NTNU
Det skapende universitet

Utvikling av testmiljø for Network on Chip

Andreas Hepsø

Master i elektronikk
Oppgaven levert: Juni 2010
Hovedveileder: Kjetil Svarstad, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Bakgrunn for oppgaven er tidligere prosjekt innen Network-on-chip for AHEAD hvor det er utviklet rutere for et slikt nettverk på FPGA. Disse er vanskelige å teste, og det er et behov for å lage moduler som kan generere og måle trafikk på nettet mens det kjører på FPGA, og det er dette som er hovedoppgaven,

Oppgaven tar sikte på:

- kartlegge flere mulige testfasiliteter
- velge ut og implementere de høyest prioriterte
- implementere og teste testmiljøet

Det skal utføres tester for å kartlegge relasjonen mellom båndbredde og modulenes plassering og antall. Hvis tiden tillater det skal testmiljøet også prøves ut for å trafikk-planlegge en større applikasjon som f.eks. en video-skalerer.

Oppgaven gitt: 15. januar 2010
Hovedveileder: Kjetil Svarstad, IET

Sammendrag

Ved utviklingen av nye produkter er det ønskelig å ha muligheten til å teste produktet for å forsikre korrekt oppførsel. For AHEADs Network on Chip løsning vil en slik testing kreve et skreddersydd testmiljø.

Arbeidet i denne oppgaven kartlegger en rekke relevante testfasiliteter, for så å evaluere alle disse med hensyn på implementerbarhet, samt areal- og tidsbegrensninger. Videre er en prioritetsliste opprettet der alle testfasilitetene rangeres etter prioritet.

Ut ifra denne prioritetslisten er det implementert en rekke moduler som tilbyr meget nøye trafikksimulering med enten en pseudotilfeldig eller fast bitrate, samt lagring av samtlige pakkers tidsforsinkelse gjennom rutersystemet. Det er også designet en ny arbiter for å bedre utsultingen av den lokale inngangen ved høy pågang på ruteren. Modulene som er designet i denne oppgaven er.

- Konfigurerbar trafikkgenerator
- Trafikkmonitor
- Kontrollmodul for utlesning av data
- Arbiter

Videre er alle modulene simulert for korrekt oppførsel, samt at systemet er implementert og testet på målplattformen Suzaku-S. Testene viser at testmiljøet er meget anvendelig med hensyn på å simulere kompliserte trafikkbilder, samt gi relevant informasjon og vranglåser og bugs som kan benyttes i videre utvikling av systemet. Testmiljøet er også benyttet til å trafikkplanlegge en videoskalerer, der testmiljøets rolle er å angi om den gitte modulplasseringen tilfredsstiller throughputkravene hver modul har.

Innhold

Forkortelser

FPGA = Field Programmable Gate Array

AHEAD = Ambient Hardware Embedded Architectures on Demand

NoC = Network on Chip

TM = TrafficGenerator

TM = TrafficMonitor

HW = HardWare, Norsk = Maskinvare

SW = SoftWare, Norsk = Programvare

PT = Processing Time

NL = Normalized Load

NB = Normalisert Belastning

Figurer

Tabeller

Kapittel 1 – Innledning

1.1 – Testing

Gjennom alle årene med teknologisk utvikling, har testing vært en viktig del av denne prosessen. Testingen er den delen av utviklingen som sikrer egenskaper slik som produktets ytelse, funksjonalitet og holdbarhet. Dette gjenspeiles også i definisjonen av ordet test:

"En prosedyre for kritisk evaluering; et hjelpemiddel for å bestemme tilstedeværelsen, kvaliteten, eller sannheten av noe; en prøvelse"

I denne oppgaven er ønsket å presentere et testmiljø, slik at brukere av dette testmiljøet kan hjelpes med å avgjøre akkurat det spørsmålet *de* stiller til systemet. I vårt tilfelle vil dette systemet være AHEAD prosjektets *"Network on Chip"*.

1.2 – AHEAD

AHEAD prosjektet ble startet av Kjetil Svarstad ved Institutt for elektronikk og telekommunikasjon og er forkortelsen for **A**mbient **H**ardware, **E**mbedded **A**rchitectures on **D**emand. Prosjektet tar sikte på å tilby mindre regnekraftige enheter maskinvareakselerasjon av tungdrevne applikasjoner. Dette gjennomføres ved at enheten leverer en maskinvarebeskrivelse av applikasjonen og et brukergrensesnitt (programvare) mellom maskinvaren og brukeren. Når AHEAD-noden har mottatt disse to elementene, settes en kommunikasjonslink opp imot brukeren, og systemet er klart til bruk.

1.3 – Tidligere arbeid

Gjennom 2008 og 2009 jobbet Ivar Ersland og Andreas Hepsø med å utvikle og ferdigstille AHEAD prosjektets Network on Chip. Dette nettverket er en essensiell del av funksjonaliteten til systemet, og står nå klar til testing. Ethvert Network on Chip er gjerne skreddersydd til et spesifikt formål, slik at det å bruke et generelt testoppsett vil være uaktuelt. I denne oppgaven er det konstruert et testmiljø for dette nettverket som tar sikte på å tilby den funksjonaliteten som behøves for å bekrefte korrekt funksjonalitet og virke som et hjelpemiddel med hensyn på utlegg og sammenkobling av moduler internt på nettverket.

1.4 – Mitt bidrag

- Spesifikasjon og avveining av testfasiliteter
- Utvikling av testfunksjonalitet
- Realisering av testmoduler
- Uttesting av typiske systemer

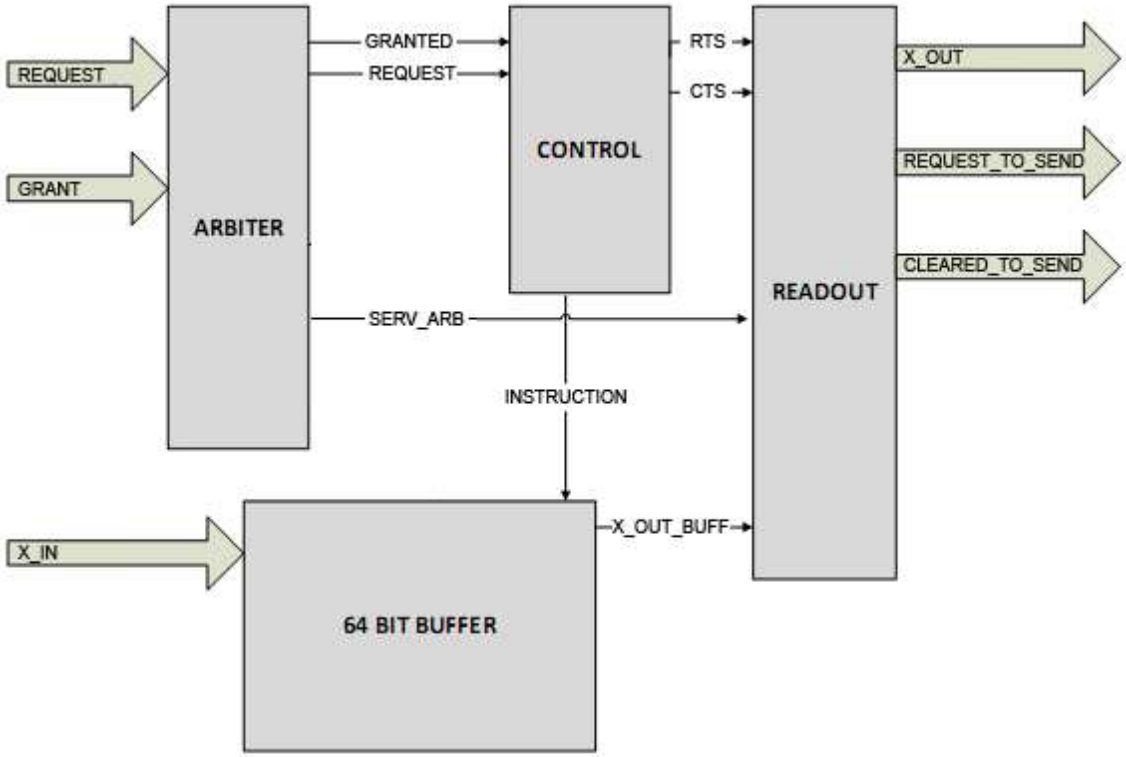
Kapittel 2 – Teori

I dette kapittelet vil teorien for Network on Chip presenteres. Denne teorien er presentert i større detalj i oppgaven skrevet av Andreas Hepsø høsten 2009 [1] og oppgaven skrevet av Ivar Ersland våren 2009 [2]. Den gjennomgangen som er gitt her, skal gi en grunnleggende forståelse for hvordan ruteren er bygd opp og hvordan pakketransaksjoner gjennomføres.

2.1 – Ruterens interne oppbygning

Ruteren som er presentert i [1], har fem interne moduler som gir den funksjonaliteten som er ønskelig. Disse fem modulene er Arbiter, Control, Readout og to ganger 64 bit Buffer.

Sammenkoblingen mellom disse modulene er vist i figuren under.



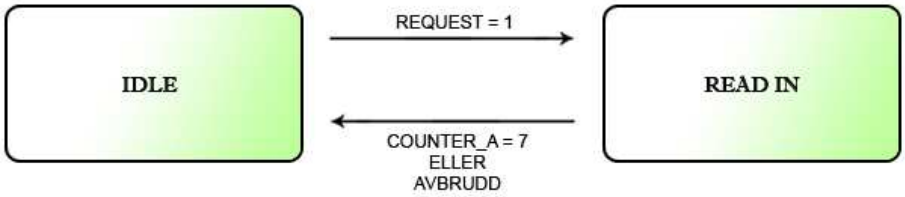
Figur 2.1 – Blokkdiagram av ruteren

2.1.1 - Arbiter

Arbiters funksjonalitet er å ta seg av alle innkommende forespørsler til ruteren, og gi tilgang til ruteren på en rettferdig måte. Når arbiter har bestemt hvilken retning som skal få tilgang til ruteren, gjenspeiles dette i signalet serv_arb. Dette signalet gis til readout-modulen.

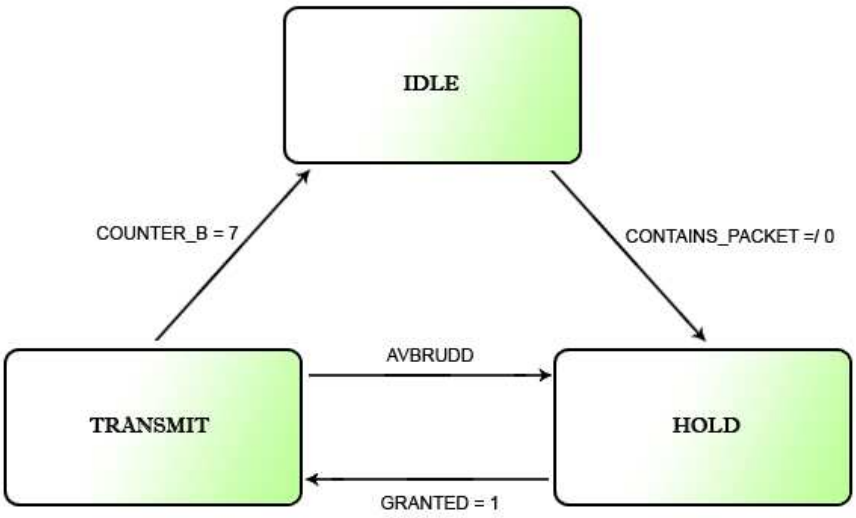
2.1.2 - Control

Control modulen bruker informasjonen den har fått av arbiter som stimuli til tilstandsmaskinene. Denne informasjonen består av enten et høyt request- eller grant-signal som betyr enten at det er en innkommende pakke, eller at man har fått tillatelse til å sende en pakke. Ved en innkommende pakke vil denne informasjonen leveres til tilstandsmaskinen for innlesning som gjengitt i figuren under.



Figur 2.2 – Tilstandsmaskinen for innlesning [1, s.4]

Hvis signalet er et grant signal, vil det tilsi at det allerede er lest inn en pakke som skal sendes videre, slik at tilstandsmaskinen for utlesning har gått ifra IDLE til HOLD, der den forespør adgang til ruteren som skal motta pakken i avventing på et høyt grant signal.



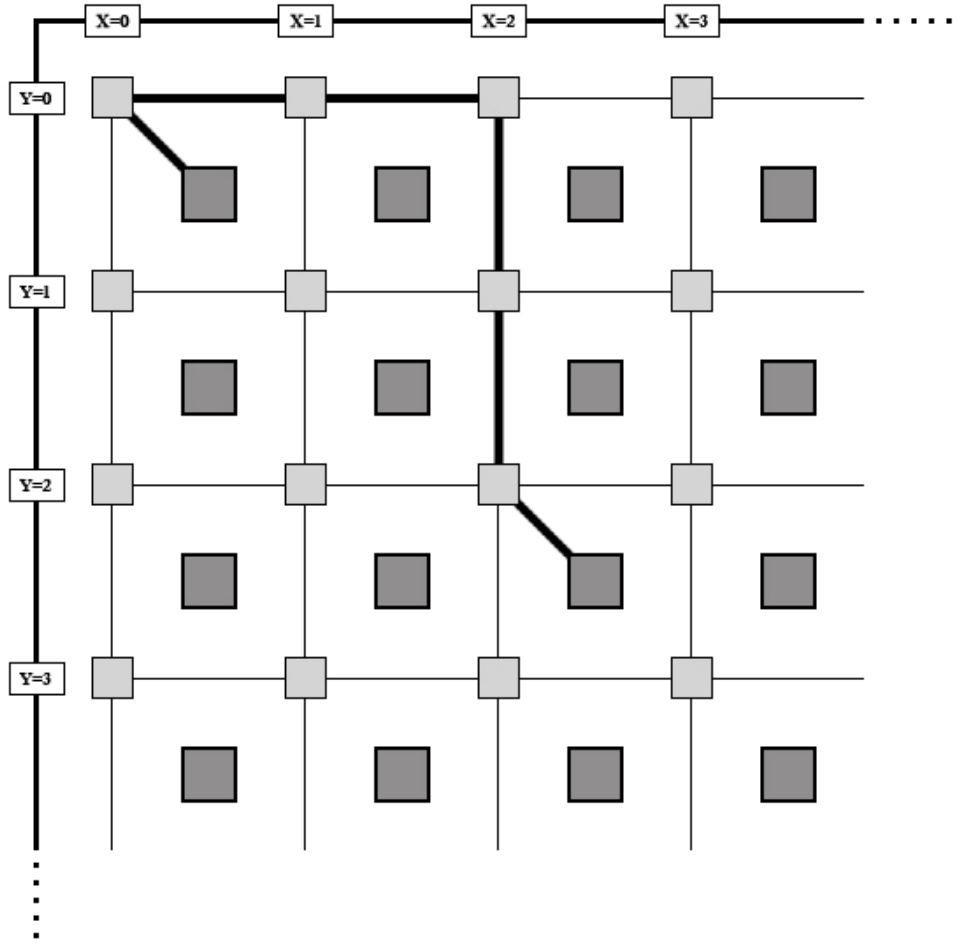
Figur 2.3 – Tilstandsmaskinen for utlesning [1, s.5]

2.1.3 - Readout

For å holde kompleksitetsnivået på kontrollogikken lav, er det implementert en modul med navn readout. Denne modulen sikrer at alle signaler blir rutet til de korrekte utgangene. Denne signalrutingen gjør den på bakgrunn av informasjonen den får fra arbiter, control, bufferne og den interne XY-algoritmen.

2.2 – XY-algoritmen

For å sikre en enkel og presis ruting av pakkene, er XY-algoritmen implementert. Måten denne algoritmen fungerer på er at alle ruterne i ruternettverket har hver sin spesifikke posisjon, gitt av en X og Y koordinat. Når en pakke da sendes ut i ruternettverket, vil readout modulen evaluere destinasjonsadressen til pakken, angitt som de første fire bitene i pakken, opp imot den nåværende ruterens koordinater. Algoritmens virkemåte deriveres av navnet, som tilsier at pakken rutes først til korrekt X-koordinat, før den rutes videre til korrekt Y-koordinat og dermed korrekt destinasjon. En slik pakketransaksjon er gjengitt i figuren under.

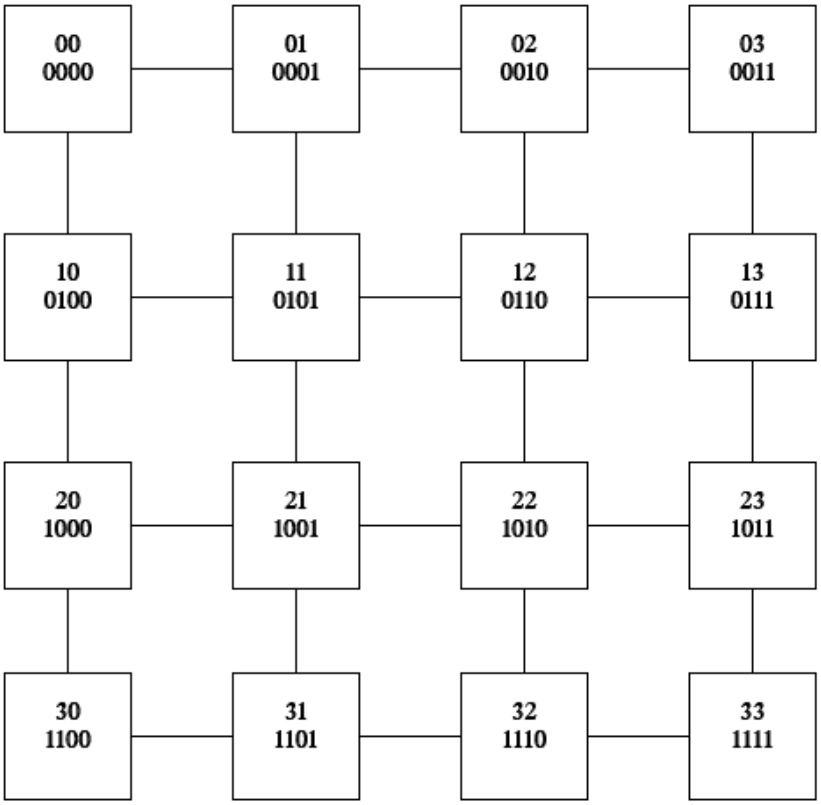


Figur 2.4 – XY-algoritmen [1, s.3]

Her kan det observeres at pakken sendes fra modulen koblet til ruteren med [X,Y] koordinater lik [0,0]. Fram til korrekt X-koordinat [2,0], før den sendes til korrekt Y-koordinat og dermed korrekt adresse til destinasjonen [2,2]. Det er viktig å få med seg at når destinasjonsruteren har lest inn pakken, vil den gjennomføre en siste pakketransaksjon imot den lokale modulen. Dette gjelder også ved sending av en pakke fra en lokal modul.

