantitative comparison of reconfigurable, tiled, and conventional architectures on bit-level computation. *Field-Programmable Custom Computing Machines, FCCM 2004. 12th Annual IEEE Symposium on*, april 2004.

[20] J. V. Oldfield and R. C. Dorf, editors. *Field-Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. John Wiley and Sons Inc, 1995.

[21] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Pérez-Uribe, and A. Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In T. Higuchi, M. Iwata, and W. Liu, editors, *Evolvable Systems: from Biology to Hardware, ICES 96*, volume 1259 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 1996.

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

[23] M. Sipper, M. Goeke, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. The firefly machine: Online evolware. In *Proc. of 1997 International Conference on Evolutionary Computation (CEC97)*, pages 181–186. IEEE, 1997.

[24] Andre Stauffer, Daniel Mange, Gianluca Tempesti, and Christof Teuscher. Biowatch: A giant electronic bio-inspired watch. *eh*, 00:0185, 2001.

[25] Gianluca Tempesti, Daniel Mange, André Stauffer, and Christof Teuscher. The biowall: an electronic tissue for prototyping bio-inspired systems. Proceedings of the 2002 NASA/DOD Conference on Evolvable Hardware (EH.02), 2002.

[26] G. Tufte and P. C. Haddow. Towards development on a silicon-based cellular computation machine. *Natural Computation*, 4(4):387–416, 2005.

[27] G. Tufte and J. Thomassen. Size matters: Scaling of organism and genomes for development of emergent structurs. In *CODESOAR at Genetic and Evolutionary Computation (GECCO 2006)*, ACM Conference Proceedings. ACM, 2006.

[28] L. Wolpert. *Principles of Development, Second edition.* Oxford University Press, 2002.

[29] Xilinx. Xst user guide. [http://toolbox.xilinx.com/docsan/xilinx8/books/docs/xst/xst.pdf].

[30] Xilinx. Virtex-ii pro and virtex-ii pro x platform fpgas: Complete data sheet. [http://www.xilinx.com/bvdocs/publications/ds083.pdf], 2005.

[31] Xilinx. Platform flash in-system programmable configuration (proms). [http://direct.xilinx.com/bvdocs/publications/ds123.pdf], mai 2006.

Tillegg A

Filer og syntetisering

Dette vedlegget har en oversikt over alle filene som trengs for å syntetisere Sblock-matrisen og hva som er i de forskjellige filene. I tillegg beskrives hvordan systemet syntetiseres i Xilinx Project Navigator.

VHDL

Systemet er delt opp i ulike VHDL-filer på en hierarkisk måte og innholder en del VHDL-filer som er skrevet manuellt og noen filer som er generert av GoreGen som er en del av Xilinx ISE.

package.vhd

Denne modulen innholder en del konstanter og typer som brukes mange steder i systemet. Det er i denne filen størrelsen på Sblock-matrisen blir satt.

constraints.ucf

Denne filen bestemmer timing-krav og pin-konfigurasjon for systemet.

toplevel.vhd

Dette er toppnivå-filen i systemet. Denne har svært mye av routingen, og kobler blant annet sammen alle Sblockene i matrisen. I tillegg er det en del moduler som opprettes her.

- *matrixbram.xco*; Generert BRAM-modul som lagrer Sblock-matrisen

- *rulecounter.xco*; Generert teller som bestemmer hvilken regel som skal sjekkes
- *sblock_counter.xco*; Generert teller som bestemmer hvilken Sblock som skal få satt verdien sin og adresserer rulecounter.xco
- *sblock_decoder*; Generert dekoder som bruker sblock_counter for å finne riktig Sblock å sette

controller.vhd

Dette er kontroller-enheten. Tar seg av tolkning av instruksjoner og setter kontrollsignalene til Sblock-matrisen.

- *instr_store.xco*; Generert BRAM-intruksjonslager
- *com40*; Kommunikasjonsmodulen som tar seg av kommunikasjon med PCI-FIFO-bufferet. Denne filen er hentet fra [6]

rulestore.vhd

Dette er en wrapperfil for den genererte BRAM-modulen:

- *rulebram.xco*; BRAM som lagrer regler

sblock.vhd

Denne filen beskriver en Sblock med funksjonalitet og development-prosess.

- *devunit.vhd*; Regelsjekkeren, hentet fra [6]
- *type2lutconfig*; Generert distribuert minne som lagrer mapping mellom Type og LUT-konfigurasjon

Syntetisering

Syntetisering gjøres i Xilinx Project Navigator. Dette gjøres ved å åpne prosjektfilen sblock2006.ise. Deretter velger man toplevel-filen, og velger den prosessen man ønsker fra prosessvinduet (for eksempel syntetisering eller generering av bit-fil).

Tillegg B

Instruksjonsmanual

Dette vedlegget er hentet fra [1] og oppdatert med nye instruksjoner som er endret eller kommet til under arbeidet med denne oppgaven. Alle nye instruksjoner er merket med '*', endrede 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.

