NTNU
Det skapende universitet

Quality of Service for Network on Chip

Ivar Ersland

Master i elektronikk
Oppgaven levert: Juni 2009
Hovedveileder: Kjetil Svarstad, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Det er utviklet en rutermodul med QoS-støtte for bruk i en mesh QNoC-arkitektur. Ruteren buffrer hele 8 byte pakker, med to ulike prioritetsnivåer. Videre utfører ruteren en enkel arbitrering av inngangene og ruter videre etter den deterministiske x-y-algoritmen. Nettverket er testet for små pakkerater.

Oppgaven består i å videreutvikle denne modulen slik at den bedre kan håndtere store pakkerater. Dette består i å forbedre arbiteringen, samt vurdere adaptive rutingalgoritmer. Det sees her spesielt på muligheten for å unngå deadlock i nettverket.

Videre skal nettverket integreres mot en Microblaze soft-prosessor som kjører Linux-kjerne og testes på et Suzaku-S-kort utviklet av Atmark Techno INC.Det skal gjøres en vurdering av nødvendig båndbredde og maksimal ressursbruk for nettverket basert på trafikk-analyser av en eller to applikasjoner.

Oppgaven gitt: 15. januar 2009
Hovedveileder: Kjetil Svarstad, IET

SAMMENDRAG

Moderne enbrikkesystemer bygges i større og større grad opp av heterogene strukturer der ulike applikasjoner med forskjellige oppgaver kommuniserer med hverandre. Strenge krav stilles til pålitelighet i form av korrekthet, enkelhet, robusthet, stabilitet, rettferdighet og optimalitet i slike systemer. Network on Chip(NoC)'s viktigste oppgave er å skjule kommunikasjonsprotokollen for de ulike applikasjonene slik at de kan utvikles uavhengig av brikkens infrastruktur. Videre bidrar NoC til å møte strenge tidsfrister i sanntidssystemer. Arbeidet viser at NoC er spesielt anvendbart på FPGA, der infrastruktur er en begrenset ressurs. Videre kan NoC bedre utnyttelse av dynamisk rekonfigurering.

Undersøkelser viser at en nettverksarkitektur med SAF-svitsjing og to virtuelle kanaler per ruter gir gode resultater for båndbredde og QoS. QoS med to prioritetsnivåer oppnås ved at hver ruter arbitrerer på en inngangskø samt at alle pakker til en hver tid er fullstendig buffret i en ruter i nettverket.

Syntese til Xilinx Spartan-3 FPGA viser at nettverket med 4x4 mesh-topologi gir en teoretisk båndbredde på $1,3 GBit/s$. Videre viser analyser at nettverket er anvendbart på et antall FPGA'er og med en rekke ulike applikasjoner. Arbeidet har vist at QoS for NoC er både fordelaktig og oppnåelig i moderne enbrikkesystemer.

FORORD

Denne rapporten er levert som en del av Sivilingeniørstudiet i Elektronikk ved NTNU i Trondheim. Oppgaven er utført våren 2009 under institutt for Elektronikk og Telekomunikasjon. Som veileder har Kjetil Svarstad hvert til stor hjelp under arbeidet.

INNHOLD

Forkortelser

ALU - Arithmetic Logic Unit *Aritmetisk-logisk enhet*
APSRA - APplikasjonsSpesifikk RutingAlgoritme, *Eng: Application Spesific Routing Algorithm*
CRC - Cyclinc Redundancy Check.
CTS - Clear To Send
Buffre - Lagre informasjon i lokalt minne
CLB - Configurable logic block. *Norsk: Konfigurerbar logisk blokk*
FF - Flip Flop, minneselle
FPGA - Field Programmable Gate Array
FSM - Tilstandsmaskin *Finite State Mashine*
GALS - Globalt Asynkron, Lokalt Synkron, *Eng: Globaly Asyncronous Localy Synkronous*
HW - Maskinvare(Eng: Hardware)
IC - Integrated Circuit. *Norsk: Integrert krets.*
IIT - Israel Institute of Technology
MIT - Massachusetts Institute of Thechnology
NoC - Network on Chip
QoS - Quality of Service
RT - Real Time. *Sanntid*
RTR - Run Time Reconfiguration. *Norsk: Kjøretids-rekonfigurering.*
RTS - Request To Send
SAF - Store-And-Forward
SW - Programvare, *Eng: Software*
VCT - Vitual-Cut-Trough
VHDL - (Very High Speed Integrated Circuits) Hardware Description Language
WH - Wormhole

FIGURER

TABELLER

KAPITTEL 1

INTRODUKSJON OG MOTIVASJON

1.1 Enbrikkesystemer møter det 21 århundre

I 1965 skrev den amerikanske forskeren Goordon E. Moore en artikkel publisert i magasinet *Electronics* hvor han hevder at antall transistorer integrert på en enkelt brikke vil dobles vært andre år[4]. Moores prediksjon holder fremdeles og i dag, 44 år senere, er over 1 milliard transistorer integrert på en enkelt integrert krets. Med økende antall tilgjengelige transistorer oppstår imidlertid nye utfordringer. Syntese og kompilatorteknologier klarer ikke å utnytte alle de tilgjengelige ressursene og dette gapet, kalt *produktivitetsgapet*, betyr at det vil være behov for flere og flere designteam[5].

En grunnlegende metode for å begrense produktivitetsgapet er ved gjenbruk av maskinvaremoduler. Mer og mer komplekse komponenter fra transistorer til avanserte ALU'er har blitt byggeblokker i små og store systemer. Ved å flytte seg oppover i nivå ettersom antall tilgjengelige transistorer øker, prøver designerne å holde følge med IC-produsentene og begrense produktivitetsgapet[6].

1.2 Rekonfigurerbar maskinvare(HW)

En nøkkelfaktor for å kunne bevege seg oppover i nivå ved gjenbruk av moduler ligger i innføring av rekonfigurerbar HW. Rekonfigurerbar HW er enheter der konfigurasjonen av logiske porter og deres sammenkobling kan forandres etter at brikken er produsert. Ved å kunne skrive maskinvarekonfigurasjon flere ganger til samme enhet kombineres maskinvarens korte beregningstid med programvarens fleksibilitet.

Moderne rekonfigurerbare enheter, som for eksempel FPGA, gjør det mulig å rekonfigurere delvis(partielt) under kjøring(RTR) av maskinvare. På denne måten kan en del av systemet utføre en beregning eller annen oppgave mens en annen del rekonfigureres for å håndtere nye oppgaver [7].

1.3 Network on Chip i et enbrikkesystem

En utfordring ved bruk av FPGA er at de interne koblingene typisk er veldig finkornet og dermed krever store ressurser. Når en brikke delvis rekonfigureres brytes tilkoblingen til de resterende maskinvaremodulene noe som gjør det vanskelig å koble den nye enheten til på samme måte som den gamle[5]. Så, hvordan kommuniserer de ulike maskinvaremodulene med hverandre når de samtidig byttes ut med uregelmessing periode?

En løsning med økende popularitet er integrert Network on Chip. NoC definerer en kommunikasjonsprotokoll der de ulike maskinvaremodulene kommuniserer med hverandre ved å sende pakker til hverandre over et internt nettverk. Dette gjør det enklere å bytte ut moduler ved å rekonfigurere deler av FPGA'en. I et nettverk av rutere der hver maskinvaremodul har en tilhørende ruter, slipper funksjonelle maskinvaremoduler å håndtere kommunikasjon. Trenger en funksjonell modul(for eksempel filter, flash controller eller ALU) å sende informasjon til en annen, genererer den ganske enkelt en pakke med den riktige destinasjonsadressen og lar nettverket ta seg av levering av pakken.

Fleksibilitet og plattformuavhengighet gjør det integrerte nettverket til en nøkkelegenskap for å gjøre det mulig å utnytte over 1 milliard transistorer integrert i et enbrikkesystem. Arbeidet presentert i denne rapporten tar for seg design og evaluering av et slikt nettverk. Spesielt vektlegges vurdering av ulike bufferstørrelser og innføring av prioritetsnivåer i et NoC. Prioritetsnivåer gjør at prioriterte pakker vil få tilgang til nettverket før lavere prioriterte pakker. Innføringen av prioritetsnivåer med garantert båndbredde kalles ofte Quality of Service(QoS) og er en viktig egenskap for å oppnå et pålitelig nettverk. Egenskapen kan også sies å være spesielt viktig i enbrikkesystemer der man ofte har kritiske tidsfrister i forbindelse med RT-oppgaver.

Utgangspunktet for arbeidet er et nettverk utviklet som prosjektoppgave høsten 2008. Det antas at dette nettverket kan videreutvikles til å håndtere store pakkerater og gi god QoS. Oppgaven utføres under AHEAD-prosjektet, et sammarbeid mellom institutt for elektronikk og telekommunikasjon og Sintef i Trondheim.

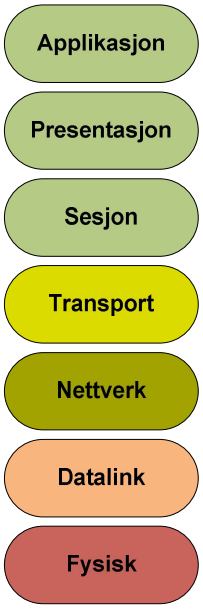
Rapporten tar først for seg teorien bak emnet NoC(kap 2). Videre presenteres tidligere arbeider som vurderes som viktige innenfor temaet(kap 3). I kapittel 4 vurderes ulike teknikker for å møte spesifikasjonene beskrevet over, før det konkluderes med en løsning. En maskinvareimplementasjon av den valgte løsningen er presentert i kapittel 5. For å analysere nettverket presenteres en rekke simuleringer og resultater i kapittel 6 sammen med en vurdering av nettverkets ytelse. Videre foreslås en metode for å teste nettverket

på FPGA(kap 7) før designet evalueres sammen med forslag for videre studier(kap 8).

KAPITTEL 2

TEORI

Dette kapittelet gir en kort presentasjon av teorien bak nettverk for FPGA. Dette omfatter presentasjon av populære topologier, svitsjestrategier, algoritmer for ruting samt metoder for design av pålitelige og stabile nettverk.



Figur 2.1: OSI-modellen, de 7 abstraksjonslagene

Network on Chip(NoC) eller nettverk for enbrikkesystemer tar for seg konsepter for store nettverk(fjernnett), skalerer dem ned og integrerer dem på en enkelt brikke, her i form av en FPGA. NoC er et nettverk av noder organisert i en topologi der hver node består av en ruter og en funksjonell modul. Den funksjonelle modulen er typisk en applikasjon som utfører en oppgave, mens ruteren sørger for at applikasjonens respons kan sendes i form av en melding. Meldigen kan da transporteres til andre applikasjoner på brikken eller til periferier utenfor brikken. NoC'er er forskjellige fra fjernnett ved at de må ta hånd om strengere krav slik som energiforbruk, lav kommunikasjonsforsinkelse og ressursforbruk. For å oppnå dette må NoC'er håndtere alle de 7 lagene i OSI-modellen vist i figur 2.1.

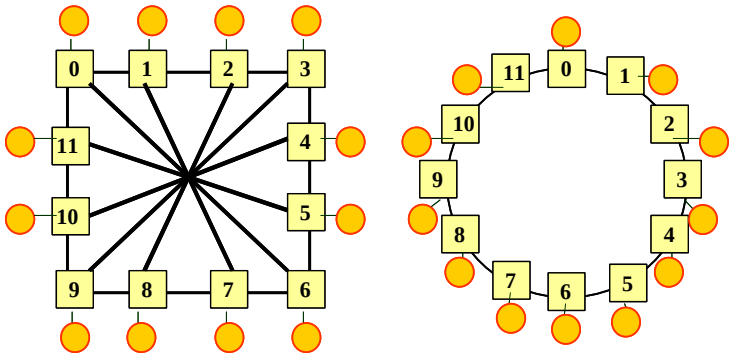
Det nederste laget i OSI-modellen, kalt **fysisk lag**, tar for seg mekaniske og elektriske egenskaper ved arkitekturen. Dette inkluderer forsinkelse i ledere mellom to punkter, *set* og *reset* av register og andre fysiske egenskaper. Med en FPGA som målplattform(se kapittel 2.11) vil disse faktorene tas hånd om av synteseverktøyet for den aktuelle FPGA'en. En FPGA inneholder programmerbar logikk, kalt logiske blokker, samt et hierarki av rekonfigurerbare sammenkoblinger. Register og kombinatorikk realiseres på en FPGA i form av en rekke CLB'er der hver CLB består av et antall LUT'er(oppslagstabell) og FF'er(minneselle) i tillegg til ulike former for logikk. Videre kan en FPGA rekonfigureres ved at de logiske blokkenes funksjonalitet samt deres sammenkoblinger forandres. Nettverk for enbrikkesystemer har i de siste årene fått betydelig oppmerksomhet som en løsning for på sammenkoblingsproblemet i komplekse enbrikkesystemer. Hovedgrunnen til dette er at NoC bidrar til løsning av elektriske problemer for $>1\mu m$-teknologier(submicron) på grunn av at NoC strukturerer og administrerer globale ledere[8],[9],[10].

Datalinklaget regulerer tilgang til fysiske medium og avgjør konflikter i form av arbitrering. I NoC vil dette typisk bety en beslutning om hvem som skal få tilgang til en ressurs i form av en link eller kanal. Dette er nærmere presentert i kapittel 2.7.

Nettverks- og transportlaget en ansvarlig for svitsjing(kap 2.2) og ruting(kap 2.3) av datapakker igjennom nettverket. Svitsjeteknikker brukes til å etablere en sammenkobling mens ruting bestemmer en meldings bane igjennom nettverket fra en kilde til en destinasjon.

De tre øverste lagene, **applikasjons, presentasjons og sesjonslaget** er programmvarelag som definerer langvarige kommunikasjoner mellom noder samt presentasjon og kryptering av meldinger.

2.1 Nettverkstopologier

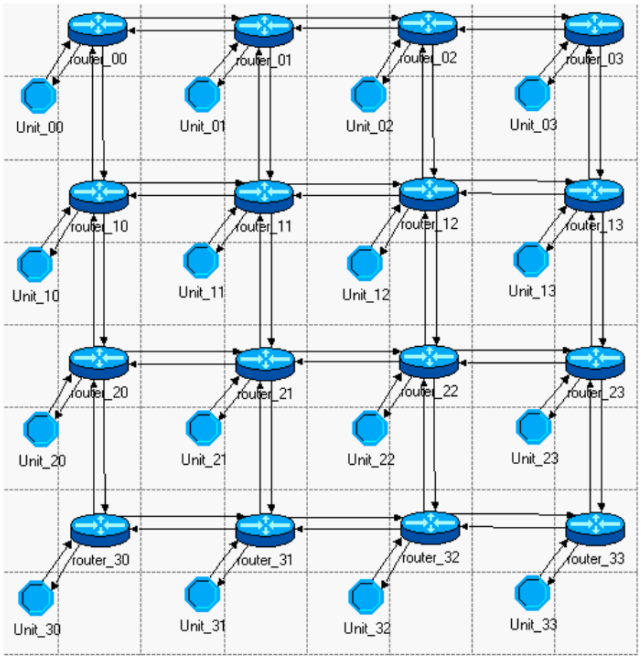


Figur 2.2: a) Spidergon b) Ring ,[11]

For å utnytte ruterne maksimalt er det viktig at de blir koblet sammen på en effektiv måte. Dette kan oppnås ved å organisere ruterne i noder eller knutepunkter. Topologien til et nettverk definerer nodenes fysiske organisering i forhold til hverandre samt hvordan de er koblet sammen. Topologiene klassifiseres som direkte, indirekte eller irregulære.

Nabonode: *En nodes naboer er de noder som har direkte tilkobling til denne noden. Data ankommer en node i samme rekkefølge som de ble sendt ut av sin nabo.*

I **direkte topologier** har hver ruter en direkte kobling til sin nabonode igjennom en fysisk link. De fleste direkte nettverk har en ortogonal implementasjon, hvor nodene organiseres i et $n-$dimensjonalt ortogonalt rom. Direkte topologier som for eksempel mesh(figur 2.3), Spidergon, Ring(figur 2.2) og Torus er foreslått av ulike instanser. Ring-nettverket(figur 2.2b) er en av de enkleste topologiene for NoC men topologien er lite brukt på grunn av dårlig skalerbarhet. En ring av noder gir enkel ruting men nettverkets evne til å håndtere trafikk vil minke når man tilfører flere noder i nettverket. Alternativt kan vi tenke oss en stjernekobling av noder. Dette vil gi lik avstand mellom alle nodene men denne topologien lider naturlig nok av at all trafikk må gå igjennom sentrum av nettverket. STMicroelectronics har videreutviklet denne topologien ved å koble sammen nabonoder slik figur 2.2a viser. Denne topologien har vist god ytelse men er også relativt kompleks å implementere på grunn av kompleks ruting[11].



Figur 2.3: 2D Mesh,[12]

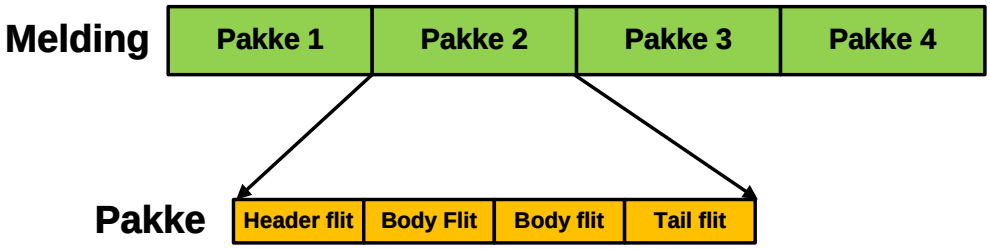
Den effektive 2D mesh-topologien (figur 2.3) er den mest vanlige i moderne NoC. Hver node har et antall naboer(maksimalt 4) samt en lokal funksjonell modul. Ruterne vil få lik nærhet til alle sine naboer og hver node kan dermed bare kommunisere direkte med sin nabo. Informasjon kan imidlertid formidles videre av en ruter i nettverket slik at alle nodene kan kommunisere indirekte med hverandre ved at ruteren videresender informasjon som ikke er ment for denne noden. Samtidlig skalerer mesh-nettverkets areal lineært med antall noder i nettverket noe som letter det fysiske designet av nettverket. Som vil skal se i kapittel 4.1 vil også mesh-topologier være meget fordelaktige når de brukes i kombinasjon med dynamisk rekonfigurering. I en utvidet versjon av mesh-topologien kommuniserer nodene på kolonnen ytterst til høyre i nettverket direkte med nodene ytterst til venstre i nettverket. Topologien kalles da Torus.

Indirekte nettverkstopologier omfatter topologier der hver node er koblet til en ekstern svitsj, og der denne svitsjen er koblet videre til andre svitsjer via punkt-til-punkt-linker. I stedet for å ha rutere, integreres hver applikasjon med et nettverksgrensesnitt for kommunikasjon med svitsjene[13]. Balkan *et al.*[14] med sitt *mesh-tre* og SPINnettverket(se kapittel 3.4) med sitt *Fat-tree* har utforsket mulighetene som ligger i indirekte nettverk.

Irregulere- eller ad-hoc-topologier forsøker å øke den tilgjengelige båndbredden ved å tilpasse nettverket bedre til den eller de applikasjonene som skal benytte nettverket. Et enkelt eksempel er å fjerne enkelte noder i en mesh-topologi for å gi plass til applikasjoner som krever store ressurser. Xpipes(se kapittel 3.5) og Æthereal(se kapittel 3.1)

tillater irregulære topologier mens Holsmark, Palesi og Kumar undersøker rutingmetoder i irregulære topologier[15], [13].

2.2 Svitsjestrategier



Figur 2.4: En melding deles opp i fire pakker

I et NoC avgjør svitsjestrategier hvordan data flyter igjennom ruterne. Svitsjestrategier avgjør kornethet til en dataoverføring samt anvendt svitsjeteknikk[13],[16]. Meldinger som skal sendes over nettverket deles inn i datapakker som igjen deles inn i et antall flit'er der en strø av pakker fra en kilde til en destinasjon kalles en flyt[17]. Størrelsen på en pakke er som oftest en fastsatt størrelse og meldinger som inneholder mer informasjon enn det som det er plass til i en pakke vil bli delt opp i et antall pakker. Figur 2.4 viser et eksempel på en melding som deles opp i fire pakker der hver pakke igjen deles i fire flit'er. De to hovedmodusene for sending av en slik pakke fra en avsender til en mottaker kalles kretssvitsjing og pakkesvitsjing.

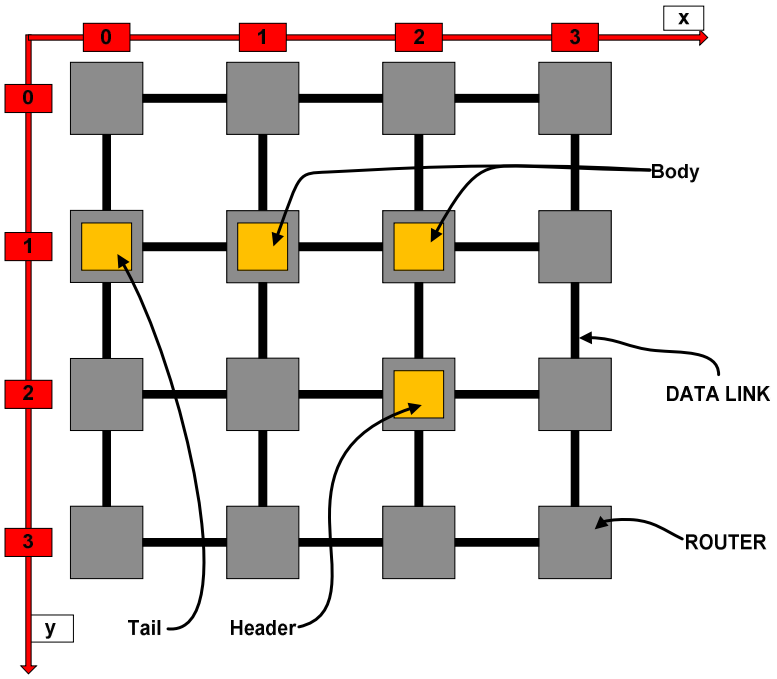
I kretssvitsjing settes en direkte fysisk rute opp mellom avsender og mottaker. Den fysiske ruten kan bestå av en serie rutere og datalinker og når en fysisk rute er allokert til en pakke sendes den i sin helhet over denne ruten før ruten blir lukket. Denne direkte linken mellom sender og mottaker gjør at pakken kan benytte hele linkens båndbredde samtidig som dette gir veldig lav forsinkelse etter at ruten er satt opp. Kretsvitsjing gir imidlertid dårlig skalerbarhet for et NoC. Som nevnt tidligere er ledere den mest begrensede ressursen i en FPGA. I og med at kretssvitsjing krever en direkte rute mellom alle nodene i nettverket vil et nettverk som bruker kun kretssvitsjing skalere eksponentielt. I et nettverk av 16 noder tregs det altså $16 \cdot 15 = 240$ linker for å koble alle nodene sammen med alle andre noder.

I pakkesvitsjing settes det opp en fast link mellom en ruter og dens naborutere istedenfor å sette opp en link til alle andre rutere i nettverket. En pakke transmitteres så langs denne oppsatte ruten til neste ruter før denne ruteren igjen setter opp en ny rute i nettverket.

En slik overføring av en pakke fra node til node i nettverket kan utføres ved at pakken

sendes flit for flit over en fysisk datalink. Størrelsen på denne datalinken avgjør flit'ens størrelse slik at en bussbredde på 8 bit gir en flitstørrelse på 8 bit. Med dette kan pakken fra figur 2.4 sendes over datalinken i fire steg, der header'en sendes først før de resterende tre flit'ene følger fortløpende i rekkefølge. Tre ulike strategier, kalt Wormhole(WH), Store And Forward(SAF) og Virtual Cut Trought(VCT) er blitt populære for å utføre en transmisjon. De tre strategiene forklares under.

2.2.1 Wormhole(WH) svitsjing



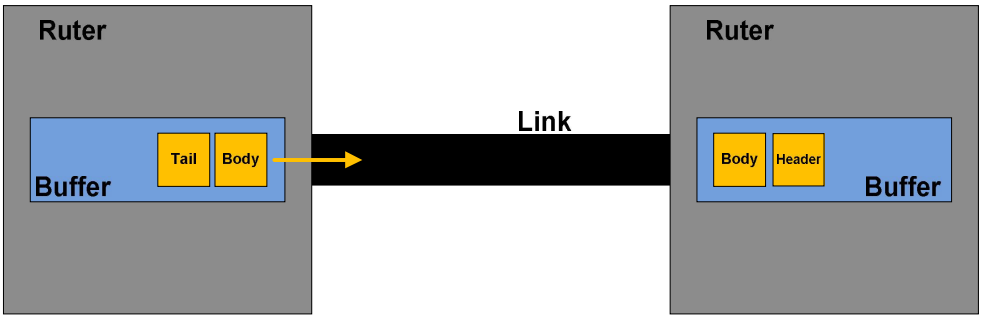
Figur 2.5: Wormhole-svitsjing i et 4x4 mesh

Wormhole svitsjing er en metode for ruting av en pakke der en *header-flit*(se figur 2.4) setter opp en rute igjennom nettverket og de resterende flit'ene i pakken følger etter i en kø. Pakkens *body-flit'er* er da bundet til denne ruten i den rekkefølgen de blir transmittert av avsenderen. Figur 2.5 viser hvordan pakken fra figur 2.4 vil traversere igjennom et mesh-nettverk ved bruk av WH-svitsjing. En pakke vil bre seg som en orm igjennom nettverket og opptar da en kontinuerlig rekke med rutere lik pakkens størrelse i antall flit'er[18]. Når alle flit'ene er rutet igjennom er ruter, lukker *tail-fliten* ruten og ruteren er klar til å håndtere nye pakker. En header som prøver å få tilgang til en ruter som er opptatt med ruting av en annen pakke vil bli blokkert, noe som igjen fører til at de etterfølgende flit'ene i pakken blir blokkert langs den oppsatte ruten[19],[13].

Ved å la hver ruter bare buffre èn flit om gangen kan man oppnå meget arealeffektive

rutere samtidig som forsinkelsen i nettverket vil være uavhengig av pakkens reiselengde. Som vi ser av figur 2.5 vil imidlertid de små bufferne føre til at pakken får en lang hale etter seg. Denne halen vil blokkere all annen trafikk på tvers av og forbi den aktuelle pakken noe som gjør nettverket sårbart for vranglås og opphopninger.

2.2.2 Store-And-Forward(SAF) svitsjing



Figur 2.6: Store-And-Forward-svitsjing

Ved å øke bufferstørrelsen i hver ruter kan pakkens hale reduseres til bare å omfatte to rutere. Med en satt pakkestørrelse lar vi hver ruters buffer være like stort eller større enn denne pakkestørrelsen. Mer presist kan vi si at pakkestørrelse vil være begrenset av ruternes bufferressurser og en pakke kan ikke være større enn det minste bufferet den skal innom. På denne måten kan en ruter buffre en hel pakke før den rutes videre og med dette unngår man at pakken brer seg ut over mange rutere i nettverket. Som vi ser av figur 2.6 transmitteres flit'ene en og en over en datalink fra en ruter til en annen helt til alle flit'ene er transmittert. Når hele pakken er mottatt hos en ruter setter så denne ruteren opp en ny rute til neste ruter hvor pakken så transmitteres videre på samme måte. Ved å benytte denne metoden sørger man for at alle pakker i nettverket til en hver tid vil være fullstendig buffret i minst en ruter noe som gir pålitelighet og stabilitet i nettverket. Når en bane er satt opp mellom to rutere sendes hele pakken på en enhetstid som vil være tilnærmet lik for alle pakker[18]. Dette gjør det enklere å avgjøre hvor lang reisetid en pakke vil få, noe som vil være spesielt gunstig i forbindelse med sanntidsapplikasjoner.

2.2.3 Virtual-Cut-Through(VCT) svitsjing

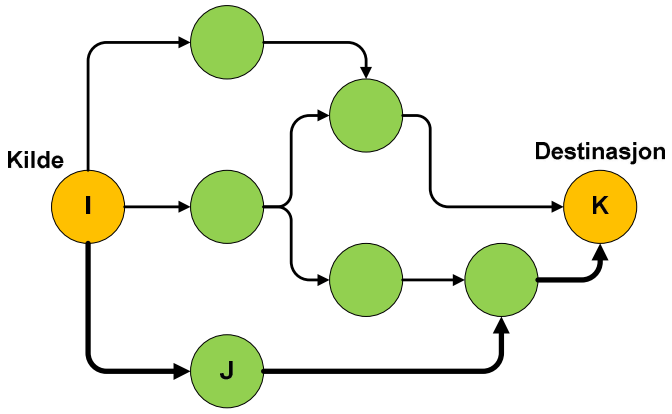
En teknikk for å redusere forsinkelsen i en ruter som følge av den tiden det tar å buffre en hel pakke kalles Virtual Cut Through og ble introdusert av Kermani og Keinrock[19]. Med denne teknikken rutes en flit videre til neste ruter langs pakkens vei så lenge det er plass til hele pakken hos neste ruter. Dette krever naturlig nok at alle ruterne har plass

til å buffre minst en hel pakke. I forhold til SAF oppnår man at den gjennomsnittlige reisetiden for en pakke reduseres på grunn av at flit'er ikke vil stoppe opp og vente på resten av sin pakke så lenge det er plass til pakken i neste ruter på veien.