2.3 – Network on Chip

Ruteren som er presentert i [1], har muligheten for å betjene fem forskjellige innganger. Dette gjør at ruteren kan kobles opp i et såkalt N x N meshoppsett, med mulighet for oppkobling imot en lokal modul. Et eksempel av et 4 x 4 meshoppsett er vist i figuren under. Det er et slikt oppsett som er brukt i videre testing i denne oppgaven.



Figur 2.5 – 4 x 4 meshoppsett [1, s.8]

Når en modul kobles til den lokale inngangen på en av disse ruterne, har den muligheten til å nå andre rutere på nettverket ved å starte en pakketransaksjon. For å gjennomføre en slik transaksjon benytter ruteren seg av to handshake signaler, Request to Send og Cleared to Send. Ved å gjennomføre pakketransaksjonen på denne måten, forsikrer man seg om at begge ruterne er i korrekt tilstand for overføring. Når overføringen er korrekt satt opp, vil ruteren som skal sende pakken begynne å skifte ut verdiene i bufferet sitt på korrekt utgang, slik at ruteren som mottar pakken kan lese av utgangen og skifte verdiene inn i en av sine tilgjengelige buffere. Når pakken er korrekt overført, vil headerflitens adresse leses av, og ruteren vil i henhold til sin interne XY-algoritme avgjøre i hvilken retning pakken skal sendes videre. På denne måten traverserer pakken gjennom nettverket helt til den når sin korrekte destinasjon.

Kapittel 3 – Spesifikasjon

I et ferdigstilt testmiljø er det flere fasiliteter tilgjengelige. Disse er der for å støtte og hjelpe brukeren av systemet, slik at testingen kan gjennomføres så enkelt, grundig og nøyaktig som mulig. I vårt tilfelle er det ønskelig å teste dataflyt og båndbredde mellom moduler i et tenkt system mhp krav til båndbredde.

Ved utvikling av et testmiljø vil det være fysisk umulig å implementere alle tenkelige testfasiliteter. Derfor vil det være nødvendig og først kartlegge de testfasiliteter som kan være ønskelig å implementere. Videre må man evaluere hvilke behov og kriterier testmiljøet skal møte, slik at man enklere kan prioritere hvilke fasiliteter som til slutt vil bli implementert.

3.1 – Kartlegging av testfasiliteter

- Trafikkgenerator
    - Sanntidsendring av dataraten produsert i trafikkgeneratorene
    - Realistisk trafikkbilde med reaktive trafikkgeneratorer
    - Trafikkplanlegging av kjent applikasjon
- Sanntidsoversikt over benyttet båndbredde
- Dynamisk implementering av moduler/trafikkgeneratorer
- Brukervennlig GUI
- Koblingstest
- Overvåke tidsforsinkelsen til hver pakke
- Høy nøyaktighet med hensyn på timing og datamengde.
- Test av nettverkets feilrate
- Maskinvarestyrt stepping / clock-gating
    - Utlesning av relevant data ved hvert steg
- Innsamling og presentasjon av statistisk tallmateriale

3.2 – Evaluering av testfasiliteter

I dette delkapittelet vil alle kartlagte testfasiliteter evalueres. Denne evalueringen setter fokus på hvilket bidrag den gjeldende fasiliteten vil gi til brukeren av systemet, gjennomførbarhet og hvilke bidrag implementering av denne fasiliteten innebærer med hensyn på areal- og tidsbegrensninger.

Trafikkgenerator

For å kunne gjennomføre en kartlegging av relasjonen mellom båndbredde og modulenes plassering og antall, er det nødvendig å ha en eller flere trafikkgenererende kilder. Derfor vil det være en selvfølge for dette testmiljøet at man må implementere en trafikkgenerator.

Sanntidsendring av dataraten i trafikkgeneratorene

Ved å kunne dynamisk endre dataraten i trafikkgeneratorene kan man enklere endre den normaliserte belastingen for å finne metningspunktet til den gitte modulplasseringen. Dette gir oss en indikasjon på kretsens maksimale yteevne.

+ Enklere å justere parametrene i testmiljøet.

- Gir mer overhead for designet.

Realistisk trafikkbilde med reaktive trafikkgeneratorer

Ved å ha et dynamisk trafikkbilde der modulene har et reaktivt responsmønster, vil kretsen kunne simulere trafikk som er mer sannferdig enn ved at alle trafikkgeneratorene pumper ut data i en konstant rate.

+ Gir et mer realistisk bilde av trafikken i kretsen.

- Gir mer overhead, samt at det kan være vanskelig å implementere korrekt.

Trafikkplanlegging av kjent applikasjon

Ved å implementere støtte for trafikkplanlegging av en kjent applikasjon, kan man se hvordan akkurat denne applikasjonen påvirker kretsen.

+ Gir et meget nøyaktig bilde over hvordan trafikken i den gitte applikasjonen oppfører seg.

- Krever en dyp forståelse for den gitte applikasjonen. Krever også en allsidig trafikkgenerator, slik at denne kan etterligne trafikken presist nok.

Sanntidsoversikt over benyttet båndbredde:

Ved å ha en sanntidsoversikt over benyttet båndbredde kan man enkelt se når kretsen når metningspunktet sitt.

+ Gir en meget finkornet oversikt over ledige trafikkresurser på kretsen.

- vanskelig å realisere ved testing på FPGA da dataene må lagres fortløpende, eller sendes til µBlaze for prosessering.

Dynamisk implementering av moduler/trafikkgeneratorer

Ved å ha muligheten til dynamisk implementering av moduler, vil man kunne endre trafikkbildet plutselig, og på denne måten kunne simulere en mer korrekt oppførsel for AHEAD systemet.

+ Gir oss en bedre trafikksimulering av systemet.

- Krever enten at støtten for dynamisk implementering implementeres, eller at trafikkgeneratorene kan dynamisk (re)konfigureres gjennom konfigurasjonspakker.

Brukervennlig GUI (Graphical User Interface)

Ved å ha et brukervennlig GUI, blir det enklere for brukeren av testmiljøet og gjennomføre tester.

+ Brukervennlig

- Liten støtte for grafiske grensesnitt i uClinux.

Koblingstest

Ved å ha en koblingstest innebygd i systemet, kan man sjekke om koblingene er korrekt satt opp mellom ruterne.

+ Oppdager eventuelle feilkoblinger

- Lite behov for denne funksjonaliteten utover en innledende testing, mulig å sette opp en slik test gjennom brukergrensesnittet.

Overvåke tidsforsinkelsen til hver pakke

Ved å overvåke tidsforsinkelsen til hver pakke kan man se hvor lang tid pakken faktisk bruker på å traversere gjennom ruterne.

+ Gir oss nødvendig informasjon for å se om arbitreringen i ruterne er rettferdig.

- Krever tidsstempling av pakkene i VHDL, noe som kan være vanskelig å implementere men hensyn på det tilgjengelige arealet.

Høy nøyaktighet med hensyn på timing og datamengde

Ved å opprettholde en høy oppløsning på målingen av forsinkelse og datamengde, vil det resultere i at beregningene som gjøres på bakgrunn av denne informasjonen også blir mer nøyaktig.

+ Mer nøyaktige resultater.

- Større datamengde som må prosesseres.

Test av nettverkets feilrate

Ved å teste nettverkets feilrate, kan man få mer nøyaktige resultater på den faktiske båndbredden som er oppnåelig på kretsen. Konstanten vil ligge mellom 0 og 1 og representerer andelen av vellykkede sendinger.

+ Mer nøyaktige resultater med tanke på båndbredde

- Resurskrevende og vanskelig å implementere da man må ha en mekanisme som detekterer pakker som feiler under sending.

Maskinvarestyrt stepping/clock-gating

Ved å ha maskinvarestyrt stepping/clock-gating kan man stoppe kretsen hver N'te klokkesykel.

+ Gir oss muligheten til å utføre off-chip prosessering uten at kretsen påvirkes.

- Vil øke testtid og overhead.

Utlesning av relevant data ved hvert steg

Ved å benytte oss av stepping/clock-gating kan man lese ut innsamlet informasjon hver gang kretsen stoppes. Dette gir også muligheten til å bestemme hvor finkornet informasjonen skal gjengis. (Fra hver klokkesykel, til hele testens tidsforløp)

+ Gir oss god kontroll over hvor finkornet informasjonen skal være, samt at det er mulig å la SW gjennomføre en del av de store utregningene.

- Krever en mekanisme som steppingen/clock-gatingen kan lytte på (håndtrykk mellom HW og SW)

Innsamling og presentasjon av tallmateriale

Ved å samle inn og presentere relevant tallmateriale, kan man få en mer lettlest framvisning av kretsens faktiske ytelse.

+ Bedrer forståelsen for kretsens ytelse.

- Krever at man leser ut data med jevne mellomrom.

3.3 – Prioritering og utvelgelse av testfasiliteter

Med tanke på at det vil være umulig å implementere alle tenkelige fasiliteter i testmiljøet, er det viktig at man prioriterer og skaper en oversikt over hva som er viktig med tanke på oppgaven og hvilke resultater man er ute etter å oppnå. Som resultat av dette er prioritetslisten under satt opp. De oppføringene som er strøket ut vil ikke bli implementert med tanke på gjennomførbarhet og areal- og tidsbegrensninger. Dette er også videre diskutert i neste delkapittel.

1. Trafikkgenerator.
2. Dynamisk implementering av moduler/trafikkgeneratorer.
3. Sanntidsendring av dataraten produsert i trafikkgeneratorene.
4. Overvåke tidsforsinkelsen til hver pakke
5. ~~Realistisk trafikkbilde med reaktive trafikkgeneratorer.~~
6. Maskinvarestyrt stepping / clock-gating.
    a. Utlesning av relevant data ved hvert steg
    b. Innsamling og presentasjon av statistisk tallmateriale.
7. Høy nøyaktighet med hensyn på timing og datamengde.

~~8. Brukervennlig GUI~~

~~9. Sanntidsoversikt over benyttet båndbredde~~

10. ~~Trafikkplanlegging av kjent applikasjon.~~

~~11. Koblingstest~~

~~12. Test av nettverkets feilrate~~

3.4 – Testfasiliteter som ikke implementeres

Realistisk trafikkbilde med reaktive trafikkgeneratorer vil ikke bli implementert. Dette grunnet at hvis en slik funksjonalitet skulle blitt implementert, hadde det vært ønskelig at hver trafikkgenerator hadde inneholdt en matrise med en oversikt over hvilken adresse pakken skulle blitt sendt til og hvor lang tid det skulle tatt før denne pakken skulle blitt produsert. Dette ville krevd at hver matrise har en lengde lik antall rutere i systemet, der hvert element i matrisen inneholder en adresse på lengde $\log_2$(#Routers) og et 12 bits delaysignal. Grunnet arealkravene dette innebærer vil et slikt system være uaktuelt til man benytter seg av en plattform med større arealresurser.

Brukervennlig GUI er vanskelig å implementere grunnet uClinux's begrensede støtte for dette. Grensesnittet vil fortsatt være brukervennlig, men ikke grafisk.

Trafikkplanlegging av kjent applikasjon vil ikke bli implementert i den forstand at systemet tilbyr simulering av en rekke kjente applikasjoner. Det systemet vil tilby er en generell trafikkgenerator, slik at brukeren kan generere en test gjennom brukergrensesnittet som simulerer den ønskelige applikasjonen.

Sanntidsoversikt over benyttet båndbredde vil ikke bli implementert grunnet at dette vil kreve at kretsen kjører uavbrutt, noe som i vårt tilfelle er umulig da den interne klokken må stoppes for å få tilgang på dataene. Ved å prøve å lese ut dataene uten å stoppe kretsen vil man få problemet med at observatøren (utlesning av data) påvirker dataene, ved og nettopp observere [3].

Koblingstest vil heller ikke bli implementert. Dette grunnet at avkastningen ved å implementere en slik test ikke veier opp for tidsforbruket det tar å utvikle den og at en slik test vil være mulig å implementere gjennom konfigurering av trafikkgeneratorene.

Test av nettverkets feilrate er heller ikke en fasilitet som vil bli implementert. Først og fremst grunnet areal- og tidsbegrensningene som ligger til grunne i oppgaven. For å kunne implementere en slik fasilitet, kreves en mekanisme som oppdager pakker som feiler under sending. Med tanke på at alle koblinger imellom ruterne har små avstander og at alle signaler er hardkoblet, vil en slik test være lite anvendelig.

Kapittel 4 – Implementering av nye moduler

For å kunne utvikle et testmiljø med testfasilitetene gitt av kapittel 3.3 er det nødvendig å implementere flere nye moduler. Disse modulene skal ta seg av datagenerering, dataovervåkning og datainnsamling. Det er også konstruert en ny arbiter grunnet deaktiveringen av Quality of Service tjenesten. QoS ble deaktivert da kontrollogikken i ruteren ikke hadde implementert støtte for avbrudd i henhold til bruken av denne tjenesten.

4.1 – Arbiter v.2

Etter noen initielle tester, ble det avdekket en rekke mangler med arbiteren presentert i [1]. Den desidert største mangelen var nedprioriteringen av den lokale inngangen. Logikken som var implementert i arbiter v.1 var at den lokale inngangen kun fikk sende sine pakker når all trafikk gjennom ruteren hadde opphørt. Dette innebærer at når det traverserer en datastrøm med høy throughput gjennom en ruter, vil den lokale inngangen risikere utsultning

Det er også tatt høyde for at ruteren presentert i [1], var designet for bruk med Quality of Service, slik at ved deaktivering av QoS ble denne funksjonaliteten overflødig. Med fem mulige innganger vil det være $2^5 = 32$ mulige scenarioer arbiteren må behandle. For å senke kompleksiteten er det konstruert en enklere arbiter som kun tar høyde for hvem som fikk grant sist. Det arbiteren vil gjøre er å nekte retningen som fikk grant sist en ny grant, så lenge det er en annen retning som også vil ha tilgang til ruteren. For å holde arbiteren arealeffektiv, er det innført en statisk prioritetsrekkefølge som er: Nord, Vest, Sør, Øst, Lokal. Ved å implementere en slik statisk prioritetsrekkefølge risikerer man ved flere samtidige forespørsler utsulting av en eller flere innganger. Dette er videre diskutert i kapittel 5.1.4 og 7.2.1.

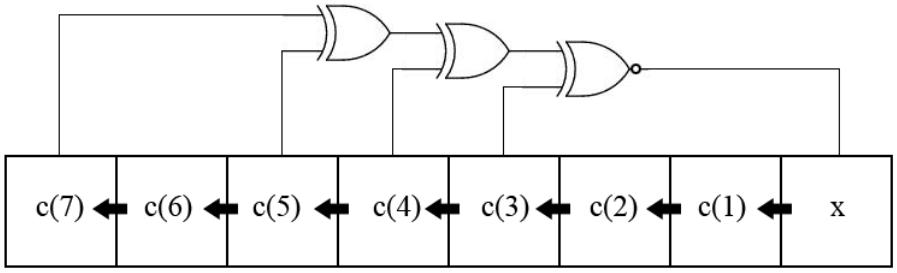
Hver gang arbiteren har arbitrert de gjeldende forespørslene, vil det være en periode hvor tilstandsmaskinen for innlesning av pakker er utilgjengelig. For å forenkle logikken i arbiteren er denne tilstandsmaskinen utstyrt med et signal packet_read. Dette signalet rutes inn i arbiteren og varsler at tilstandsmaskinen er klar til å motta en pakke, slik at arbiter ikke risikerer å arbitrere på feil tidspunkt.

4.2 – 8 Bit Linear Feedback Shift Register

Ved å implementere et LFSR vil man har muligheten til å simulere en pseudotilfeldig bitrate i trafikkgeneratoren. Måten dette registeret oppnår sin pseudotilfeldige funksjonalitet på, er ved å skifte inn en verdi på utgangsverdien, som er basert på den logiske funksjonen:

$$x = \text{not}(c(7)\ \text{xor}\ c(5)\ \text{xor}\ c(4)\ \text{xor}\ c(3))$$

Som også er vist i figuren under.



Figur 4.1 – 8 bit LFSR

Et 8 bit LFSR vil ha en periode på 2^8=256. Dette vil bety at når det har produsert 256 verdier, vil de samme verdiene gjengis om og om igjen. Dette betyr også at hvis perioden for datainnsamling er et multiplum av 256, vil pakketellerne gi ut de samme verdiene hver utlesning, grunnet den periodiske oppførselen til modulen.

4.3 – CLK gater

En nødvendighet for å kunne samle inn data, samtidig som kretsen utfører sitt arbeide er at man må ha muligheten til å sette testkretsen på pause, uten at dette påvirker muligheten for å lese ut dataene kretsen har produsert. Dette løses enkelt ved å implementere en clock gater.

Denne funksjonaliteten er implementert som en BUFGCE primitiv i Spartan 3 FPGA-en, da dette sikrer at kretsen som skal klokkes, fortsatt benytter de interne klokkerutingene. Hvis man ikke hadde benyttet denne primitiven, ville man risikert at den gatede klokken ikke hadde blitt gjenkjent som en klokke av syntheseverktøyet og på denne måten utsette kretsen koblet på denne klokken for eventuelt clock-skew og andre klokkerelaterte problemer.

Entiteten for denne enheten er vist i figuren under.

```
entity BUFGCE is
    port(
    I   : in std_logic;
    CE  : in std_logic;
    O   : out std_logic
    );
end entity;
```

Figur 4.2 – BUFGCE entitet

Her er *I (input)* den globale klokken fra Suzaku-S brettet, *CE (clock enable)* er et kontrollsignal som ved høy verdi setter utgangen O (output) lik inngangen, ellers null. Utgangen på BUFGCE elementet er koblet til alle deler av kretsen som må stoppes ved utlesning av data. For dette designet vil dette kun omfatte ruterne og trafikkgeneratorene. Dette grunnet at pakketellerne ikke er en aktiv modul, slik at den vil ikke utføre arbeid så lenge det ikke er aktivitet på ruternettverket.