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

? config

Konfigurer SBlockene i matrisen med korrekt LUT-verdi i henhold til typen. Dette må gjøres hver gang en Sblock har endret typen sin.

ubrukt	0	00111
31–6	5	4–0

* customClear

Setter den interne telleren til 0.

ubrukt	0	11100
31–6	5	4–0

* customInc

Øker den interne telleren med 1.

ubrukt	0	11101
31–6	5	4–0

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

end

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

ubrukt	0	01111
31–6	5	4–0

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

? jumpEqual

Hopper til angitt posisjon når antall kjørte development-step tilsvarer gitt verdi eller når den andre telleren har en gitt verdi. Hvilken teller som brukes angis av bit 17.

adresse	ubrukt	teller	verdi	ubrukt	0	10110
31–24	23–18	17	16–8	7–6	5	4–0

- *verdi*; Antall development-step som skal ha blitt kjørt før instruksjonen hopper.
- *teller*; 0 er development-telleren, 1 den andre.
- *adresse*; Hopp til denne adressen i instruksjonslageret

nop

Ingen operasjon.

ubrukt	0	00000
31–6	5	4–0

readback

Les tilbake tilstandsdata fra SBlockmatrisen og lagre disse i BRAM.

ubrukt	0	01000
31–6	5	4–0

? readType

Les en enkelt SBlocks informasjon fra BRAM og send den til PCI-bussen. De 5 minst signifikante bit er type, det 6. er tilstand.

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

? readSums

Leser ut verdier lagret ved å kjøre storeSum.

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

- *antall*; Antall verdier som skal leses

resetDevCounter

Nullstiller registeret som teller antall development som har blitt kjørt.

ubrukt	0	10111
31–6	5	4–0

run

Kjør SBlockmatrise.

sykler	ubrukt	0	01001
31–8	7–6	5	4–0

- *sykler*; Antall sykler SBlockmatrisen skal kjøres

? readUsedRules

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

regeldel	ubrukt	0	10101
31–30	29–6	5	4–0

- *regeldel*; De 32 første, neste eller siste reglene.

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.

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

* storeSum

Teller opp alle 1'ere i Sblock-matrisen.

ubrukt	0	11011
31–6	5	4–0

writeLUTConv

Skriv til LUT konverteringstabell. Denne tabellen benyttes for å oversette et gitt SBlocktypenummer 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

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 C.
- *nummer*; Nummer på regel. Høyt nummer betyr høy prioritet. Hver regel må ha sitt unike nummer, og prioritet mellom regler er dermed entydig gitt.

writeType

Skriv en enkelt SBlocks type og tilstand til BRAM.

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

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

Tillegg C

Regelformat

Vedlegget er hentet fra [6].

Regler beskrives internt på samme måte som de kodes i *writeRule*-instruksjonen.

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.
- *Resultat*; Hva som skal skje dersom denne regelen slår til.

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.

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 D

Kommunikasjonsprotokoll

Vedlegget er hentet fra [6].

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

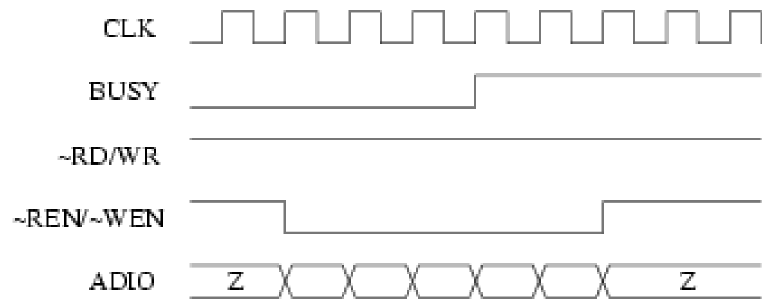
- *AS/˜DS*: Høy: adresse sendes på ADIO, lav: data sendes på ADIO. Dette signalet sier hvilken dataordtype som sendes over ADIO. To typer finnes: Adresse og data. Bruker-FPGA-en og programvaren som sender disse bestemmer hvordan disse skal tolkes.
- *EMPTY*: FIFO-buffer er tomt
- *BUSY*: FIFO-buffer 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.

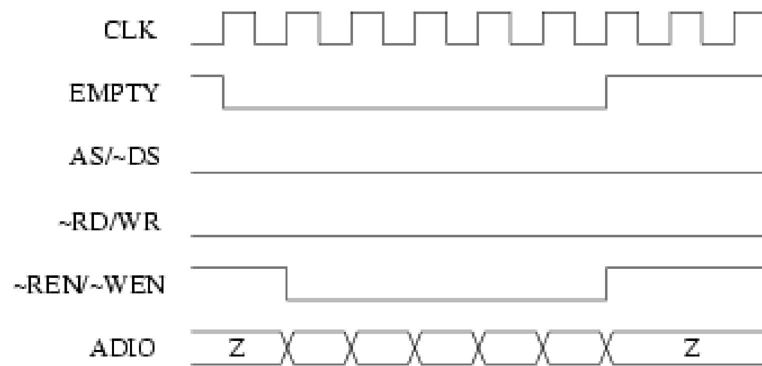
Klokken må gå på 33MHz–40MHz. Dette betyr at kommunikasjonshastigheten begrenses av hastigheten på CPCI-bussen som har en øvre teoretisk grense ved 132MB pr. sekund.

Figur D.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 D.2 viser et eksempel på lesing fra FIFO-buffer.

Se brukermanualen til BenERA [18] for mer informasjon.



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



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