2.3 Rutingalgoritmer

Nettversklagets hovedoppgave er å rute pakker fra en kilde til en destinasjon. Rutingalgoritmen er den delen av nettverkslaget som er ansvarlig for å avgjøre hvilken utgang en pakke som ankommer en ruter skal rutes til. Mer presist kan vi si at rutingalgoritmen spesifiserer en protokoll for sending av m meldinger langs n mulighet baner[18]. Til forskjell fra ruting i et fjernnettverk har ikke ruterne i et NoC noen programvaremodul for håndtering av komplekse rutingalgoritmer. Algoritmer som for eksempel *bredde først søk* og *Dijkstra's korteste vei*[20] vil med andre ord medføre en uforholdsmessig stor kostnad som rene maskinvaremoduler. Krav stilles til rutingen i form av korrekthet, enkelhet, robusthet, stabilitet, rettferdighet og optimalitet[17].

Videre kan rutingalgoritmer generelt deles inn i deterministiske og adaptive. I en deterministisk(statisk), eller glemsk, ruter settes en pakkes bane opp utelukkende på grunnlag av avsender- og destinasjonsadresse. Det vil si at parametre som varierer utover i et nettverks levetid, som for eksempel opphopning, ikke vil bli tatt hensyn til. Noe mer komplekst vil en adaptiv(dynamisk) ruter også ta hensyn til dynamiske parametre og dermed kunne tilpasse pakkens videre rute etter det[19].



Figur 2.7: Optimal rute fra kilde til destinasjon

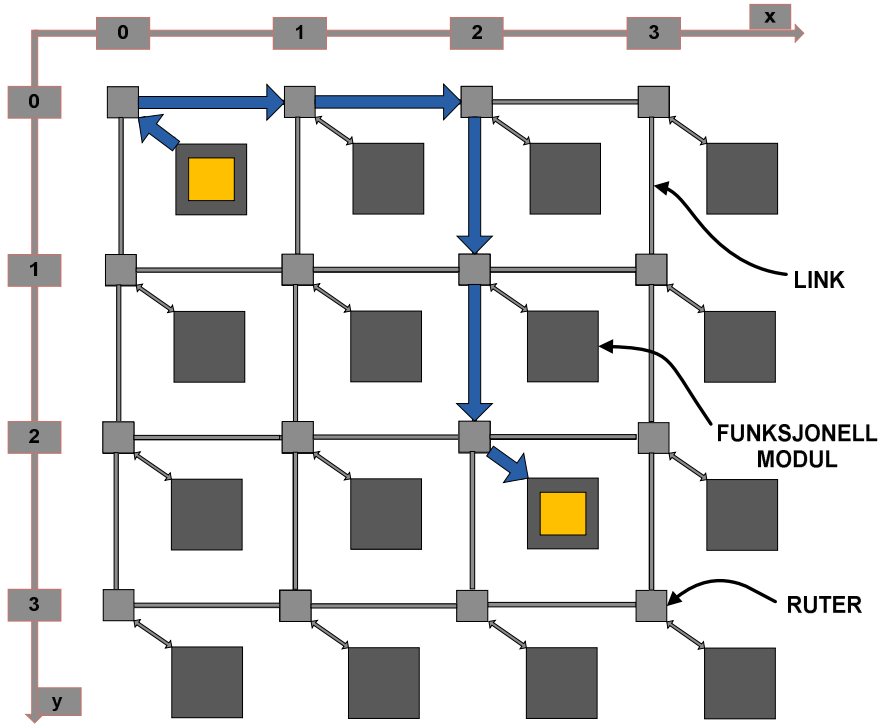
Ideelt sett finnes det en optimal rute for alle pakker i ett NoC. Rutingalgoritmens oppgave blir da å sørge for at så mange pakker som mulig følger sin optimale rute. Med en optimal rute menes her den ruten som gir den korteste totale reisetiden for alle pakker i nettverket og som samtidig møter de tidsfrister som eventuelt eksisterer. Av dette følger at den mest optimale ruten ikke nødvendigvis er den korteste men den som gjør at færrest pakker

krysser hverandres baner eller forsinker hverandre. Figur 2.7 viser et antall ruter mellom en kilde og en destinasjon der den nederste ruten her som et eksempel er markert som den optimale ruten mellom de to nodene.

Optimaliseringsprinsippet: *Hvis en ruter J befinner seg langs den optimale ruten mellom to andre rutere I og K så vil den optimale ruten fra J til K også falle langs den samme ruten(fig 2.7).*

Som en konsekvens av optimaliseringsprinsippet følger at et antall optimale ruter kan eksistere men at disse vil følge en rettet asyklisk graf med avsender i en ende og mottaker i den andre. Videre ser vi at den optimale ruten ikke kan inneholde noen sykler da en rute som ikke inneholder denne sykelen vil være mer optimal. Grafteorien er beskrevet i [20].

2.3.1 X-Y-algoritmen



Figur 2.8: Ruting av en pakke fra node (0,0) til (2,2) ved bruk av x-y-algoritmen

På grunn av sin enkelhet er den deterministiske x-y-algoritmen brukt i en rekke ulike applikasjoner. Figur 2.8 viser ruting av en pakke i et 2D mesh nettverk ved bruk av x-y-algoritmen. Nettverket settes inn i et koordinatsystem der hver ruter representeres med sine (x, y) koordinater i nettverket. Som figuren viser, vil en pakke rutes først i x-retning

og så i y-retning. Med andre ord rutes pakken først langs avsenders rad til den kommer til mottakers kolonne før den rutes langs denne kolonnen til den ankommer mottakers rad. Dette betyr at hver pakke gjør maksimalt en sving i nettverket og at denne vil komme når riktig rad møter riktig kolonne. Algoritmen er enkel å implementere, og brukt i kombinasjon med mesh-topologien garanterer den at pakken alltid vil ta korteste vei fra avsender til mottaker[19],[18]. Vi merker oss her at algoritmen har vist en tendens til opphopning av pakker rundt sentrum i nettverket ved stor trafikk/pakketetthet[19].

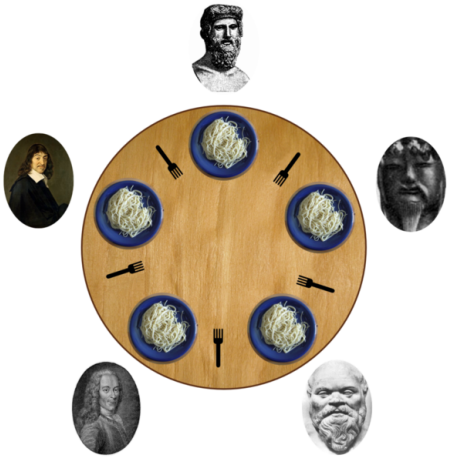
2.3.2 Adaptive rutingalgoritmer

Adaptive algoritmer kan få sin informasjon fra sine naborutere eller fra alle andre rutere i nettverket[17]. Denne informasjonen kan komme i form av at en nabo gir beskjed om at dens buffere er fulle eller at en del av nettverket kringkaster at det har en opphopning av pakker i området eller er midlertidig ute av drift.

Adaptivitet i et NoC kan i mange sammenhenger sees på som det samme som å tilpasse seg til en spesifikk applikasjon. Ved applikasjonsspesifikk ruting, ruter man pakker igjennom nettverket ut fra forutsetninger gitt av de funksjonelle moduler som skal benytte nettverket. Dette vil si at visse typer informasjon kan benytte en annen rutingalgoritme for å kunne nå sin destinasjon raskere. Utfordringen med mange typer adaptive algoritmer er dog at økt ytelse går på bekostning av introduksjon av vranglås og livelock[21] samt økt kompleksitet i nettverket. En rekke vranglåsfrie rutingalgoritmer er fremlagt [22],[23],[24],[25] men i de fleste tilfeller går vranglåsfrihet på bekostning av adaptivitet i nettverket[21].

Adaptive eller applikasjonsspesifikke rutingalgoritmer realiseres ofte i form av en oppslagstabell. I hver node i nettverket bygges en tabell med en postering for alle andre noder i nettverket. For hver destinasjonsadresse inneholder da denne tabellen typisk en primærutgang som en pakke skal rutes til sammen med en sekundær utgang om ikke primærutgangen er ledig. Dette krever store ressurser på grunn av at hver nettverksnode må ha en tabell over alle andre noder i nettverket. Ved innføring av en ny node må en ny postering opprettes i oppslagstabellen for alle andre noder i nettverket. Dette gjør at ressurskravet for å realisere disse tabellene vil øke eksponentielt med antall noder i nettverket. Metoder som den fremlagt i [26] kan imidlertid redusere størrelsen på en slik tabell noe.

2.4 Vranglås



Figur 2.9: Dining philosophers problem, [27]

Vranglås definerer situasjoner som forsinker en pakkeleveranse til uendelig tid ved at et antall pakker blir blokkert i nettverket. Vranglås kan oppstå fordi nettverket lar pakker holde på en ressurs samtidig som de ber om en annen. Dette problemet, som kalles *Dining philosophers problem*, beskrives av Cormen *et al.* i [20]. Alle filosofene rundt bordet(fig 2.9) starter med å ta en gaffel men ingen av dem får spist noe fordi ingen av dem har to gafler. Relatert til NoC betyr dette som oftest at en pakke befinner seg i et buffer og ber om tilgang til et annet buffer. Pakken vil nå holde på det bufferet den befinner seg i helt til den har fått tilgang til det nye bufferet. Om en annen pakke holder på bufferet den første pakken ber om, vil den første pakken bli blokkert fra å få tilgang til dette bufferet så lenge ikke den andre pakken har fått tilgang til et nytt buffer.

Deteksjon av vranglås samt gjenopprettingsmekanismer krever store ressurser og kan videre føre til uforutsette forsinkelser[28]. Dette gjør vranglåsfrie nettverk meget attraktive.

2.5 Quality of Service

Quality of Service, forkortet QoS, er evnen til å kunne håndtere meldinger av ulik viktighet. Med et økende antall multimediaapplikasjoner oppstår et behov for å kunne garantere for håndtering av visse typer informasjon[17]. For en bruker som streamer en video fra nettet vil det være mye viktigere at videostrømmen ankommer riktig enn for eksempel at en ny meil med et stort vedlegg blir lastet ned med en gang. E-meil, filoverføringer, webtilgang og fjerninnlogging er eksempler på applikasjoner der en viss forsinkelse kan tillates, men der bitfeil ikke kan aksepteres. Ser vi derimot på applikasjo-

ner for video og lyd er kravene til båndbredde ofte mye større, men her kan til gjengjeld enkelte bitfeil tillates[17]. Ulike applikasjoner krever ulike former for QoS og et NoC må tilfredsstille de QoS krav som kreves fra applikasjonene som skal benytte det.

Vi deler generelt QoS krav for enbrikkesystem inn i tre deler: *beste eevne(BE), garanterte tjenester(GS) og differented services*. BE tar bare hensyn til korrekthet og kan dermed ikke gi noen garantier for leveringstid. Pakkene leveres så fort som mulig over et forbindelsesløst nettverk. GS gis en mer konkret forpliktelse med tanke på ytelse, i tillegg til basistjenestene tilbydd igjennom BE. Videre kan en gitt tjeneste tilbys med ulik grad av forpliktelse slik at for eksempel en dataoverføring kan gis både med og uten garanti om at dataen leveres. GS tilbys typisk ved å bruke tilkoblingsorientert svitsjing for eksempel ved innføring av virtuelle kanaler(kap 2.6).

Med QoS i et NoC menes at man kan garantere for at en høyere prioritert pakker blir håndtert før lavere prioriterte pakker i en node i et nettverk. Dette baserer seg på at ruterne har støtte for å kunne avbryte behandling av en pakke om en høyere prioritert pakke ankommer noden.

Type	**Beskrivelse**	**Prioritet**
Signalering	Hastemeldinger, korte pakker, avbrudd og kontrollsignaler som krever kort transportforsinkelse	Høyest
Sanntidsmeldinger	Sanntidsapplikasjonspakker	
Lesing og skriving LUT	Kortidsminne- og register-tilgang	
Blokkoverføringer	Lange meldinger og blokker av data	Lavest

Tabell 2.1: Prioritering for ulike typer meldinger,[1]

Vi opplever at enbrikkesystemer blir mer og mer heterogene og dynamiske. Med dette menes at applikasjoner som tidligere ikke hadde noe med hverandre å gjøre nå bygges sammen på en brikke for å håndtere felles oppgaver. I mobiltelefoner intraherer tradisjonelle oppgaver som telefoni og SMS med nye applikasjoner som for eksempler foto, video og til og med kunstig intelligens. Mens brukeren krever en viss oppførsel av systemet kan denne sammenbyggingen av ulike applikasjoner føre til upålitelighet og ustabilitet. Essensen av å bruke QoS blir derfor å kunne tilby et stabilt og pålitelig system for brukeren.

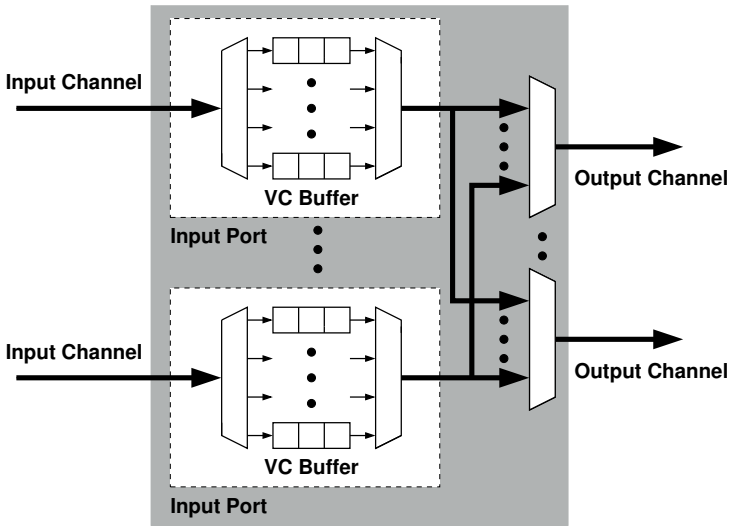
Med større større enbrikkesystemer øker viktigheten av å kunne tilby vanntett QoS med *garanterte tjenester* og for å oppnå dette krevers at enbrikkedesignet er både lokalt og globalt pålitelig[29]. Med QoS forplikter altså nettverket seg til visse tjenester i form av korrekthet, pakkelevering og maksimal forsinkelse.

Ingen enkeltteknikk er funnet for å møte alle QoS-krav og en kombinasjon av ulike designteknikker er designerens eneste mulighet for å møte de krav som stilles fra de ulike applikasjonene. En mulighet her kan være å designe hver ruter med så mye kapasitet

i form av bufferplass og båndbredde at alle pakker enkelt kan flyte igjennom uten å bli forsinket. Dette vil naturlig nok innebære en stor kostnad for nettverket og en mer tilpasset tilnærming vil være fordelaktig. Med dette oppstår en avveiing mellom hvilke tjenester man ønsker å tilby mot hvilke ressurser man har tilgjengelig.

QoS i et nettverk realiseres som oftest ved at pakkene gis en prioritet etter deres viktighet. Med dette vil pakker på ulike prioritetsnivåer bli behandlet etter deres prioritet slik at viktige meldinger kan nå sin destinasjon raskt. En alternativ mulighet er å gi visse noder i nettverket høyere prioritet, og dermed båndbredde, enn andre. Om to noder med korte tidsfrister skal kommunisere med hverandre kan det være ønskelig at dette prioriteres over annen kryssende trafikk. Med en FPGA som målplattform vil det her være mulig, ved å rekonfigurere deler av nettverket, å dynamisk tildele en høyere båndbredde til enkelte deler av nettverket om dette skulle være nødvendig.

2.6 Virtuelle kanaler



Figur 2.10: En mulig arkitektur for implementasjon av virtuelle kanaler,[30]

Sammenkoblede nettverk er sammensatt av to typer ressurser: buffer og kanaler. Typisk assosieres ett buffer med hver kanal slik at vi får en fysisk rute igjennom en ruter og videre til neste ruter. Når en pakke er allokert til et buffer vil da ingen annen pakke kunne benytte seg av kanalen som er assosiert med dette bufferet.

Virtuelle kanaler utvider WH-svitsjeteknikker for å forbedre gjennomstrømning og forsinkelse. Ved å legge til virtuelle kanaler kan vi forbedre ytelse ved å la flit'er fra flere ulike

pakker bli sendt igjennom en og samme fysiske kanal. Dette løser problemet med tradisjonell WH-svitsjing der det ikke vil være mulig å sende flere pakker på sammen kanal, og dermed heller ikke mulig å sende pakker forbi blokkerte pakker i nettverket[30],[31].

En pakke som ankommer en ruter vil bli tildelt en rute igjennom denne og videre i nettverket. Denne prosessen kalles flytkontroll og er en grunnleggende faktor for å kunne utnytte multiple kanaler i en ruter. Fra pakkens synspunkt vil det ikke finnes noen annen rute igjennom en ruter, og det ser dermed ut til og bare finnes en kanal igjennom den aktuelle ruteren. En pakke som ankommer ruteren fra en annen vinkel vil også bli tildelt en slik rute gjennom ruteren, og på samme måte vil også denne pakken bare se en mulig rute igjennom ruteren. De to pakkene vil ikke se hverandre og vil dermed heller ikke ta hensyn til hverandre. Det blir da opp til ruteren å skille dem fra hverandre ved å allokere individuelle ruter til de to pakkene.

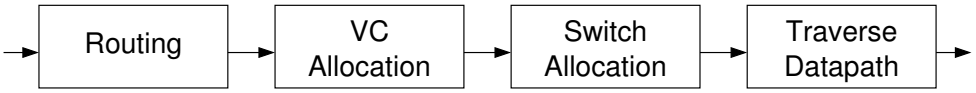
2.7 Arbitrering

For at en pakke skal få tilgang til en ressurs i et nettverk kreves en eller annen form for arbitrering. Arbitreringsoperasjoner blir fort veldig komplekse når nettverket opererer med mange pakker og ressurser og smarte metoder kan dermed spare store ressurser på en FPGA. En velkjent metode for å designe arbitrere med et stort antall innganger eller mulige senarioer er å organisere dem som et tre av mindre arbitere. Med denne ordningen vil hver arbiter propagere forespørsler oppover i treet til en rotarbiter som igjen gir en *grant* eller en bevilgning nedover i treet slik at den riktige utgangen blir satt[32]. Det er her viktig å merke seg at lange kjeder av arbitere i en ruter i et NoC kan gi uforholdsmessig stor forsinkelse. En parallellisering av arbitreringsressursen vil som regel føre med seg en større kostnad i form av ressurser på en FPGA, men vil også gi redusert forsinkelse i forhold til lange kjeder[33].

2.8 Klokkedistribusjon

Klokketrær kan konsumere en signifikant del av det totale effektforbruket på en brikke, og klokkedistribusjon blir dermed en viktig del av moderne synkrone enbrikkesystemer. I forbindelse med nettverk skiller vi i hovedsak mellom synkron og asynkron kommunikasjon, der synkron kommunikasjon betyr at en klokke distribueres utover brikken for å synkronisere informasjonen som distribueres over ledere. Ulike deler av en brikke kan operere på ulike klokkefrekvenser men en globalt synkron brikke vil si at den samme klokken distribueres til hele brikken. Dette vil si at alle FF'er på brikken settes på samme flanke, og med samme frekvens[13]. I en ASIC kan det å distribuere en global klokke over hele brikken være forbundet med en del utfordringer. Moderne FPGA'er her derimot innebygget HW for distribusjon av *lav-scew* global klokke.

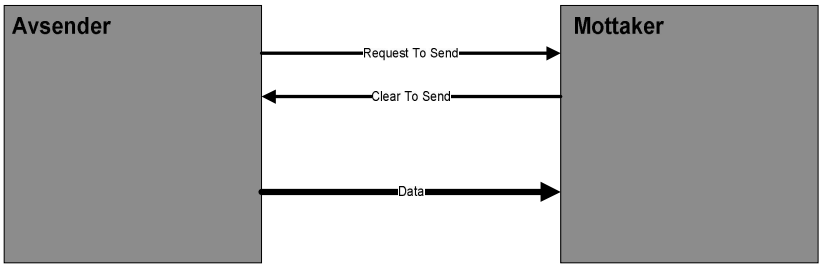
2.9 Pipelineing(samlebånd)



Figur 2.11: Pipeline'ing ved bruk av virtuelle kanaler,[30]

En enkel måte å implementere kontroll ved benyttelse av synkrone rutere er å pipeline de enkelte kontrolloperasjonene. Målet med dette er å gjøre operasjonsfrekvensen for nettverket så uavhengig av kontrollogikken som mulig. Et eksempel her kan være pipelinestegene presentert i figur 2.11. Pipeline'n har en dybde på fire, med steg for ruting, kanalallokering, svitsjallokering og sending av pakke. På denne måten vil tiden det tar å traversere en ruters pipeline være veldig liten i forhold til den totale forsinkelsen i en rutes pipeline. Videre kan man utnytte pipeline'ing til å beregne en pakkes videre rute en sykel før den faktisk skal brukes. Med denne metoden, som kalles lookahead-ruting, kan man redusere forsinkelsen igjennom ruteren ved å gjemme bort deler av kontrollogikken[30].

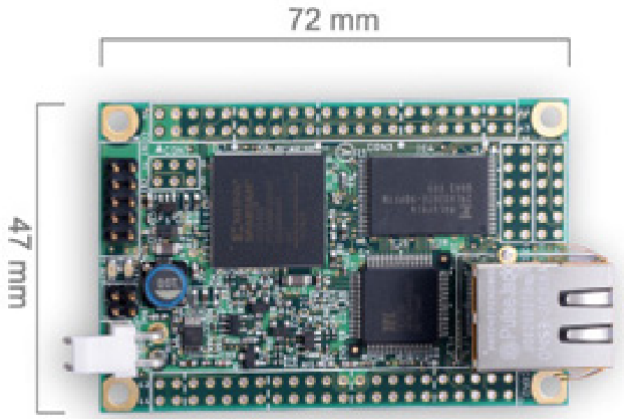
2.10 Håndtrykk



Figur 2.12: Håndtrykksignaler

For at to rutere skal kunne kommunisere serielt på en linje trenger de en protokoll for å hindre avsender i å sende data før mottaker er klar til å ta imot dem. En vanlig måte å håndtere dette i maskinvare er ved å ha to signaler mellom sender og mottaker, *request* og *acknowledge* eller som brukt i RS232, *Request To Send(RTS)* og *Clear To Send(CTS)* for å synkronisere kommunikasjonen. Når avsender ønsker å sende data, settes RTS for å indikere til mottaker at data er klar. Mottaker setter så CTS når den er klar til å motta data[34].

2.11 Plattformen



Figur 2.13: Suzaku-S plattform, [35]

AHEAD-prosjektets målplattform er et Suzaku-S-kort utviklet av Atmark Techno INC. Kortet, avbildet i figur 2.13, inneholder en Xilinx Spartan-3 FPGA(XC3S1000 FT256) med en Microblaze soft-prosessor som kjører en Linux-kjerne(uClinux 2.4). I tillegg har kortet minne(8 Kbyte BRAM, 8 MByte FLASH og 16 MByte SDRAM) og periferier (Ethernet og UART)[35].

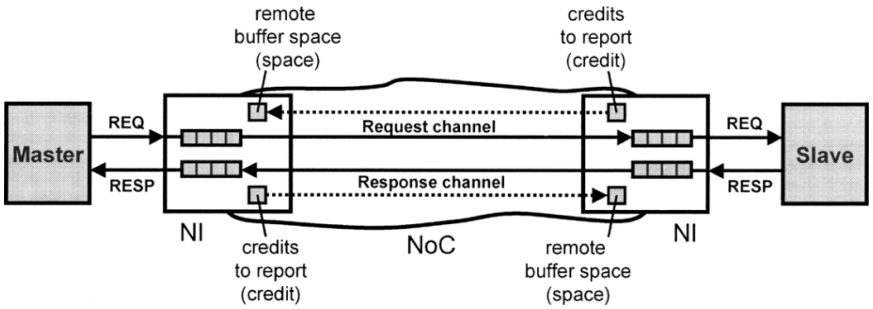
Plattformens FPGA, en Spartan-3 fra Xilinx, inneholder 1,920 CLBs med 7,680 slices(4 slices i hver CLB). Hver slice består av to LUTs og to FFs(register) samt mentelogikk og aritmetiske porter. Spartan serien støtter ikke glitch-free rekonfigurering og anbefales dermed ikke brukt i kombinasjon med partiell rekonfigurering. For mer om dette refereres leseren til FPGAens datablad[36].

KAPITTEL 3

BAKGRUNN

Dette kapittelet gir en oversikt over de viktigste arbeidene innen for emnet Network on Chip. Det er her spesielt fokusert på å gi en kort oppsummering av de arbeider som i størst grad har influert arbeidet presentert i denne rapporten. Dette inkluderer artikler fra et antall arbeider gjort av ulike forskergrupper samt arbeider gjort innenfor AHEAD-prosjektet ved NTNU i Trondheim.

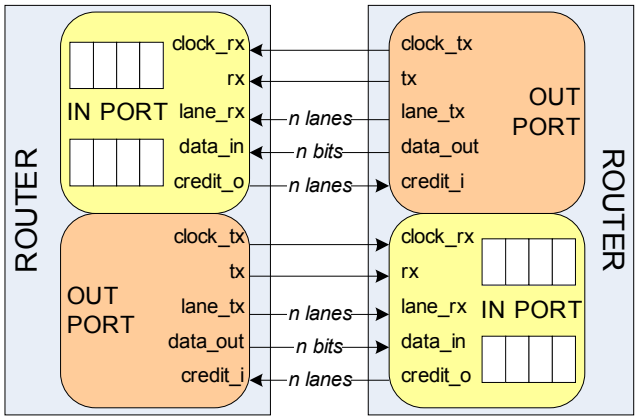
3.1 Æthereal



Figur 3.1: Æthereal, kobling mellom to rutere, [37]

Philips har utviklet et indirekte, synkront nettverk kalt Æthereal. Nettverket fokuserer på bakoverkompabilitet. Ved å benytte en transaksjonsprotokoll kan nettverket kommunisere med tradisjonelle bussløsninger som AXI, OCP og DTL. Denne protokollen tilbyr et delt minne slik at en master-ruter kan forespørre en sending. Slaven svarer da med en responsmelding(for eksempel status til en eksekvering). Figur 3.1 over viser hvordan to noder setter opp en peer-to-peer kommunikasjon mellom hverandre. Masteren styrer kommunikasjonen over en request-kanal, mens slaven svarer på en responskanal. På denne måten kan nettverket tilby garanterte tjenester for bruk i sanntidsoppgaver og kritiske funksjoner i tillegg til beste-evene-tjenester for å utnytte lavere ressurskrav og gi en potensielt bedre gjennomsnittlig ytelse[37],[29].

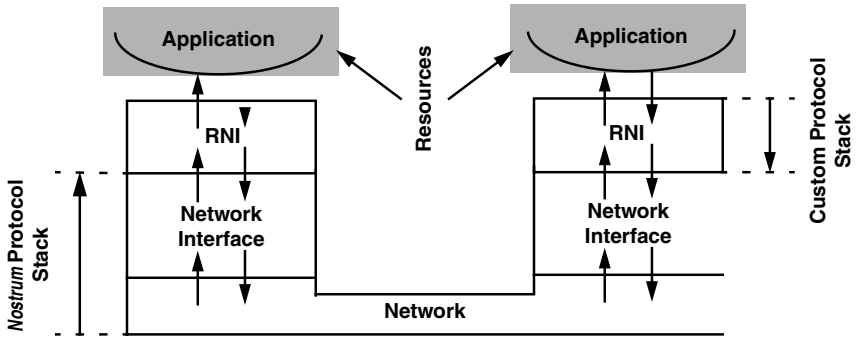
3.2 HERMES



Figur 3.2: Hermes, rutergrensesnitt, [38]

Hermes-nettverket, utviklet ved PUCRS i Brasil, er et direkte nettverk med en 2-D mesh-topologi som anvender WH-svitsjing sammen med x-y-ruting. Hermes-svitsjen har en kontrolllogikk med 5 bidireksjonale porter: Øst, Vest, Nord, Sør og Lokal. Hver port har en inngangsbuffe for midlerdidig lagring av informasjon før en lokal port oppretter mellom svitsjen og dens lokale kjerne. Nettverket bruker en generisk pakkestørrelse med en flit-størrelse på 8 bit. De to første flit'ene i en pakke inneholder en header som beskriver destinasjonsadresse samt antall flit'er i pakken. Svitsjen teller opp alle flit'ene som propagerer igjennom ruteren og lukker ruten når alle er passert[39],[13].