4.4 – TM control (TrafficMonitor control)

For å kunne overføre dataene fra HW til SW må man først stoppe kretsen ved hjelp av CLK-gater modulen, for så å varsle SW gjennom et håndtrykksignal om at det foreligger data klar for utlesning. Når dataene er utlest må SW sende sitt håndtrykksignal til HW for å varsle om at utlesningen er ferdig og at tellerne kan resettes, og kretsen startes igjen. Det er dette grensesnittet som er implementert i TM kontrolleren.

```
entity TM_control is
    port(
    clk             : in std_logic;
    reset           : in std_logic;
    CLK_GATE_FREQ   : in std_logic_vector(3 downto 0);
    readout_finished: in std_logic;
    start_testing   : in std_logic;
    clock_enable    : out std_logic;
    readout         : out std_logic
    );
end entity;
```

Figur 4.3 – TM_control entitet

Den ønskelige oppførselen til TM_control enheten, er at hvis systemet har en pakkestrøm som kjører på 100 % av normalisert belastning, skal mottageren ha mulighet til å lagre alle mottatte tidsforsinkelser i hele perioden for en utlesning. Dette vil si at når TG-ens BRAM-matrise har en lengde på 512 og at det i følge figur 4.2 i [1, s.13] tar tolv klokkesykler å overføre en pakke mellom to rutere, vil en naturlig periode for utlesning være 512 * 12 = 6144 Klokkesykler.

Fra dette kan to konstanter angis, en konstant som angir tiden det tar å sende en pakke og en konstant som angir hvor mange pakker man maksimalt kan produsere i løpet av en utlesning (Med CLK_GATE_FREQ = 1).

$$\text{delay}_{\text{SEND}} = 12, \qquad \#\text{packages}_{\text{MAX}} = 512$$

Logikken for avbruddet i henhold til denne verdien er at når den interne telleren når clk_gate_freq ◦ 6144 vil et avbrudd opprettes og readout settes høyt og clock_enable settes lavt. Dette vil si at ruterne og trafikkgeneratorene stoppes av CLK-gateren og trafikkmonitorene blir bedt om å lese ut pakketellerne sine. TM kontrolleren setter også et internt pausesignal høyt, slik at man stopper den interne telleren i avventing på en lav-> høy transisjon av programvarens håndtrykksignal readout_finished.

Start_testing signalet gis fra SW for å varsle om at denne nå er klar til å starte testingen av kretsen. Dette signalet er implementert fordi trafikkgeneratorene trenger konfigurering før testen kan startes og det er upraktisk å pause kretsen under denne prosessen.

4.5 – TrafficMonitor

Denne modulen har som hensikt å holde oversikt over hvor mange pakker som traverserer gjennom en gitt ruter. For å kunne detektere at en pakke traverserer gjennom ruteren, må man lytte på et signal. I TrafficMonitoren-en er dette signalet CTS (Cleared_to_Send). Grunnen til dette er at ruterens oppbygning legger til rette for bruken av dette signalet, da kontrollenhetens CTS signal er felles for alle 5 retningene. På denne måten slipper man å detektere hvilken inngang signalet kommer fra. *(Det er arbiters serv signal som demuxer kontrollenhetens CTS signalet til riktig utgang i modulen readout.)*

```
entity TM is
port(   clk                 : in std_logic;
        reset               : in std_logic;
        CTS                 : in std_logic;
        readout_finished    : in std_logic;
        packet_counter      : out integer
);
end entity;
```

Figur 4.4 – TrafikkMonitor entitet

CTS	Signal fra kontrollenheten som går høyt når en pakke ankommer ruteren
Readout_finished	Håndtrykksignal fra SW som går høyt når SW er ferdig med å lese ut alle dataene
Packet_counter	Integer som angir hvor mange pakker som er ankommet ruteren.

Modulen har to innebygde flankedetektorer for signalene CTS og readout_finished. Grunnen til dette er at CTS er høy under hele overføringen av en pakke, slik at man må lytte på en $CTS_{lav->høy}$ transisjon. Readout_finished er et håndtrykksignal som styres fra SW, og det vil med hensyn på hastigheten til kretsen være umulig å holde dette signalet høyt kun én klokkeflanke.

4.6 – TrafficGenerator

For å kunne kartlegge relasjonen mellom benyttet båndbredde, og modulenes plassering og antall, er det nødvendig å generere trafikk. Til dette formålet er TrafficGenerator-en utviklet. Dens oppgave i testmiljøet er å etterligne realistisk datakommunikasjon, uten å inneholde noen form for spesifikk logikk. Dette oppnås ved at modulen i utgangspunktet er meget generell, med ved hjelp av konfigurering har den muligheten til å endre oppførselen slik at det er mulig å etterligne den datatrafikken som er ønskelig.

```
entity TG is
port(
    ungated_clk     :in  std_logic;
    clk             :in  std_logic;
    reset           :in  std_logic;
    start_testing   :in  std_logic;
    x_in            :in  std_logic_vector(7 downto 0);
    request         :in  std_logic;
    grant           :in  std_logic;
    SW_enable       :in  std_logic;
    SW_addr         :in  std_logic_vector(8 downto 0);
    global_time     :in  std_logic_vector(31 downto 0);
    x_out           :out std_logic_vector(7 downto 0);
    RTS             :out std_logic;
    CTS             :out std_logic;
    tstamp_out      :out std_logic_vector(31 downto 0)
    );
end entity;
```

Figur 4.5 – TrafficGenerator entitet

Som det kan observeres av figur 4.5, benytter trafikkgeneratoren seg av de samme portene (request, grant, RTS, CTS) som finnes i ruterens kontroll-logikk. Dette grunnet at ruterens design ikke inneholder noen spesifikk rutine/logikk for behandling av pakker som har nådd sin destinasjon. Ruteren vil kun observere at pakken skal leses ut på den lokale utgangen, så er det opp til modulen som sitter i andre enden å tilpasse seg standarden for pakkeoverføring.

De resterende portene blir brukt i forbindelse med kontrollering av testprosedyren, samt tidsstempling og utlesning og kontroll av BRAM-modulens tidsstemplinger. Funksjonen til disse signalene er gitt av tabellen på neste side.

Signal	Virkeområde	Funksjon
Start_testing	Testprosedyre	Signalisere at konfigurasjonspakkene fra SW er sendt og programvaren er klar til å starte testprosedyren.
Global_time	Tidsstempling	Global klokke som brukes i tidsstemplingen av pakkene
Ungated_clk	Utlesning og kontroll av BRAM	Nødvendig for å lese ut BRAM-en når resten av logikken er stanset
SW_enable	Utlesning og kontroll av BRAM	Aktiverer utlesning av BRAM-modulen
SW_addr	Utlesning og kontroll av BRAM	Angir hvilken adresse i BRAM-modulen som skal leses ut
Tstamp_out	Utlesning og kontroll av BRAM	Gir verdien fra BRAM modulen angitt av SW_addr signalet.

Tabell 4.1 – Funksjonaliteten til trafikkgeneratorens signal

Ved en eventuell endring av ruterdesignet er det viktig å vite at for å benytte seg av trafikkgeneratoren, må designet bruke samme standard for pakkeoverføring som det gjeldende designet.

4.6.1 – Konfigurasjonspakke

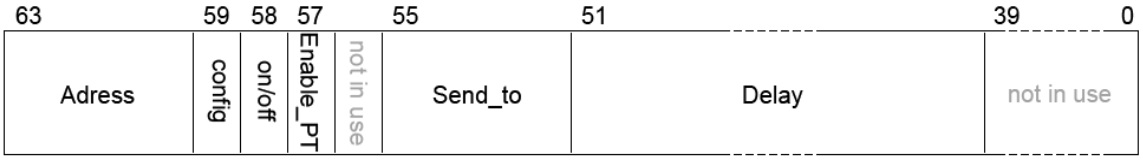
Ved å implementere rekonfigurering av trafikkgeneratorene gjennom et pakkegrensesnitt, kan man ha et fast oppsett implementert på FPGA-en, mens man fortsatt kan endre hvordan testen skal utløpe ved å endre vesentlige variabler i hver TG.

For å kunne implementere en slik funksjonalitet, må denne konfigurasjonspakken inneholde nok variabler til at systemet kan gjøres generelt nok. Tabell 4.2 viser hvilke variabler som er implementert i en slik pakke, bitbredden og hvilke funksjoner variabelen har.

Variabel	Bitbredde	Funksjon
Config	1 bit	Er høy hvis pakken er en konfigurasjonspakke, lav ellers
On_off	1 bit	Er høy hvis modulen skal produsere trafikk
Enable PT	1 bit	Angir om LFSR_8 modulen skal være aktiv eller ikke, slik at responstiden er pseudorandom mellom 0 til 255 klokkeflanker.
Send_to	4 bit	Gir adressen denne modulen skal sende pakker til
Delay	12 bit	Gir forsinkelsen som produseringen av pakker skal ha. Verdien er gitt i antall klokkeflanker og kan ta verdiene [0, 4095]. Noe som gir trafikkgeneratoren muligheten til å produsere pakker med en normalisert belastning i intervallet [0.29, 100] %.

Tabell 4.2 – Signalliste for TrafikkGeneratoren

En pakke består av 64 bit fordelt over 8 flits. Som man kan se av figuren under er de fire første bitene adressen til pakken da dette er universelt for alle pakker. Bit nummer 59 er konfigurasjonsbitet, som angir om pakken er en konfigurasjonspakke eller ikke. Videre kommer variablene som angitt i tabellen over. Det er verdt å merke seg at bit nummer 56 ikke er i bruk, dette grunnet at det er enklere å lese og konfigurere en konfigurasjonspakke i programvaren når man benytter seg av fire og fire bit grunnet den heksadesimale representasjonen.



Figur 4.6 – Konfigurasjonspakkens oppsett

4.6.2 – Utvikling av timestamp-protokollen for trafikkgeneratoren

For å kunne implementere tidsstempling i trafikkgeneratorene er det noen punkter som man må ta hensyn til. Dette er at tidsstemplingen er pakkebasert og at utlesningen av data vil ha en periode mellom 6144 klokkesykler og 15*6144 klokkesykler. Dette vil si at man enten må mellomlagre alle tidsstemplingene man mottar, eller lese ut tidsstemplingen for hver pakke som mottas. For å kunne lese ut alle tidsstemplingene må man trigge utlesningen på $CTS_{HØY->LAV}$ transisjonen i kontroll-logikken, for da å lese ut tidsstemplingen til SW for mellomlagring og senere prosessering.

Den valgte implementeringen følger den første ideen, slik at alle pakker mellomlagres inne på selve ruteren. Denne utviklingen har vært en trestegs utvikling som representeres i de tre neste delkapitlene.

4.6.2.1 – Utvikling av tidsstempling med mellomlagring av verdier og intern prosessering

For å implementere tidsstempling av pakkene, ble trafikkgeneratoren endret for å imøtekomme mellomlagring av disse dataverdiene. Det ble opprettet en rekke med N antall M bits std_logic_vector verdier og logikk som leste ut og plasserte tidsstemplingen i en sirkulær måte, slik at når rekken var fullt, skrev logikken over den eldste verdien. Ved utlesning ble de N verdiene summert og midlet med N, for så å bli lest ut til utgangen for tidsstemplingen. På denne måten hadde SW kun ett signal å forholde seg til, og kan lese verdien rett ut uten prosessering.

Ved å tenke over hvor stort ressursforbruk dette designet indikerer, er det nødvendig med visse endringer med tanke på resursbegrensningen i Spartan 3 brikken.

4.6.2.2 – utvikling av tidsstempling med mellomlagring av verdier og ekstern prosessering

Ved å la SW ta seg av prosesseringen av de mellomlagrede verdiene, sparer man resursforbruk i HW. På denne måten kan man implementere mer logikk i HW, eller møte resurskrav, -begrensninger. Denne implementeringen krever at SW har mulighet for å nå hver verdi i tidsstemplingsmatrisen, og dermed må det opprettes en multiplekser. Som en følge av dette må også SW ha tilgang til et signal av lengde $\log_2(N)$, som sier hvilken verdi i matrisen den vil ha adgang til.

Denne logikkimplementering vil spare en del resurser, men vil ikke få plass ved siden av et 4x4 rutermesh inne på Spartan 3 brikken. Derfor er det umulig å implementere mellomlagring av tidsstemplingen i logikken, uten å korte ned lengden på rekken til en verdi som effektivt vil gjøre verdiene mer eller mindre urelevante i testsammenheng.

4.6.2.3 – utvikling av tidsstempling med mellomlagring av verdier i BRAM og ekstern prosessering

Grunnet plassbegrensninger i Spartan 3 brikken, er det nødvendig å benytte seg av interne primitiver. Spartan 3 XC3S1000 brikken tilbyr 24 BRAM primitiver. Ved å benytte seg av BRAM primitiven kan man ifølge [2, s.4] lagre 18KiB per enhet. Dette gir oss muligheten til å implementere en matrise på 512 ganger 36 bit per trafikkgenerator. (36 bit grunnet at hver BRAM primitiv støtter maksimum en 9 bit adressevektor)

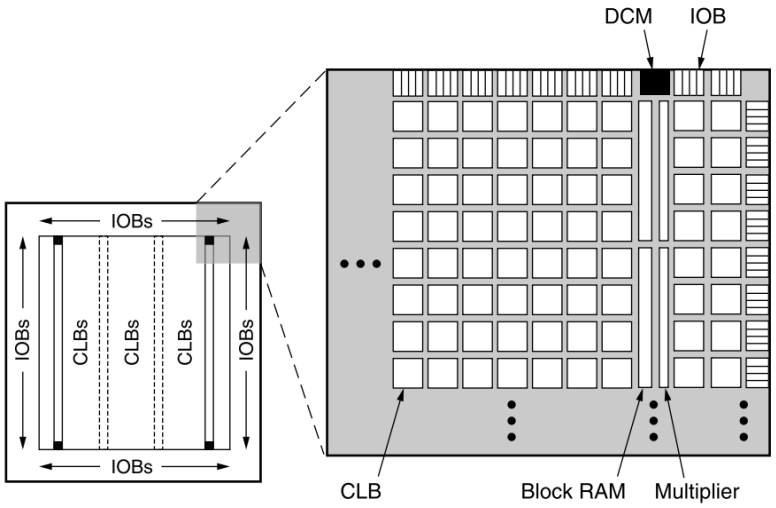
I [1, s.13] kan man observere at ruteren bruker 12 klokkeflanker på å sende en pakke fra en ruter til en annen, dette vil føre til at i løpet av 6144 klokkeflanker kan

en ruter produsere et maksimum av 6144/12 = 512 pakker. Som vil sikre at alle tidsforsinkelsene blir lagret i BRAM-en før utlesning.

Det er også verdt å merke seg at BRAM primitiven støtter dual-port, slik at man kan betjene BRAM-en uanhengig fra både HW og SW i vårt tilfelle. Dette egner seg godt for dette designet, der HW benytter seg av BRAM-en når kretsen er aktiv, mens SW benytter seg av BRAM-en når kretsen er stoppet.

4.6.2.4 – Begrensninger ved bruk av BRAM

Med bruken av BRAM moduler følger det med visse begrensninger når man benytter seg av Spartan 3 brikken. Disse begrensningene kommer i form av at selv om brikken inneholder 24 BRAM moduler og 24 dedikerte multiplikatorer, er det kun mulig å benytte seg av en sammensetning på totalt 24 moduler. Denne begrensningen er grunnet at de to modulene deler felles logikk som kan oppfattes av plasseringen gitt av figuren under.

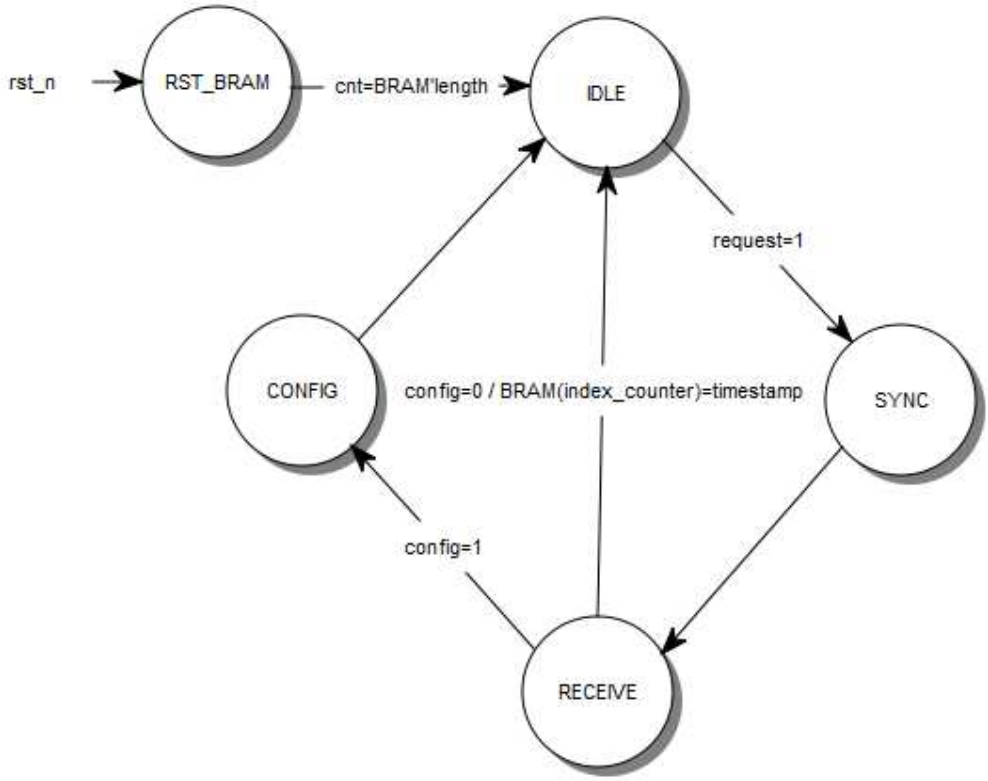


Figur 4.7 – Felles logikk for BRAM og dedikert multiplikator [4, s.4]

4.6.3 – Tilstandsmaskinene

For å kontrollere funksjonaliteten til trafikkgeneratoren, er det implementert to tilstandsmaskiner som speiler kontrollogikken i ruterens kontrollenhet og tar seg av korrekt oppførsel for alle interne registre/signal.

4.6.3.1 – Tilstandsmaskinen for innlesning av pakker



Figur 4.8 – tilstandsmaskinen for innlesning

RST_BRAM: Denne tilstanden er implementert grunnet at BRAM primitiven ikke har noen reset funksjonalitet som skriver over registeret. Ved multiple tester etter hverandre, med en reset av systemet imellom, vil man risikere å lese ut gamle verdier. Når alle verdiene i BRAM matrisen er skrevet over går tilstandsmaskinen videre inn i IDLE.

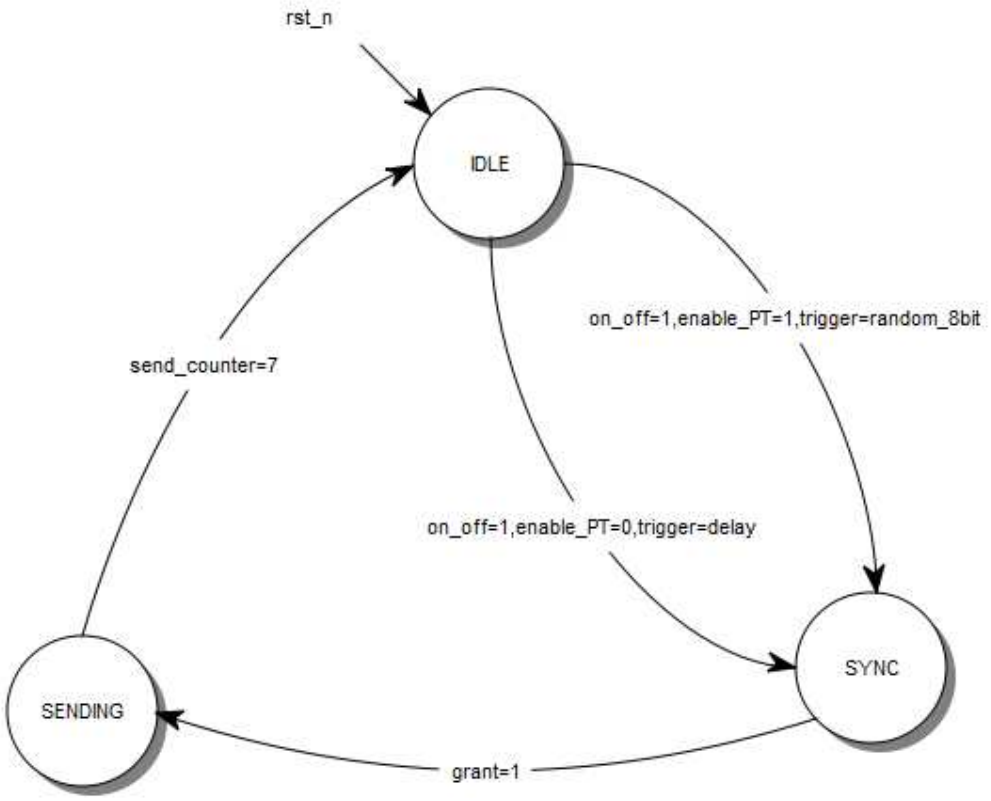
IDLE: I denne tilstanden venter tilstandsmaskinen på et requestsignal. Når en annen ruter ønsker å sende en pakke ved å sette RTS høy, vil tilstandsmaskinen svare ved å sette sitt handshakesignal CTS høyt og entre tilstanden SYNC.

SYNC: Denne tilstanden er implementert grunnet synkroniseringsbehovet mellom sender og mottager, dette grunnet tiden det tar før sender registrerer handshakesignalet CTS.

RECEIVE: I denne tilstanden vil pakken leses inn og sjekkes for høyt konfigurasjonsbit. Hvis det er en konfigurasjonspakke entrer tilstandsmaksinen tilstanden CONFIG. Hvis det ikke er en konfigurasjonspakke, lagres differansen mellom nåtid og stemplingstid av pakken i BRAM-matrisen.

CONFIG: Her leses konfigurasjonsverdiene inn i sine respektive registre i henhold til konfigurasjonspakken gitt i kapittel 4.6.1.

4.6.3.2 – Tilstandsmaskin for utlesning av pakker v.1



Figur 4.9 – Tilstandsmaskinen for utlesning v.1

Denne tilstandsmaskinen er låst til IDLE tilstanden ved reset, dette grunnet at den trenger en konfigurasjon for å aktivere sending av pakker. Når en slik konfigurasjon er mottatt, vil den i henhold til konfigurasjonen begynne å produsere pakker.

IDLE: I denne tilstanden venter tilstandsmaskinen på at triggerverdien skal nå enten verdien på LFSR8 modulen ved høyt enable_PT bit, eller verdien på delay signalet ved lavt enable_PT bit. Dette vil sende tilstandsmaskinen inn i SYNC tilstanden.

SYNC: Her vil tilstandsmaskinen sende sitt handshakesignal til mottageren og vente på svar, når svaret ankommer i form av et høyt grant signal, vil tilstandsmaskinen entre SENDING tilstanden.

SENDING: Her leses de åtte flitene ut på utgangen etter tur, slik at pakken blir overført til mottager.

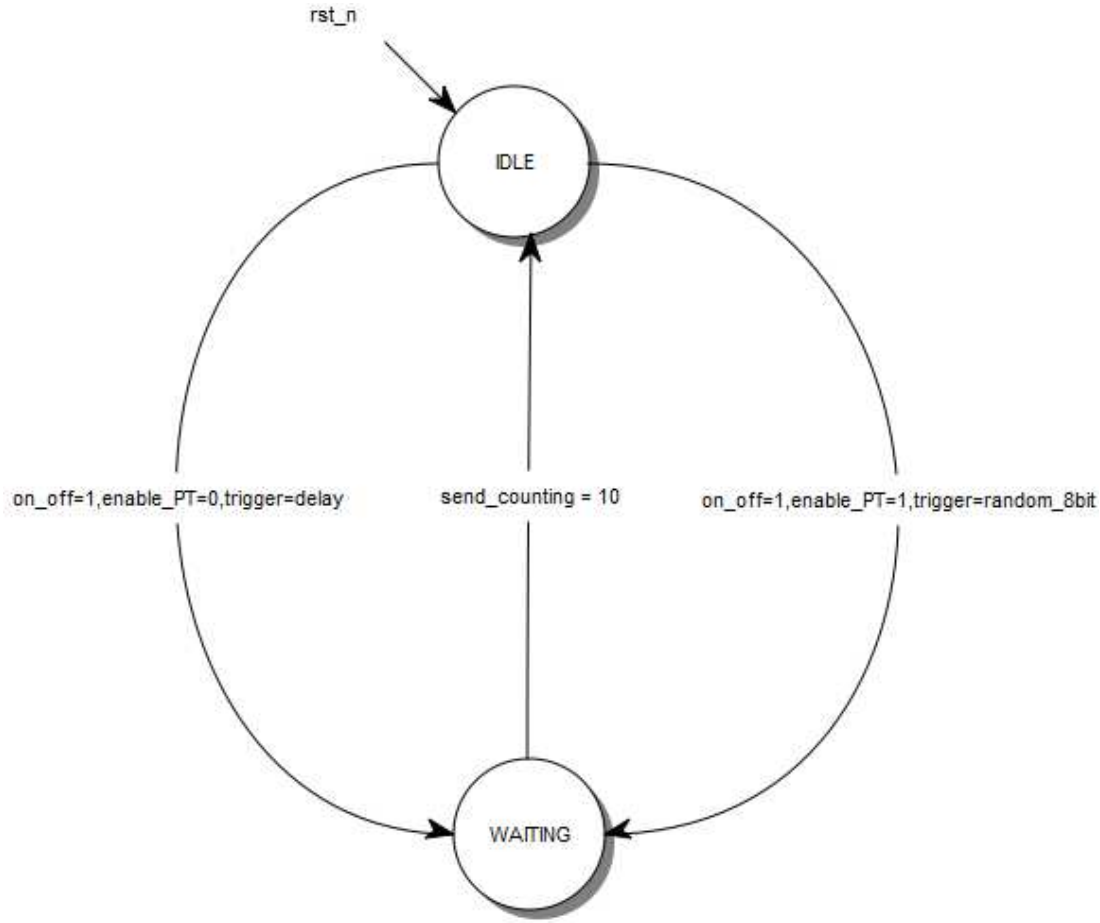
Denne syklusen vil tilstandsmaskinen repetere til den mottar en ny konfigurasjon eller at systemet resettes.

4.6.4 – Tilstandsmaskinene v.2

Under testingen av videoskalereren ble det avdekket et problem med trafikkgeneratorens oppførsel. Dette problemet bestod av at antall pakker som trafikkgeneratoren produserte sank når det var innkommende trafikk til trafikkgeneratoren, selv om ruterens totale båndbredde ikke var utnyttet.

4.6.4.1 – Tilstandsmaskin for inkrementering av sendeteller

Slik som systemet var konstruert, var hensiktet at trafikkgeneratorens throughput skulle være direkte gitt av konfigurasjonspakkens delay-verdi (eller LFSR8 verdi). Slik som systemet faktisk fungerte var throughputen også direkte avhengig av utlesningstiden på pakkene. For å fikse dette problemet ble logikken for initiering av en pakkeutlesning flyttet inn i en egen tilstandsmaskin. Ved å gjøre dette kunne man dermed bryte avhengigheten til utlesningstiden og oppnå den throughputen som konfigurasjonen skulle tilsi.

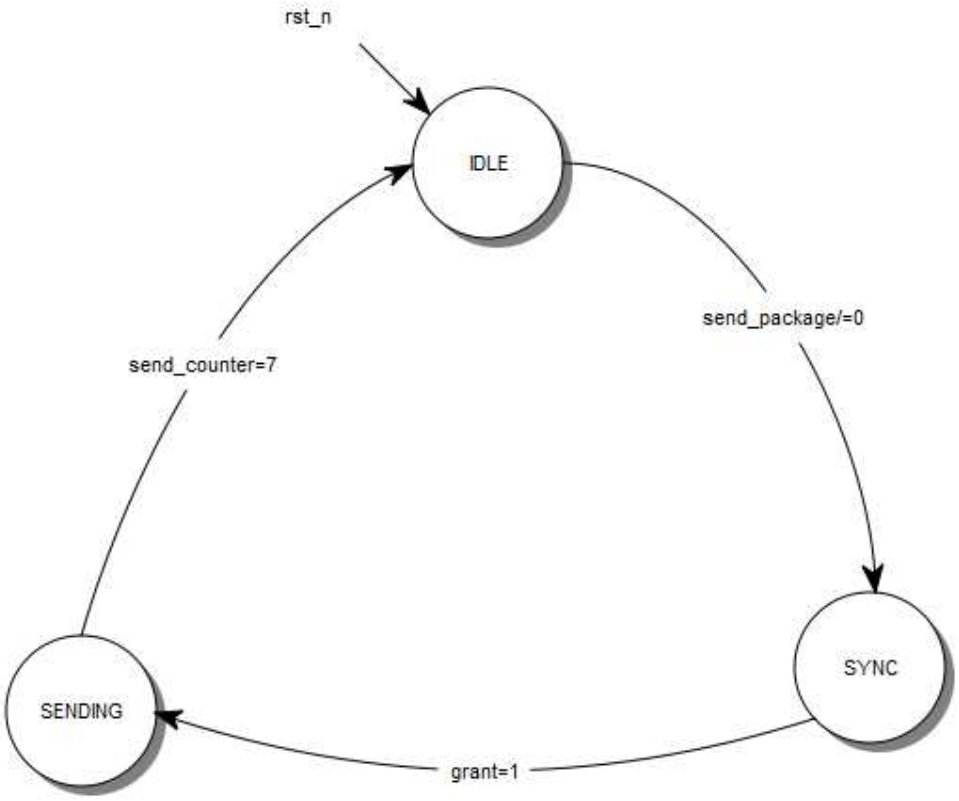


Figur 4.10 – Tilstandsmaskin som styrer throughputen i trafikkgeneratoren

Figuren over viser at logikken for å sende en pakke er uforandret. Forskjellen ligger i at i stedet for å fysisk starte en pakkesending, vil nå en overgang mellom IDLE og WAITING inkrementere signalet send_package. På denne måten vil produksjonen av pakker være uavhengig av tiden det faktisk tar å sende pakkene.

4.6.4.2 – tilstandsmaskin for utlesning av pakker v.2

Som et resultat av den nye tilstandsmaskinen måtte tilstandsmaskinen for utlesninger av pakker modifiseres. En transisjon mellom IDLE og SYNC er nå aktivert av signalet send_package. Når dette signalet har en verdi høyere enn null, vil trafikkgeneratoren sende en pakke, på denne måten sikrer man seg at i perioder der den inngående trafikken hemmer pakkesending, vil utlesningen av pakker i tiden etter øke for å oppnå den throughputen som er ønskelig.



Figur 4.11 – Tilstandsmaskin for utlesning v.2

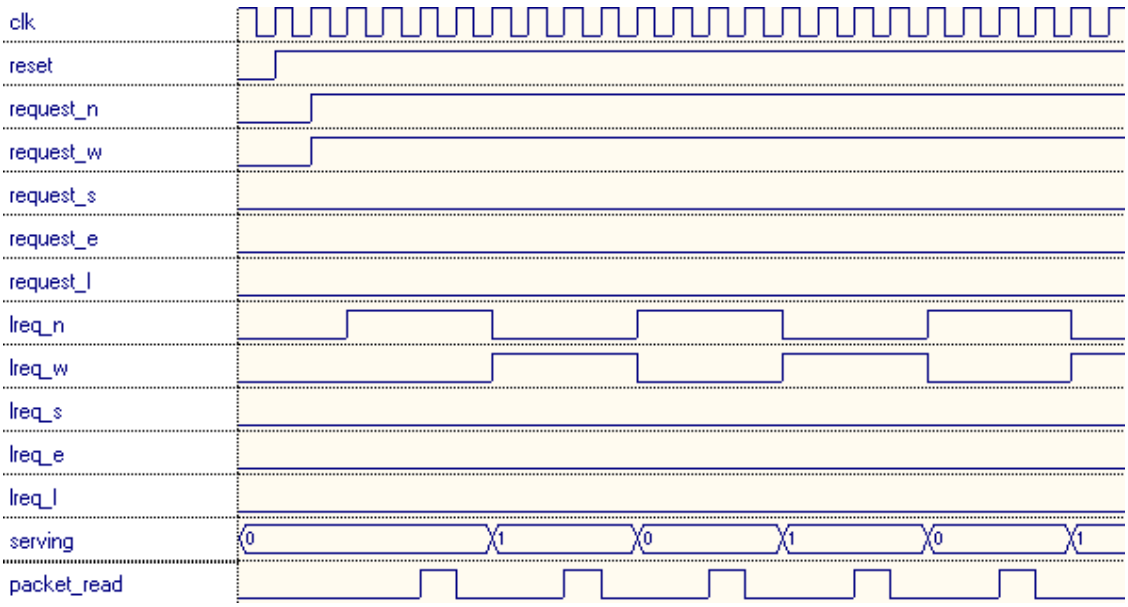
Kapittel 5 – Simulering

For å sikre funksjonaliteten til de nyutviklede modulene, samt vise korrekt oppførsel presenteres her et utvalg simuleringer av systemet. Simuleringene i dette kapittelet er gjennomført ved hjelp av Active Hdl Student Edition 7.2 på en 50 MHz klokke. Simuleringene som er gjennomført tar sikte på å bekrefte normal funksjonalitet og avdekke eventuelle feil og mangler ved hjørnetilfeller. Dette er et preventivt formål, da det er ønskelig å fjerne alle slike feil og mangler før implementering og test.

5.1 – Arbiter

I samhold med at støtten for Quality of Service ble deaktivert, ble en ny arbiter designet med hensyn på enkelt design og enkel forståelse.

5.1.1 – Test #1



Figur 5.1 – Arbiter test #1 – dobbel request

Ved et konstant høyt requestsignal fra både den nordlige og vestlige inngangen, er den ønskede oppførselen at grant signalet alternerer imellom de to inngangen for å unngå utsultning.

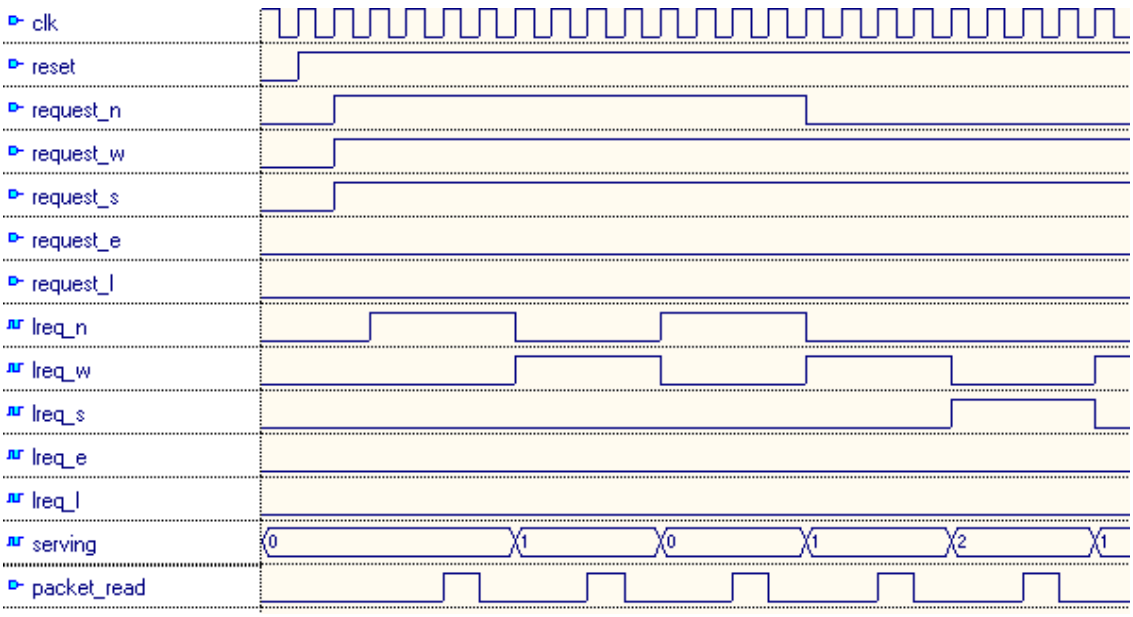
Etter den nordlige retningen (0) har fått sitt grantsignal grunnet høyest prioritet, vil arbiteren benytte seg av det høye lreq_n signalet, som signaliserer at det var denne retningen som mottok forrige grant signal. Dermed avvises en ny grant til samme retning og den vestlige retningen (1) får det neste grant signalet. Når arbiter arbitrerer for tredje gang begynner syklusen på nytt og dermed vil man oppnå den ønskede effekten med alternerende grantsignal mellom de to inngangene. Det kan her observeres at packet_read signalet går høyt før en pakke på 8 flit har muligheten til å overføres, dette er grunnet at testen skal få et mindre tidsspenn.