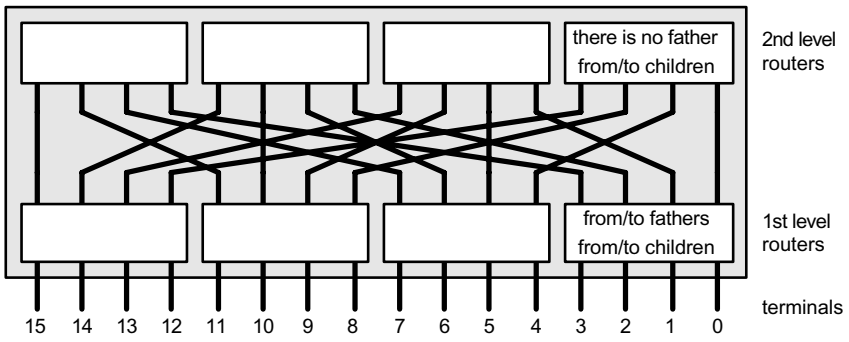
3.3 Nostrum



Figur 3.3: Nostrum, nettverksgrensesnitt, [40]

Et svensk prosjekt ved KTH i Stocholm har utviklet et direkte nettverk med 2-D mesh-topologi, kalt Nostrum. Deres mål er å kombinere tradisjonell mapping av applikasjoner til maskinvare med moderne kommunikasjonsinfrastruktur i form av NoC. For å oppnå dette er et *Nostrum Backbone* utviklet, som skal sørge for en pålitelig struktur der designeren kan velge mellom et sett av implementeringer samt ulike nivåer av pålitelighet, kompleksitet og tjenester. Figur 3.3 viser nostrum-nettstammens grensesnitt mot applikasjoner i et enbrikkesystem[40].

3.4 SPIN



Figur 3.4: SPIN nettverk med Fat-tree topologi, [41]

SPIN(Scaleable Programmable Integrated Network)-nettverket utviklet ved University of Pittsburg er et generisk, skalerbart NoC som baserer seg på WH-svitjing og peer-to-

peer bidireksjonale datalinker. Ruterne er organisert etter en *fat-tree*-topologi som er et tre med alle noder og terminaler som løvnoder[41].

3.5 Xpipes

University of Bologna i samarbeid med Stanford i California har utviklet en kompilator for å meget parametriserbare NoC. Ved bruk av *XpipesCompiler* kan man generere NoC som benytter WH-svitjing med street-sign-ruting. Ruting skjer ved at ruterne aksesserer en oppslagstabell basert på pakkens destinasjonsadresse. Videre tilbyr nettverket CRC(Cyclinc Redundancy Check) som feildeteksjon med retransmisjon av korrupte data. Av energihensyn foretrekkes retransmisjon av flit'er ved feil, fremfor feilkorreksjon. Nettverket benytter pipeline'ing med 7 pipeline-steg[42].

3.6 Ruting i NoC

Både innen for maskinnettverk og nettverk for enbrikkesystemer er en rekke adaptive algoritmer presentert med ulike nivåer av ytelse, pålitelighet og grad av tilpassbarhet. Bolotin *et al.* [43] utvider x-y-algoritmen med hardkodede ruter for vranglåsfri kommunikasjon, mens en ikke-minimal vranglåsfri rutinglgoritme beskrives av [23] for integrasjon i irregulære mesh-topologier.

Forskere ved universitetet i Hong Kong har utviklet en adaptiv feiltolerant rutingalgoritme som benytter WH-svitsjing. Algoritmen tar utgangspunkt i at to typer feil kan oppstå, en fysisk link feiler eller en hel node feiler. Ved å la hver node inneholde informasjon om de andre nodenes status kan feilede noder unngås under ruting av pakker[44]. De to neste seksjonene gir en kort beskrivelse av to metoder for å oppnå vranglåsfri ruting i NoC.

3.6.1 Odd-Even Turn Model for adaptiv ruting

Ge-Ming Choi har tatt for seg Ni og Glass sin *turn model*[25] og videreutviklet den til tre nye, delvis adaptive, rutingalgoritmer. Ideen bak *turn model* er å finne det minimale antall svinger i en pakkes rute som bryter alle sykler som kan oppstå i denne ruten. Chois videreutvikling baserer seg på at nettverket deles inn i odd- og par-rutere, der disse er organisert annenhver i en mesh-topologi. Videre baserer rutingen seg på to hovedregler:

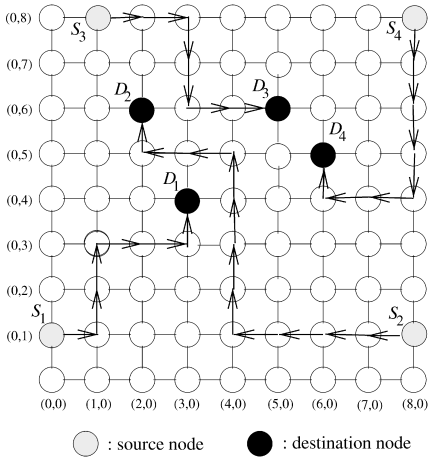
1. Enhver pakke kan ikke ta en ØN-sving[1] i noen par-node og ikke NV-sving i noen

[1]En pakke ankommer en ruter fra øst og svinger mot nord

odd-node.

2. Enhver pakke kan ikke ta en ØS-sving i noen par-node og ikke SV-sving i noen odd-node.

Arbeidet viser at enhver rutingalgoritme som følger reglene over, vil være vranglåsfri så lenge det ikke er lov med en 180-graders vending. Videre viser Choi at x-y-algoritmen utkonkurrerer andre algoritmer for uniform trafikk, men at *odd-even turn model* gir noe bedre resultater for ikke uniform trafikk[45]. Under, i figur 3.5, ser vi et eksempel på hvordan pakker kan rutes i et 9x9 mesh ved bruk av denne metoden.

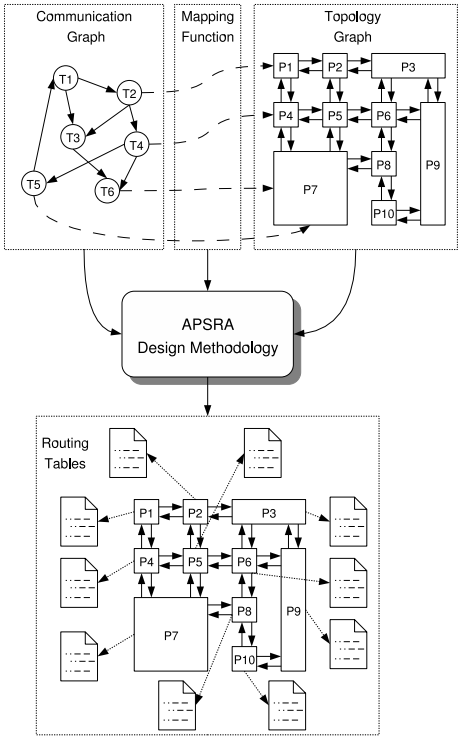


Figur 3.5: Rutingseksempel ved bruk av odd-even turn model, [45]

3.6.2 Metode for design av vranglåsfri ruting

Duato har foreslått en generell teori for utvikling av adaptive vranglåsfrie rutingalgoritmer for nettverk som benytter WH-svitsjing[22]. Metoden tar bare hensyn til nettverkstopologi under bygging av rutingtabeller og Palesi *et al.* utvider dette og presenterer en metode for å generere en rutingalgoritme når kommunikasjonsgrafen[2] for applikasjonen er kjent. Dette gjøres ved å bygge en *applikasjonsspesifikk kanalavhegighetsgraf* basert på algoritmen under[21].

[2]En rettet graf der hver kant t_i representerer en oppgave, og hver node $c_{i,j} = (t_i, t_j)$ representerer en kommnikasjon fra t_i til t_j.



Figur 3.6: Oversikt over APSRA designmetoden, [21]

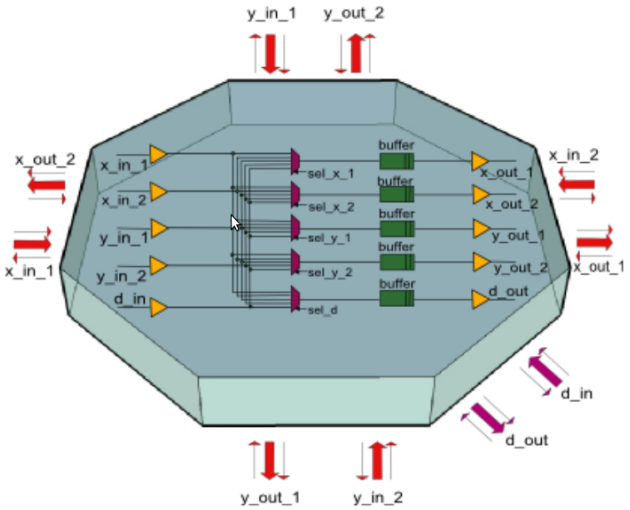
Algoritme 3.1 APSRA designmetoden

1: La R være en minimal fullt adaptiv rutingfuksjon.
2: Bygg en ASCDG relativt til R, CG, TG og M.[3]
3: Pakk ut en sykel fra ASCDG.
4: Bruk en heuristikk for å kutte en kant(fjerne en avhengighet) fra sykelen for å oppdatere R.
5: Gå til 2.

Ved å benytte algoritmen over kan man lage en ASCDG uten sykler og dermed vranglåsfri. Med CG og TG som input kan man lage individuelle rutingtabeller for hver node i nettverket hvis størrelse vil være avhengig av nettverkets størrelse samt kommunikasjonstetthet[4] Etter denne metoden viser forfatterne at for 2-dimensjonale topologier, vil algoritmer designet etter APSRA metoden gi høyere adaptivitet og bedre ytelse sammenlignet med universelle algoritmer.

[4]Forhold mellom kommunikasjonen og antall oppgaver(se side 145 i [21]).

3.7 Prosjektet, Glas



Figur 3.7: Glas, ruterstruktur , [2]

I 2006 hadde den franske studenten Jean-Christophe Glas utveksling ved NTNU i Trondheim. I den forbindelse designet han et integrert nettverk for Xilinx Suzaku-S-plattformen. Arbeidet ble gjort som en del av AHEAD-prosjektet.

Glas har designet et rutingnettverk basert på pakkesvitsjing. Pakkene deles inn i flit'er(à 8 bit) som rutes fra node til node. Figur 3.7 viser hvordan en ruter er designet. Glas benytter WH-svitsjing med x-y-ruting som beskrevet i kapittel 2.2.1 og 2.3.1. Når en flit ankommer en ruter, rutes den til riktig utgangsbuffer ved bruk av multiplexere. I tillegg benytter ruterne to håndtrykksignaler *request* og *achnowledge* for å synkronisere kommunikasjonen.

	Brukt	Tilgjengelig	Utnyttelsesgrad
Antall Slices	121	7680	1%
Antall Flip Flops	85	15360	1%
Antall 4 inngangs LUT	234	15360	1%
Antall Slices i 4*4 mesh NoC	1896	15360	24%

Tabell 3.1: Noen resultater fra syntese til Xilinx Spartan-3, [2]

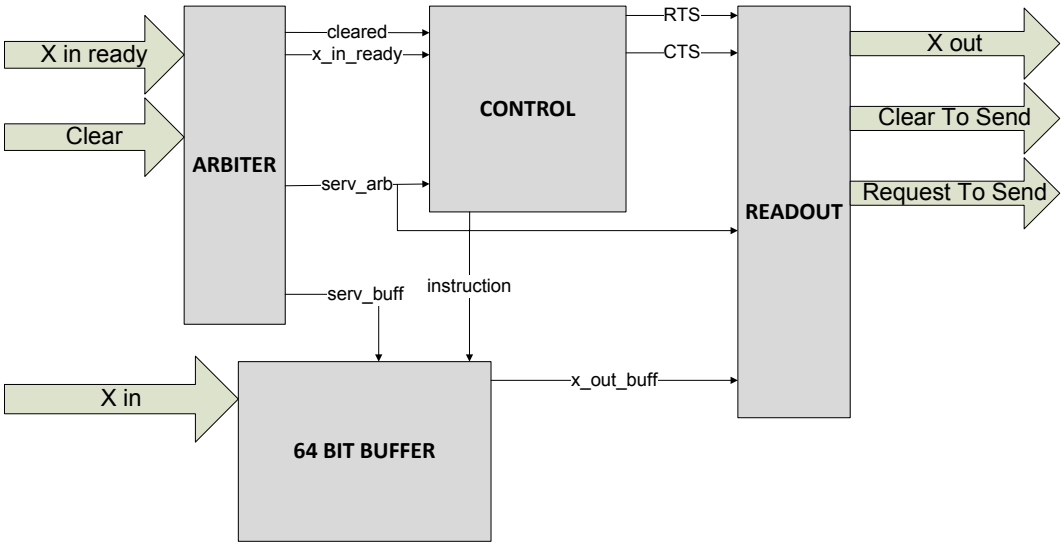
Glas har vist at et integrert nettverk ikke vil ta opp uforholdsmessig store ressurser på Susaku S-plattformen. Nettverket ruter imidlertid bare etter et beste-evnenoe som ikke vil kunne sørge for garantert båndbredde og dermed heller ikke QoS-støtte.

3.8 Prosjekt, Ersland

Denne seksjonene gir en kort presentasjon av det arbeidet som ble gjort i forbindelse med prosjektet mitt høsten 2008. For mer detaljer rundt implementering, resultater og evalueringer refereres leseren til [3]

Med utgangspunkt i nettverket designet av Glas i 2006(se kap 3.7) ble ulike metoder vurdert for innføring av QoS-støtte på dette nettverket. Simuleringer og tester viste imidlertid at Glas sitt nettverk vanskelig lar seg videreutvikle til også å kunne håndtere flere ulike prioritetsnivåer. Nettverket utviklet av Glas har ingen strategi for å formidle til andre dele av nettverket at en pakke er avbrutt noe som vil være et krav for god QoS. På dette grunnlag ble en ruter med en 64 bit lineær kanal, implementert som et skiftregister, implementert og vurdert. Under følger noen detaljer om hvordan ruteren er implementert og hvilke resultater som er oppnådd.

3.8.1 Implementasjon



Figur 3.8: The Router Modules

Ruteren er implementert bestående av de fire submodulene Arbiter, Buffer, Control og Readout. Figur 3.8 viser de fire modulene med deres innganger, utganger og signaler for intern og ekstern kommunikasjon.

Nettverket er organisert i en 4x4 mesh-topologi av 16 rutere, der hver ruter har en inngangskø og en lineær 64 bit kanal. Videre benytter nettverket SAF-svitsjing med

x-y-ruting slik at alle pakker i nettverket vil være fullstendig lagret i en ruter. QoS er implementert med to prioritetsnivåer og en arbiter på inngangen av hver ruter sørger for å rute inn den til en hver tid høyest prioriterte pakken som befinner seg der.

Denne ruteren omtales heretter som prosjektruteren.

3.8.2 Resultater

	Arbiter	Control	Buffer	Readout	Totalt
Antall Slices	9 (8,2%)	23 (20,9%)	47 (42,7%)	31 (28,2%)	110
Antall Flip Flops	4 (4,8%)	16 (19,3%)	64 (77,1%)	0 (0%)	84
Antall 4 inngangs LUT	15 (7,5%)	45 (22,4%)	89 (44,3%)	52 (25,8%)	201

Tabell 3.2: En ruter, resultater av syntese,[3]

Tabell 3.2 viser resultater fra syntes av en ruter slik den er beskrevet i kapittel 3.8.1. Kolonnene viser ressursbruken for de ulike modulene samt de nødvendige ressursene behøvd for å realisere en ruter på den aktuelle FPGA'en. Tallene i parentes angir hvor stor del av ruteren som brukes til å realisere den aktuelle modulen.

Av det totale antall slice'r(110) behøvd for å realisere ruteren går altså 42% til buffring og 21% til kontrolllogikk. Disse delene er naturlig nok de to største delene av ruteren. En enkel arbitrering og rutingalgoritme gjør at Arbiter- og Readout-modulene blir relativt små enheter i ruteren.

	Beregnet	Brukt	Tilgjengelig	Utnyttelsesgrad
Anstall Slices	1760	1446	7680	18%
Antall Flip Flops	1328	1328	15360	8%
Antall 4 inngangs LUT	3216	2709	15360	17%

Tabell 3.3: 4*4 mesh, resultater av syntese,[3]

Resultatene i tabell 3.3 viser at nettverket ikke vil ta opp uforholdsmessig store dele av den aktuelle FPGA'en. Nettverket vil trenge 362 av 1,920 tilgjengelige CLB'er, noe som tilsvarer 18% av de tilgjengelige ressursene. Dette vil gi 1558 CLB'er til de funksjonelle enhetene.

Det er i dette arbeidet ikke gjort noen florplanning og resultatene kan dermed sprike noe fra de faktiske tallene. De beregnede verdiene presentert i tabell 3.3 viser resultater av å plassere 16 uavhengige rutere på FPGA'en. Ved å optimalisere plasseringen av ruterne kan imidlertid synteseverktøyet redusere ressursbruken noe. Da det er vanskelig å si noe om hvilke optimaliseringer som er gjort er det vanskelig å si noe eksakt om den faktiske ressursbruken uten å teste nettverket på det tiltenkte kortet.

$$8 \cdot 8\ Bit \cdot \frac{100\ MHz}{10\ klokkeflanker} = 640\ Mbit/s \tag{3.1}$$

Ruting av 8 flit'er på 10 klokkeflanker gir en teoretisk båndbredde på $640 Mbit/s$.

KAPITTEL 4

DESIGNVALG

Dette kapittelet gir en presentasjon av ulike løsninger som er vurdert i forbindelse med videreutvikling av nettverket foreslått i prosjektet. Kapittelet gir en vurdering av ulike løsninger og hvordan disse kan tilby QoS, pålitelighet og stabilitet. Seksjon 4.1 gir en begrunnelse for valg av topologi. De neste seksjonene gir en gjennomgang av de svitsjeteknikker som er vurdert, før det i seksjon 4.4.3 konkluderes med en løsning. Til slutt velges en arbitreringsstrategi(seksjon 4.5) spesielt tilpasset denne løsningen.

4.1 Ruter og topologi

Som beskrevet i kapittel 2.1 beskriver nettverkstopologien hvordan nodene er organisert i forhold til hverandre samt hvordan de er koblet sammen. Ved valg av topologi er det viktig å ta hensyn til behovet for nærhet mellom nodene og hvilken rutingalgoritme man skal bruke. Som vi har sett er en rekke direkte, indirekte og irregulære topologier foreslått, men 2-D mesh topologien(med diverse modifikasjoner) ser ut til å være en klar favoritt hos de ulike forskningsgruppene innenfor NoC. Topologien kan tilby gode elektriske egenskaper i form av lik forsinkelse mellom alle noder. Samtidig gir den homogene oppbygningen god arealutnyttelse og gjør topologien anvendbar brukt i kombinasjon med dynamisk rekonfigurering. Alle de funksjonelle modulenes areal vil være likt. Dette gjør det enkelt å avgjøre plassering av en eventuell ny modul som skal konfigureres til FPGA'en. Videre gir mesh-topologien, på grunn av nettverkets organisering som et nett av noder, enkel adressering av nodene og dermed også enkel ruting i nettverket. I tillegg

merker vi oss at simuleringer og tester gjort av Bononi og Concer viser at 2D-mesh gir bedre datagjennomstrømning enn ringnettverk[46].

4.2 Valg av bufferstruktur og rutingalgoritme

Det mest sentrale valget når det kommer til nettverkets funksjonalitet og ressursutnyttelse ligger i valg av rutingalgoritme og bufferstørrelse og -struktur. Det vil her, som oftest, være naturlig og først velge rutingalgoritme, for så å velge en bufferstruktur som passer til denne. En av grunnene til dette er at rutingalgoritmen avgjør hvordan pakkene beveger seg igjennom nettverket og dermed setter premisser for hvordan nettverket skal håndtere trafikken av pakker. I seksjonene under utdypes denne problemstillingen sammen med en presentasjon av ulike bufferstrukturer og hvordan disse kan brukes i kombinasjon med smart arbitrering og tilpassede rutingalgoritmer. Alle betraktningene gjort i dette kapittelet er gjort med utgangspunkt i et regulært mesh-nettverk. Bufferressurser markeres med blått i illustrasjonene tilknyttet dette kapittelet.

I prosjektarbeidet[3] ble en løsning med en 64 bit lineær kanal for buffring av hele pakker undersøkt. Denne arkitekturen, i den form den ble benyttet i prosjektet, ble funnet å ha enkelte begrensninger når det kommer til håndtering av stor trafikk. Med et hovedfokus på forbedret ytelse, i form av båndbredde og QoS, undersøkes ulike retninger for videreføring av denne arkitekturen slik at nettverket bedre kan møte de krav ulike applikasjoner stiller til det. Med en god bufferstruktur og rutingalgoriteme i kombinasjon med smart arbitrering vil nettverket bli mer pålitelig, gi sikrere QoS, og kunne tilby en bedre båndbredde enn det tidligere foreslåtte nettverket. For å oppnå dette ser vi først på ulike utfordringer og løsninger på systemnivå med en vurdering av hvordan ulike valg påvirker hverandre og i hvilken grad forandring av en parameter vil legge premisser for andre deler av arkitekturen.

Ved først å velge rutingalgoritme som kan sette videre krav til systemet kan man få en oversikt over hva som er nødvendig og begrensende for de andre modulene i ruteren. Bufferstruktur som passer til denne rutingalgoritmen og som oppfyller systemets ytelses- og ressurskrav kan da lettere realiseres. Under følger en oversikt over de fordeler, ulemper og utfordringer noen utvalgte løsninger kan gi.

4.3 Innføre adaptivitet

En løsning brukt av en rekke ulike forskergrupper er innføring av ulike former for adaptive rutingalgoritmer. Hovedideen bak dette er at buffring av pakker er meget ressurskrevende, og for å holde buffernivået på et minimum velger man å rute pakker rundt hverandre i stedet for å sende dem forbi hverandre i en ruter. Metoden brukes også for å lede pakker bort fra deler av nettvert der man har opphopning av pakker. Ved å innføre

adaptivitet i nettverket vil nettverket selv ha muligheten til å forutse og forhindre mulige kritiske situasjoner som kan oppstå. På denne måten kan man holde behovet for buffring av pakker på et minimum samtidig med at adaptivitet kan gi en mer optimal utnyttelse av nettverket.

På grunn av det reduserte behovet for bufferressurser adaptive rutingalgoritmer fører med seg, brukes adaptivitet ofte i kombinasjon med WH-svitsjing. Det er her imidlertid viktig å være bevisst på at adaptivitet vil føre med seg økt kompleksitet i form at ruting og mer avansert kontrollogikk.

Det vanligste problemet i et nettverk er en situasjon der to pakker møter hverandre hode mot hode i en node. Ved bruk av adaptivitet kan denne situasjonen løses ved at en av de to pakkene rutes rundt den andre. Med dette utgangspunktet må en av pakkene ta en annen rute enn den som opprinnelig var utpekt til denne pakken Det blir da opp til den aktuelle ruteren og avgjøre hvilken av pakkene som skal rutes rundt og hvilken vei denne pakken da skal ta. For å tilby QoS må pakkenes prioritet ivaretas og vi kan her tenke oss at pakken med lavest prioritet rutes rundt den andre pakken. Har pakkene lik prioritet kan forhold som for eksempel hvilken pakke som er nærmest midten av nettverket avgjøre hvilken pakke som skal rutes rundt. Dette fordi det antas at pakken nærmest sentrum av nettverket har større sjanse for å finne en rute rundt. En tredje mulighet er at den som har lengst til sin destinasjon rutes rundt, da det antas at denne pakken vil bli minst forsinket i forhold til sin totale reisetid.

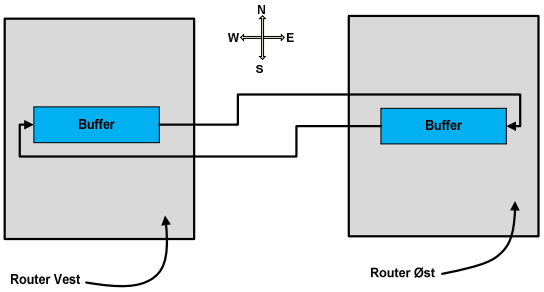
Etter å ha avgjort hvilken av pakkene som skal rutes rundt må ruteren avgjøre hvilken retning den aktuelle pakken skal rutes til. Annen trafikk i nettverket samt nettverkets topologi er bare to av parametrene som kan være med å bestemme denne rutingen. Som vi ser blir slike operasjoner ofte veldig komplekse og kan dermed føre til kompleks arbitrering, ruting og kontrollogikk. Vi ser her også et behov innføring av asynkrone signaler for signalisering til andre rutere som eventuelt måtte holde på flit'er som tilhører pakken som er påvirket av et adaptivt valg. En vanlig løsning for dette er at nettverket gjøres globalt asynkront og lokalt synkront(GALS). Dette betyr at de funksjonelle modulene kommuniserer synkront med sin lokale ruter mens ruterne sender pakken asynkront mellom hverandre. Ved design for ASIC-implementering har flere forskergrupper pekt på fordelen ved å slippe å distribuere en global klokke utover hele brikken. De fleste moderne FPGA'er har imidlertid innebygget maskinvare for klokkedistribusjon, og vi vil dermed ikke møte dette problemet i samme grad her.

4.4 Svitsjeteknikk

Kompleksiteten og uforutsigbarheten ved adaptive rutingalgoritmer gjør at vi her ønsker å se på andre teknikker for adressering av opphopninger og vranglåser i nettverket. Etter som de tilgjengelige ressursene på FPGA'ene blir større åpnes nye muligheter når det kommer til svitsjeteknikker og ruting i NoC. En mulighet her er å la pakker traversere

forbi hverandre, eller på tvers av hverandre, inne i en ruter. For å gjøre dette mulig må en ruter kunne holde på flit'er fra flere ulike pakker samtidig, noe som vil medføre større buffer i hver enkelt ruter og dermed kreves større ressurser for å realisere nettverket. I tillegg er det viktig at bufferne organiseres på en slik måte at pakkene enkelt kan skilles fra hverandre og at hver flit blir rutet i riktig i forhold til den oppsatte ruten. De tre neste seksjonene tar for seg et utvalg av metoder for å rute flere pakker forbi og på tvers av hverandre inne i en ruter.

4.4.1 Sende og motta samtidig



Figur 4.1: Sende og motta samtidig

En enkel metode for å kunne håndtere møtende pakker er og la hver ruter ha en lineær bufferkanal, slik det er implementert i prosjektet, og innføre muligheten for å lese inn på en ende av kanalen samtidig som man leser ut fra andre enden. Figur 4.1 illustrerer hvordan metoden håndterer to møtende pakker ved bruk av en lineær kanal i hver ruter. De to ruterne, her kalt ruter vest og ruter øst, holder hver sin pakke som de ønsker å sende mot hverandre. Med ruteren beskrevet i prosjektoppgaven som utgangspunkt innfører vi en ekstra modus der ruterne kan sende og motta samtidig på en enkelt kanal. Dette er mulig fordi nettverket benytter endireksjonale ledere og dermed har to busser mellom hver ruter. I denne modusen stenger de to ruterne ned alle andre funksjoner og snakker kun med hverandre. Ruter øst og ruter vest skifter da ut sine pakker fra enden av sitt buffer samtidig som de leser inn pakken fra den andre ruteren på andre enden av det samme bufferet. Prosjektruteren er implementert slik at den skifter rundt når den sender pakker(se mer om dette i kapittel 3.4.2 i kilde [3]) noe som føre til at ruteren vil holde på pakken helt til hele pakken er sendt. I og med at vi her sender om mottar pakker på samme kanal samtidig vil denne egenskapen ikke være opprettholdt. Det vil da være vesentlig at utvekslingen av pakker mellom ruter vest og ruter øst foregår uten avbrudd og uten forstyrrelse fra andre deler av nettverket da dette vil føre til pakketap eller korrupte data.

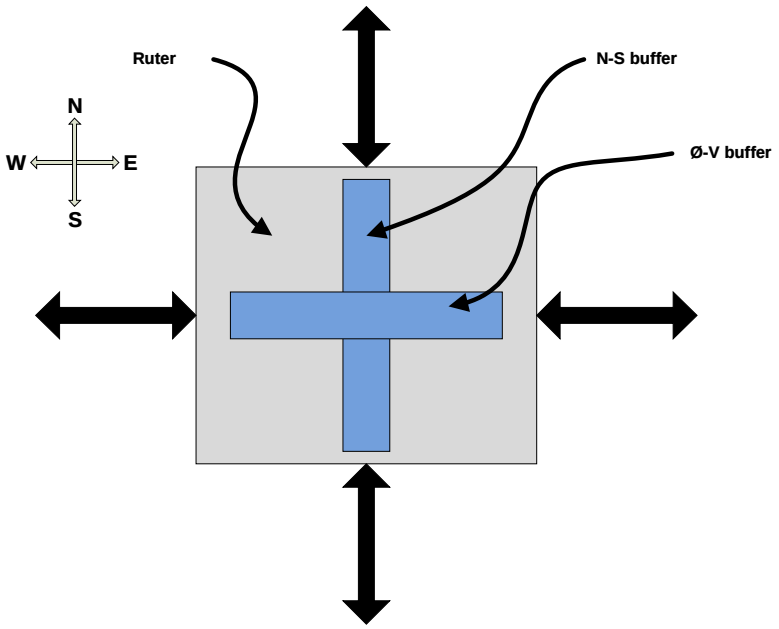
Med utgangspunkt i prosjektruteren ser vi at metoden ikke krever flere ressurser til buffring. Samtidig vil metoden, på grunn av muligheten for å sende og motta samtidig, gi en potensielt forbedret båndbredde. Implementert riktig kan vi også garantere at alle

pakker blir levert uten at vranglås oppstår. Dette fordi to rutere alltid vil ha mulighet til å utveksle pakker selv om alle rutere i nettverket holder på en pakke. Noe ekstra ressurser behøves for å implementere modusen for utveksling av pakker, men ut fra beregningene gjort i prosjektrapporten i kapittel 4.2.1 antas det at ressursene til kontrollogikk vil være betydelig mindre enn de ressursene som behøves for å innføre ekstra buffring.

Metoden kan imidlertid ikke garantere for QoS da den gir mulighet for prioritetsinversjon. Ved utveksling av pakker vil de to aktuelle ruterne ikke ta hensyn til pakkenes prioritet og en eventuell ventende pakke fra en annen del av nettverket vil da ikke bli håndtert etter den prioriteten den er gitt.

4.4.2 N-S og Ø-V ruting

Ved bruk av x-y-algoritmen ser vi at hver pakke aldri vil gjøre mer enn en sving i nettverket. Denne svingen vil skje når pakken ankommer den raden mottakeren ligger på fra den kolonnen avsender ligger på slik vi ser av figur 2.8. På dette grunnlaget kan vi si at pakker som beveger seg mer enn to rutere bort fra sin avsender alltid vil traversere rett frem i en ruter flere ganger enn den svinger. Det vil si at pakker som kommer fra øst som regel skal videre vestover, pakker som kommer fra nord skal videre sørover og så videre. Ved å innføre egne kanaler mellom øst og vest, og mellom nord og sør i en ruter, kan vi dermed forenkle arbitrering. Dette vil gjøre ruting raskere og mindre kompleks for tilfeller der pakker skal videre i den retning de kommer fra.



Figur 4.2: NS,ØV-bufferstruktur

Figur 4.2 viser hvordan to kanaler kan organiseres på tvers av hverandre for å oppnå disse egenskapene. Har vi to pakker i nettverket som beveger seg på tvers av hverandre vil disse kunne rutes igjennom samme ruter samtidig. Fordi de to kanalene er uavhengige av hverandre kan en pakke fra nord mot sør rutes igjennom den ene kanalen samtidig som en annen pakke fra øst mot vest rutes igjennom den andre kanalen. På denne måten reduserer vi leveringstiden for pakkene, samtidig som vi oppnår forbedret båndbredde for nettverket.

Metoden krever imidlertid en mekanisme for å håndtere pakker som skal svinge i nettverket. Dette gjøres ved at den aktuelle pakken flyttes fra den ene kanalen(for eksempel N-S buffer) til den andre kanalen(Ø-V buffer). I og med at FPGA'er har begrensede ressurser for interne koblinger vil det lønne seg å skifte pakken fra det ene bufferet til det andre, flit for flit.

En arbitrering avgjør om en flit/pakke skal flyttes over til den andre kanalen. Det er da viktig at eksisterende pakker ikke blir overskrevet samtidig som høyere prioriterte pakker ikke blir avbrutt eller forsinket på annen måte. Det vil dermed bli nødvendig med to nivå av arbitrering. På det første nivået arbitreres det individuelt i hver kanal over hvilken retning som skal leses inn av henholdsvis nord og sør, og øst og vest. På neste nivå håndteres flytting av pakker fra den ene kanalen til den andre.

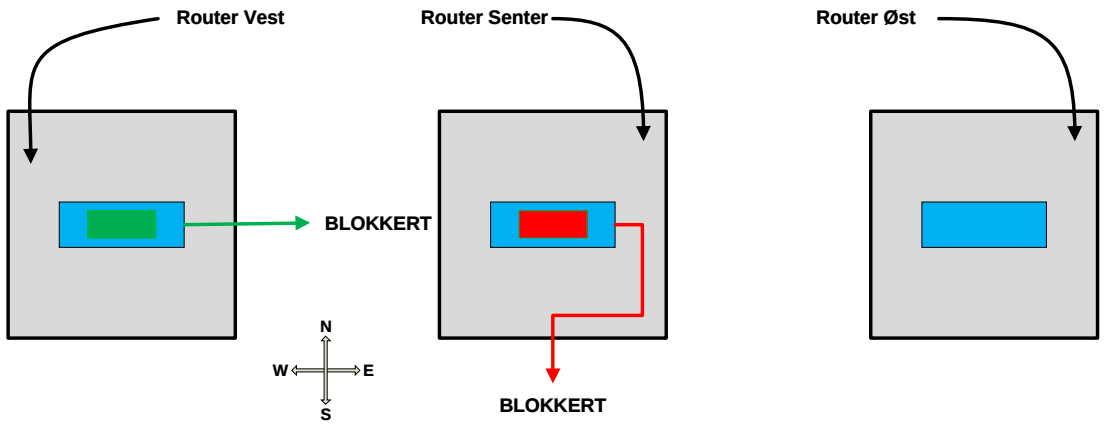
Det vil være naturlig å gi flytting av pakker prioritet fremfor innlesing av nye pakker. Innlesing av ny pakker kan lettere føre til opphopning av pakker, vranglås eller overskrivning av eksisterende pakker. Vi ser imidlertid at det å gi flytting prioritet ikke vil opprettholde QoS støtten da en høyere prioritert pakke som er klar for innlesing vil bli avbrutt til fordel for flytting av en lavere prioritert pakke. En alternativ løsning her vil være å la de to kanalene operere som en ekstra inngang på den andre kanalen, noe som gjør at en enkelt arbitrering avgjør om nye pakker skal leses inn eller pakker skal flyttes over fra den andre kanalen. Som vi ser vil metoden gi avansert arbitrering for å møte de krav som stilles til nettverket i form av god QoS. Brukt i kombinasjon med *Odd-Even Turn Model* beskrevet i kapittel 3.6.1 kunne likevel arbitreringen hvert gjort noe enklere da denne rutingalgoritmen gir premisser for hvilke noder det er mulig å gjøre en sving.

Metoden håndterer ikke vranglåsproblemet da to pakker som støter mot hverandre ikke vil ha noen mulighet til å traversere forbi hverandre uten innføring av ytterligere logikk eller buffer. Allikevel kan vi, ved å innføre muligheten for å la hver kanal kunne lese inn og ut samtidig, garantere at alle pakker blir levert uten vranglås(se kap 4.4). Videre vil metoden gi økt båndbredde som følge av flere kanaler, men dette må sees i forhold til de økte bufferkostnadene som følge av innøring av to kanaler istedenfor en. I tillegg seg vi at de to kanalene bare utnyttes når to pakker beveger seg på tvers av hverandre og det er dermed ønskelig å se på muligheter for å utnytte to kanaler bedre.

4.4.3 Innføring av virtuelle kanaler

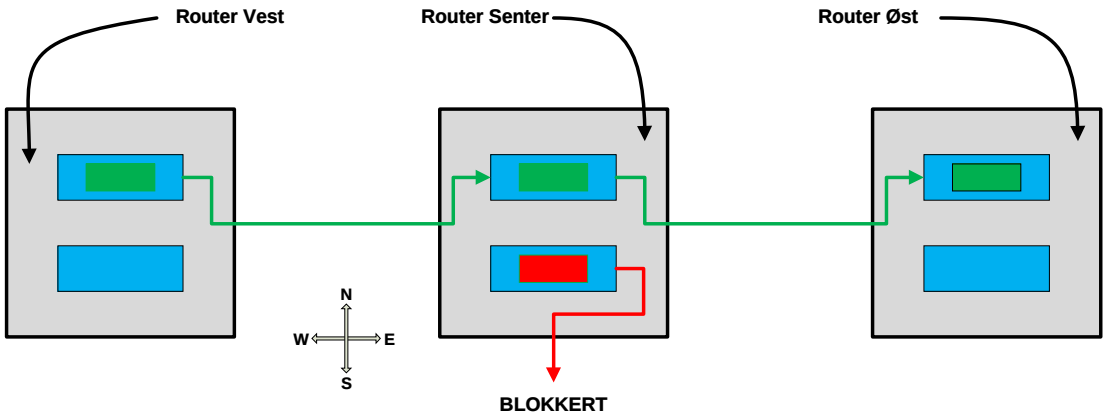
Som vi så i kapittel 4.4.2 over kan det i mange tilfeller være fordelaktig å implementere flere kanaler i hver ruter. Metoden over utnytter imidlertid ikke de mulighetene som ligger i å kunne rute flere pakker igjennom en ruter samtidig og vi ønsker dermed å se på muligheter for å la de to kanalene operere mer uavhengig av hverandre. Dette kan ses på som om man legger til filer på en innfartsvei, der et nettverk uten virtuelle kanaler vil være det samme som en vei med bare en fil. Ingen biler kan passere hverandre og en bil som stopper på veien vil føre til at alle biler bak også må stoppe. Har vi flere filer vil flere biler kunne kjøre på veien samt at raskere biler kan passere treger biler.

I prosjektruteren hadde vi bare èn fysisk kanal igjennom ruteren noe som gjør at de to pakkene som ankommer ruteren fra ulike vinkler, selv om de ikke ser hverandre, i realiteten må benytte den samme fysiske ruten for å få satt opp sin vei igjennom ruteren.



Figur 4.3: En virtuell kanal

Figur 4.3 viser to pakker, en rød og en grønn, som beveger seg igjennom et mesh-nettverk. Den røde pakken ankommer *ruter senter* og rutes videre sørover for å sendes til sin destinasjon. Ruteren i sør er imidlertid opptatt med å håndtere andre pakker og blokkerer dermed sendingen. Den grønne pakken, som også skal igjennom *ruter senter*, vil da også bli blokkert som følge av dette. Det eksisterer altså ingen rute igjennom denne ruteren som kan allokeres til den grønne pakken og den må dermed vente til ruter senter har fått rutet den røde pakken sørover før den kan rutes videre.



Figur 4.4: To virtuelle kanaler

Ved å innføre flere fysiske kanaler igjennom en ruter kan vi, om nødvendig, sette opp et antall samtidige ruter igjennom ruteren noe som i praksis betyr at de to pakkene som ankommer ruteren fra ulike vinkler kan bli rutet igjennom ruteren samtidig. Selv om de to pakkene her bare ser en mulig rute igjennom ruteren, finnes det altså en annen rute benyttet av den andre pakken. I figur 4.4 ser vi den samme situasjonen som over i figur 4.3 men nå har hver rute to virtuelle kanaler i stedet for en. Den røde pakken, som er blokkert, opptar den ene kanalen. Den grønne pakken, som ankommer ruter senter fra øst, vil nå bli allokert en rute igjennom ruterens andre kanal, og videre til ruter øst. På denne måten vil ikke blokkeringen av den røde pakken forsinke eller stoppe den grønne pakkens vei igjennom nettverket og den kan uforstyrret fortsette videre mot sin destinasjon.

Antallet kanaler kan variere ut fra topologi, og en ruter som har mange innganger vil ha stor sannsynlighet for ankomst av flere samtidige pakker og kan dermed ha behov for mange kanaler. Med fem innganger og en pakke per inngang kan en ruter ha maksimalt fem pakker klare på inngangene(se kapittel 2.1 for mer om dette). For å kunne rute fem pakker samtidig behøves fem kanaler noe som vil være meget ressurskrevende i form av logikk og bufferressurser. En vurdering av behovet for båndbredde opp mot ressursforbruk vil her være nødvendig for å avgjøre et passende antall virtuelle kanaler. Med utgangpunkt i prosjektruteren, som buffrer hele 64bit pakker(SAF-svitjsing), vurderes dermed en videreføring med to kanaler à 64 bit. Med en slik løsning kan dermed hver ruter holde på to hele pakker samtidig som nettverket vil opprettholde funksjonaliteten fra prosjektet med at alle pakker til en hver tid vil være fullstendig lagret i en ruter(se kapittel 3.8).

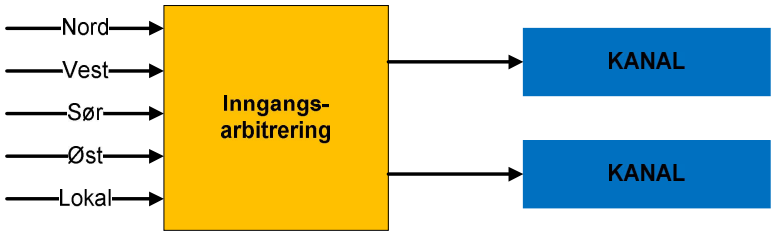
For å gjøre ruteren så fleksibel som mulig er det ønskelig at de to kanalene i størst mulig grad gjøres uavhengig av hverandre noe som vil si at inn og utlesing på den ene kanalen i minst mulig grad påvirkes av aktivitet på den andre kanalen. Som et ekstrem kan vi her tenke oss at vi plasserer to prosjektrutere på samme fysiske plassering og lar de to

ruterne operere fullstendig uavhengig av hverandre. En pakke som ankommer en node vil bli allokert en av de to tilgjengelige ruterne, som igjen vil rute pakken videre i nettverket. Pakker kan nå leses inn og ut fra den ene ruteren fullstendig uavhengig av den andre og omvendt. Den gode fleksibiliteten forbundet med denne metoden går på bekostning av en markant økning i ressursforbruk. Dette gjør igjen at et 4x4 mesh-nettverk, som det vi hadde i prosjektet, vil oppta 38% av ressursene på den aktuelle Xilinix Spartan 3 FPGA'en. I tillegg kommer arbitreringsressurser for å allokere en ruter til de ulike pakkene samt hvilken av de to ruterne som skal for sende først. For å begrense denne ressursbruken, ønsker vi å se på metoder for å, i større grad, integrere de to kanalene med hverandre men samtidig opprettholde fleksibiliteten og påliteligheten.

Mellom hver node vil det kun være en fysisk kanal. Med dette menes at det kun finnes en buss mellom hver node og at det dermed bare vil være mulig å sende en pakke fra en ruter til en annen om gangen. Arbitrering mellom de fire inngangene vil så avgjøre hvilke av de fire inngangene som skal rutes inn på ruterens bufferressurser. Dette gjør at det kun vil være mulig for en ruter å motta en pakke om gangen. Videre fører dette til at kontrolllokikken samt arbitreringsstrategien kan forenkles en god del i forhold til det og ha to fullstendig uavhengige rutere på samme fysiske plassering. Ved ankomst av en eller flere pakker vil da altså èn arbitrering allokere èn bufferkanal til den pakken med høyest prioritet. kontrollogikk sørger deretter for at den aktuelle pakken blir lest inn på kanalen den har fått tildelt. Ved å ha en felles arbitrering for de to kanalene kan vi redusere ressursbruken samtidig som vi reduserer pakkens forsinkelse på vei inn i ruteren. Da denne arkitekturen ser ut til å være realiserbar på FPGA samtidig som den gir pålitelighet, stabilitet og god QoS er den valgt som arkitektur for videreutvikling av prosjektruteren. Denne videreutviklingen presenteres i kapittel 5.

Når vi nå har valgt en arkitektur for ruteren som benytter to virtuelle kanaler for ruting av pakker ved bruk av SAF-svitsjing ønsker vi videre å se på arbitreringsstrategier i kombinasjon med muligheten for å pipeline de ulike stegene i rutingen av en pakke igjennom en ruter. En arkitektur som den vi når har valgt vil bestå av to store beregnende deler, en arbitrering over hvilken av naboruterne som til en hver tid skal få sende pakker til denne ruteren samt kontrollogikk for å styre inn og utlesing fra de to kanalene. Vi ønsker nå å se på strategier for smart arbitrering av en inngangskø for å oppnå god QoS og båndbredde.

4.5 Arbitrering



Figur 4.5: En inngansarbiter

En rekke ulike strategier for god arbitrering er presentert i litteraturen. Ved valg av arbitreringsstrategi og struktur for arbiteren er det viktig å tenke på at arbiteren som oftest går direkte på den kritiske stien igjennom ruteren, og at den dermed vil være direkte begrensende for ruterens operasjonsfrekvens. Avhengig av bufferstruktur kan ulike arbitreringsmetoder vurderes, der enkelte av dem er diskutert tidligere i dette kapittelet.

Da vi nå har valgt en arkitektur med to virtuelle kanaler ønsker vi å se de på arbitreringsmetoder spesielt tilpasset denne teknikken. En arbitrering mellom fem innganger med fem tilhørende prioriteter, slik det er beskrevet tidligere i dette kapittelet, kan bli veldig kompleks og det er her viktig å fokusere på å gjøre arbitreringen så enkel som mulig. Det er allikevel viktig at ruteren kan håndtere alle situasjoner som kan oppstå.

Med dette oppstår en avveining mellom god QoS og ressursforbruk i arbiteren. Det kan i mange tilfeller være viktigere at situasjoner som oppstår ofte håndteres bra, enn at alle situasjoner håndteres på samme måte. Et første steg for å gjøre arbitreringen enklere vil være å nedprioritere utsending av nye pakker. Med dette oppnår vi at antallet arbitreringssituasjoner halveres. Samtidig resulterer dette i at ruterne prioriterer levering av de pakkene som til en hver tid måtte befinne seg i nettverket, fremfor å lese ut nye pakker til nettverket fra sin lokale modul. Dette vil også redusere sannsynligheten for vranglås i nettverket.

For å gjøre ruterens operasjonsfrekvens mindre avhengig av arbiterens kritiske sti, klokkes arbitreringen. I stedet for å rute den høyest prioriterte pakken direkte inn til bufferet bruker arbiteren først en klokkesykel på å avgjøre hvilken retning som skal håndteres, før signalene blir rutet inn. På denne måten kan kontrollogikken styre inn og utlesing fra de to kanalene uten at arbitreringen vil forsinke denne prosessen i form av bidrag til ruterens kritiske sti.

4.6 Oppsummmering

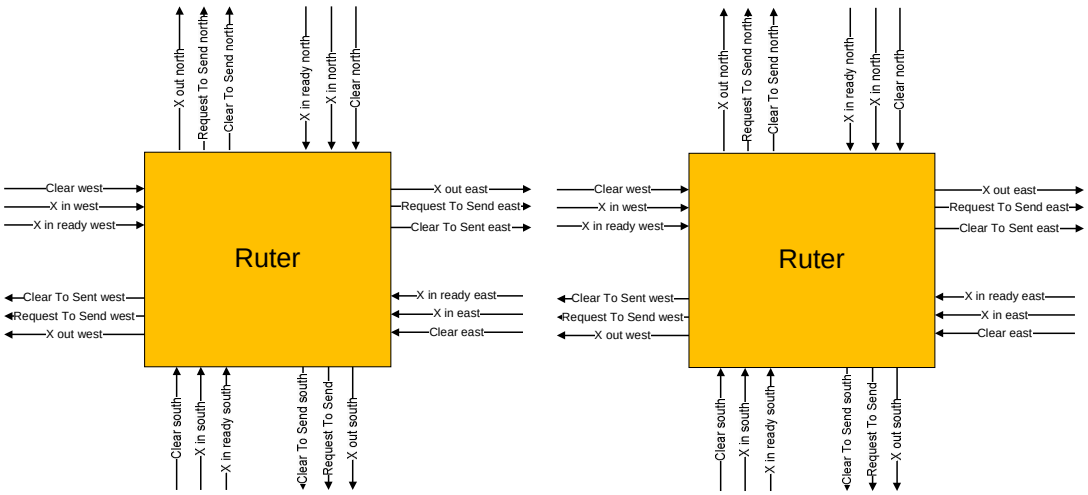
Etter vurderingene presentert i dette kapittelet er følgende teknikker valgt for implementasjon.

- Topologi: Direkte homogen 2D-mesh
- Svitsjeteknikk: SAF-svitsjing med to virtuelle kanaler
- Arbitregringsstrategi: Inngangskø med to prioriteter.
- Rutingalgoritme: x-y-ruting.

KAPITTEL 5

IMPLEMENTERING

Dette kapittelet beskriver maskinvareimplementering av et NoC ved bruk av VHDL. Nettverket benytter synkron kommunikasjon ved èn global klokke og alle register settes på positiv flanke. Videre implementeres systemet med asynkron aktiv lav reset for alle register.



Figur 5.1: To sammenkoblede rutere

5.1 Ruter og topologi

Figur 5.1 viser en høynivåbeskrivelse av en ruter og hvordan denne kan kobles til en omkringliggende ruter. Hver ruter er implementert med de to håndtrykksignalene *Request To Send* og *Clear To Send*, samt en 8 bit databuss, *x out*, i hver retning. Utgangene og inngangangene er enkeltrettet, noe som betyr at data bare kan flyte i en retning i hver leder. Hver utgang har dermed en korresponderende inngang hos sin naboruter og vi observerer for eksempel at utgangen *RTS east* kobles til *X in ready west* på ruterens nabo i vest.

Prosjektruteren opererer med fire innganer på hver ruter, en fra hver av retningene nord, sør, øst og vest. Dette grensesnittet er nå videreutviklet til også å omfatte kommunikasjon med en lokal funksjonell modul. Dette fører med seg noe ekstra kombinatorikk i forbindelse med arbitrering og ruting. Dette er videre presentert i kapittel 5.3.1 og 5.3.4. Grensesnittet mot lokal funksjonell enhet vil være likt det som er mellom ruterne, noe som fører til at ruteren kommuniserer med sin lokale modul på samme måte som den kommuniserer med sine naborutere. Med to 8 bit busser samt fire håndtrykksignaler i hver retning gir dette totalt $(1 + 1 + 8) \cdot 2 = 20$ ledere for å koble sammen to rutere. Under følger en kort oversikt over utgangenes funksjon.

RTS *Signal* Request To Send settes høy for å indikere at en ruter ønsker å sende en pakke til en av sine naboer. Den aktuelle naboen vil da oppdage dette ved at **x in ready** går høy.

CTS *Signal* Clear To Send settes høy for å indikere til en nabo at ruteren er klar til å ta imot en pakke. Naboen vil da oppdage dette ved at **Cleared** går høy.

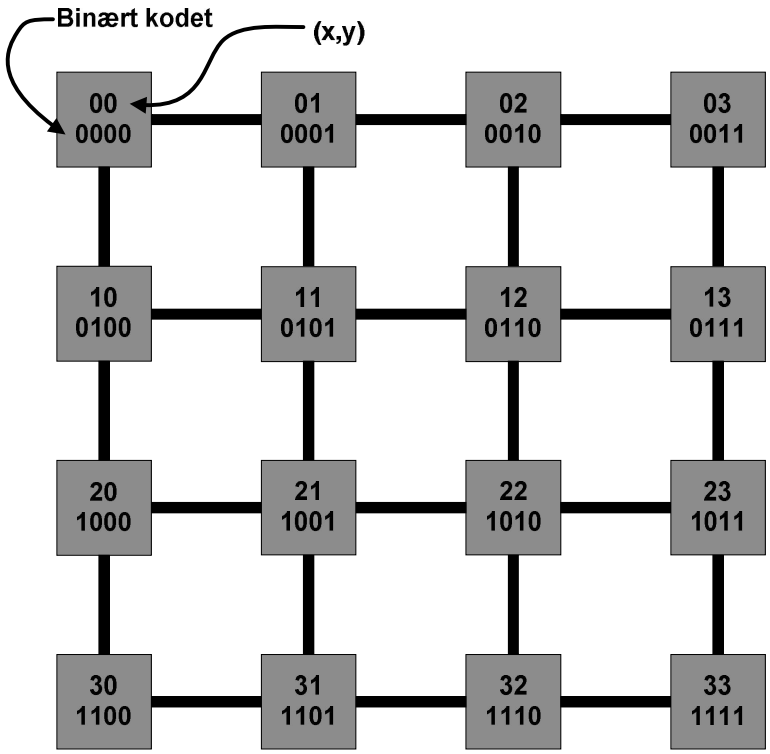
X out *8 bit databuss* Enkeltrettet buss for sending av data fra en ruter til en annen. Kobles til **X in**

Koordinat	Binær kode
0	00
1	01
2	10
3	11

Tabell 5.1: Binært kodede koordinater

Ruterne settes inn i et 4x4 mesh-nettverk etter figur 2.5 og hver ruter gis koordinater gitt av samme figur. Med 16 rutere trenger vi $\sqrt{16} = 4bit$ for å kunne gi alle ruterne en unik adresse. Koordinatene kodes binært på formen (x, y) etter tabell 5.1, noe som for eksempel vil gi ruter $(3, 2)$ i figuren koordinat *1110*. Figur 5.2 viser en oversikt over alle ruternes koordinater med deres binære koding. Vi observerer at alle rutere i samme rad vil ha de to første bitene like, mens alle rutere i samme kolonne vil ha de to siste bitene like. Et 4x4 mesh-nettverk vil ha 11 sammenkoblinger for hver rad og med fire rader gir dette 44 sammenkoblinger. Med 20 ledere per sammenkobling(som vist over)

gir dette totalt $44 \cdot 20 = 880$ endireksjonale ledere for å koble sammen alle rutere og lokale moduler i nettverket. Med denne organiseringen får vi et direkte nettverk av 16 rutere og mulighet for 16 funksjonelle enheter.

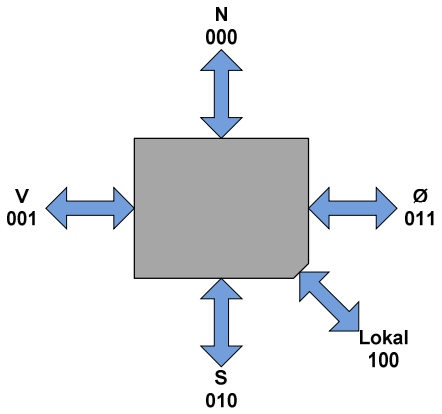


Figur 5.2: Binær koding av ruterkoordinater

Retning	Binær kode
Nord	000
Vest	001
Sør	010
Øst	011
Lokal	100

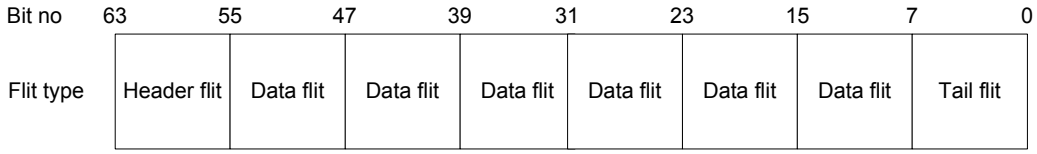
Tabell 5.2: Binært kodede retninger

Retningene orienteres etter standard karttopologi. Det vil si at opp i figur 5.2 tilsvarer nord, venstre tilsvarer vest, ned tilsvarer sør og høyre tilsvarer øst. Videre kodes retningene binært, med 5 bit, etter tabell 5.2. Figur 5.3 gir en oversikt over retningene, med deres koding, sett fra ruteren.



Figur 5.3: Binær koding av retningene

5.2 Pakkebeskrivelse



Figur 5.4: En pakke,[3]

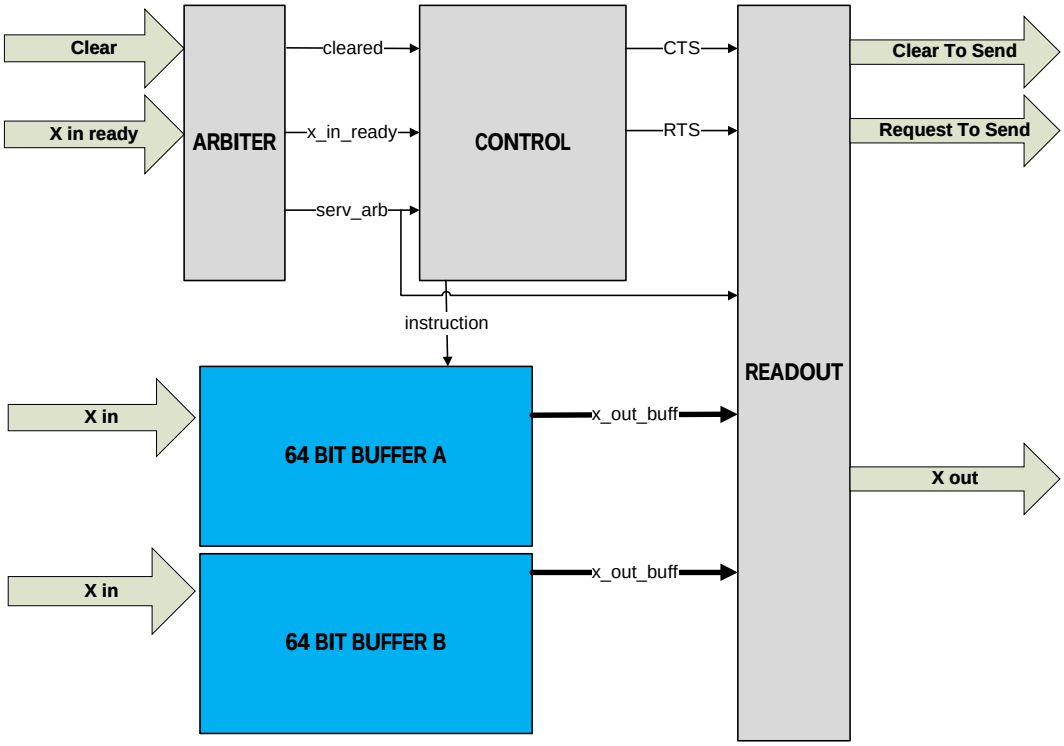
Pakkesvitsjing beskrevet i kapittel 2.2 velges som transmisjonstopologi. Pakkestørrelsen settes til 64 bit der hver pakke deles i 8 *flits* à 8 bit slik figur 5.4 viser.