5.1.2 – Test #2

Slik som arbiteren er konstruert, er det ikke tatt høyde for høy pågang fra tre innganger samtidig. Ivar Ersland antok i [2] at:

”For å spare ressurser antar vi at tre pakker ikke kan ankomme på samme klokkeflanke”

Ved å følge denne antagelsen er det kun implementert ett minnesteg. Dette vil si at arbiteren kun husker hvem som mottok det siste grantsignalet.

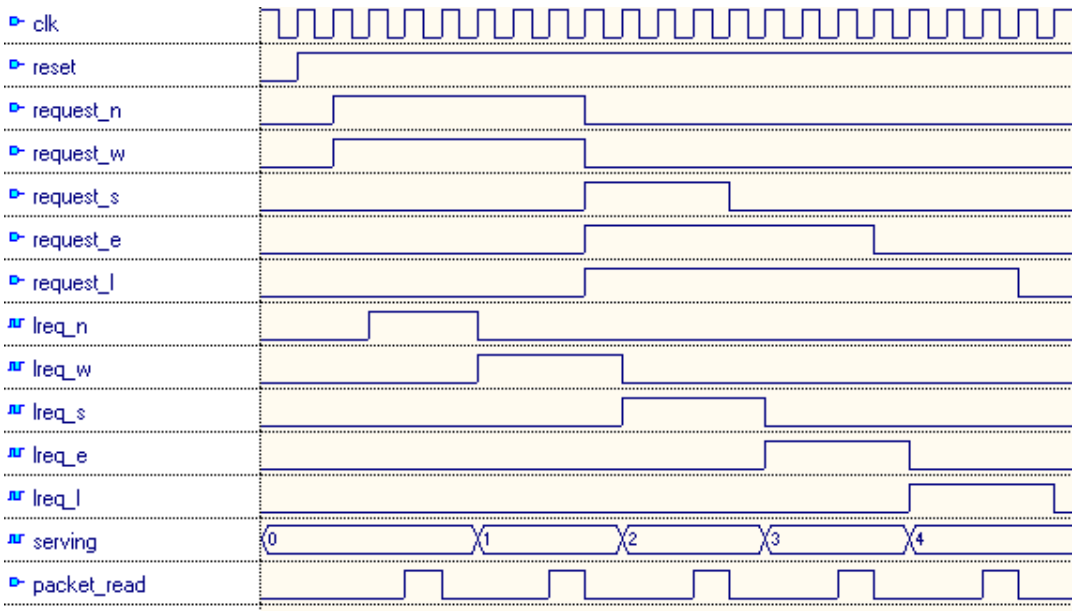


Figur 5.2 – Arbiter test #2 – tre samtidige request over tid

Figuren over viser tre kontinuerlig høye requestsignal, som i følge prioritetsrekkefølgen burde gi en grant til nord, så en grant til vest, for til slutt å gi en grant til sør. Det som skjer i et slikt tilfelle er at nord og vest får hvert sitt grantsignal, men når turen kommer til den sydlige inngangen, har ikke arbiter mer enn lreq_w å forholde seg til, slik at den nordlige inngangen vil utsulte den sydlige inngangen grunnet høyere prioritet. Det er først når den nordlige inngangen legger sitt requestsignal lavt at den sydlige inngangen får en grant.

Det er verdt å merke seg at dette problemet kun vil oppstå ved meget høy pågang fra tre eller flere innganger samtidig. Ved lavere pågang vil hver inngang rekke å legge sin request lav, slik at andre har muligheten til å komme til. Dette er vist i neste test.

5.1.3 – Test #3



Figur 5.3 – Arbiter test #3 – multiple requester

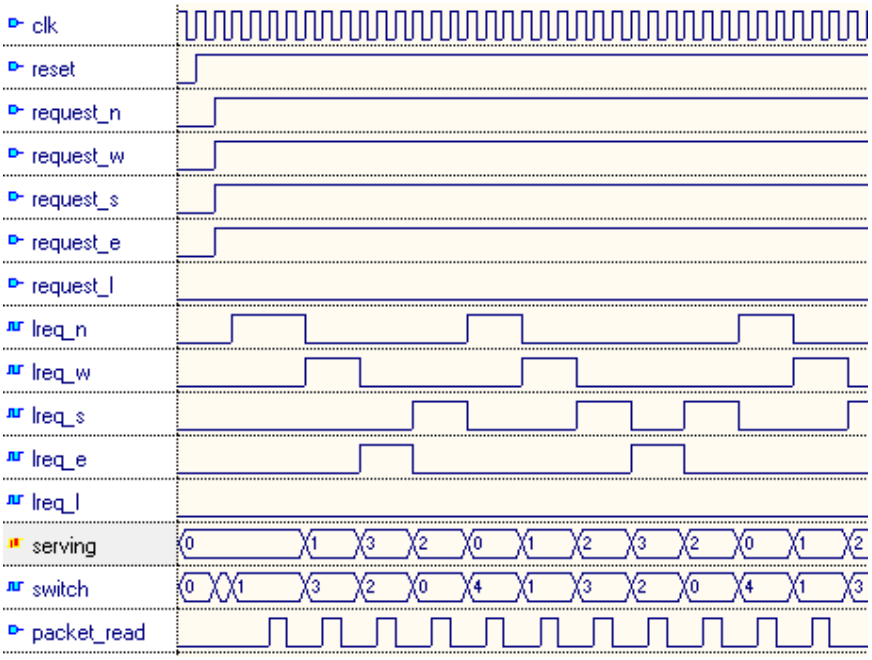
Figuren over viser at arbiteren følger den ønskede oppførselen for to samtidige requestsignal i henhold til test nummer en.

Ved 3 samtidig høye requestsignal oppnåde man i den forrige testen utsulting av inngangen med lavest prioritet. I denne testen observeres det at grant signalet (serving) alternerer mellom alle inngangene og at det dermed ikke oppstår noen utsulting.

Denne oppførselen oppnås ved lavere trafikkpågang fra inngangene, da de i et slikt tilfelle vil rekke å legge requestsignalet lavt før de er klar til å sende en ny pakke. Ved å gjøre dette, vil arbiteren gi grant til sør, øst og lokal inngang i henhold til den predefinerte prioritetsrekkefølgen.

5.1.4 – Test av ”proof of concept” arbiter

Siden arbiteren ikke er rettferdig er det designet en mer rettferdig arbiter for ”proof of concept”. Denne arbiteren har et switch signal implementert, slik at den vil endre hele prioritetsrekkefølgen basert på hvem som fikk grant sist. Dette vil bety en mer kompleks avhengighet til lreq signalene, slik at arbiter vil gi ut en mer rettferdig fordeling av grant signal.



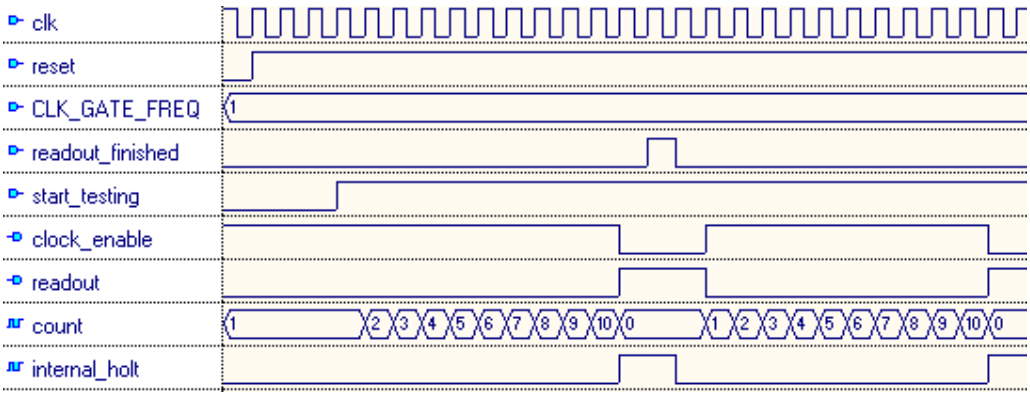
Figur 5.4 – Arbiter test #4 – fire samtidige request med alternerende prioritet

I figuren over er det utført en enkel test av denne ”proof of concept” arbiteren. Her observeres det fire kontinuerlig høye requestsignal. I følge oppførselen til den opprinnelige arbiteren ville den sørlige og østlige inngangen ha blitt utsatt for utsulting, mens her observeres det at man har en jevnere fordeling av grant signalet og at ingen av inngangen blir utsatt for utsulting.

Denne arbiteren er ikke brukt i videre testing av designet, da den under syntese hadde et arealforbruk som er ~60% høyere enn den gjeldene arbiteren. Den er kun testet for ”proof of concept” for å vise at dette er en måte å forbedre en delvis urettferdig arbiter på hvis det er nødvendig. Arbiterens kildekode ligger i filvedlegget som følger med oppgaven.

5.2 – TM control

Denne modulen styrer klokken for det interne ruternettverket. Med hensyn på tidsforløpet for simuleringen er logikken beskrevet i kapittel 4.4 endret for denne testen. I stedet for å trigge en utlesning på CLK_GATE_FREQ*6144 er dette nå endret til CLK_GATE_FREQ*10.

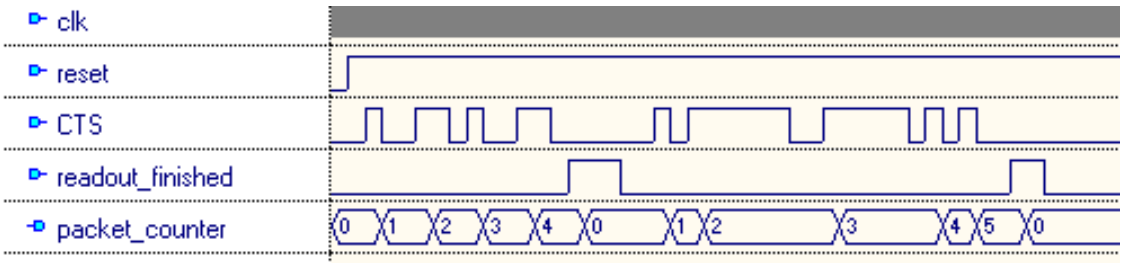


Figur 5.5 – TM control test

I figuren over observeres det at TM control-enheten starter den interne telleren når start_testing signalet fra SW går høyt. Signalet count inkrementeres til det når CLK_GATE_FREQ*10, før den setter clock_enable signalet lavt og readoutsignalet høyt. Dette vil signalisere programvaren om at det ligger data klar til utlesning. Når denne utlesningen er ferdig vil programvaren varsle ved å sette signalet readout_finished høyt, noe som vil starte syklusen på nytt igjen.

5.3 – TrafficMonitor

Denne modulen skal kun telle pakker i henhold til beskrivelsen i kapittel 4.5. Denne oppførselen er enkel å vise og er gjengitt i figuren under.



Figur 5.6 – Test av Trafikkmonitoren

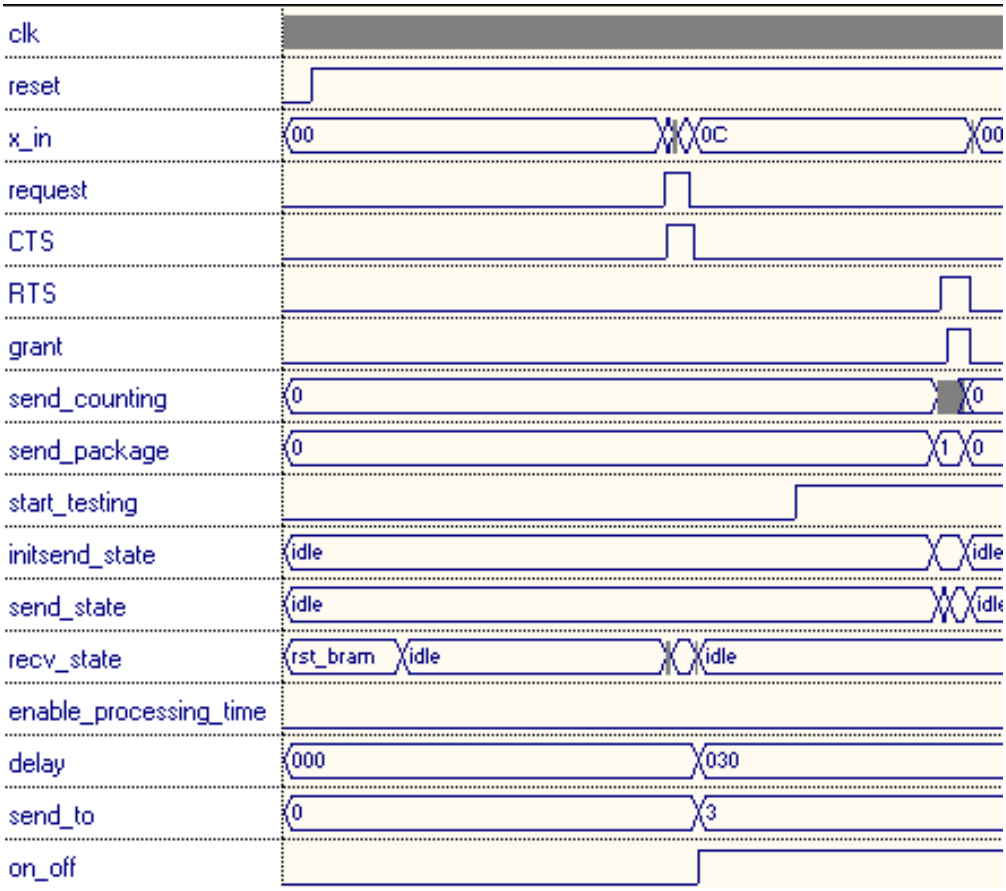
Når trafikkmonitoren observerer en transisjon fra lav til høy på CTS inngangen, vil den inkrementere den interne pakketelleren sin, packet_counter. Når verdien blir lest av programvaren, går readout_finished høy og modulen resetter den interne pakketelleren.

5.4 – TrafficGenerator – Enkel konfigurasjonstest

For å teste trafikkgeneratoren er det nødvendig å se at den mottar en konfigurasjon og setter de interne variablene i henhold til de mottatte dataene. I denne testen konfigureres trafikkgeneratoren med følgende data.

- On/Off 1
- Enable-PT 0
- Send til 3
- Delay 48 (30 heksadesimalt)

Det tar 12 klokkesykler å sende en pakke, slik at når forsinkelsen er 48, vil TG-en produsere pakker med en normalisert belastning på 12/(12+48) = 0,2 = 20 %

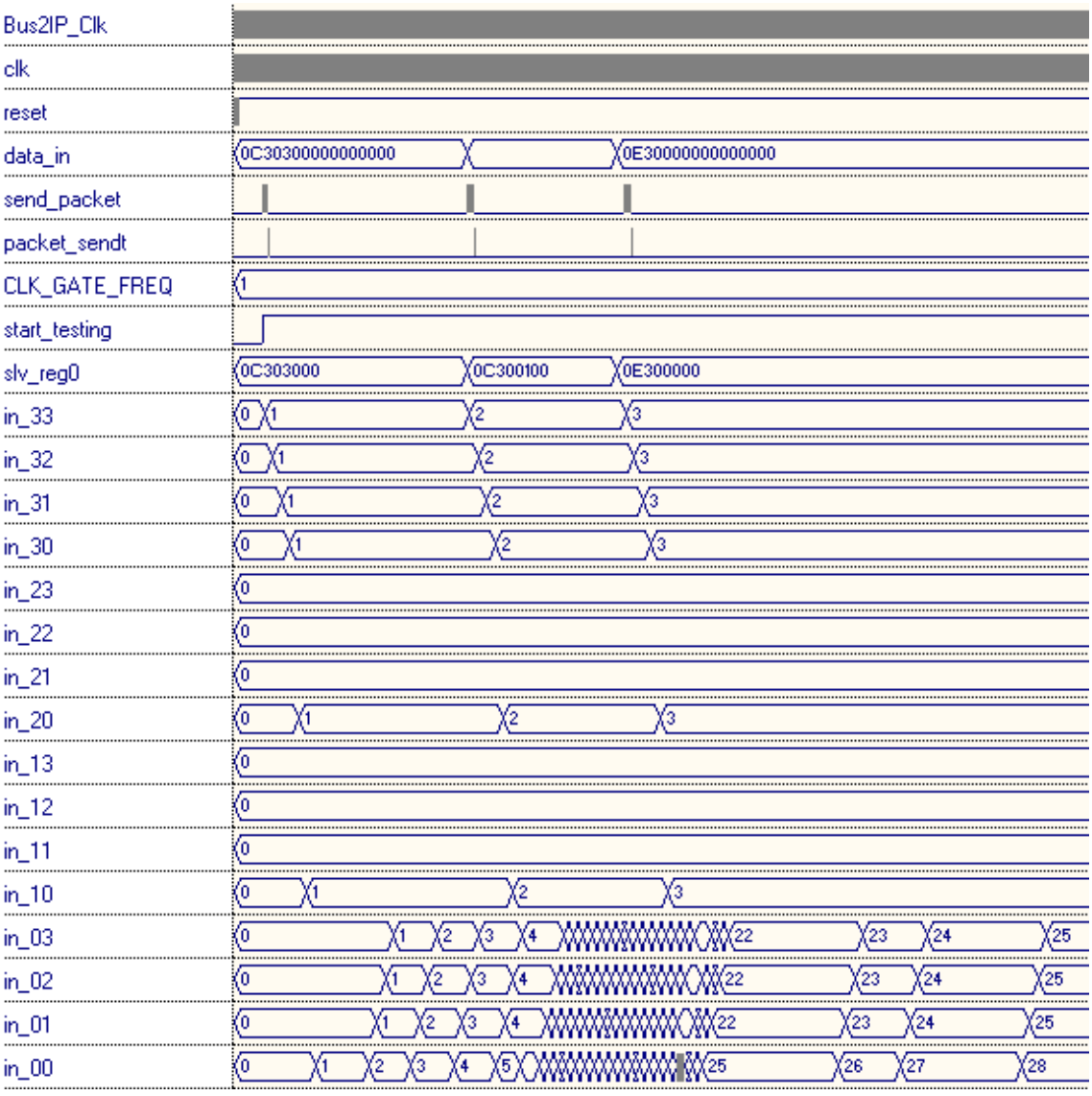


Figur 5.7 – Singel rekonfigurasjon av TG

I denne testen vil først tilstandsmaskinen for mottak av pakker resette hele BRAM-rekken i tilstanden rst_bram. Videre observeres en høy request, med et påfølgende høyt CTS signal. Dette var mottaket av konfigurasjonspakken, slik at videre observeres det at on_off går høy, send_to får verdien 3, delay får den heksadesimale verdien 030 og enable_processing_time holdes fortsatt lav. Når konfigurasjonen er mottatt, ser man at RTS signalet går høyt, med et påfølgende grant signal. Noe som vil si at konfigurasjonen er korrekt og trafikkgeneratoren har begynt å produsere trafikk.