7	6	5	4	3	2	1	0
SL	Addr(3)	Addr(2)	Addr(1)	Addr(0)	DATA	DATA	DATA

Figur 5.5: En header-flit,[3]

Figur 5.5 viser en *header-flit*. Header-flit'en inneholder en prioritets-tag(bit nr 8), også kalt Service Level(SL), relatert til kommunikasjonskrav. Èn bit prioritet gir to ulike prioritetsnivåer 0 og 1 der 0 er høyest prioritet. Videre inneholder bitene fra 6 til 3, av header'en, destinasjonsadressen. Denne 4 bit adressen korresponderer med destinasjonsruterens koordinater, noe som for eksempel betyr at en pakke med destinasjonsadresse *1010* vil rutes til ruter 22 slik det er vist i figur 2.3.1.

5.3 Ruter men 2x64 bit buffer



Figur 5.6: Rutermodulen

Ruteren implementeres bestående av de tre submodulene Arbiter, Control og Readout i tillegg til to lineære, uavhengige bufferkanaler, hver i form at et 64 bit skiftregister. **Arbiter** avgjør hvilken retning som skal få tilgang til denne ruterens ressurser, **Control** styrer inn og utlesing fra de to kanalene i tillegg til å avgjøre ruterens videre handlinger, mens **Readout** ruter signalene ut slik at pakkene blir sendt i riktig retning ut fra rutingalgoritmen. VHDL-kode for de fire modulene og deres sammenkobling er vedlagt i tillegg A.

Figur 5.7: Pipeline for ruteren

En pakke rutes igjennom ruteren i de fem stegene vist i figur 5.7. Med rødt, under de ulike stegene, vises hvilke moduler som styrer de ulike stegene og med blått over ser vi

hvor mange sykler de ulike stegene omfatter. Ved ankomst av en ny pakke til ruteren vil arbiter først bruke en klokkesykel på å detektere forespørselen, lese pakkens prioritet og arbitrere over hvilken retning som skal få sende. Control vil så bruke en sykel på å allokere et buffer til pakken som skal leses inn før den gir avsender klarsignal om å starte sending av pakken. Den neste sykelen bruker sender til å detektere at klarsignal er gitt og initiere utlesing av pakken. 8 sykler brukes så på å skifte inn de 8 flit'ene en og en, før en sykel brukes av Readout for å lese destinasjonsadressen fra bufferet og avgjøre hvilke retning pakken skal rutes videre til. Seksjonene under beskriver maksinvareimplementasjonen av de ulike modulene med en oppsummering av viktig funksjonalitet, signaler og andre ressurser benyttet av de ulike modulene.

5.3.1 Arbiter

For å kunne tilby QoS er det viktig at høyt prioriterte pakker håndteres før lavere prioriterte pakker. Arbiters oppgave er å sørge for at den høyest prioriterte pakken som til en hver tid befinner seg på inngangen er den som blir rutet inn til bufferet. En Arbiter med samme formål ble implementert i forbindelse med prosjektarbeidet og denne er nå videreutviklet for også å håndtere utlesing lokalt. Videre er Arbiter forbedret med tanke på, i størst mulig grad, å tilby god QoS samt at den i minst mulig grad gir bidrag til ruterens kritiske sti. Konkret betyr dette at potensielt vanskelige situasjoner håndteres bedre, samt at arbitreringen er skilt ut som et eget steg i ruterens pipeline(se figur 5.7).

Med fem innganger har vi altså fem mulige utfall av inngans-arbitreringen(nord, sør, øst, vest eller lokal). Videre observerer vi at fem innganger gir $2^5 = 32$ ulike arbitreringssituasjoner. Arbiterenen holder kontroll på hvilken inngang som nå leses inn, og vil beordre kontrollmodulen om å fortsette å lese inn den samme inngangen så lenge det ikke ankommer noen høyere prioritert pakke. Ved å holde informasjon om hvilken retning som nå er under innlesing kan arbiter ut fra innkommende forespørsler avgjør om denne fortsatt skal leses inn eller avbrytes til fordel for en høyere prioritert pakke. Med denne implementasjonen må altså arbiteren håndtere $32^2 = 1024$ mulige senarioer eller overganger fra en retning til en annen. Innlesing fra lokal funksjonell modul prioriteres imidlertid alltid lavest uavhengig av pakkens prioritet. Dette gjør at Arbiter i praksis bare arbitrerer på de 4 naboruterne, og gir den lokale modulen mulighet til å lese ut hvis ingen andre forespørsler(*x in ready*) er satt. Dette gir $2^4 + 1 = 17$ ulike arbitreringssituasjoner og $17^2 = 289$ ulike senarioer. Dette resonnementet oppsummeres i punktene under der punkt to er det som benyttes av den implementerte arbiteren.

- 5 innganger med 2 tilhørende prioriteter gir $2^5 = 32$ arbitreringssituasjoner og $32^2 = 1024$ mulige overganger fra en retning til en annen.

- $4+1$ innganger med 2 tilhørende prioriteter gir $2^4+1 = 17$ arbitreringssituasjoner og $17^2 = 289$ mulige overganger fra en retning til en annen.

Arbiteren benytter fem ulike metoder for å avgjøre hvilken retning som skal håndteres, avhengig av hvor mange samtidige forespørsler som er satt[1]. Hvis **ingen forespørsler** er satt vil arbiteren håndtere lokale pakker om noen er klare, og hvis **en forespørsel** er satt vil arbiteren velge å håndtere denne.

I situasjoner der **to forespørsler** er satt, det vil si at to naboer prøver å sende en pakke til denne ruteren, vil Arbiter håndtere den pakken med høyest prioritet. Om de to pakkene har lik prioritet vil Arbiter håndtere den av pakkene som eventuelt allerede er delvis innlest. 6 av de 17 tilfellene inneholder to forespørsler, og koden under viser et eksempel der forespørsel foreligger fra sør og øst.

- Registeret SL_reg(0) holder prioriteten til pakken fra øst.
- Registeret SL_reg(1) holder prioriteten til pakken fra sør.
- Registeret SL_reg(2) holder prioriteten til pakken fra vest.
- Registeret SL_reg(3) holder prioriteten til pakken fra nord.
- Registeret serv lagrer retningen som håndteres(jamfør tabell 5.2)

Listing 5.1: Arbitrering mellom sør og øst

```
1 if SL_reg(1) = SL_reg(0) then
2     if serving(1 downto 0) = "10" then
3           serv <= serving;
4     else serv <= "011"; end if;
5 elsif SL_reg(1) = '0'
6     then serv <= "010";
7 else serv <= "011";
8 end if;
```

De vanskeligste situasjonene oppstår når pakker ankommer fra tre retninger samtidig. **Tre forespørsler** gir $2^3 = 8$ mulige prioritetskombinasjoner. For å spare ressurser antar vi at tre pakker ikke kan ankomme på samme klokkeflanke. Med dette som utgangspunkt velger Arbiter den av pakkene som har prioritet 0 hvis noen av dem har det. Om flere av pakkene har prioritet 0 eller at ingen av dem har det vil Arbiter fortsette å lese inn den pakken som er under innlesing. Antar vi at pakkene ikke ankommer samtidig vil, i de tilfellene der to eller tre av pakkene har prioritet 0, Arbiter allerede håndtere den høyest prioriterte pakken. Det samme gjelder hvis alle de tre pakkene har prioritet 1. Eksempelet under viser hvordan Arbiter håndterer tre ventende forespørsler fra vest, sør og øst.

Listing 5.2: Arbitrering med tre forespørsler

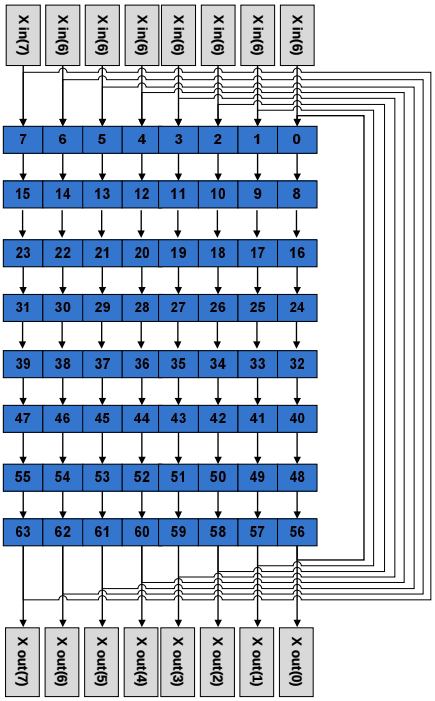
```
1 case SL_reg(2 downto 0) is
```

[1] Hvor mange av naboruterne som prøver å sende en pakke til denne ruteren.

```
2       when "011" => serv <= "001";
3       when "101" => serv <= "010";
4       when "110" => serv <= "011";
5       when others => serv <= serving;
6   end case;
```

Den siste muligheten, med **fire samtidige forespørsler** anses som meget usannsynlig og for å spare ressurser på FPGA'en er QoS deaktivert i denne situasjonen. Videre vil fire samtidige forespørsler gi stor sannsynlighet for vranglås og Arbiter vil alltid velge nord slik at en av pakkene kan leses inn og rutes videre så fort som mulig.

5.3.2 Bufferstruktur



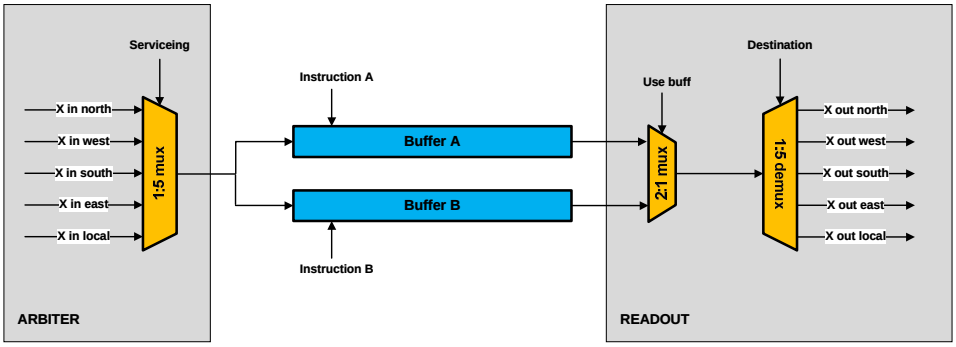
Intruksjon	Binær kode
IDLE	10,11
READ IN	00
READ OUT	01

Figur 5.8: Bufferet, et 64 bit skiftregister

I prosjektarbeidet ble en 64 bit, lineær bufferkanal implementert slik figur 5.8 viser. Bufferet er implementert som et skiftregister, der data leses inn på bufferet på bitene 0-7, skiftes igjennom slik figuren viser, og leses ut fra bitene 55-63 i enden av kanalen. Bufferet skifter altså inn en flit(8 bit) per klokkesykel og hver flit vil dermed bruke 8 klokkesykler på å bli skiftet igjennom hele kanalen.

For kontroll av kanalen er bufferet implementert med de tre instruksjonene *idle*, *readin*

og *readout* slik tabellen tilknyttet figur 5.8 viser. Ved instruksjon **IDLE** vil bufferet bare holde på de data som ligger der, uten å gjøre noen form for skifting. Gis instruksjonen **READIN** vil bufferet lese inn nye data fra bitene $0-7$, skifte dem igjennom kanalen og dermed skrive over de data som ligger i bufferet. **READOUT** beordrer bufferet til å skifte ut de data som ligger lagret i kanalen. Dette gjøres ved at flit'ene når enden av bufferkanalen, hvor de leses ut til riktig retning, før de skiftes rundt til starten av bufferet igjen. På denne måten vil bufferet ha lagret alle de 8 flit'ene fra en pakke helt til hele pakken er lest ut. Om en utlesing blir avbrutt av mottaker, eller at en annen feil oppstår, vil pakken fremdeles være lagret i bufferet slik at ruteren kan retransmittere den om nødvendig.



Figur 5.9: To virtuelle kanaler

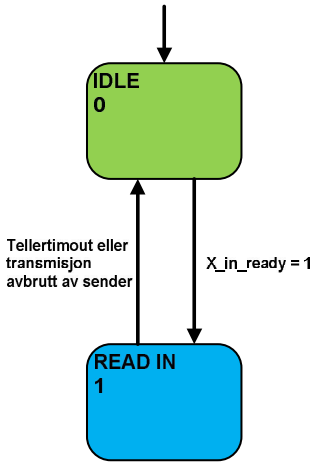
Som vi ser av modulbeskrivelsen i figur 5.6 benytter ruteren, slik den er implementert i dette arbeidet, to slike bufferkanaler. Denne videreutviklingen fra prosjektruteren gjør at nettverket nå er implementert med to virtuelle kanaler for hver ruter der signalene mux'es inn og ut fra de to kanalene slik figur 5.9 viser. Den eneste forandringen som er gjort på selve bufferet er at en mux for å rute signalene inn fra de ulike retningene er flyttet ut fra selve buffermodulen slik at den ikke blir duplisert i ruteren.

De to kanalene, kanal A og kanal B, opererer fullstendig uavhengig av hverandre, og styres av to tilstandsmaskiner i modulen Control. Med denne implementteringen kan den ene kanalen leses inn, samtidig som den andre kanalen leses ut, og motsatt. På grunn av at de to kanalene har en felles fysisk link ut og inn, vil det imidlertid ikke være mulig å lese inn eller sende ut to pakker samtidig. Denne problemstillingen er videre diskutert i kapittel 8.

5.3.3 Control

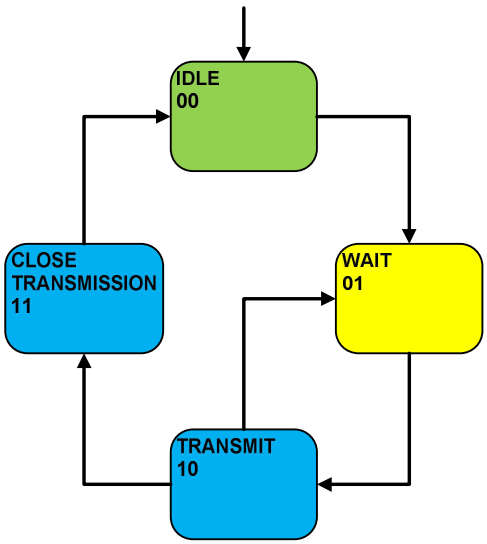
Modulen Control styrer ruterens handlinger i form av innlesing, utlesing, avbrudd og gjenopptagelse av avbrutte pakker. Basert på informasjon fra Arbiter om hvilken retning som skal håndteres beregner kontrollogikken ruterens videre handlinger ved bruk av to

tilstandsmaskiner(FSM), en for innlesing og en for utlesing. Da kontrollogikken vil være direkte begrensende for ruterens maksimale operasjonsfrekvens er implementasjonen av denne modulen bygget opp slik at de ulike tilstandene i størst mulig grad gir likt bidrag til ruterens kritiske sti. Videre holder kontrollmodulen informasjon om hvor mange pakker som er buffret, hvilke kanaler som er klare for utlesing og hvilke kanaler som er åpen for innlesing av nye pakker. Under en transmisjon vil altså sender lese ut pakken ved å benytte sin tilstandsmaskin for utlesing, mens mottaker leser inn pakken ved å benytte sin tilstandsmaskin for innlesing.



Figur 5.10: Tilstandsmaskin(FSM) for innlesing av pakker

Tilstandsmaskinen for **innlesing** av pakker består av de to tilstandene IDLE(*0*) og READ IN(*1*) som markert i figur 5.10. I tilstanden IDLE vil kontrollmodulen vente på signal fra Arbiter om at en ny pakke er klar til innlesing. I tilstanden READ IN vil kontrollmodulen gi instruksjon til en av bufferkanalen om å lese inn den ankommende pakken. Den første pakken som ankommer vil leses inn på kanal A. Ettersom nye pakker ankommer ruteren vil disse bli allokert en av kanalene avhengig av hvilken som er ledig. Om begge buffere er fulle vil kontrollogikken stenge for innlesing av nye pakker. Under innlesing vil en teller telle inn flit'ene, èn og èn, helt til alle de 8 flit'ene er mottatt. Ved ankomst av en høyere prioritert pakke under innlesing, vil Arbiter markere dette ved å indikere innlesing av en ny retning. Kontrollmodulen vil da avbryte innlesing av pakken for å lese inne den høyere prioriterte pakken. Når en pakke så er lest inn på en av kanalene setter kontrollmodulen kanalen klar til utlesing og gjør den samtidig utilgjengelig for innlesing av nye pakker.



Figur 5.11: Tilstandsmaskin(FSM) for sending av pakker

Utlesing av en pakke gjøres ved bruk av de fire tilstandene IDLE, WAIT, TRANSMIT og CLOSE TRANSMISSION vist av figur 5.11. Tilstandsmaskinen vil være i tilstanden IDLE så lenge det ikke befinner seg noen pakker i noen av kanalene. Når en pakke er lest inn vil tilstandsmaskinen gå til tilstanden WAIT for å lese ut fra bufferet pakken befinner seg i. I tilstanden WAIT sender kontrollmodulen en forespørsel til den av naboruterne pakken skal sendes til ved å sette *RTS* høy. Den så vente i denne tilstanden til klarering er gitt fra mottaker. Når klarering av mottatt går modulen til tilstanden TRANSMIT, og pakken skiftes ut fra bufferet, flit for flit, så lenge transmisjonen ikke blir avbrutt av mottaker. Blir sendingen avbrutt vil tilstandsmaskinen sørge for å skifte bufferet helt rundt slik at pakken igjen ligger klar til utlesing. Når en pakke er lest ut setter tilstandsmaskinen igjen kanalen klar til innlesing av nye pakker.

5.3.4 Readout

Readout er modulen som ruter signaler og buffer ut til riktig retning ut fra rutingalgoritmen. Rutingen skjer ved bruk av de tre parametrene *use_buff*, *serv* og *destination* der disse settes av henholdsvis Arbiter, Control og Readout selv. To signaler, *RTS* og *CTS*, i tillegg til bussen *x_out* skal rutes ut.

serv	Signal fra arbiter som markerer hvilken retning som håndteres nå. Klareringssignalet **CTS** rutes til denne retningen. En mux sørger for at kun èn retning kan få klarsignal til å sende.
use_buff	Register fra kontroll som marker hvilken kanal som skal rutes ut. READOUT leser så destinasjonsaddressen fra kanalen som skal rutes ut og beregner, etter x-y-algoritmen, ut fra dette hvilken retning pakken skal sendes til. Denne informasjonen lagres så i et register **destination**
destination	Register som holder retningen pakken i bufferet skal rutes til. Forespørselsignalet **RTS** og utgangen fra bufferet **x_out** rutes til denne retningen.

Som vi observerte av figur 5.2 vil en meshorganisering av ruterne føre til at destinasjonsadressene kan kodes på formen (x, y) slik at READOUT først ser på x-koordinaten og ruter pakken enten øst eller vest helt til pakken har nådd riktig kolonne. Deretter ser READOUT på y-koordinaten og ruter pakken til riktig rad. Når pakken så har ankommet riktig modul, og dermed har både riktig rad og kolonne vil READOUT rute pakken ut til sin lokale funksjonelle enhet.

KAPITTEL 6

SIMULERINGER, SYNTESE OG EVALUERING

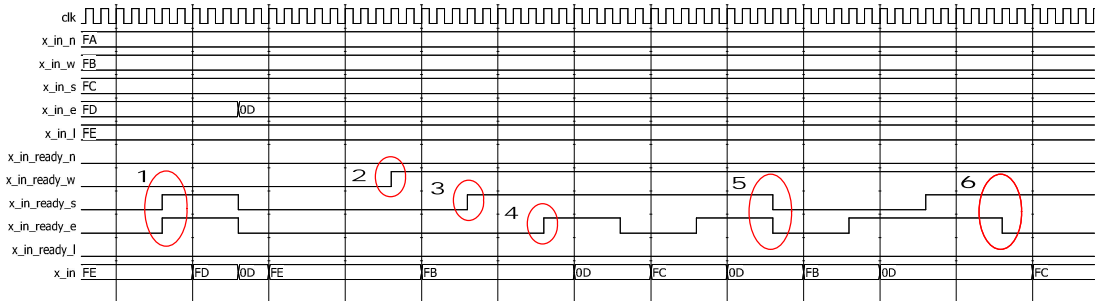
6.1 Simuleringer

For å vise nettverkets funksjonalitet presenteres her et utvalg simuleringer av systemet. Simuleringene er gjort ved bruk av Modelsim SE PLUS 6.3f fra Mentor Graphics med 50 MHz klokke. Intensjonen med simuleringene er å vise at nettverket slik det er presentert i kapittel 5, fungerer etter hensikten og at konseptet er realiserbart. Fokus er lagt på simulering av høyfrekvente situasjoner samt et utvalg hjørnetilfeller, men nettverket er ikke fullstendig testet.

6.1.1 Arbitrering

Enkle situasjoner med en pakke om gangen ble simulert i prosjektrapporten og leseren refereres til [3] for videre informasjon rundt dette. Under design av arbiteren er de fleste situasjoner simulert og det presenteres her er lite utvalg hjørnetilfeller for å vise dens funksjonalitet. På de fem inngansbussene settes henholdsvis *FA, FB, FC, FD* og *FE*. FA heksadesimalt gir den binære verdien *1111 1010* og ut fra figur 5.5 gir dette en pakke med prioritet 1, adresse 1111 og tre databit 010. Det sentrale for arbiteren er pakkenes prioritet og med dataene over vil alle pakkene få laveste prioritet(prioritet 1). Verdiene

A-E er satt på andre del av header'en for at inngangene skal være enklere å skille fra hverandre under simuleringen.



Figur 6.1: Arbiter, simuleringer

1. To pakker med samme prioritet ankommer fra sør og øst samtidig. Arbiteren velger å håndtere pakke fra sør.

2. Ingen pakker er klare og arbiteren ruter dermed inn lokal modul. En pakke ankommer fra øst, og arbiteren velger å håndtere denne fremfor å lese inn lokal modul.

3. En ny pakke ankommer fra sør under innlesing av pakken fra vest. De to pakkene har samme prioritet og arbiteren prioriterer å fortsette innlesing av pakken fra vest.

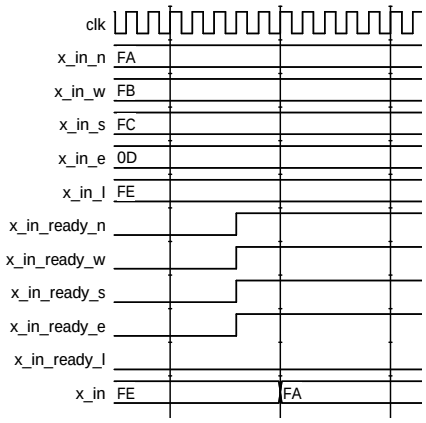
4. En tredje pakke ankommer arbiteren, denne gangen fra øst, men denne pakken har høyere prioritet(0). Arbiteren velger å håndtere denne pakken før de to lavere prioriterte pakkene.

5. Pakkene fra sør og øst er lest inn og arbiteren velger å håndtere pakken fra vest da denne fremdeles venter på å bli lest inn[1]

6. Igjen ankommer en høyere prioritert pakke fra øst.

Jamfører vi punkt 1 over med punkt 5 i kapittel 4.1.1 i prosjektrapporten[3] observerer vi at arbiteren nå er videreutviklet til å også å håndtere situasjoner der to pakker med samme prioritet ankommer på samme klokkeflanke. Vi merker oss at alle de simulerte situasjonene over håndteres riktig av Arbiter, samt at arbiteren nå håndterer to og tre innkommende pakker bedre(med bedre QoS) enn det som var tilfellet for ruteren utviklet i prosjektet.

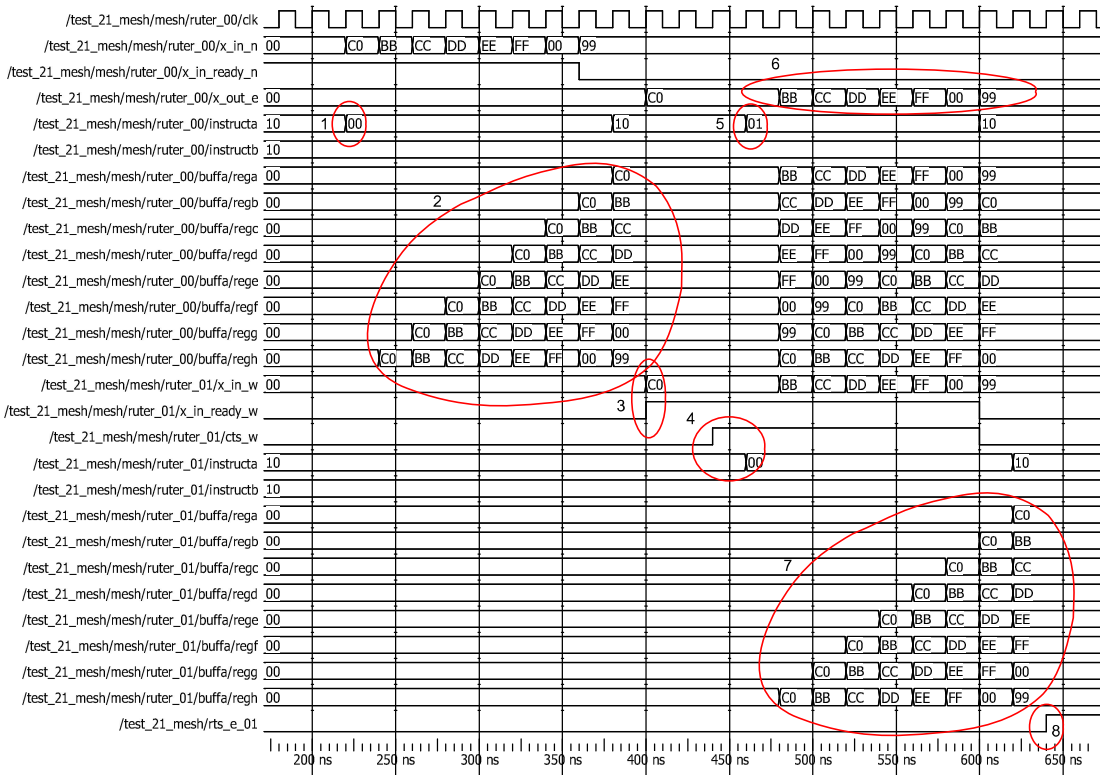
[1] Da ruteren ikke har mulighet for å lese inn to pakker samtidig vil ikke denne situasjonen kunne nås lovlig.



Figur 6.2: Arbiter, fire forespørsler

Figur 6.2 viser en situasjon med ankomst av fire pakker til en ruter. Som beskrevet i kapittel 5.3.1 vil arbiteren koble ut QoS, og prioritere innlesing av pakken fra nord. Pakken fra øst har høyest prioritet(0) og skulle blitt lest inn først om QoS hadde hvert aktivert. Situasjonen med fire pakker som skal sendes via en og samme ruter er imidlertid kritisk med tanke på vranglås og arbiteren prioriterer dermed å lese inn retning nord(tilfeldig valgt) så fort som mulig, fremfor å bruke store ressurser på å velge pakken med høyest prioritet.

6.1.2 Inn og utlesing fra to kanaler



Figur 6.3: Ruter, inn/utlesing av pakke

Figur 6.3 viser hvordan en pakke sendes fra en ruter til neste. De to aktuelle ruterne, ruter 00 og ruter 01, befinner seg i det nordvestre hjørnet av et mesh-nettverk og har henholdsvis 0000 og 0001 som koordinater(se figur 5.2). Pakken som sendes består av 8 flit'er kodet heksadesimalt til *C0,BB,CC,DD,EE,FF,00,99*. Som beskrevet i kapittel 5.2 bærer den første flit'en pakkens prioritet og destinasjonsadresse gitt av figur 5.5. Med den heksadesimale verdien C0, som kodes binært til 1100 0000, vil pakken få prioritet 0 og destinasjon 1000(node 02). Ut fra x-y-algoritmen beskrevet i kapittel 2.3.1, vil pakken dermed rutes rett østover fra ruter 00 til ruter 01. Under følger hendelsene i kronologisk rekkefølge etter tallene i figur 6.3.

1. En pakke ankommer ruter 00 fra nord. Pakken er klar til innlesing og kontrollmodulen gir beskjed om å lese inn på kanal A.

2. Pakken skiftes inn på kanalen, èn og èn flit. Den første flit'en ender etter 8 sykler opp i enden av registeret.

3. Pakken er lest inn, og destinasjonsadressen leses fra enden av bufferet. Destinasjonen, som er *1000*, befinner seg rett øst for ruter 00 og Readout velger dermed å

rute pakken østover. Første flit settes klar på bussen, slik at mottaker kan arbitrere på pakkens prioritet, og ruteren ber om å få sende pakken til sin østre nabo ved å sette RTS_east høy.

4. Ruter 01 er klar til å lese inn pakken, og gir dermed klarsignal til ruter 00 om å starte sending.

5. Ruter 00 leser pakken ut fra kanal A.

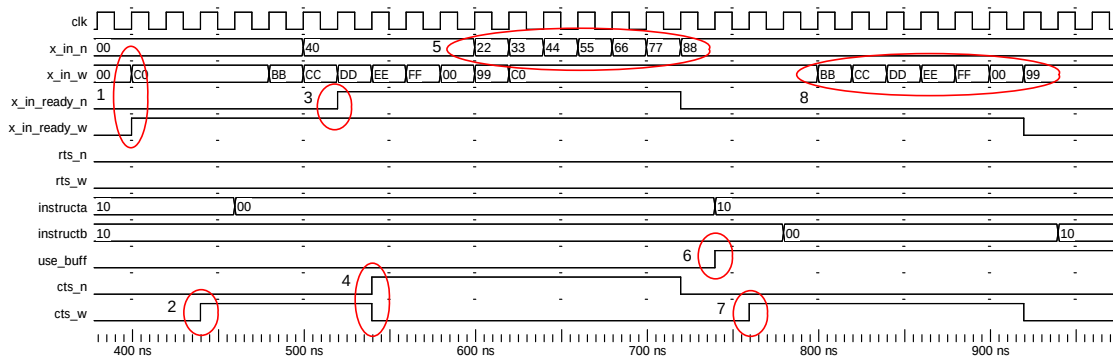
6. Flit'ene fra pakken skiftes fortløpende ut over bussen. Vi merker oss her at flit'en ikke slettes fra bufferet, men skiftes rundt. På denne måten kan ruter 00, om sendingen skulle bli avbrutt under utlesing, prøve å sende pakken på nytt.

7. Ruter 01 skifter pakken inn på kanal A.

8. Ruter 01 ruter pakken videre østover.

Simuleringen over viser at ruting av en pakke i et mesh-nettverk ved bruk av ruteren beskrevet i kapittel 5, fungerer etter hensikten. Bufferet bidrar til QoS ved å holde på flit'ene i en pakke, helt til hele pakken er lest ut. Samtidig merker vi oss at det går 12 klokkesykler fra ruter 00 er klar til å sende, og til ruter 01 er klar til å sende den samme pakken. Dette betyr at en pakke bruker 12 sykler på å traversere igjennom en ruter hvis den ikke støter på noen kryssende trafikk.

6.1.3 To pakker traverserer forbi hverandre



Figur 6.4: Ruter, avbrudd til fordel for høyere prioritert pakke

Simuleringene i figur 6.4 viser sending av den samme pakken som over, men den blir når avbrutt av en annen, høyere prioritert, pakke. Simuleringen viser avbrudd og gjenopptagelse av innlesing i ruter 01. Vi kaller pakken fra simuleringen i figur 6.3 pakke A og den nye pakken pakke B. Pakke A vil da som tidligere få prioritet 1 og destinasjon 02.

Pakke B består av de 8 flit'ene *40,22,33,44,55,66,77,88* og vil dermed få prioritet 0 og destinasjon 02, på samme måte som beskrevet i kapittel 6.1.2. Vi merker oss at de to pakkene, A og B, har samme destinasjon men at pakke B har høyere prioritet enn A.

1. Pakke A ankommer ruter 01 fra vest på samme måte som over i kapittel 6.1.2.
2. Ruter 01 gir klarering til pakke A om å starte sending av pakken, og pakken leses inn på kanal A.
3. Pakke B, med høyere prioritet, ankommer ruteren fra nord[2].
4. Ruter 01 avbryter innlesing av pakken fra vest til fordel for pakken fra nord.
5. Pakke B fra nord leses inn.
6. Pakke B er ferdig innlest og ruteren gjør klar til utlesing av den samme pakken. Samtidig bytter ruteren innlesingskanal slik at en ny pakke kan leses inn.
7. Ruter 01 gir klarsignal til sin vestre nabo om at den er klar til å motta pakken.
8. Pakke A fra vest leses inn på kanal B mens ruteren venter på å få lese ut pakke A fra kanal A.

Situasjonen over simulerer bare en av de mange mulighetene som ligger i bruk av to virtuelle kanaler. Som beskrevet i kapittel 5 har ruteren også mulighet for å sende pakke B videre under innlesing av pakke B. I og med at ruter 01 i simuleringen over ikke får klarsignal til å sende pakke B videre, vil den vente til den får det før den sender pakkene ut fra de to kanalene i den rekkefølge de ankom bufferet. Det vil si at den høyest prioriterte pakken, pakke B, vil bli sendt videre før pakke A.

Vi observerer at den høyprioriterte pakken, pakke B, glir rett igjennom ruteren uten å bli forsinket av pakke B. Dette til tross for at pakke B avbryter innlesing av pakke A. Pakke A bruker 27 sykler fra den er klar til innlesing til ruter 01 til den er klar til utlesing fra den samme ruteren. Den blir dermed forsinket $27 - 13 = 14$ sykler i forhold til simuleringen i figur 6.1.2. Vi merker oss at denne forsinkelsen vil være avhengig av når pakken blir avbrutt og hvor lenge den høyprioriterte pakken eventuelt må vente før den blir sendt videre.

6.2 Syntese

Her presenteres resultater og analyser fra syntese til Xilinx XC3S1000 256 FT FPGA gjort ved bruk av Xilinx ISE Project Navigator 10.1.03

[2]Merk at det ikke vil befinne seg noen node nord for node 01 men at andre noder nærmere sentrum av nettverket kan møte denne situasjonen.

6.2.1 Ruter

	Arbiter	Control	Buffer	Readout	Totalt
Antall Slices	53 (25,4%)	39 (18,3%)	2x39 (37,3%)	39 (18,7%)	209
Antall Flip Flops	15 (8,2%)	31 (17,0%)	2x64 (70,3%)	8 (4,4%)	182
Antall 4 inngangs LUT	96 (24,7%)	76 (19,5%)	2x73 (37,5%)	71 (18,3%)	389
Maksimal klokkefrekvens	132 MHz	164 MHz	258 MHz	260 MHz	

Tabell 6.1: En ruter, resultater av syntese

Tabell 6.1 viser resultater fra syntes av en ruter slik den er beskrevet i kapittel 5. Kolonnene viser ressursbruken for de ulike modulene samt de totale ressursene for å realisere en ruter på den aktuelle FPGA'en. Tallene i parentes angir hvor stor del av ruteren som brukes til å realisere den aktuelle modulen. Av det totale antall Slice'r(209) for å realisere èn ruter går altså 37,3% til buffring, 18,7% til kontrollogikk og 18,7% til ruting.

Arbiter har lavest operasjonsfrekvens(132 MHz), noe som betyr at denne modulen har den lengste kritiske stien. I og med at arbitrering er skilt ut som et eget steg i ruterens pipline, vil da denne modulen(Arbiter) også sette ruterens samlede operasjonsfrekvens.

Topologi	Antall slice'r	Slice'r per ruter	Operasjonsfrekvens
2x2 mesh	620(8%)	155	135 MHz
3x3 mesh	1491(19%)	166	125 MHz
4x4 mesh	2773(36%)	173	123 MHz
5x5 mesh	4198(54%)[3]	167	123 MHz

Tabell 6.2: Mesh-nettverk, resultater av syntese

Tabell 6.2 viser resultater av syntese for en rekke mesh-nettverk. Slice'r per ruter angir forholdet mellom ressursforbruk og antall noder for de ulike topologiene. Tallene evalueres i neste avsnitt.

6.2.2 Maksimal klokkefrekvens og skalerbarhet

Med 4x4 mesh vil en operasjonsfrekvens for nettverket på 123 MHz gi en teoretisk båndbredde på 1,3 GBit slik formel 6.1 viser. Det vil si at hver ruter kan lese inn 1,3 Gbit/s data eller omtrent 19 millioner pakker per sekund. Vi ser at klokkefrekvensen er den samme for 5x5 som 4x4 noe som vil gi samme båndbredde for denne topologien. I og med at vi har $5^2 - 4^2 = 9$ flere noder i 5x5 mesh enn i 4x4, vil imidlertid det totale antallet pakker nettverket kan håndtere være større og også skalere lineært med antall noder om vi utvider ytterligere.

[3]Med 4 av 6 addressebit

$$8 \cdot 8\,Bit \cdot \frac{123\,MHz}{12\,klokkeflanker} \cdot 2 = 1312\,Mbit/s = 1,3\,Gbit/s \tag{6.1}$$

Ved å pipeline kontrollogikken er ruterens operasjonsfrekvens gjort helt uavhengig av tilstandsmaskinene i kontrollmodulen. Der den kritiske stien for prosjektruteren gikk igjennom ruterens kontrollogikk bidrar denne nå bare til pakkens reisetid i form av økt forsinkelse(12 sykler mot 10 i prosjektruteren). Vi ser her fordelen med pipeline'ing av kontrollogikken, da dette resulterer i forbedret båndbredde. Hadde ikke den nye ruteren hatt muligheten for samtidig innlesing ville allikevel en pipeline'ing av kontrollogikken gitt en forbedring i båndbredde på minimum $1,3/2 - 640$[4] $= 16Mbit/s$[5].

Ser vi på skalerbarheten for nettverket finner vi av tabell 6.2 at et lavt antall noder(2x2) vil gi et noe lavere ressursforbruk per ruter enn for større nettverk(4x4). Dette kommer av at synteseverktøyet, for mindre nettverk, vil optimalisere bort en del unødvendige signaler og register. Alle ruterne er like, med inn og utganger til fire eksterne retninger, men i mindre nettverk som 2x2 vil i realiteten ingen av ruterene ta i mot pakker fra mer enn to andre retninger. Samtidig kan ruting gjøres enklere i slike nettverk. Dette, sammen med behovet for flere adressebit(ruterkoordinater) i større nettverk, gjør at nettverket vil skalere med et marginalt stigende ressursforbruk per ruter. På grunnlag av disse punktene kan vi si at skalerbarheten for nettverket er god.

6.3 Evaluering i forhold til tidligere arbeid

I prosjektarbeidet ble det vist at nettverket er bedre, både med tanke på båndbredde og ressursforbruk, enn tidligere arbeid gjort under AHEAD ved NTNU i Trondheim. Dette nettverket er nå videreutviklet og ved å sammenligne tabell 3.3 og 6.2, ser vi at for et 4x4 mesh opptar det videreutviklede nettverket $36 - 18 = 18\%$ større ressurser på Spartan 3. Dette er nesten en dobling av ressursforbruken(fra 1446 til 2773) i form av slice'r på FPGA'en. Hovedgrunnene til dette er en mer avansert arbitreringsstrategi for å tilby bedre QoS, samt innføring av en ekstra bufferkanal. Videre ser vi imidlertid at pipeline'ing av arbitrerings- og kontrollogikk gir bedre klokkefrekvens og med en ekstra kanal oppnås en båndbredde på $1,3GBit/s$ mot $640MBit/s$ for prosjektnettverket.

For å kunne sammenligne med HERMES-nettverket[6] presentert i kapittel 3.2 er et 2x2 mesh-nettverk syntetisert til Virtex II (XC2v500) og analysert. Denne FPGA'en har mindre ressurser enn Spartan 3, men er også raskere slik at vi oppnår en bedre båndbredde. Syntese av et 2x2 meshnettverk til Virtex II med *speedgrade -6* gir 651 Slice'r med en maksimal klokkefrekvens på $212,5MHz$. Hermes oppnår, for denne FPGA'en,

[4] Teoretisk båndbredde for prosjektruteren

[5] Dette tallet vil i realiteten være en god del større da den nye ruteren har mer avansert arbitreringslogikk, noe som er begrensende for operasjonsfrekvensen.

[6] Tallene det sammenlignes med her og i neste avsnitt er hentet fra tabell 1 i [39]

en båndbredde på $500Mb/s$ ved $25MHz$ ved bruk av 3058 slice'er.

- Vi ser da at nettverket foreslått i kapittel 5 gir en markant bedre båndbredde, samt at ressursforbruket vil være 1/5 sammenlignet med Hermes-nettverket.

Benytter vi formel 6.1 får vi, ved å bruke Virtex II, en teoretisk båndbredde på $2,25GBit/s$ per ruter om nettverket kjører på maksimal klokkefrekvens($212,5MHz$) eller $267MBit/s$ med en operasjonsfrekvens på $25MHz$. Det skal her legges til at HERMES-nettverket bare tilbyr beste-evneruting og dermed ikke har støtte for QoS.

Marescaux presenterer et ressursforbruk for Virtex II på $611Slices$ med en båndbredde på $320Mbit/s$ ved $40MHz$. Ved $40MHz$ gir nettverket foreslått i denne rapporten en båndbredde på $426Mbit/s$ per ruter for 2x2 mesh. Ser vi videre på nettverket presentert av Bartic opptar det $557Slices$ og gir en båndbredde på $800Mbit/s$ per virtuell kanal ved 50 MHz. Med to kanaler vil vårt foreslåtte nettverk gi en båndbredde på $267MBit/s$ per virtuelle kanal.

- Nettverket foreslått i dette arbeidet gir bedre båndbredde enn et nettverk foreslått av Marescaux
- Nettverket foreslått i dette arbeidet gir noe lavere båndbredde med omtrent samme ressursforbruk sammenlignet med et nettverk foreslått av Bartic.

I og med at HERMES-nettverket benytter samme topologi med samme rutingalgoritme som arbeidet foreslått i denne rapporten, kan disse to arbeid til en viss grad sammenlignes. På grunn av store forskjeller i hvilke tjenester som tilbys, er tallene fra Marescaux og Martic bare ment som en pekepinn på hvilken båndbredde og hvilket ressursforbruk som er realistisk for implementasjon av NoC på FPGA. Uten å vite konkret hvordan de ulike har beregnet sin båndbredde ser vi likevel at nettverket presentert i denne rapporten gir noe bedre resultater enn HERMES og Marascaux og noe lavere båndbredde enn Bartic.

6.4 Applikasjonsvurdering, HD-dekoding

Tidligere er filterapplikasjoner, flash-kontroller og små ALU'er foreslått som mulige anvendelsesområder for NoC. Vi ønsker nå å se hvordan det designede nettverket kan anvendes i kombinasjon med meget båndbreddekrevende applikasjoner. Som et eksempel foreslås her en metode for å utnytte nettverket til dekoding av høydefinisjonsvideo på FPGA. Det er her gjort en teoretisk analyse av båndbreddekrav og anvendbarhet i forbindelse med HD-dekoding.

I stedet for at en videofil lagrer alle bilder i en video, kodes videoen som en serie av vektorer. HD-dekoding er en metode for å generere en videostrøm fra en komprimert video. Dette betyr at dekoderen, ut fra den komprimerte videofilen, må generere en bitfil for hver pixel på skjermen videoen skal vises på. Dekoderen må da gjøre den samme beregningen for alle de $1920 \cdot 1080 = 2,1 millioner$ pixlene i et HD-bilde. Applikasjoner som utfører samme bergning mange ganger kan med stor fordel parallelliseres, noe som ofte utnyttes i grafikkort. Vi tenker oss nå at dekoderen, i stedet for å gjøre en operasjon om gangen, gjør 16 parallelle operasjoner. Disse 16 operasjonene skilles så i hver sin maskinvaremodul og legges ut på en FPGA. De 16 modulene kobles så til hver sin ruter i det designede 4x4 mesh-nettverket.

Store ressurser kreves for HD-dekoding, og vi velger dermed en Xilinx Virtex 5 FPGA for å håndtere dekodingen. Syntese av 4x4 mesh-nettverket til denne FPGA'en gir:

- 2861 slice'r med en operasjosfrekvens på 354 MHz

Etter at videoen er dekodet, og videostrømmen er generert, må denne sendes til skjermen, som vil ligge utenfor FPGA'en. Lar vi en av ruterne være FPGA'ens grensesnitt mot omverden betyr dette at all trafikk må gå til og fra denne ruteren. Videre gir en rå HD-videostrøm følgende datarate:

- $1920 \cdot 1080$ pixler/bilde $\cdot 30$ bilder/sekund $\cdot 8$ bit/bilde = 500 millioner $bit/sekund$

For å rute videostrømen fra de 16 dekodermodulene til sjermen igjennom en enkelt ruter krever altså dette en båndbredde på $500 MBit/s$. En operasjonsfrekvens for nettverket på 354 MHz gir en teoretisk båndbredde:

- $8 \cdot 8 \cdot 354/12 \cdot 2 = 3776 MBit/s = 3,78 GBit/s$

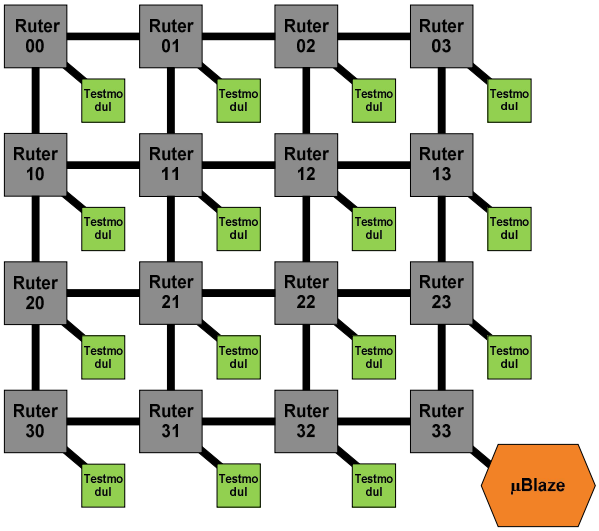
Vi ser altså at nettverket har god nok båndbredde til å rute videostrømmen fra dekoderne til skjermen. Det skal her tillegges at nettverket også må rute den komprimerte videostrømmen til dekoderne for at de skal kunne dekode videoen. Dataraten for denne strømmen vil imidlertid ikke ha signifikant størrelse i forhold til den dekomprimerte videoen.

4x4 mesh-nettverket tar opp 2861 slice'r på Virtex 5. Denne FPGA'en finnes i flere størrelser der den største (XCV5VLX330T) har 207360 slice'r. Et 4x4 mesh-nettverk vil dermed oppta omtrent 1‰ av de tilgjengelige ressursene på denne FPGA'en. Det vil her sannsynligvis ikke være behov for en så stor FPGA, og det vil være naturlig å velge den minste FPGA'en som har plass til ett 4x4 mesh-nettverk i tillegg til 16 HD-dekodere. Det er ikke gjort noen vurdering av ressursforbruket forbundet med HD-dekoding på FPGA.

KAPITTEL 7

TEST PÅ FPGA

Dette kapittelet presenterer en strategi for testing at et mesh-nettverk på Spartan 3 FPGA'en. En testmodulen er implementert i VHDL og simulert med Modelsim SE PLUS 6.3f. Programvaredelen av testen er skrevet i C og kompilert for kjøring på µBlaze. Testen er syntetisert, generert og lastet ned på Suzaku S-kortet ved bruk av en HP dv2115 laptop med Ubuntu 8.04 OS som kjører kjerne(kernel) 2-6-24. Arbeidet er ikke fullført og en videreføring vil være nødvendig for å kunne kjøre testen etter spesifikasjonene.



Figur 7.1: Testnettverket

7.1 Testmetode

Testen som presenteres i dette kapittelet tar utgangspunkt i et 4x4 mesh-nettverk av rutere, en μBlaze softcoreprosessor og 15 testmoduler.

Den eneste muligheten for å lese data fra en FPGA er igjennom FPGA'ens I/O. I og med at antall I/O på en FPGA en sterkt begrenset vil det være nødvendig med et definert grensesnitt mot omverden. For å oppnå dette settes ruter 33 i det sørvestre hjørnet av mesh-nettverket til å sørge for kommunikasjon mellom nodene i nettverket og prosessoren. Dette vil si at ruter 33 vil se prosessoren som sin lokale funksjonelle modul og kommuniserer da med denne som om den var en maskinvaremodul. Som vi skal se i kapittel 7.3 foregår denne kommunikasjonen i realiteten igjennom et HW-SW grensesnitt

Nettverket, med testmodulene og prosessoren legges ut på FPGA'en slik figur 7.1 viser. Prosessoren(μBlaze) kobles til ruter 33 i det sørvestre hjørnet av mesh-nettverket mens testmodulene kobles til de resterende 15 ruterne. Med denne tilkoblingen vil det altså være mulig for et program, kjørende på μBlaze, å kommunisere med maskinvare på FPGA'en. En konfigurasjonsfil genereres for maskinvaredelen ved bruk av Xilinx EDK 8.2.02i. For en detaljert gjennomgang av denne prosessen refereres leseren til [47], [48], [49].

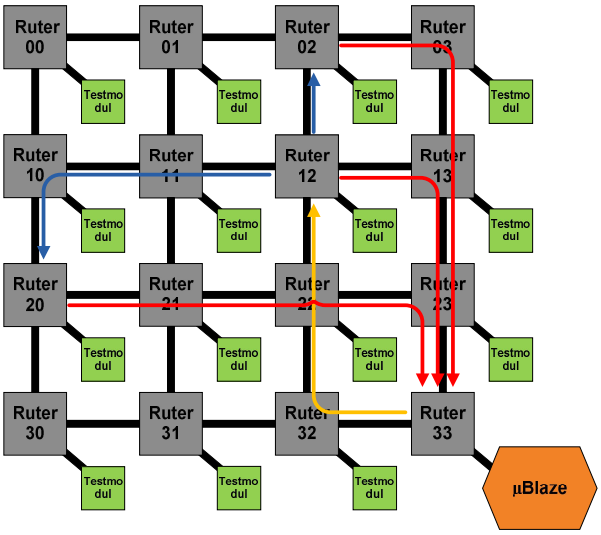
- **Nettverket** ruter pakker til og fra de 15 testmodulene
- **Testmodulene** analyserer og svarer på de pakkene de mottar
- **Prosessoren** generer pakker til nettverket og leser ut de pakker den får i retur.

Målet med testen er å finne ut hvordan nettverket håndterer kryssende trafikk samt hva som skjer ved store pakkerater av ulike prioritet. For at omverden skal kunne lese ut informasjon om hvilke pakker som befinner seg i nettverket må altså testmodulene generere denne informasjonen og sende den til adresse 33 hvor prosessoren befinner seg.

7.2 Testmodulen(maskinvare)

Testmodulen er implementert i to versjoner der den ene er en videreføring av den andre. Som ved implementering av selve nettverket, er det også her lagt vekt på at alle testmodulene skal være like og operere på samme måte. Under test sender prosessoren ut en testpakke med en adresse. Når denne testpakken har ankommet noden den er adressert til vil testmodulen lese ut denne pakken og analysere den.

I en **enkel versjon** genererer testmodulen, ved mottak av en testpakke, en ny pakke med samme data som det den mottok men med destinasjon 33. På denne måten vil prosessoren, en tid etter at testpakken er sendt ut, motta en kvittering fra mottaker. Med denne kvitteringen kan testprogrammet konkludere med at pakken er rutet riktig frem til sin destinasjon og at kvitteringen er rutet riktig tilbake til prosessoren. Her kunne det også hvert lagt til ytterligere funksjoner som for eksempel en tidsstempling av pakkene, informasjon om hvilken rute pakkene har tatt igjennom nettverket eller om de er avbrutt(og eventuelt hvor mange ganger) av andre pakker.



Figur 7.2: Ruting av testpakker i testnettverket

Svakheten med modulen over er at all kommunikasjon vil foregå mellom en enkelt node i nettverket og prosessoren. Med denne metoden blir det vanskelig å generere kryssende trafikk i nettverket. For kunne generere et mer realistisk trafikksenario er testmodulen videreutviklet.

I en **avansert versjon** generer prosessoren på samme måte som over en testpakke med en adresse i nettverket. Det som skjer videre kan deles i tre steg.

1. Testmodulen som mottar den gule pakken fra prosessoren genererer to ny testpakker(markert med blått)
2. Testmodulen som mottar den gule pakken gir en kvittering tilbake til prosessoren med samme data den mottok
3. De to testmodulene som mottar de blå pakkene gir en kvittering tilbake til prosessoren

Figur 7.2 viser et eksempel på en slik testpakke(markert med gult) med destinasjon 12.

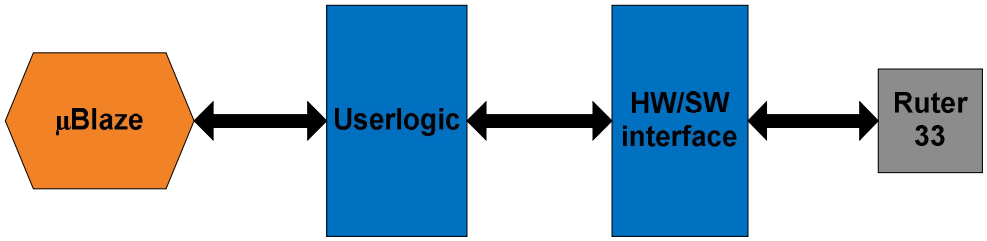
Når testmodulen(ruter 12) mottar testpakken genereres nå to nye pakker til nettverket. I eksempelet er disse markert med blått og har destinasjon 20 og 02. Testpakken fra prosessoren vil bestå av 8 flit'er og av disse brukes de fire første til å lage en ny testpakke med en annen adresse i nettverket(blå pakke til ruter 20). På samme måte brukes de fire siste flit'ene fra den originale testpakken(gul pakke) til å generere den andre pakken(blå pakke til ruter 02).

Når testmodulen har gjort dette sender den en kvittering(rød pakke fra ruter 12 til 33) tilbake til prosessoren for å bekrefte at testpakken er mottatt og at to nye testpakker er generert. I tillegg til kvitteringen fra ruter 12 mottar også prosessoren kvittering fra ruterne som mottar de to nye testpakkene. Prosessoren vil som et resultat av dette motta tre kvitteringer fra de ruterne som har mottatt de tre testpakkene. Prosessoren vil nå altså, ved å sende ut èn testpakke, generere seks pakker i nettverket, inkludert mulig kryssende pakker. På denne måten kan prosessoren observere hvilken rekkefølge bekreftelsene kommer tilbake og dermed, på en bedre måte, analysere nettverkets ytelse.

7.3 Testprogrammet(programvare)

Programvaredelen består av et testprogram som kjører på μBlaze. For å håndtere schedulering og synkronisering av oppgaver i prosessoren kjører prosessoren en linux-kjerne(uClinux 2.4) utviklet av Atmark Techno INC. Kjernen kompileres og lastes til kortet via serielt RS232-grensesnitt.

For å teste nettverket med testmodulene beskrevet over kjører μBlaze et testprogram. Dette programmet lar brukeren generere pakker til nettverket. Samtidig skal programmet lese ut de pakker som kommer tilbake til μBlaze fra nettverket.



Figur 7.3: Testnettverk, grensesnitt mellom prosessor og maskinvare

For at programvare som kjører på μBlaze skal kunne kommunisere med nettverket opprettes et grensesnitt mellom prosessor og maskinvare. Dette består i de to modulene vist i figur 7.3. En HP Pavilion dv2115 er satt opp med Ubuntu 8.04 for å kommunisere med Suzaku-kortet. PC'en kommuniserer med kortet via serielt RS232-grensesnitt og Ubuntu benytter GtkTerm som terminal for å skrive til/lese fra Suzaku. For å synkroni-

sere kommunikasjonen settes terminalen opp med *baud rate = 115200* og ingen paritet. Userlogic inneholder fem slave-register, à 32 bit, hvis formål er gitt av oversikten under.

slv_reg0	Holde de 32 minst signifikante bit i testpakken fra testprogrammet
slv_reg1	Holde de 32 mest signifikante bit i testpakken fra testprogrammet
slv_reg2	Holde de 32 minst signifikante bit i kvitteringen fra testmodulen
slv_reg3	Holde de 32 mest signifikante bit i kvitteringen fra testmodulen
slv_reg4	Holde status og resetinformasjon

Vi starter med å teste om testprogrammet klarer å skrive til/lese fra de fem registrene. Programmet gir da følgene print til *GtkTerm*.

Listing 7.1: Lese og skrive til maskinvare

```
1
2  ---------------------------------------------
3  This is a packet generator for the 4*4 QNoC
4  ---------------------------------------------
5  Press 1 to generate own testpacket
6  Press 2 to generate random testpackets
7  Press 3 to write testdata
8  Press 4 to run hw-reset
9  Press 5 to cancel hw-reset
10 Press 0 to terminate
11 3
12 Enter number of slave registers
13 5
14 Read for slv_reg0 : 7. Correct
15 Read for slv_reg1 : 10. Correct
16 Read for slv_reg2 : 13. Correct
17 Read for slv_reg3 : 16. Correct
18 Read for slv_reg4 : 19. Correct
19 ---------------------------------------------
```

Som vist skriver testprogrammet verdier til de fem registrene før det leser de samme verdiene tilbake. Dette fungerer etter hensikten.

For å sende en testpakke ut i nettverket skriver testprogrammet pakken til *slv_reg0* (første del) og *slv_reg1* (andre del) og setter statusregisteret *slv_reg4* til å sende testpakken ut i nettverket. Testprogrammet har to moduser der den ene lar brukeren velge hvilken testpakke som skal sendes ut. Brukeren blir da bedt om å velge adresse for de tre pakkene(den gule og de to blå pakkene i figur 7.2) og programmet generer den ønskede pakken. Den andre modusen lar brukeren velge et antall tilfeldige pakker som testprogrammet printer til terminalen før de sendes ut i nettverket. Under vises terminal-print fra de to modusene.