5.5 – TrafficGenerator – Fullstendig systemtest.

For å teste at trafikkgeneratorene fungerer etter hensikt, er det nødvendig å teste de på det høyeste hierarkiske nivået mulig. Dette vil si at designet i sin helhet vil testes, med fokus på korrekt oppførsel til både system og trafikkgeneratorene.



Figur 5.8 – Konfigurering og rekonfigurering av ruter null

I figuren over kan man observere at i henhold til oppbygningen beskrevet i kapitel 4.6.1 sendes det en konfigurasjonspakke til ruter null med følgende informasjon.

- Config 1
- On/Off 1
- Enable-PT 0
- Send til 3
- Delay 48 (30 heksadesimalt)

Dette vil konfigurere ruteren til å sende en pakke til ruter 3 med 48 klokkesyklers mellomrom, noe som kan observeres ved at pakketellerne fra null til tre inkrementeres med 60 klokkesyklers mellomrom. (Grunnet 12 klokkesyklers sendetid)

Videre observeres det at dataene på data_in inngangen endres slik at innholdet i konfigurasjonspakken nå er endret til.

- Config 1
- On/Off 1
- Enable-PT 0
- Send til 3
- Delay 1

Et intern delay i trafikkgeneratoren på 1 klokkesyklus vil tilsvare en normalisert belastning på

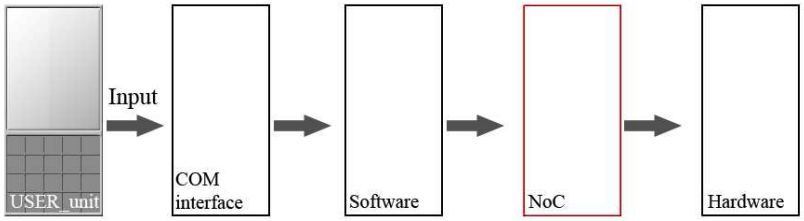
(1) $$\mathrm{NL} = \frac{\mathrm{delay}_{\mathrm{SEND}}}{\mathrm{delay}_{\mathrm{SEND}} + \mathrm{delay}} = \frac{12}{12+1} \approx 92\%$$

Denne konfigurasjonspakken sendes til ruter null som observeres ved at send_packet har en lav->høy transisjon. Med en gang denne konfigurasjonspakken mottas på ruter 0, observeres det at produksjonsraten av pakker økes i henhold til den nye delay-verdien.

Til slutt så sendes det en ny pakke der tidsforsinkelsen settes til null, dette er trivielt da også enable-PT settes høy. Dette vil aktivere LFSR8 enheten i TG-en, som kan observeres ved at pakketellerne inkrementeres med uregelmessige intervall i henhold til den pseudotilfeldige funksjonaliteten til LFSR8 enheten.

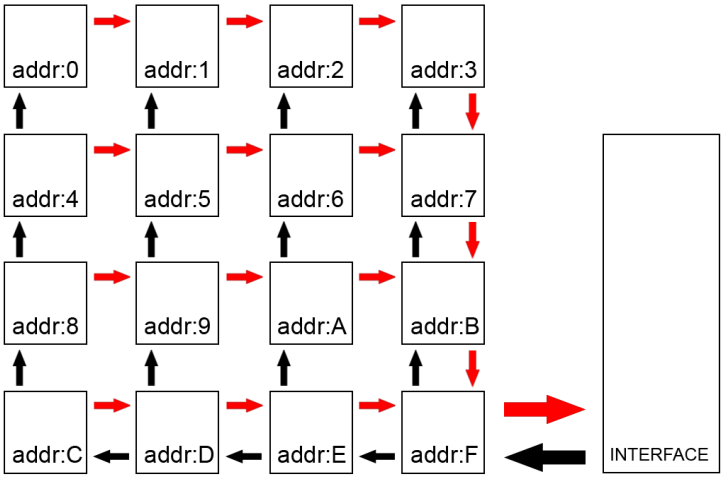
Kapittel 6 – Test på FPGA

For å kunne teste et design, er det viktig å ha en inngående forståelse for hvordan dette systemet skal fungere. AHEAD prosjektet tar sikte på å tilby mindre regnekraftige enheter maskinvareakselerasjon av ønskede applikasjoner. Måten en AHEAD node skal gjennomføre dette på, er ved å motta en konfigurasjonsfil, for så å dynamisk rekonfigurere en FPGA til å tilby det gitte designet. Videre skal AHEAD noden motta programvaren som skal kjøre oppå den gitte maskinvaren, og sette opp en kommunikasjonslink mellom programvaren og brukeren. Den viktige delen for testingen av dette designet er kommunikasjonen mellom maskinvaren og programvaren, da det er denne delen som benytter seg av det interne nettverket, bedre kjent som NoC (Network on Chip)



6.1 – AHEAD datavei

Slik som ruternettverket er oppbygd vil programvaren opprette en pakke på bakgrunn av informasjonen den har mottatt fra brukeren, for så og sende denne inn i ruternettverket gjennom ruteren som befinner seg i det nederste høyre hjørnet. Dette innebærer at med et mesh oppsett, forklart i [2, s.7], vil all trafikk fra brukeren/programvaren traversere mot venstre og oppover (svarte piler, figur 6.2), mens all trafikk til brukeren/programvaren vil traversere mot høyre og nedover (røde piler, figur 6.2).

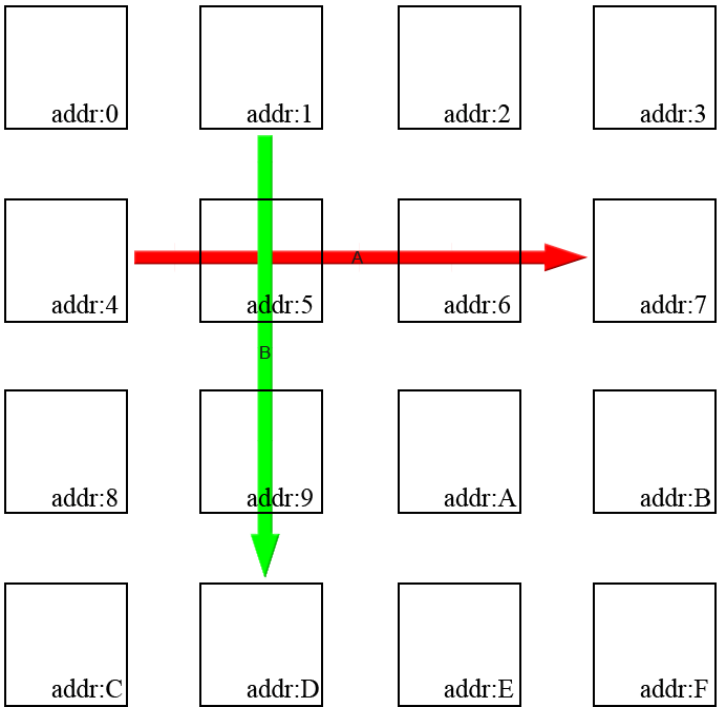


Figur 6.2 – Trafikkmønster for kommunikasjon med brukeren/programvaren

Ut ifra dette kan man enklere forstå hvordan trafikken traverserer gjennom ruteren og hvordan man enklere kan plassere modulene for å unngå flaskehalser.

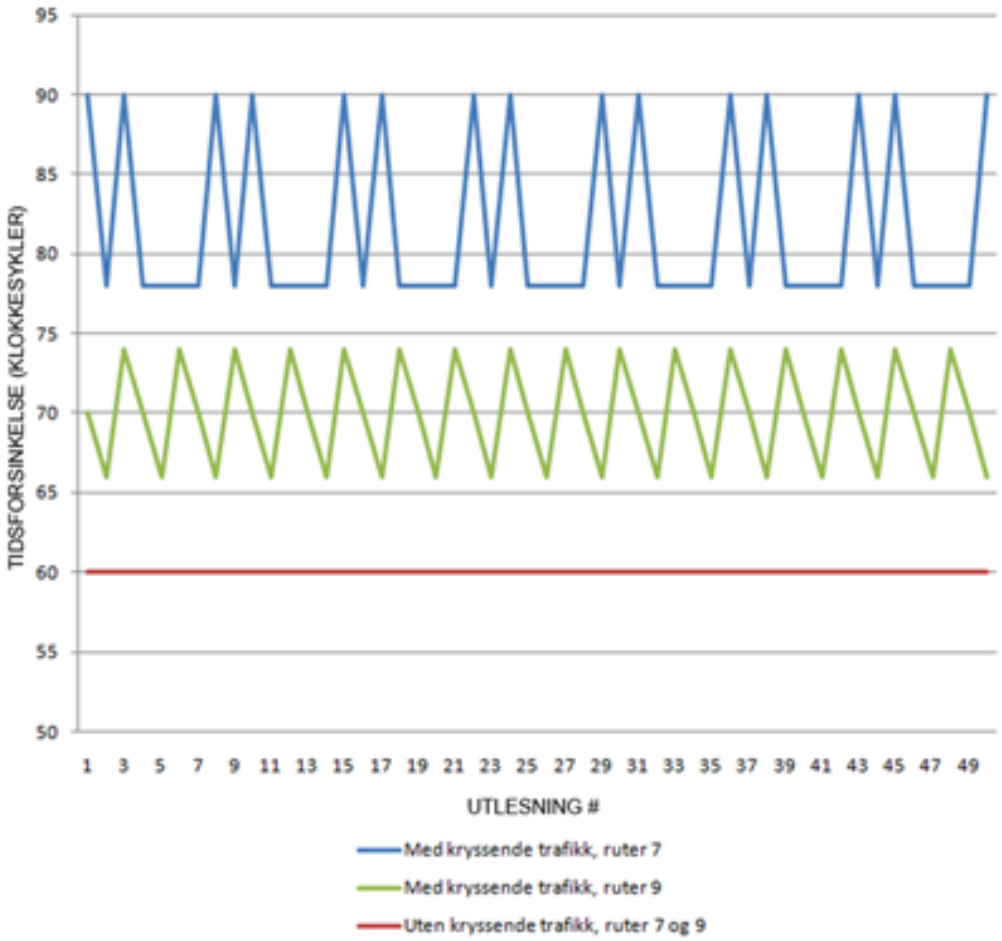
6.1 – Første test – enkel sending med og uten kryssende trafikk

For å kunne se hvordan en kryssende datastrøm påvirker tidsforsinkelsen og antall genererte pakker, settes det opp to datastrømmer i ruternettverket. Den første datastrømmen, datastrøm A, går fra ruter 4 til ruter 7. Den andre datastrømmen, datastrøm B, går fra ruter 1 til ruter D i henhold til figuren under. For å øke sannsynligheten for at begge datastrømmene benytter seg av ruter 5 på samme tid, samtidig som man kan observere når og hvor dette skjer, vil det være hensiktsmessig at datastrøm A har høy throughput, mens datastrøm B har lav til middels throughput. I denne testen er throughputen til henholdsvis datastrøm A og B satt til 100 % og 30 % av normalisert belastning.



Figur 6.3 – Kryssende datastrømmer

Testen vil først aktivere datastrøm A, for å se hvordan verdiene på tidsforsinkelse og antall pakker er uten kryssende trafikk. Videre vil datastrøm B aktiveres, slik at man kan sammenligne resultatene.

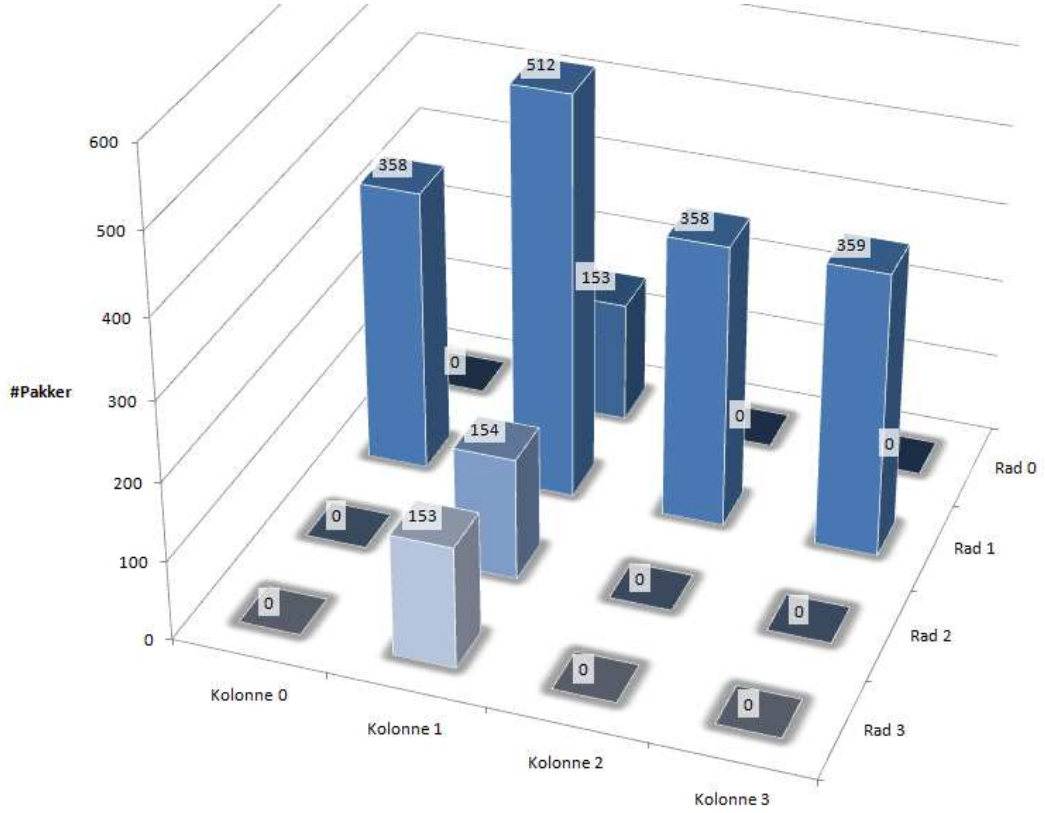


Figur 6.4 – Tidsforsinkelsene ved test #1

Etter å ha kjørt testen med kun én av datastrømmene aktivert, får man en tidsforsinkelse som angis av den røde (nederste) linjen i figuren over. Den ligger på konstant 60 klokkesykler forsinkelse, som er et resultat av at den ikke har noen kryssende datastrøm. Når man aktiverer den kryssende trafikken, vil tidsforsinkelsen til begge datastrømmene få et høyere gjennomsnitt, samtidig som de varierer periodisk. Det er viktig å observere at grunnet forskjellen i den normaliserte belastningen på pakkestrøm A og B, vil ikke utlesning N for ruter 7 og 9 samsvare i tid, slik at en direkte sammenligning mellom grafene er umulig.

Videre observeres det at den høyeste tidsforsinkelsen til datastrøm A er lik 90 klokkesykler, noe som er 150 % av tidsforsinkelsen uten krysstrafikk. Siden datastrøm B har en normalisert belastning på 30 %, kunne man tenkt seg at hver gang den produserte en pakke, økte tidsforsinkelsen med maksimalt 12 klokkesykler. Det er verdt å tenke på at datastrøm A produserer pakker og sender de så fort ruteren klarer. Et avbrudd på 12 klokkesykler vil få ringvirkninger og forsinke

alle pakkene som er produsert, slik at man på denne måten kan forvente en høyere maksimalverdi enn 12.



Figur 6.5 – Antall sendte pakker i systemet ved test #1

Når testsystemet kjører med én aktiv pakkestrøm på 100 % av normalisert belastning, vil den i henhold til kapittel 4.4 klare å produsere 512 pakker i løpet av perioden mellom utlesning. Ved å ha kryssende trafikk, kan man observere at antall produserte pakker påvirkes kraftig. Det observeres av figuren over at ruter nr 5 (Rad 1, Kolonne 1), har 512 pakketraverseringer, mens de resterende ruterne i pakkestrøm A (Rad 1), har kun 358/359. Videre ser man at pakkestrøm B, (Kolonne 1), har 153/154 pakketraverseringer.

Pakkestrøm A kjørte under denne testen på en normalisert belastning på 100 %, mens pakkestrøm B kjørte på 30 %. Dette vil tilsi at man kan regne ut antall pakker hver av pakkestrømmene skal generere i løpet av perioden for en utlesning.