Listing 7.2: Generere egen pakke

```
1
2  ---------------------------------------------
3  This is a packet generator for the 4*4 QNoC
4  ---------------------------------------------
5  Press 1 to generate own testpacket
6  Press 2 to generate random testpackets
7  Press 3 to write testdata
8  Press 4 to run hw-reset
9  Press 5 to cancel hw-reset
10 Press 0 to terminate
11 1
12 Tast inn adresse 0: A
13 Tast inn adresse 1: B
14 Tast inn adresse 2: C
15 Addr0: a Addr1 : b Addr2: c
16 Sending packet: 50580000 60000000
```

```
17  Read from slv_reg0 and 1: 50580000 60000000
18  Current status: 1
19  ------------------------------------------------
```

Som vi ser over velger brukeren å generere en egen pakke. Brukeren blir da bedt om å gi inn tre verdier. Disse tre verdiene representerer adressene for hendoldsvis den gule og de to blå pakkene i figur 7.2. I dette eksempelet får de tre pakkene adresse 22, 32 og 03. Alle pakkene gis prioritet 0 slik at adresse 22 gir headeren 01010000 eller 50 kodet heksadesimalt. De tre siste bitene markerer at dette er en gul pakke. For de blå pakkene setter testmodulene disse bitene til 111. Testprogrammet kontrollerer til slutt om pakken er skrevet riktig til slaveregistrene.

Listing 7.3: Generere tre tilfeldige pakker

```
1   Enter number of ramdom packets to generate
2   3
3   Packet 1:
4   Addr0: 94 Addr1 : ce Addr2: af
5   Sending packet: 20700000 78000000
6   Read from slv_reg0 and 1: 20700000 78000000
7   Current status: 1
8   Packet 2:
9   Addr0: 4b Addr1 : 13 Addr2: 94
10  Sending packet: 58180000 a0000000
11  Read from slv_reg0 and 1: 58180000 a0000000
12  Current status: 1
13  Packet 3:
14  Addr0: 35 Addr1 : 49 Addr2: 49
15  Sending packet: 28480000 48000000
16  Read from slv_reg0 and 1: 28480000 48000000
17  Current status: 1
18  ------------------------------------------------
```

Brukeren har her valgt å generere tilfeldige pakker til nettverket. Brukeren vil da få spørsmål om hvor mange pakker han vil generere. Deretter sender deretter ut dette antallet pakker med tilfeldige destinasjoner i nettverket.

Den andre delen av testprogrammet består i å detektere kvitteringene som kommer tilbake til prosessoren(markert med rødt i fig 7.2) Når *HW/SW interface* har mottatt en pakke fra nettverket, skriver den pakken til *slv_reg2* og *slv_reg3*. I tillegg setter *HW/SW interface* statusregisteret for å markere til programvaren at en kvittering er ankommet. Programvaren leser pakken ved å lese innholdet i *slv_reg2* og *slv_reg3*. Denne delen av testprogrammet er ikke ferdigstilt og en videreføring vil her være nødvendig for å kunne teste sending av pakker i nettverket. Undersøkelser viser at grensesnittet(*HW/SW interface*) ikke opererer etter hensikten. Når testprogrammet har satt statusregisteret til å sende ut testpakken reagerer ikke grensesnittet på denne informasjonen.

KAPITTEL 8

DISKUSJON

Nettverk for enbrikkesystemer, eller NoC, er et stort emne som spenner seg over et stort antall fagområder. Ulike utfordringer kan løses med ulike teknologier, på ulike abstraksjonslag enten som programvaremoduler eller som raske maskinvareimplementasjonener. Arbeidet i denne rapporten tar for seg maskinvareimplementering av enkelte teknologier. Rapporten er likevel ikke ment som en totalvurdering av hva som kan være en optimal implementasjon av NoC. Ulike nettverkslag kan utelukke hverandre, og arbeidet presentert i denne rapporten tar bare for seg en liten del av de utfordringene NoC kan møte på. Det skal her også nevnes at NoC er et forholdsvis nytt fagfelt, introdusert i 2001. Da mange forskere er overbevist om at NoC vil være en nøkkelfaktor for å dekke produktivitetsgapet samt å utnytte dynamisk rekonfigurering antas det at mye arbeid vil bli gjort innenfor dette emnet i de kommende årene.

Flere moderne FPGA'er har støtte for dynamisk rekonfigurering men på grunn blant annet ufordring rundt ruting av signaler mellom moduler er mulighetene som ligger i dynamisk rekonfigurering sjelden utnyttet av brukerne. Det er grunn til å tro at FPGA-produsentene i fremtiden vil komme med egne NoC, spesielt designet for deres FPGA'er, som en konfigurasjon eller som innebygget maskinvare i FPGA'en.

Utgangspunktet for arbeidet var et NoC utviklet under AHEAD-prosjektet ved NTNU høsten 2008. Med en hypotese om at dette nettverket kan videreutvikles til å tilby bedre QoS samt å håndtere større pakkerater ble en rekke løsninger for svitsjing, ruting og arbitrering vurdert(kap 4). Arbeidet, som har bestått i et litteraturstudium, en implementasjonsdel og en del for praktisk testing ble utført over fem måneder våren 2009.

Under følger en diskusjon av de valgene som er gjort, med en vurdering av de resultatene som er oppnådd. Videre foreslås metoder for å tilpasse det implementerte nettverket til andre topologier, videreføre arbitreringsstrategi samt forbedre funksjonalitet, ytelse, ressursforbruk og båndbredde ved optimalisering og ytterligere pipeline'ing. For flere detaljer rundt de ulike designvalg som er gjort refereres leseren til kapittel 4.

8.1 Nettverkstopologier

En rekke topologier er vurdert og undersøkt før en 2D-mesh-topologi er valgt. Topologien viser god og stabil ytelse i tillegg til at enkel adressering og ruting minimerer behovet for kompleks kombinatorikk i ruterne. På grunn av mesh-topologiens tendens til opphopning av pakker rundt sentrum av nettverket kan det her imidlertid være interessant å se på alternative topologier. En enkel videreføring her kan være å teste ut torus-topologien. Denne topologien er imidlertid lite anvendbar på FPGA på grunn av behovet for å rute signaler fra ytterkant til ytterkant i nettverket(se kap 2.3). Mange moderne FPGA'er inneholder globale ledere som kunne vært brukt til dette formålet men det er usikkert om dette vil gi lik nærhet mellom alle noder. Forfatteren anser en videreutvikling til *folded torus* eller *hyperkube* som mer interessant da disse oppfyller nærhetskravet bedre. Denne topologien vil gi en noe mer avansert adressering og ruting, men ruterens kontrolllogikk, arbitreringsmetode og grensesnitt antas å kunne videreføres til denne topologien uten større forandringer. En mulig videreutvikling av adresseringen av nodene i nettverket er at avsender definerer hvor mange hopp en pakke skal gjøre i x-retning og i y-retning. Dette gjør at nodene slipper å holde informasjon om hvor de befinner seg i nettverket i form av sine (x, y)-koordinater. Med dette kan vi, i et 4x4-mesh-nettverk spare ett 4 bit register per ruter kun ved denne modifikasjonen. I tillegg kan nettverket da skaleres opp uten at man trenger å legge til flere adressebit.

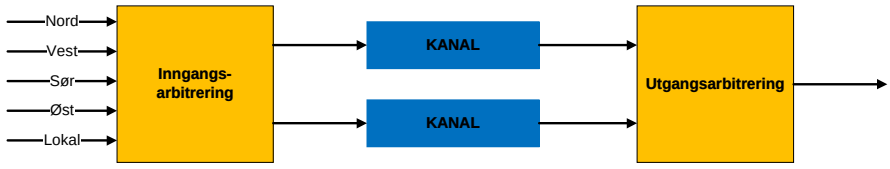
8.2 Arbitrering og ruting

Ruteren er implementert med en inngangsarbitrering for å velge den av pakkene på de fem inngangene nord, sør, øst, vest og lokal som til en hver tid har høyest prioritet. Arbiteren er delt opp i fem deler, beskrevet i kapittel 5.3.1 for å gjøre den så parallell som mulig. Med denne implementasjonen er målet å gi alle arbitreringssituasjoner så lik forsinkelse som mulig. Arbiteren gir direkte bidrag til ruterens kritiske sti og vil dermed være begrensende for nettverkets operasjonsfrekvens noe som fører til at en mest mulig parallell implementering vil være fordelaktig.

Det er likevel slik at den vanskeligste situasjonen, der 3 pakker er klare samtidig på ruterens innganger, vil gi større bidrag til kritisk sti enn de andre arbitreringssituasjonene. Av denne grunn kan en ytterligere pipeline'ing av arbitreringen være fordelaktig med tanke på båndbredde. Arbiteren bruker en klokkesykel på å avgjøre hvilken retning som

skal håndteres og vi kan her tenke oss at tre samtidige forespørsler kan pipeline's og dermed sees på som en kombinasjon av $2+1$ forespørsler. Med dette menes at arbiteren først arbitrerer på to av inngangene før den på neste klokkesykel sammenligner denne arbitreringen med den tredje inngangen. På denne måten kan vi med et tredje steg også arbitrere med en eventuell fjerde forespørsel. Dette vil gi økt maksimal forsinkelse igjennom arbiteren men med denne pipeline'ingen kan en og to forespørsler håndteres raskere enn før. Det antas dermed at dette vil gi en kortere gjennomsnittlig forsinkelse.

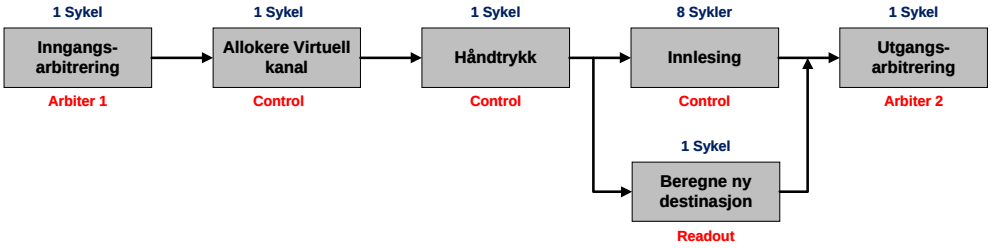
Vi merker oss her imidlertid at forsinkelsen gjennom kontrollogikken er omtrent like lang som forsinkelsen igjennom arbiteren. Kontrolllogikken må dermed ytterligere pipeline's for å få fullt utbytte av en eventuell pipeline'ing av arbitreringen.



Figur 8.1: Videreutvikling av arbitreringsstrategien, jamfør figur 4.5

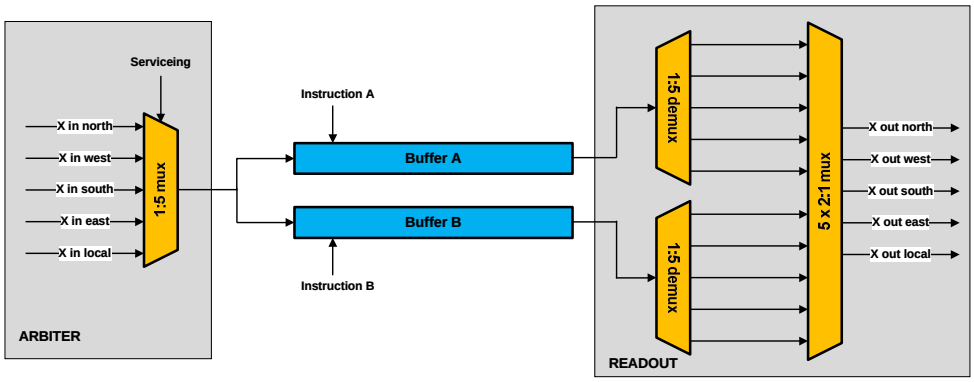
Videre er ruteren bygget opp slik at det ved relativt enkle grep vil være mulig å videreutvikle den til også å utføre en arbitrering på utgangen av de to bufferkanalene. Slik ruteren er bygget opp nå, vil den alltid sende den pakken som først ankommer bufferet. Med en videreutvikling ønsker vi å innføre muligheten for også å kunne arbitrere mellom de to kanalene. Om en høyere prioritert pakke ankommer en kanal under utlesing av den andre kanalen vil ruteren da ha mulighet til å avbryte utlesing av denne pakken til fordel for den sist ankomne pakken(om denne har høyere prioritet). Med dette vil det altså være mulig for en pakke å traversere forbi en annen pakke også inne i ruterens kanaler. Konkret kan dette gjøres ved at de to kanalene arver prioriteten til pakken som er buffret der. Kontrollmodulen kan da, i sin TRANSMIT-tilstand(se figur 5.11), sjekke om kanalen den nå leser ut er kanalen med høyest prioritet. Ingen modifikasjoner behøves på mottakersiden(innlesingsmekanismen) da ruteren allerede har logikk for å håndtere avbrudd fra sender. Denne løsningen vil i enda større grad prioritere noen pakker fremfor andre men kan også gi større total reisetid for alle pakker i og med at metoden gir mulighet for flere avbrudd.

En løsning med lookahead-ruting er også vurdert. Løsningen som er implementert baserer seg på at *readout*-modulen leser destinasjonsadressen fra pakkens første flit i det den er på vei inn i bufferet. Problemer rundt avbrudd gjorde imidlertid at denne løsningen ble forkastet til fordel for et ekstra pipelinesteg. Med en smartere implementering kunne denne løsningen gitt gode resultater. En mulighet er at videre ruting for pakkene avgjøres under innlesing og at hver kanal dermed blir tagget med informasjon om hvilken retning den skal rutes videre til.



Figur 8.2: Videreutvikling av ruterpipeline, jamfør figur 5.7

Ved å benytte lookahead-ruting, samt at en utgangsarbitrering utføres i tillegg til inngangsarbitreringen kan ruteren benytte en pipeline som den i figur 8.2, med parallelle steg for innlesing av pakke og beregning av ny destinasjon. Denne løsningen vil gi både en inngangskø og en utgangskø, og gir dermed veldig god QoS. Dette begrunnes med at høyprioriterte pakker da kan avbryte andre pakker både på vei inn og på vei ut av en ruter. Dette krever store arbitreringsressurser og en mulighet her er og ganske enkelt la de to kanalene lese inn den første pakken som ankommer på inngangen. På denne måten kan en utgangsarbitrering velge hvilken av de to kanalene som skal få lese ut sin pakke først. Det vil da også være mulig for en ruter, om den ikke får sendt pakken med høyest prioritet med den gang, å forsøke å sende den andre pakken mens den venter. På grunn av at man her bare har to kanaler vil denne utgangsarbitreringen kunne gjøres enkelt og raskt. Det er her imidlertid viktig å påpeke at en løsning uten inngangsarbitrering ikke vil gi like god QoS som implementasjonen beskrevet i kapittel 5. På grunn av antatt lavere arbitreringskostnad anses denne videreutviklingen likevel som interessant for ytterligere studier.



Figur 8.3: Rute ut to samtidig

Som et alternativ til en utgangskø ligger muligheten for å rute ut to pakker samtidig. Denne løsningen krever at de to kanalene ikke rutes ut til samme retning da det bare finnes en fysisk link mellom hver ruter men som arkitekturen i figur 8.3 viser vil en slik videreføring føre med seg et minimum av ekstra ressurser. Løsningen vil bedre totale

reisetiden ved store pakkerater.

8.3 Ytelse og ressursforbruk

Syntese av 2x2,3x3,4x4 og 5x5 mesh-nettverk viser at nettverket skalerer lineært både for båndbredde og ressursforbruk. Samtidig ser vi av tabell 6.2 at nettverkets operasjonsfrekvens ikke øker nevneverdig når nettverket skaleres opp. Vi kan på dette grunnlag konstatere at nettverket ikke inneholder asynkrone signaler.

I kapittel 3 presenteres et utvalg av de tidligere arbeidene som vurderes som viktige under temaet NoC. Under arbeidet med prosjektet ble det gjort en vurdering av ressursforbruket på en FPGA som følge av buffring av data(pakker) i en node. Realisering av minne på FPGA krever store ressurser og mange tidligere arbeider fokuserer på å holde bufferressursene i NoC på et absolutt minimum. Ved å benytte WH-svitsjing kan man oppnå dette men det viser seg at dette vil gå på bekostning av avansert kontrollogikk. WH-svitsjing krever at hver ruter hele tide holder informasjon om hvilke pakker som passerer igjennom ruteren og hvilken retning de ulike pakkene skal rutes til. Med mange pakker, flere prioritetsnivåer og mange kanaler vil dette føre til avansert kontrolllogikk og i mange tilfeller asynkrone signaler.

Nettverket presentert i denne rapporten viser at, sammenlignet med tidligere arbeid(se kapittel 6.3), SAF-svitsjing kan redusere kostnaden i form av konstrolllogikk samtidig som denne metoden gir bedre båndbredde og QoS. Dette begrunnes med at en pakke, ved bruk av SAF-svitsjing, aldri vil oppta ressurser i mer enn to rutere om gangen, noe som gir mindre sannsynlighet for avbrudd, vranglås og livelock. Dette bekreftes av MIT i California som har demonstrert forbedret ytelse i sin J-Mashine ved bruk av større buffer[19].

Videre ser vi at med denne metoden vil alle pakker som til en hver tid befinner seg i nettverket være fullstendig buffret i minst en ruter. Dette gir bedre garantier for at en pakke ikke blir overskrevet eller feilsendt enn ved WH-svitsjing, hvor pakkene ofte har lange haler av flit'er etter seg.

Både i prosjektet og denne rapporten er en rekke bufferstrukturer vurdert. Som vi har sett er en struktur med to lineære 64 bit kanaler funnet hensiktsmessig. Vi ser imidlertid at for store nettverk vil en slik struktur ta opp uforholdsmessig store ressurser på en Spartan 3 FPGA. Med et 5x5 nettverk av rutere opptar nettverket litt over halvparten av ressursene på den aktuelle FPGA'en. Det er her viktig å vurdere, ut fra aktuelle applikasjoner, hvilke ressurser det er hensiktsmessig å benytte til nettverket. Dette inkluderer en vurdering av behovet for noder i nettverket. Spartan-serien har relativt små ressurser i forhold til nye FPGA'er man vil dermed ikke ha behov for store nettverk. Som vi så i kapittel 6.4 vil nettverket ta opp et minimum av ressurser på en Virtex 5 FPGA. Skalerer vi opp nettverket etter tabell 6.2 ser vi at et 10x10 mesh-nettverk(100

noder) vil oppta ca 8,6%(18000 slice'r) på en Virtex 5(XC5VLX330T).

I og med at ruteren, bort sett fra arbitrering kommuniserer likt med alle de fem retningene kan man, uten å modifisere nettverket, erstatte en naboruter med en funksjonell modul. Ved å benytte et 2x2 mesh-nettverk kan man da koble en funksjonell modul til alle retninger der det ikke befinner seg en ruter. Dette gir mulighet for $2 \cdot 4 + 4 = 12$ funksjonelle moduler ved å bare benytte fire rutere(2x2 mesh). Vi merker oss her imidlertid at dette vil føre til at all informasjon må rutes igjennom et lite antall noder, noe som kan føre til opphopning av pakker ved stor trafikk mellom de funksjonelle enhetene.

Nettverket skalerer lineært, også i hvor mange pakker det kan håndtere. Ved valg av størrelse for er nettverket er det her naturlig å velge det minste antall rutere som kan håndtere det største antall pakker som til en tid kan befinne seg i nettverket. Som resten av FPGA'en kan også nettverket rekonfigureres og det vil dermed være mulig å starte med et mindre antall noder, for så å øke antallet om applikasjonene skulle oppleve uforholdsmessig stor forsinkelse for pakkeleveranser.

8.4 Quality of Service

Nettverket er designet for å tilby QoS i form av beste-evne og garanterte tjenester. I tillegg er ulike løsninger vurdert for å gjøre nettverket vranglås- og livelock-fritt. Nettverket slik det er implementert har en inngangskø med fem innganger på hver ruter og mulighet for to prioritetsnivåer, 0 og 1, der 0 gir høyest prioritet. Nettverket tilbyr beste-evne arbitrering på prioritet 1. Det vil si at alle pakker med denne prioriteten vil bli levert, men at vi ikke kan garantere for leveringstid eller rekkefølge. Med to prioritetsnivåer tilbyr nettverket garanterte tjenester i form av at en pakke med prioritet 0 vil leses inn før en pakke med prioritet 1 og at pakker med prioritet 1 heller ikke vil forsinke leveranse av en pakke med prioritet 0. Nettverket benytter x-y-algoritmen for ruting av pakkene og med to virtuelle kanaler i hver ruter kan pakker sendes forbi og på tvers av hverandre i en ruter.

Rutingalgoritmen ikke mulighet for vranglås eller livelock da pakkene i nettverket bare har en mulig rute. På grunn av svitsjemetoden i nettverket vil det imidlertid være en teoretisk mulighet for vranglås. Dette kan for eksempel forekomme om to rutere har begge sine kanaler fulle og samtidig prøver å sende pakker til hverandre(jamfør *Dining philosophers problem*, kap 2.4). En mulighet ved vranglås er å rekonfigurere den aktuelle noden, og på denne måten lese ut de pakkene som befinner seg der. For å kunne oppdage vranglås imidlertid må ytterligere funksjonalitet legges til ruternes kontrollogikk.

Alternativt kan man her se på muligheten for å innføre adaptivitet i nettverket for å ytterligere minke sannsynligheten for vranglås. I kapittel 3 er enkelte løsninger vurdert men den enkle og effektive x-y-algoritmen er funnet fordelaktig både med tanke på ytelse og pålitelighet. Fokuset er lagt på å holde kontrolllogikk og arbitrering på et minimum

til fordel for større bufferressurser og bedre båndbredde. En innføring av adaptivitet i nettverket er vurdert til å medføre unødvendig store kostnader i form av økte ressurser.

Simuleringer og analyser viser at nettverket, med x-y ruting og SAF-svitsjing, tilbyr god QoS med to prioritetsnivåer og mulighet for utvidelse til flere prioritetsnivåer. En mer avansert arbitreringsstrategi gir mulighet for flere prioritetsnivåer men det kan her også være nødvendig med flere virtuelle kanaler for å kunne tilby god QoS med mange prioritetsnivåer.

Nettverket kommuniserer ved bruk av endireksjonale busser. Med dette kan data bare flyte i en retning i hver leder, og vi trenger dermed to sett med signaler og to busser for å koble sammen to rutere. Begrensinger på FPGA'en gjør at det vil være like ressurskrevende å realisere todireksjonale signaler som endireksjonale signaler. Enkeltrettet kommunikasjon er dermed valgt for det implementerte nettverket. Dette gjør at ruterne kan sende en pakke til en naboruter samtidig som den mottar en annen pakke fra den samme ruteren. Som vi har sett bedrer dette nettverkets pålitelighet og QoS betraktelig sammenlignet med andre metoder.

Nettverket er simulert med lave pakkerater for høyfrekvente situasjoner samt utvalgte hjørnetilfeller. I tillegg er nettverket simulert med kryssende og parallell trafikk. Simuleringene viser at nettverket tilbyr god QoS på to nivåer, beste-evne og garanterte tjenester. Ut fra simuleringene presentert i kapittel 6 kan vi med rimelig sikkerhet si at nettverket fungerer i henhold til designet beskrevet i kapittel 5.

8.5 Test på FPGA

Nettverket er testet på FPGA for å finne ut hvordan det håndterer ulike pakkerater og kryssende trafikk. Dette er gjort ved at et testnettverk, presentert i kapittel 7, er lastet ned på Suzaku-kortet(kap 2.11). Testpakker genereres til nettverket ved bruk av programvare kjørende på en μBlaze-prosessor integrert på kortets Spartan-3 FPGA. Denne testen er imidlertid ikke avsluttet. Nettverket syntetiserer og ser ut til å operere på FPGA'en. Pakkene som genereres fra testprogrammet ser allikevel ikke ut til å bli sendt ut i nettverket. Det antas på dette grunnlag at grensesnittet mellom nettverket og prosessoren ikke fungerer etter hensikten. Da testen anses som god vil det her være interessant med videre undersøkelser for å kunne kjøre testen etter spesifikasjonen. I kapittel 7 foreslås også enkelte metoder for å videreutvikle testmodulen.

KAPITTEL 9

KONKLUSJON

For å kunne støtte strenge QoS krav et nettverk av rutere med garanterte tjenester utviklet. Med bakgrunn i arbeid utført som prosjekt høsten 2008 er det fremlagt en hypotese om dette nettverket kan videreutvikles til å gi bedre båndbredde og QoS. Resultatet av arbeidet er et NoC designed for Xilinx FPGA.

En rekke simuleringer er utført for å verifisere nettverkets funksjonalitet. Vi kan på dette grunnlag si at nettverket, med SAF-svitsjing i kombinasjon med x-y-ruting, gir god QoS for to prioritetsnivåer. Nettverket tilbyr beste-evne QoS på laveste nivå og garanterte tjenester i form av et høyere prioritert nivå. Videre benytter nettverket to uavhengige, lineære kanaler for hver ruter. Simuleringer og analyser viser en betydelig ytelsesøkning ved å organisere bufferressursene assosiert med nettverket i et antall filer eller virtuelle kanaler i stedet for en enkelt FIFO-kø.

Nettverket implementert i dette arbeidet skalerer lineært både for ressursforbruk og båndbredde. Dette støttes av at nettverkets operasjonsfrekvens ikke reduseres om antallet noder i nettverket økes. En analytisk vurdering konkluderer med at nettverket har god nok ytelse til å rute båndbreddekrevende datastrømmer som for eksempel HD-video. Samtidig fører god QoS-støtte til pålitelighet, robusthet og forutsigbarhet.

NoC er vist å være en nøkkelfaktor for å dekke produktivitetsgapet og utnytte dynamisk rekonfigurering på FPGA. Arbeidet i denne rapporten viser at SAF-svitsjing i kombinasjon med x-y-ruting kan være fordelaktig både med tanke på ressursforbruk og båndbredde sammenlignet med mer populære teknikker som for eksempel WH. SAF er

en metode som er vidt brukt i større nettverk og det antas at ettersom de tilgjengelige ressursene på FPGA'ene øker vil flere og flere av teknikkene fra fjernnettverk også kunne anvendes for NoC. FPGA-produsentene vil i fremtiden komme med egne NoC spesielt tilpasset deres systemer.

Resultater viser at QoS er både fordelaktig og oppnåelig. Fordelaktig fordi nettverket vil redusere ressursbruken som følge av finkornede tilkoblinger mellom maskinvaremoduler, i tillegg til at harde tidsfrister og kritiske oppgaver lettere kan møtes ved bruk av informasjonsprioritet. Videre er QoS oppnåelig da det er vist at nettverket skalerer lineært og ikke vil oppta uforholdsmessig store ressurser på en Xilinx FPGA.

Følgende temaer som interessante for videre studier.

- Vurdere ulike topologier, se kapittel 8.1
- Videreutvikling av arbitreringsstrategi, se kapittel 8.2
- Ytterliger vurderinger rundt adaptivitet
- Videreføring av test for FPGA, se kapittel 7

REFERANSER

[1] Dobkin (Reuven) Rostislav, Victoria Vishnyakov, Eyal Friedman, and Ran Ginosar. An asynchronous router for multiple service levels networks on chip. *Asynchronous Circuits and Systems, International Symposium on*, 0:44–53, 2005.

[2] Jean-Christophe Glas and Kjetil Svarstad. A noc on xilinx spartan fpga. Master's thesis, NTNU, 2006.

[3] Ivar Ersland and Kjetil Svarstad. *Quality of Service for Network on Chip*. Project report, NTNU, 2008.

[4] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 1965.

[5] Ahmed Hemani, Axel Jantsch, Shashi Kumar, Adam Postula, Johnny Öberg, Mikael Millberg, and Dan Lindqvist. Network on a chip: An architecture for billion transistor era. *Proceedings of the IEEE*, 2000.

[6] Axel Jantsch and Hannu Tenhunen. *Will Networks on Chip Close the Productivity Gap?*, pages 3–18. 2003.

[7] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[8] M. Zid, A. Zitouni, A. Baganne, and R. Tourki. New generic gals noc architectures with multiple qos. pages 345–349, Sept. 2006.

[9] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. pages 684–689, 2001.

[10] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design and Test of Computers, IEEE*, 22(5):414–421, Sept.-Oct. 2005.

[11] L Bononi and N Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh. In *In Proc. Design, Automation and Test in Europe (DATE)*, Mar 2006.

[12] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2-3):105–128, February 2004.

[13] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures*. Morgan Kaufmann, 1 edition, 2009.

[14] Aydin O. Balkan, Gang Qu, and Uzi Vishkin. A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing. In *Proceedings of the Application-Specific Systems, Architectures and Processors (ASAP)*, pages 73–80, 2006.

[15] R. Holsmark, M. Palesi, and S. Kumar. Deadlock free routing algorithms for mesh topology noc systems with regions. pages 696–703, 0-0 2006.

[16] L.G. Roberts. The evolution of packet switching. *Proceedings of the IEEE*, 66(11):1307–1313, 1978.

[17] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, 4 edition, August 2002.

[18] Robert Cypher, Friedhelm Meyer auf der Heide, Christian Scheideler, and Berthold Vöcking. Universal algorithms for store-and-forward and wormhole routing. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 356–365, New York, NY, USA, 1996. ACM.

[19] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[20] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Science/Engineering/Math, 2nd edition, December 2003.