(2) $$\#\text{packages}_{\text{MAX}} \bullet \text{NL} = 512 \bullet 1.0 = 512$$

(3) $$\#\text{packages}_{\text{MAX}} \bullet \text{NL} = 512 \bullet 0.3 = 153,6$$

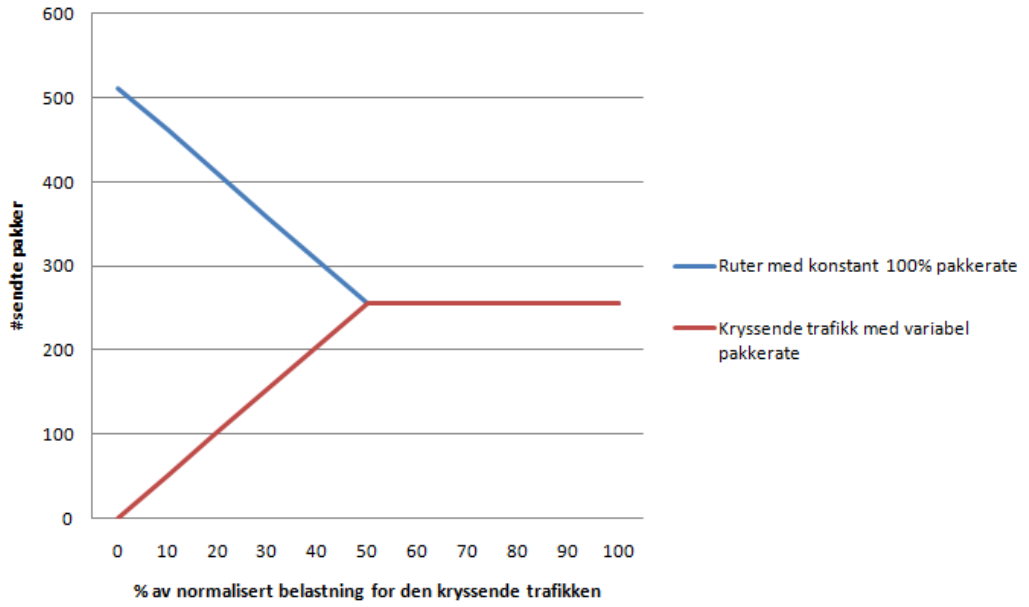
Ut ifra dette kan man se fra figur 6.5 at pakkestrøm B produserer like mange pakker som er forventet, mens pakkestrøm A produserer et gitt antall mindre. Ved å regne ut hvilken normalisert belastning pakkestrøm A kjører på, kan man finne en sammenheng mellom pakkestrøm A og B.

(4) $$\frac{358 \bullet \text{delay}_{\text{SEND}}}{\text{delay}_{\text{SEND}} \bullet \#\text{packages}_{\text{MAX}}} = 0.7$$

Ut ifra ligningen over er det enkelt å se at pakkestrøm A sin throughput har blitt redusert med pakkestrøm B sin throughput ned til 70 % normalisert belastning. Dette legger grunnlaget for neste test.

6.2 – Andre Test – Testing av krysstrafikk med variabel normalisert belastning

I test nummer en var det enkelt å se at den normaliserte belastningen til datastrøm A sank med like mye som den normaliserte belastningen til datastrøm B. Samtidig som den totale tidsforsinkelsen gjennom systemet økte. Ved fortsatt å ha datastrøm A konstant kjørende på 100 % av normalisert belastning, mens man endrer datastrøm B til å kjøre på en normalisert belastning mellom [0, 100] kan man få en oversikt over hvordan datastrøm A reagerer på den økende belastningen.

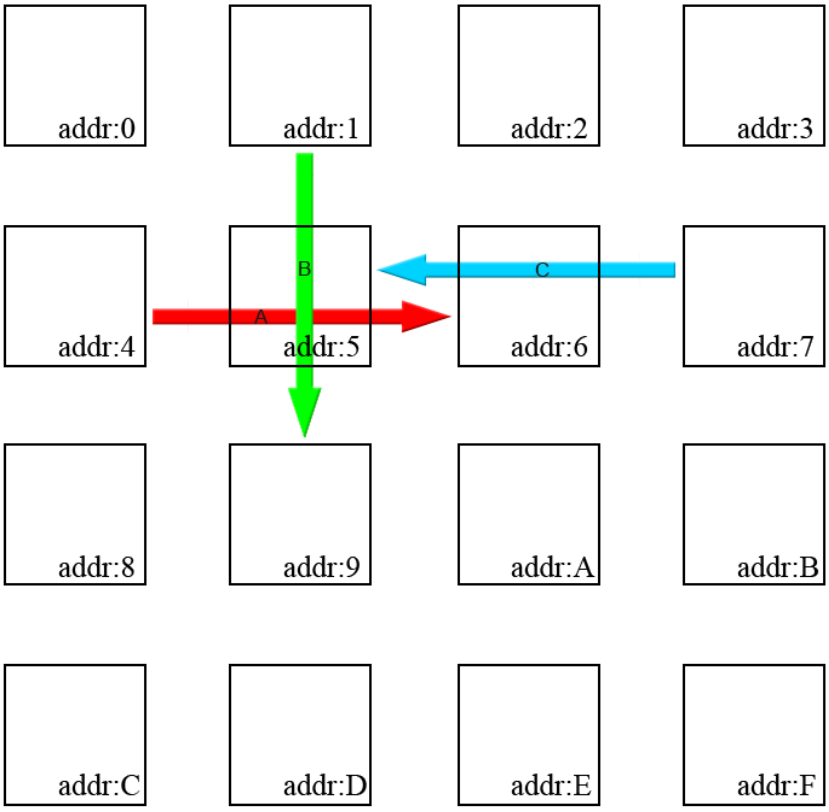


Figur 6.6 – Økende normalisert belastning på datastrøm B

I figuren over bekreftes det at datastrøm A sin normaliserte belastning er avhengig av datastrøm Bs. Det observeres at hvis to datastrømmer benytter seg av samme ruter, vil de to datastrømmenes samlede normaliserte belastning ikke kunne overskride 100 %. Dette medfører at så lenge arbiter gir grant i en rettferdig rekkefølge, vil all trafikk som traverserer gjennom samme ruter være avhengig av hverandres normaliserte belastning.

6.3 – Tredje test – Treveis sending gjennom ruter #5

I henhold til figur 6.2, ser man at i det ferdige systemet vil trafikk fra brukeren/programvaren møte motgående trafikk som er utgående fra systemet. Grunnet dette vil det være hensiktsmessig å sette opp en test som ser hvordan systemet takler slik motgående trafikk. For å gjøre testen litt mer kompleks, er det også lagt til en tredje datastrøm som krysser de to motgående datastrømmene.



Figur 6.7 – Tre kryssende datastrømmer

I denne testen kjørte alle datastrømmene på 50 % av normalisert belastning, slik at ruter 5 opplevde en reell belastning på 150 %. Resultatene fra denne testen var at alle pakketellerne leste ut null, noe som indikerer at hele systemet har gått i vranglås. Dette er videre diskutert i kapittel 7.3.3.

6.4 – Trafikkplanlegging av On-the-fly videoskalering på FPGA

I disse testene er det flere muligheter for plassering av testmodulene, det må påpekes at grunnet funnene i 6.3 er det nødvendig å sette opp testene slik at systemet ikke går i vranglås. Dette innebærer at en ruter ikke kan ha innkommende trafikkstrømmer fra tre forskjellige retninger samtidig.

Før det er mulig å regne ut den normaliserte belastningen de forskjellige modulene i testoppsettet utøver på ruterne, er det nødvendig å justere båndbredden for å kompensere for pakkedata som ikke er nyttelast.

6.4.1 – Justering av båndbredde

Båndbredden til systemet er kritisk for hvor mye informasjon systemet kan håndtere over et gitt tidsrom. Den teoretiske båndbredden til systemet hentet fra [1, s.17] er gitt som

(5) $$8\text{bit} \bullet 132\text{Mhz} = 1056\text{Mbit/s} = 132\text{MB/s}$$

Slik som båndbredden ble definert, tar den ikke høyde for at noen av bitene i en pakke ikke er nyttelast. Ved å ta høyde for at en pakke inneholder fire adressebit og ett konfigurasjonsbit, fordelt over 8 pakker, vil den reelle nyttelasten per pakke være.

(6) $$\frac{(8\text{bit} \bullet 8\text{flit}) - 5\text{bit}}{8\text{flit}} = 7.375\text{bit}$$

Ved å anvende dette tallet i utregningen av den teoretiske båndbredden, blir den nye teoretiske båndbredden justert ned i henhold til ligningen under.

(7) $$7.375\text{bit} \bullet 132\text{Mhz} \bullet 2\,\text{kanaler} = 1947\,\text{Mbit/s} = 243\,\text{MB/s}$$

For å regne ut den faktiske båndbredden, må man ta hensyn til at klokkefrekvensen synker når nettverket kobles opp imot brukergrensesnittet (interface.vhd) [1, s.17]. Det må også tas høyde for synkronisering og handshake mellom ruterne [1, s.13, figur 4.2], slik at den faktiske båndbredden er gitt av ligningen under.

(8) $$(7.375\text{bit} \bullet 8\text{flit}) \bullet \frac{118\text{Mhz}}{12\,\text{klokkesykler}} \bullet 2\,\text{kanaler} = 1160\,\text{Mbit/s} = 145\,\text{MB/s}$$

6.4.2 – Oppsett for skalering av mpeg4 videostrøm

I en konvensjonell mpeg4 encoder/decoder vil hver av disse modulene i følge [5. s.19] bestå av fem submoduler hver. Disse modulene er typisk.

Encoder

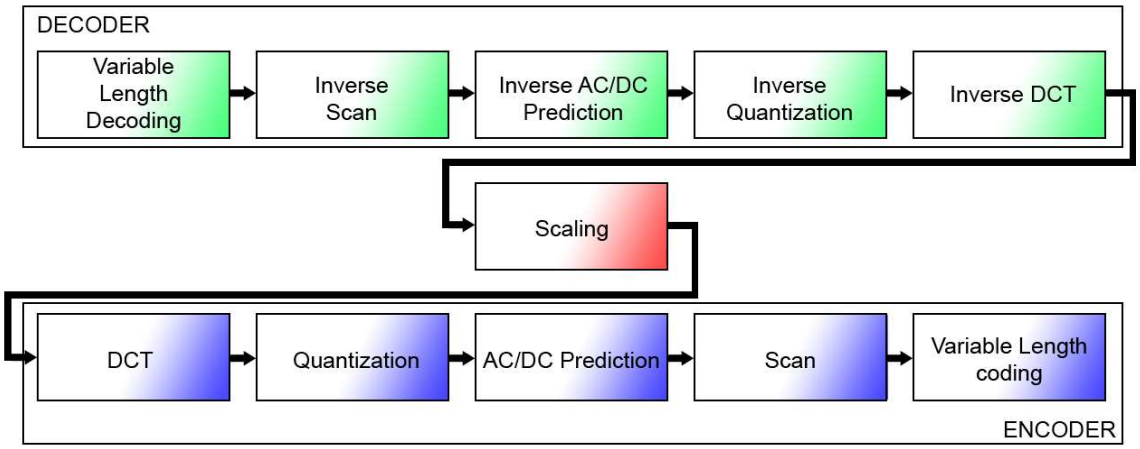
- DCT
- Quantization
- AC/DC Prediction
- Scan
- Variable Length Coding

Decoder

- Variable Length Decoding
- Inverse Scan
- Inverse AC/DC Prediction
- Inverse Quantization
- Inverse DCT

I vårt testoppsett er tankegangen at en videostrøm skal skaleres ned til å passe til vår håndholdte enhet. I følge [5, s.27] gjennomføres en slik skalering etter at videostrømmen er decodet, dette vil tilsi at brukeren er nødt til å sende sin videostrøm inn til AHEAD-noden, der den vil decodes og skaleres før den til slutt encodes for å kunne spilles av på enheten i en oppløsning som er tilpasset skjermen.

Dette innebærer at for vårt tilfelle vil en videostrøm som skal skaleres gjennomgå 11 trinn. Fem trinn i decodingen av videostrømmen, ett trinn i skaleringen og de fem siste trinnene i encodingen. Disse elleve trinnene er vist i figuren under.



Figur 6.8 – skalering av videostrøm

Når en videostrøm ankommer kodet inn i en dekoder, vil den for hver modul øke bitraten helt til den når sin endelige bitrate som en ukomprimert videostrøm. Videre vil den gjennomgå skaleringen, som vil senke eller øke bitraten henholdsvis om videoen synker eller øker i oppløsning. I prosessen hvor videostrømmen går fra dekodet til kodet vil bitraten gradvis synke igjen, til den er ferdig encodet.

Ved å anta at brukeren vil skalere en videosnutt som har en oppløsning på 1280 ganger 720 piksler, der den gjennomsnittlige bitraten ligger på 4 Mbit/s, ned til en oppløsning på 960 ganger 540 piksler, har man oppsettet klart for testen. Det som må gjøres er å regne ut bitraten til de ukomprimerte videostrømmene før og etter videoskalereren og hvilken normalisert belastning disse vil representere for systemet.

6.4.3 – Utregning av normalisert belastning for en dekodet videostrøm

Ved å benytte seg av on-the-fly videoskalering, stilles det potensielt store krav til båndbredde. Som sagt tidligere er den tradisjonelle metoden for videoskalering å skalere videoen etter den har gått fra frekvensdomenet til tidsdomenet. Med utgangspunkt i dette kan man kalkulere hvor stor en slik datastrøm vil være og hvilken normalisert belastning dette vil representere i vårt NoC.

For å regne ut hvor stor bitraten på en gitt ukomprimert videostrøm er, må man ha kjennskap til videoens parametre. Disse parametrene er.

- Størrelse, angitt i høyde i piksler * bredde i piksler
- Bitdybden på fargerepresentasjonen
- Antall rammer per sekund

Ved å ta utgangspunkt i videostrømmen som brukeren leverer til skalering, har denne en oppløsning på 1280 ganger 720 piksler. Videre antar vi at vår videostrøm har en representasjon på 8 bit per primærfarge og en bilderate på 30 rammer i sekundet. Dette vil gi en bitrate lik ligningen under.

(9) $$1280 \bullet 720 \bullet 8 \bullet 3 \bullet 30 = 664\text{Mbit} / \text{s}$$

Når denne videoen så skaleres ned til 960 ganger 540 piksler, med antagelsen om at både bitbredde og bilderate er den samme vil denne videostrømmen nå ha en bitrate lik.

(10) $$960 \bullet 540 \bullet 8 \bullet 3 \bullet 30 = 373\text{Mbit} / \text{s}$$

Når man anvender en datastrøm som angir throughput i Mbit/s, så må man ta høyde for at systemets ytelse er avhengig av systemets frekvens. Når systemet kjører på en lavere frekvens under test, er det viktig å representere datastrømmen som en normalisert belastning for å bryte koblingen mellom frekvens og ytelse. Det dette gjør

er effektivt å senke throughputen på datastrømmen som skal testes, slik at den totale belastningen i forhold til systemets ytelse er lik for nettverket under test, som for det opprinnelige nettverket.

For vårt system vil en videostrøm med en throughput på 664Mbit/s representere en normalisert belastning lik.

(11) $$\frac{664\text{Mbit/s}}{1160\text{Mbit/s}} = 57\%$$

Videostrømmen på 373 Mbit/s representerer en normalisert belastning lik.

(12) $$\frac{373\text{Mbit/s}}{1160\text{Mbit/s}} = 32\%$$

Videre vil disse belastningen ved bruk av formelen under gi henholdsvis en delayverdi på 9 og 25 klokkesykler.

(13) $$\text{delay} = \frac{\text{delay}_{\text{SEND}} \bullet \text{bitrate}_{\text{MAX}}}{\text{bitrate}} - \text{delay}_{\text{SEND}}$$

Som tidligere nevnt er

6.4.4 – Testoppsett

I denne testen presenteres det et oppsett som har som hensikt å vise at testmiljøet kan simulere oppførselen til et spesifikt oppsett med høy presisjon. På denne måten kan man bruke testmiljøet til å simulere forskjellige utlegg og detektere eventuelle brudd på båndbredde samt forsinkelse.

I denne testen antar man at kortsiktig forsinkelse av pakker ikke vil ha noen betydning for resultatet, så lenge modulene opprettholder sin gitte bitrate. Videre antas det at bitraten øker lineært i decoderen og synker lineært i encoderen. Det antas også at bitraten på den ferdig encodede videostrømmen har en bitrate på 2 Mbit/s. Dette vil gi resultatene presentert i tabell 6.1, der det angis hvilke bitrater de forskjellige modulene produserer og hvor mange pakker de er forventet å produsere. Formelen som er brukt for å regne ut den lineære økningen/senkningen i bitrate er gitt under.

(14) $$\text{bitrate}_{\text{MIN}} + \frac{\text{bitrate}_{\text{MAX}} + \text{bitrate}_{\text{MIN}}}{\text{length}} \bullet \text{step}$$ f.eks. $$4 + \frac{644-4}{5} \bullet 1 = 132$$

Verdiene for delay er gitt av formel 13

Verdiene for antall forventede pakker er gitt av formelen under.

(15) $\#\text{packages}_{MAX} \bullet \frac{\text{bitrate}}{\text{bitrate}_{MAX}}$, f.eks. $512 \bullet \frac{132}{1160} = 58,3$

Verdiene for antall forventede pakker mht delayverdien er gitt av formelen under.

(16) $\#\text{packages}_{MAX} \bullet \frac{\text{delay}_{SEND}}{\text{delay}_{SEND} + \text{delay}}$, f.eks. $512 \bullet \frac{12}{12+93} = 58,5$

Verdiene for antall forventede pakker totalt er gitt av formelen.