[21] Maurizio Palesi, Rickard Holsmark, Shashi Kumar, and Vincenzo Catania. A methodology for design of application specific deadlock-free routing algorithms for noc systems. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 142–147, New York, NY, USA, 2006. ACM.

[22] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *Parallel and Distributed Systems, IEEE Transactions on*, 4(12):1320–1331, Dec 1993.

[23] R. Holsmark and S. Kumar. Design issues and performance evaluation of mesh noc with regions. pages 40–43, Nov. 2005.

[24] Jingcao Hu and Radu Marculescu. Dyad: smart routing for networks-on-chip. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 260–263, New York, NY, USA, 2004. ACM.

[25] C.J. Glass and L.M. Ni. The turn model for adaptive routing. pages 278–287, 1992.

[26] Maurizio Palesi, Shashi Kumar, and Rickard Holsmark. *A Method for Router Table Compression for Application Specific Routing in Mesh Topology NoC Architectures*, pages 373–384. 2006.

[27] Wikipedia. *Dining Philosophers Problem.* http://en.wikipedia.org/wiki/Dining_philosophers_problem, besøkt 14.06.09.

[28] Umit Y. Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in noc design: a holistic perspective. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74, New York, NY, USA, 2005. ACM.

[29] Kees Goossens, John Dielissen, Jef Meerbergen, Peter Poplavko, Andrei RÄfdulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. *Guaranteeing the Quality of Services in Networks on Chip*, pages 61–82. 2003.

[30] Robert Mullins, Jeong-Gun Lee, and Simon Moore. Selecting a timing regime for on-chip networks. *In Proc. of the 17th UK Async. Forum,*, 2005.

[31] W.J. Dally. Virtual-channel flow control. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):194–205, Mar 1992.

[32] A. Yakovlev, A. Petrov, and L. Lavagno. A low latency asynchronous arbitration circuit. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(3):372–377, Sep 1994.

[33] L.-S. Peh and W.J. Dally. A delay model and speculative architecture for pipelined routers. pages 255–266, 2001.

[34] John Catsoulis. *Designing Embedded Hardware.* O'Reilly Media, Inc., 2 edition, May 2005.

[35] Inc. Atmark Techno. http://suzaku-en.atmark.com.

[36] Xilinx spartan-3 fpga family data sheet. 2008.

[37] A. Radulescu, J. Dielissen, S.G. Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):4–17, Jan. 2005.

[38] Aline Mello, Leonel Tedesco, Ney Calazans, and Fernando Moraes. Virtual channels in networks on chip: implementation and evaluation on hermes NoC. In *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 178–183, Florianolpolis, Brazil, 2005. ACM.

[39] Fernando Moraes, Ney Calazans, Aline Mello, Leandro MÃ¶ller, and Luciano Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, October 2004.

[40] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The nostrum backbone-a communication protocol stack for networks on chip. pages 693–696, 2004.

[41] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C.A. Zeferino. Spin: a scalable, packet switched, on-chip micro-network. pages 70–73 suppl., 2003.

[42] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[43] E. Bolotin, A. Morgenshtein, I. Cidon, R. Ginosar, and A. Kolodny. Automatic hardware-efficient soc integration by qos network on chip. pages 479–482, Dec. 2004.

[44] Jipeng Zhou and F.C.M. Lau. Adaptive fault-tolerant wormhole routing in 2d meshes. pages 8 pp.–, Apr 2001.

[45] Ge-Ming Chiu. The odd-even turn model for adaptive routing. *Parallel and Distributed Systems, IEEE Transactions on*, 11(7):729–738, Jul 2000.

[46] L. Bononi and N. Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. *Design, Automation and Test in Europe Conference and Exhibition*, 2:30, 2006.

[47] Sverre Hamre. *Tutorial for adding an adder to the suzaku hardware.* NTNU, Trondheim, 2008.

[48] Institutt for elektronikk og telekommunikasjon. *TFE 4170 Enbrikkesystemer, Laboratorieoppgave.* NTNU, Trondheim.

[49] Atmark Techno INC. *Suzaku Software Manual.* 2006.

TILLEGG A

VHDL LISTINGS

Listing A.1: Arbiter

```
1  ----------------------------------------------------------------------------------
2  -- Arbiter
3  ----------------------------------------------------------------------------------
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6  use IEEE.STD_LOGIC_ARITH.ALL;
7  use IEEE.STD_LOGIC_UNSIGNED.ALL;
8  entity arbiter  is
9      Port ( x_in_n      :        in     STD_LOGIC_VECTOR(7 downto 0);
10            x_in_w      :        in     STD_LOGIC_VECTOR(7 downto 0);
11            x_in_s      :        in     STD_LOGIC_VECTOR(7 downto 0);
12            x_in_e      :        in     STD_LOGIC_VECTOR(7 downto 0);
13            x_in_l      :        in     STD_LOGIC_VECTOR(7 downto 0);
14            x_in_ready_n:        in     STD_LOGIC;
15            x_in_ready_w:        in     STD_LOGIC;
16            x_in_ready_s:        in     STD_LOGIC;
17            x_in_ready_e:        in     STD_LOGIC;
18            x_in_ready_l:        in     STD_LOGIC;
19            clear_n     :        in     STD_LOGIC;
20            clear_w     :        in     STD_LOGIC;
21            clear_s     :        in     STD_LOGIC;
22            clear_e     :        in     STD_LOGIC;
23            clear_local :        in     STD_LOGIC;
24            clk, reset  :        in     STD_LOGIC;
25            cleared     :        out    STD_LOGIC;
26            x_in_ready_ctrl:     out    STD_LOGIC;
27            serv_arb    :        out    STD_LOGIC_VECTOR(2 downto 0);
28            x_in        :        out    STD_LOGIC_VECTOR(7 downto 0));
29 end arbiter;
30
31 architecture Behavioral of arbiter is
32
33 signal ready                    : STD_LOGIC_VECTOR(3 downto 0); -- Signal: 4 request signals coupled to
       a bus.
34 signal serv,serving              : STD_LOGIC_VECTOR(2 downto 0) := "000"; -- Registers: Holds the
      direction with the highets prioritzed packet.
35 signal SL_reg                   : STD_LOGIC_VECTOR(3 downto 0);
36 signal x_in_ready               : STD_LOGIC;
37 signal serv_buff                : STD_LOGIC_VECTOR(2 downto 0);
38 signal SL_n,SL_w,SL_s,SL_e:       STD_LOGIC;
39
40 begin
41
42     x_in_ready <= x_in_ready_n or x_in_ready_w or x_in_ready_s or x_in_ready_e or x_in_ready_l;
```

```
43      ready <=  x_in_ready_n & x_in_ready_w & x_in_ready_s & x_in_ready_e;
44      serv_arb <= serving;
45      cleared <= clear_n or clear_w or clear_s or clear_e or clear_local;
46
47          with serv_buff select
48          x_in <= x_in_n when "000",
49              x_in_w when "001",
50              x_in_s when "010",
51              x_in_e when "011",
52              x_in_l when others;
53
54        SL_n <= x_in_n(7);
55        SL_w <= x_in_w(7);
56        SL_s <= x_in_s(7);
57        SL_e <= x_in_e(7);
58
59      ---------------------------------------------------
60      -- To determine which direction to service next --
61      process (ready,reset) is
62      begin
63          if reset = '0' then
64                      serv <= (others=> '0');
65          else
66                case ready is
67                      when "0001" => serv <= "011" ;
68
69                      when "0010" => serv <= "010" ;
70
71                      when "0011" => if SL_reg(1) = SL_reg(0) then
72                                              if serving(1 downto 0) = "10" then serv <= serving; else serv
                                                    <= "011"; end if;
73                                      elsif SL_reg(1) = '0'  then serv <= "010";
74                                      else serv <= "011"; end if;
75
76                      when "0100" => serv <= "001" ;
77
78                      when "0101" => if SL_reg(2) = SL_reg(0) then
79                                              if serving(1 downto 0) = "001" then serv <= serving; else serv
                                                    <= "011"; end if;
80                                      elsif SL_reg(2) = '0' then serv <= "001";
81                                      else serv <= "011"; end if;
82  --
83                      when "0110" => if SL_reg(2) = SL_reg(1) then
84                                              if serving(1 downto 0) = "001" then serv <= serving; else serv
                                                    <= "010"; end if;
85                                      elsif SL_reg(2) = '0' then serv <= "001";
86                                      else serv <= "010"; end if;
87
88                      when "0111" => case SL_reg(2 downto 0) is
89                                          when "011" => serv <= "001";
90                                          when "101" => serv <= "010";
91                                          when "110" => serv <= "011";
92                                          when others => serv <= serving;
93                                      end case;
94
95                      when "1000" => serv <= "000";
96
97                      when "1001" => if SL_reg(3) = SL_reg(0) then
98                                              if serving(1 downto 0) = "000" then serv <= serving; else serv
                                                    <= "011"; end if;
99                                      elsif SL_reg(3) = '0' then serv <= "000";
100                                     else serv <= "011"; end if;
101
102                     when "1010" => if SL_reg(3) = SL_reg(1) then
103                                             if serving(1 downto 0) = "000" then serv <= serving; else serv
                                                    <= "010"; end if;
104                                     elsif SL_reg(1) = '0' then serv <= "010";
105                                     else serv <= "000"; end if;
106
107                     when "1011" => case SL_reg(3) & SL_reg(1 downto 0) is
108                                         when "011" => serv <= "000";
109                                         when "101" => serv <= "010";
110                                         when "110" => serv <= "011";
111                                         when others => serv <= serving;
112                                     end case;
113
114                     when "1100" => if SL_reg(3) = SL_reg(2) then
115                                             if serving(1 downto 0) = "000" then serv <= serving; else serv
                                                    <= "001"; end if;
116                                     elsif SL_reg(3) = '0' then serv <= "000";
117                                     else serv <= "001"; end if;
118
119                     when "1101" => case SL_reg(3 downto 2) & SL_reg(0) is
120                                         when "011" => serv <= "000";
121                                         when "101" => serv <= "001";
122                                         when "110" => serv <= "011";
123                                         when others => serv <= serving;
124                                     end case;
```

```
125
126                     when "1110" => case SL_reg(3 downto 1) is
127                                         when "011" => serv <= "000";
128                                         when "101" => serv <= "001";
129                                         when "110" => serv <= "010";
130                                         when others => serv <= serving;
131                                    end case;
132                     when "1111" => serv <= "000";
133                     when others => serv <= "111";
134                end case;
135
136         end if;
137     end process;
138
139     ------------------------------------------------
140     -- Serviceing signal for buffer is delayed to --
141     -- make sure the data flit is ready           --
142     process(clk,reset) is
143
144         variable ready_reg : STD_LOGIC_VECTOR(3 downto 0) := "0000";
145         variable vSL_reg : STD_LOGIC_VECTOR(3 downto 0) := "0000";
146
147     begin
148           -- Reset the registers --
149           if reset = '0' then
150               serving <= "000";
151               serv_buff <= "000";
152               vSL_reg := "0000";
153               x_in_ready_ctrl <= '1';
154
155           else
156           -- Set the registers --
157           if clk'event and clk = '1' then
158               x_in_ready_ctrl <= x_in_ready;
159
160               serving <= serv;
161               serv_buff <= serving;
162
163               -- Lagrer ny SL når ny request annkommer ---
164               if x_in_ready_e='1' and ready_reg(0)='0' then
165                   vSL_reg(0) := SL_e;
166               end if;
167
168               if x_in_ready_s='1' and ready_reg(1)='0' then
169                   vSL_reg(1) := SL_s;
170               end if;
171
172               if x_in_ready_w='1' and ready_reg(2)='0' then
173                   vSL_reg(2) := SL_w;
174               end if;
175
176               if x_in_ready_n='1' and ready_reg(3)='0' then
177                   vSL_reg(3) := SL_n;
178               end if;
179
180           ready_reg := ready;
181
182           end if;
183           end if;
184           SL_reg <= vSL_reg;
185    end process;
186 end architecture;
```

Listing A.2: Control

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5  use IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7  entity control2 is
8      Port (  x_in_ready :   in   STD_LOGIC;
9              cleared    :   in   STD_LOGIC;
10             serviceing :   in   STD_LOGIC_VECTOR(2 downto 0);
11             clk        :   in   STD_LOGIC;
12             reset      :   in   STD_LOGIC;
13             RTS        :   out  STD_LOGIC;                        -- Request To Send
14             CTS        :   out  STD_LOGIC;                        -- Clear to send
15             instructA  :   out  STD_LOGIC_VECTOR(1 downto 0);   -- Instruction for the buffer
16             instructB  :   out  STD_LOGIC_VECTOR(1 downto 0);   -- Instruction for the buffer
17             use_buff   :   out  STD_LOGIC;
18             read       :   out  STD_LOGIC);
19 end control2;
20
21 architecture behavioral of control2 is
```

```
22      signal stateA,stateB                    : STD_LOGIC_VECTOR(1 downto 0);
23      signal transmit_buff                    : STD_LOGIC;
24      begin
25
26         use_buff <= transmit_buff;
27
28         with transmit_buff select
29         instructA <= stateA when '0',
30                      stateB when others;
31
32         with transmit_buff select
33         instructB <= stateB when '0',
34                      stateA when others;
35
36      process(clk,reset) is
37      variable transmit_state                  : STD_LOGIC_VECTOR(1 downto 0);
38      variable ntransmit_state                 : STD_LOGIC_VECTOR(1 downto 0);
39      variable read_state,nread_state          : STD_LOGIC;
40      variable counterA,counterB               : INTEGER range -3 to 7;
41      variable curr_serv                       : STD_LOGIC_VECTOR(2 downto 0);   -- Direction currently
42      variable contains_packet                 : STD_LOGIC_VECTOR(1 downto 0);
43      variable ntransmit_buff                  : STD_LOGIC;
44
45      begin
46
47         if reset ='0' then
48               nread_state := '0';
49               ntransmit_state := "00";
50               ntransmit_buff := '0';
51               counterA := -1; counterB := 0;
52               RTS <= '0';
53               CTS <= '0';
54               curr_serv := "000";
55               contains_packet := "00";
56               stateA <= "10"; stateB <= "10";
57               read <= '0';
58         elsif clk'event and clk='1' then
59
60         case read_state is
61             when '0' =>             if x_in_ready = '1' then
62                                     curr_serv := serviceing;
63
64                                     if contains_packet /= "11" then
65                                         nread_state := '1';
66                                         CTS <= '1';
67                                     end if;
68                                  end if;
69             when '1' => stateA <= "00"; CTS <= '1';
70                                  if counterA = 7 then -- last flit is read
71                                     nread_state := '0' ;
72                                     stateA <= "10";
73                                     CTS <= '0';
74                                     counterA := -1;
75                                     read <= '0';
76                                     if transmit_buff = '1' then
77                                         contains_packet(0) := '1';
78                                     else
79                                         contains_packet(1) := '1';
80                                     end if;
81                                  elsif curr_serv /= serviceing then
82                                     counterA := 0; curr_serv := serviceing;
83                                     nread_state := '0';
84                                  else
85                                     if counterA = 6 then
86                                         CTS <= '0';
87                                     end if;
88                                     counterA := counterA + 1;
89                                  end if;
90             when others => null;
91         end case;
92
93         case transmit_state is
94             when "00" =>         if contains_packet(1) = '1' then
95                                      ntransmit_state := "01";
96                                      ntransmit_buff := '1';-- RTS <= '1';
97                                      read <= '1';
98                                  elsif contains_packet(0) = '1' then
99                                      ntransmit_state := "01";
100                                     ntransmit_buff := '0';-- RTS <= '1';
101                                     read <= '1';
102                                 end if;
103
104             when "01" => stateB <= "10"; read <= '0'; RTS <= '1';  -- Waiting to read out
105                                 if cleared = '1' then
106                                    ntransmit_state := "10";
107                                    stateB <= "01";
108                                    counterB := counterB + 1;
109                                 end if;
```

```
110             when "10" => stateB <= "01"; -- Transmit
111                                  if counterB = 7 then
112                                     if cleared = '1' then -- last flit transmitted
113                                        stateB <= "10";--
114                                        ntransmit_state := "11"; RTS <= '0';
115                                     else -- if the packet is cancelled
116                                        ntransmit_state := "01"; counterB := 0;
117                                     end if;
118                                  else -- Transmitting
119                                     counterB := counterB + 1;
120                                  end if;
121             when "11" => null; -- Gjenopptagelse av avbrutt pakke.
122                                  counterB := 0; ntransmit_state := "00";
123                                  stateB <= "10";
124                                  if transmit_buff = '1' then
125                                     contains_packet(1) := '0';
126                                  else
127                                     contains_packet(0) := '0';
128                                  end if;
129             when others => null;
130         end case;
131       read_state := nread_state;
132       transmit_state := ntransmit_state;
133       transmit_buff <= ntransmit_buff;
134         end if;
135     end process;
136
137 end architecture;
```

Listing A.3: 64 bit buffer

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5  use IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7  entity buffer_64bit is
8      Port ( x_in         :        in     STD_LOGIC_VECTOR(7 downto 0);
9             instruction  :        in     STD_LOGIC_VECTOR(1 downto 0); -- Intruksjon fra Control
10            clk          :        in     STD_LOGIC;
11            reset        :        in     STD_LOGIC;
12            x_out        :        out    STD_LOGIC_VECTOR(7 downto 0); -- Utangen fra bufferet
13            adresse      :        out    STD_LOGIC_VECTOR(3 downto 0)); -- Pakken som ligger lagret sin
                  destinasjonsadresse
14 end buffer_64bit;
15
16 architecture Behavioral of buffer_64bit is
17
18 signal regA,regB,regC,regD,regE,regF,regG,regH : STD_LOGIC_VECTOR(7 downto 0);
19
20 begin
21 ------------------------
22 ---- Utgangen settes --
23 x_out <= regA;
24 adresse <= regA(6 downto 3);
25
26 --------------------------
27 -- 64 bit shiftregister --
28 process(clk,reset) is
29
30 begin
31
32     ---- Reset ----
33     if reset  = '0' then
34            regA <= x"00";regB <= x"00";regC <= x"00";regD <= x"00";
35            regE <= x"00";regF <= x"00";regG <= x"00";regH <= x"00";
36
37    elsif clk'event and clk = '1' then
38          case instruction is
39                ---- Read In ----
40             when "00" =>
41                   regA <= regB ; regB <= regC ; regC <= regD ; regD <= regE;
42                   regE <= regF ; regF <= regG ; regG <= regH ; regH <= x_in;
43
44             ---- Read Out ----
45             when "01" =>
46
47                   regA <= regB ; regB <= regC ; regC <= regD ; regD <= regE;
48                   regE <= regF ; regF <= regG ; regG <= regH ; regH <= regA;
49
50             ---- IDLE ----
51             when others => null;
52         end case;
53    end if;
54 end process;
```

```
55  end Behavioral;
```

Listing A.4: Readout

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use IEEE.STD_LOGIC_ARITH.ALL;
5   use IEEE.STD_LOGIC_UNSIGNED.ALL;
6
7   entity readout  is
8       Port ( x_out_n      :        out     STD_LOGIC_VECTOR(7 downto 0);
9              x_out_w      :        out     STD_LOGIC_VECTOR(7 downto 0);
10             x_out_s      :        out     STD_LOGIC_VECTOR(7 downto 0);
11             x_out_e      :        out     STD_LOGIC_VECTOR(7 downto 0);
12             x_out_l      :        out     STD_LOGIC_VECTOR(7 downto 0);
13             RTS_n        :        out     STD_LOGIC;
14             RTS_w        :        out     STD_LOGIC;
15             RTS_s        :        out     STD_LOGIC;
16             RTS_e        :        out     STD_LOGIC;
17             RTS_local    :        out     STD_LOGIC;
18             CTS_n        :        out     STD_LOGIC;
19             CTS_w        :        out     STD_LOGIC;
20             CTS_s        :        out     STD_LOGIC;
21             CTS_e        :        out     STD_LOGIC;
22             CTS_l        :        out     STD_LOGIC;
23             clk          :        in      STD_LOGIC;
24             coord        :        in      STD_LOGIC_VECTOR(3 downto 0);
25             serv         :        in      STD_LOGIC_VECTOR(2 downto 0);
26             RTS_arb      :        in      STD_LOGIC;
27             CTS_arb      :        in      STD_LOGIC;
28             use_buff     :        in      STD_LOGIC;
29             x_out_buffA  :        in      STD_LOGIC_VECTOR(7 downto 0);
30             x_out_buffB  :        in      STD_LOGIC_VECTOR(7 downto 0);
31             reset        :        in      STD_LOGIC;
32             adr_buffA    :        in      STD_LOGIC_VECTOR(3 downto 0);
33             adr_buffB    :        in      STD_LOGIC_VECTOR(3 downto 0);
34             read         :        in      STD_LOGIC);
35
36  end readout;
37
38  architecture Behavioral of readout is
39
40      signal x_out        : STD_LOGIC_VECTOR(39 downto 0);
41      signal RTS          : STD_LOGIC_VECTOR(4 downto 0);
42      signal CTS          : STD_LOGIC_VECTOR(4 downto 0);
43      signal destination  : STD_LOGIC_VECTOR(2 downto 0);
44      signal x_out_buff   : STD_LOGIC_VECTOR(7 downto 0);
45      signal adr_buff     : STD_LOGIC_VECTOR(3 downto 0);
46      signal adresse      : STD_LOGIC_VECTOR(3 downto 0);
47
48  begin
49
50      x_out_n <= x_out(31 downto 24);
51      x_out_w <= x_out(23 downto 16);
52      x_out_s <= x_out(15 downto 8);
53      x_out_e <= x_out(7 downto 0);
54      x_out_l <= x_out(39 downto 32);
55
56      RTS_n <= RTS(3);
57      RTS_w <= RTS(2);
58      RTS_s <= RTS(1);
59      RTS_e <= RTS(0);
60      RTS_local <= RTS(4);
61
62      CTS_n <= CTS(3);
63      CTS_w <= CTS(2);
64      CTS_s <= CTS(1);
65      CTS_e <= CTS(0);
66      CTS_l <= CTS(4);
67
68      with use_buff select
69      adr_buff <= adr_buffA when '1',
70                  adr_buffB when others;
71
72      with use_buff select
73      x_out_buff <= x_out_buffA when '1',
74                    x_out_buffB when others;
75
76      --------------------------------
77      -- demux to route flits      --
78      with destination select
79      x_out <= x"00" & x_out_buff & x"000000" when "000",
80               x"0000" & x_out_buff & x"0000" when "001",
81               x"000000" & x_out_buff & x"00" when "010",
82               x"00000000" & x_out_buff       when "011",
```

```
83              x_out_buff & x"00000000"       when others;
84
85      ---------------------------------
86      -- demux to route Request      --
87      with destination select
88      RTS <= '0' & RTS_arb & "000" when "000",
89             "00" & RTS_arb & "00" when "001",
90             "000" & RTS_arb & '0' when "010",
91             "0000" & RTS_arb      when "011",
92             RTS_arb & "0000"      when others;
93
94      ---------------------------------------
95      -- demux to rute Clear To Send       --
96      with serv select
97      CTS <= '0' & CTS_arb & "000" when "000",
98             "00" & CTS_arb & "00" when "001",
99             "000" & CTS_arb & '0' when "010",
100            "0000" & CTS_arb      when "011",
101            CTS_arb & "0000"      when others;
102
103     -----------------------------------------
104     -- process to read destination address --
105     -- and detrmine routing direction      --
106     process (clk,reset) is
107         variable adr               : STD_LOGIC_VECTOR(3 downto 0):="0000"; -- 4 bit destination address
108         variable RTS_arb_prev      : STD_LOGIC;
109     begin
110
111         if reset = '0' then
112                 destination <= "000";
113         elsif clk'event and clk = '1' then
114
115               if read = '1' then
116                     adr := adr_buff;
117               end if;
118
119               ---------------------------
120               -- x-y routing algorithm --
121               if adr(3 downto 2) > coord(3 downto 2) then
122                     destination <= "011";
123               elsif adr(3 downto 2) < coord(3 downto 2) then
124                     destination <= "001";
125               else
126
127                     if adr(1 downto 0) > coord(1 downto 0) then
128                           destination <= "010";
129                     elsif adr(1 downto 0) < coord(1 downto 0) then
130                           destination <= "000";
131                     else
132                           destination <= "100";
133                     end if;
134
135               end if;
136
137         RTS_arb_prev := RTS_arb;
138
139                 adresse <= adr;
140         end if;
141
142     end process;
143 end architecture;
```

Listing A.5: Router

```
1
2   library ieee;
3   use ieee.std_logic_1164.all;
4
5   entity router is
6   port(  reset       :in              std_logic;
7          clk         :in              std_logic;
8          x_in_n      :in              STD_LOGIC_VECTOR(7 downto 0);
9          x_in_w      :in              STD_LOGIC_VECTOR(7 downto 0);
10         x_in_s      :in              STD_LOGIC_VECTOR(7 downto 0);
11         x_in_e      :in              STD_LOGIC_VECTOR(7 downto 0);
12         x_in_l      :in              STD_LOGIC_VECTOR(7 downto 0);
13         x_in_ready_n:in              STD_LOGIC;
14         x_in_ready_w:in              STD_LOGIC;
15         x_in_ready_s:in              STD_LOGIC;
16         x_in_ready_e:in              STD_LOGIC;
17         x_in_ready_l:in              STD_LOGIC;
18         clear_n     :in              STD_LOGIC;
19         clear_w     :in              STD_LOGIC;
20         clear_s     :in              STD_LOGIC;
21         clear_e     :in              STD_LOGIC;
22         clear_local :in              STD_LOGIC;
```

```
23          x_out_n     :out            STD_LOGIC_VECTOR(7 downto 0);
24          x_out_w     :out            STD_LOGIC_VECTOR(7 downto 0);
25          x_out_s     :out            STD_LOGIC_VECTOR(7 downto 0);
26          x_out_e     :out            STD_LOGIC_VECTOR(7 downto 0);
27          x_out_l     :out            STD_LOGIC_VECTOR(7 downto 0);
28          RTS_n       :out            STD_LOGIC;
29          RTS_w       :out            STD_LOGIC;
30          RTS_s       :out            STD_LOGIC;
31          RTS_e       :out            STD_LOGIC;
32          RTS_local   :out            STD_LOGIC;
33          CTS_n       :out            STD_LOGIC;
34          CTS_w       :out            STD_LOGIC;
35          CTS_s       :out            STD_LOGIC;
36          CTS_e       :out            STD_LOGIC;
37          CTS_l       :out            STD_LOGIC;
38          coord       :in             STD_LOGIC_VECTOR(3 downto 0)
39          );
40  end entity;
41
42  architecture behavioral of router is
43
44        signal serv_arb     :             STD_LOGIC_VECTOR(2 downto 0);
45        signal serv_buff    :             STD_LOGIC_VECTOR(2 downto 0);
46
47        signal RTS,CTS      :            STD_LOGIC;
48        signal RTS_arb,CTS_arb :         STD_LOGIC;
49        signal x_outA,x_outB :           STD_LOGIC_VECTOR(7 downto 0);
50
51        signal x_in_ready    :           std_logic;
52        signal instructA      :           std_logic_vector(1 downto 0);
53        signal instructB      :           std_logic_vector(1 downto 0);
54
55        signal cleared      :            STD_LOGIC;
56
57        signal x_out_buffA   :            STD_LOGIC_VECTOR(7 downto 0);
58        signal x_out_buffB   :            STD_LOGIC_VECTOR(7 downto 0);
59
60        signal adr_buffA      :            STD_LOGIC_VECTOR(3 downto 0);
61        signal adr_buffB      :            STD_LOGIC_VECTOR(3 downto 0);
62
63        signal use_buff      :            std_logic;
64
65        signal read         :              STD_LOGIC;
66        signal x_in            :         STD_LOGIC_VECTOR(7 downto 0);
67
68  begin --- Node 00 ---
69
70  arb:      entity work.arbiter(behavioral)
71            port map(x_in_n,x_in_w,x_in_s,x_in_e,x_in_l,
72            x_in_ready_n,x_in_ready_w,x_in_ready_s,x_in_ready_e,x_in_ready_l,
73            clear_n,clear_w,clear_s,clear_e,clear_local,
74            clk,reset,
75            cleared,x_in_ready,serv_arb,x_in);
76
77  ctrl:     entity work.control2(behavioral)
78            port map(x_in_ready,cleared,serv_arb,clk,reset,RTS,CTS,instructA,instructB,use_buff,read);
79
80  buffA:    entity work.buffer_64bit(behavioral)
81            port map(x_in,
82            instructA,clk,reset,x_outA,adr_buffA);
83
84  buffB:    entity work.buffer_64bit(behavioral)
85            port map(x_in,
86            instructB,clk,reset,x_outB,adr_buffB);
87
88  readout:  entity work.readout(behavioral)
89            port map(x_out_n,x_out_w,x_out_s,x_out_e,x_out_l,
90            RTS_n,RTS_w,RTS_s,RTS_e,RTS_local,
91            CTS_n,CTS_w,CTS_s,CTS_e,CTS_l,
92            clk,
93            coord,serv_arb,RTS_arb,CTS_arb,
94            use_buff,
95            x_out_buffA,x_out_buffB,
96            reset,adr_buffA,adr_buffB,
97            read);
98
99  RTS_arb <= RTS;
100 CTS_arb <= CTS;
101
102 x_out_buffA <= x_outA ; x_out_buffB <= x_outB;
103
104 end architecture;
```