(17) $\text{tot\_package} = \sum_{N=0}^{4} \#\text{packages from input}_N$

Modul	Bitrate	Delay verdi	#Forventede pakker	#Forventede pakker mht delayverdi	#Forventede pakker totalt	#Forventede pakker totalt mht delayverdi
Variable Length Decoder	132	93	58.3	58,5	58,3	58,5
Inverse Scan	260	42	114,8	113,8	173,1	172,3
Inverse AC/DC prediction	388	24	171,3	170,7	286,1	284,5
Inverse Quantization	516	15	227,8	227,6	399,1	398,3
IDCT	664	9	293,1	292,6	520,9	520,2
Scaling	373	25	164,6	166,1	457,7	458,7
DCT	299	35	132,0	130,7	296,6	296,8
Quantization	225	50	99,3	99,1	231,3	229,8
AC/DC Prediction	150	81	66,2	66,1	165,5	165,2
Scan	76	171	33,5	33,6	99,7	99,7
Variable Length Coder*	2	4095	0,9	1,5	36,4	35,1

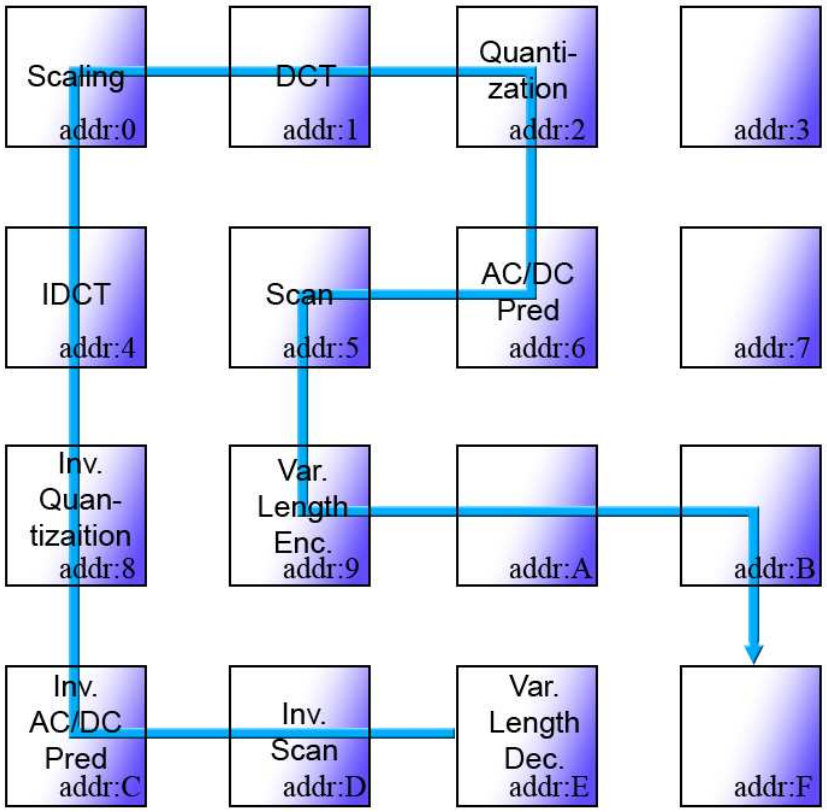
Tabell 6.1 – Verdier for bruk i test av on-the-fly videoskalering

* Siden delayverdien vår har en bitbredde lik 12, med en maksimalverdi på 2^{12}-1=4095, vil den minste datastrømmen trafikkgeneratoren klarer å produsere være lik.

(18) $$\frac{\text{throughput}_{MAX} \bullet \text{delay}_{SEND}}{\text{delay}_{SEND} + \text{delay}_{MAX}} = \frac{1160\text{Mbit/s} \bullet 12}{12 + (2^{12} - 1)} = 3.39\text{Mbit/s}$$

Av denne grunn vil Variable Length Coder modulen produsere en bitstrøm på 3.39 Mbit/s og dermed ha et forventet antall produserte pakker på 1.5 stk per 6144 klokkesykler.

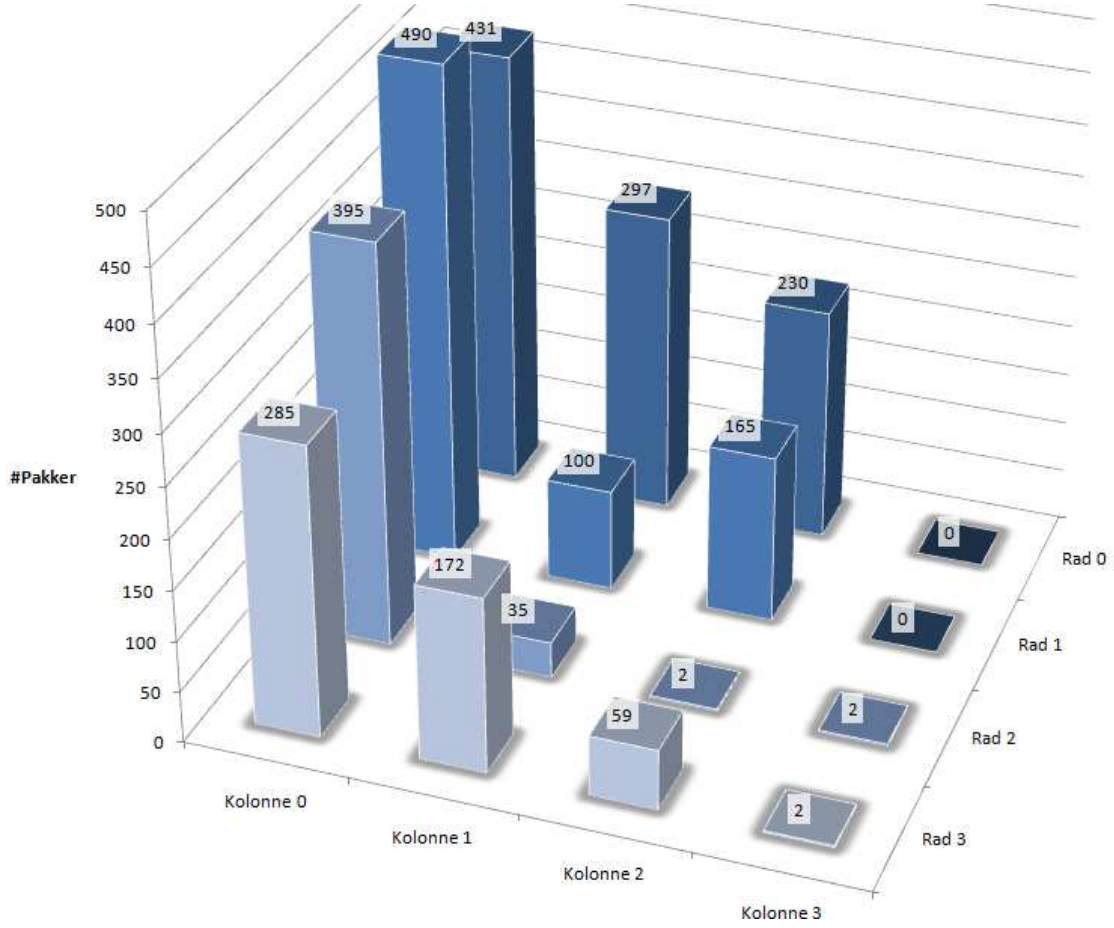
Testen vil benytte seg av alle 11 modulene i et oppsett slik som vist i figuren under. Man ser også at datastrømmen følger mønsteret slik tidligere vist i figur 6.8. Datastrømmen på 8 Mbit/s fra brukeren/programvaren er ikke tatt med i denne testen grunnet oppkoblingen imot grensesnittet på ruter Fs lokale inngang.



Figur 6.9 – Testoppsett for trafikkplanlegging av videoskalerer

6.4.5 – Resultater

Som det observeres av figuren under ser man den forventede økningen i antall leverte pakker fram til datastrømmen når IDCT modulen i Rad 1, Kolonne 0. Deretter avtar antall leverte pakker i henhold til tabell 6.1.



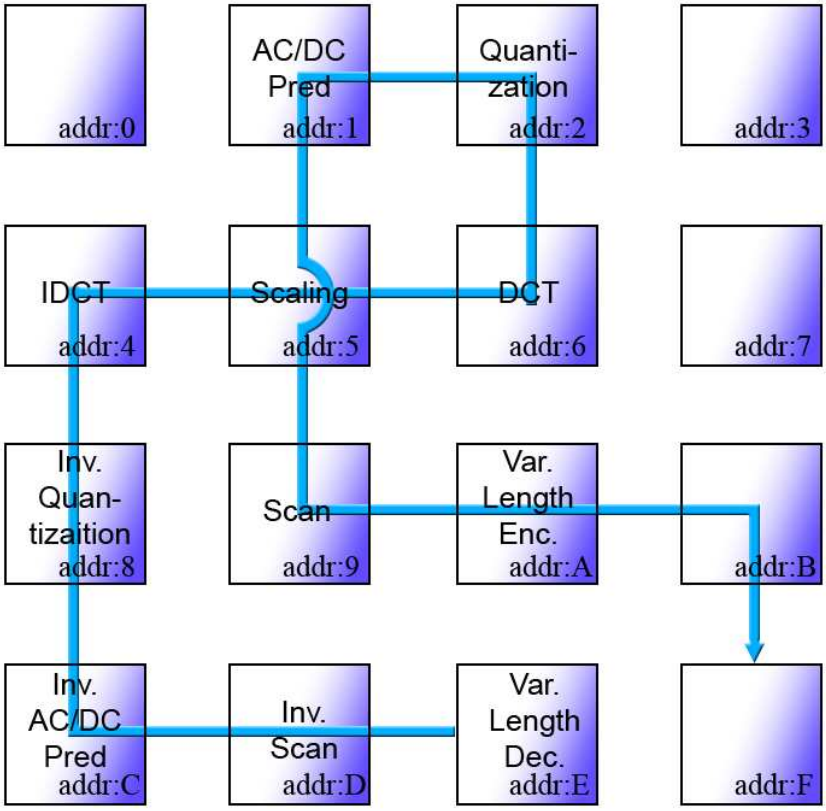
Figur 6.10 – Resultat av test #3

Ved å observere størrelsen på tallene i forhold til modulenes plassering gitt av figur 6.9, samt forventet antall pakker som traverserer ruteren i henhold til tabell 6.1, ser man at tallene stemmer meget bra, med unntak av IDCT og Scaling modulene. Disse to modulene har et avvik på henholdsvis 5,8 % og 6,1 % og det er de to modulene med mest trafikk.

For de andre verdiene er det største avviket på 4,1 pakker (Inverse Quantization), mens det største prosentavviket er gitt av en pakkeforskjell på 0,5 pakker (Trafikken fra Variable Length Encoder), som resulterer i et avvik på 25 %.

6.5 – On-the-fly videoskalering på FPGA med kryssende datavei

Hensikten med denne testen er å se om oppsettet klarer å detektere oppsett som er mindre fordelaktig for traversering av data mellom modulene. Testen benytter seg av de samme modulene som i forrige test, men i denne testen vil datastrømmen krysse seg selv, slik at ruter nummer fem vil oppleve en økt trafikkpågang. Oppsettet kan observeres av figuren under.



Figur 6.11 – Testoppsett for trafikkplanlegging av videoskalerer med kryssende datavei

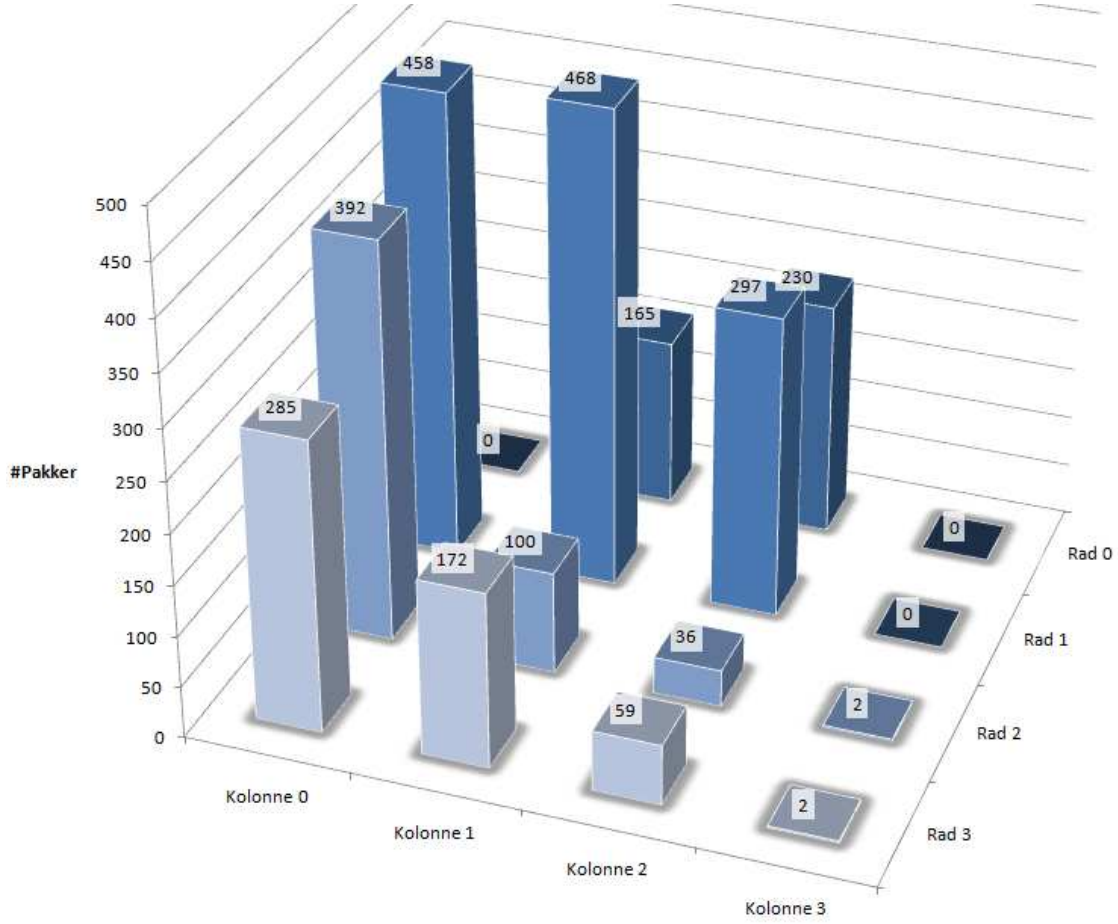
Ved å endre oppsettet for modulplasseringene, vil man også i dette tilfellet endre tabellen som inneholder testverdiene. Denne endringen kommer i form av at man nå må ta høyde for at ruteren som er koblet til Scaling-modulen nå også må behandle trafikken fra AC/DC Prediction-modulen. Dette vil øke antallet forventede pakketraverseringer gjennom Scaling-modulen med 66 pakker, som også kan observeres i tabellen under.

Modul	Bitrate	Delay verdi	#Forventede pakker	#Forventede pakker mht delayverdi	#Forventede pakker totalt	#Forventede pakker totalt mht delayverdi
Variable Length Decoder	132	93	58.3	58,5	58,3	58,5
Inverse Scan	260	42	114,8	113,8	173,1	172,3
Inverse AC/DC prediction	388	24	171,3	170,7	286,1	284,5
Inverse Quantization	516	15	227,8	227,6	399,1	398,3
IDCT	664	9	293,1	292,6	520,9	520,2
Scaling	373	25	164,6	166,1	523,9	524,8
DCT	299	35	132,0	130,7	296,6	296,8
Quantization	225	50	99,3	99,1	231,3	229,8
AC/DC Prediction	150	81	66,2	66,1	165,5	165,2
Scan	76	171	33,5	33,6	99,7	99,7
Variable Length Coder*	2	4095	0,9	1,5	36,4	35,1

Tabell 6.2 – Verdier for bruk i test av on-the-fly videoskalering med kryssende datavei

6.5.1 – Resultater

I denne testen var datastrømmen gitt et mindre fordelaktig utlegg, i forventning om at dette skulle gjenspeiles i resultatet fra utlesningen. Disse resultatene er gjengitt i figuren under.



Figur 6.12 – Resultat av test #4

Som det observeres av resultatene har det i henhold til tabell 6.2 traversert et mindre antall pakker gjennom IDCT og Scaling modulene i forhold til det som er forventet. Dette skaper et avvik på henholdsvis 11,9 % og 10,8 %. Av de andre modulene har man et avvik på Quantization-modulen på 6 pakker, som gir et prosentvis avvik på 1,6 %, og det samme avviket på pakkestrømmen fra Variable Length Encoder-modulen på 0,5 pakker (25 %).

Kapittel 7 – Diskusjon

Behovet for testing er et nødvendig steg i utviklingen og produksjon av nye produkter. Hver test vil være produsert for å avdekke spesifikke feil eller mangler, slik at sluttproduktet vil ha en bedre funksjonalitet og en lavere sannsynlighet for feil. Arbeidet i denne oppgaven har arbeidet imot det samme målet, nettopp å tilby et testmiljø som forenkler videreutviklingen av systemet og avdekker ukjente feil og mangler.

Videreutviklingen forenkles ved at testmiljøet enkelt kan konfigureres og tilbyr verdifull informasjon om hvordan modulenes plassering og antall korrelerer med ytelsen. Ved å benytte seg av denne informasjonen kan man videreutvikle ruteren i søken etter bedre ytelse. Det er også mulig å benytte denne informasjonen i utviklingen og optimaliseringen en styringsmekanisme for plassering av moduler.

Systemet avdekker feil og mangler ved at man enkelt og hurtig kan simulere komplekse trafikkmiljøer. Denne funksjonaliteten har systemet også bevist ved å avdekke situasjonen som fører til vranglås i rutersystemet.

7.1 – Spesifikasjon

En rekke testfasiliteter er kartlagt, evaluert og prioritert før de er implementert i testmiljøet gjennom utviklingen av en rekke nye moduler. Kartleggingen tok for seg alle testfasiliteter som kan være ønskelig i et slikt system. Det vil selvfølgelig være utallige slike fasiliteter, men med tanke på funksjonaliteten testmiljøet har som hensikt å tilby, og arealrestriksjonene gitt av utviklerplattformen, er de mest relevante oppført.

Det største spørsmålet i denne fasen var implementeringen av reaktive trafikkgeneratorer. Det var ønskelig å implementere et system som utførte sine handlinger på bakgrunn av den innkommende trafikkens sendeadresse. På bakgrunn av denne informasjonen kunne trafikkgeneratoren avgjøre hvor mange pakker som skulle produseres, hvor lang tid det skulle være mellom hver pakke og hvem som skulle motta pakkene. Dette ville økt fleksibiliteten til systemet betraktelig, samt at man hadde fått en bedre oversikt over trafikkavhengigheten i systemer som on-the-fly videoskalereren. Det største ankepunktet for dette systemet er arealbegrensningene gitt av utviklerplattformen, da et slikt system ville krevd lagring av mange variabler inne på trafikkgeneratoren. Det ville også ha medført mer kontrollogikk da man må ha benyttet flere enn én pakke per konfigurasjon.

7.2 – Implementering av nye moduler

Det er i denne oppgaven presentert fire nye moduler som gjengitt i listen under.

- Arbiter
- TrafficMonitor (TM)
- TM_control
- TrafficGenerator (TG)

7.2.1 - Arbiter

Det ble presentert en ny arbiter først og fremst grunnet den diskriminerende arbitreringen av den lokale inngangen. For å holde kompleksiteten på arbiteren nede og klokkefrekvensen høy, ble arbiter implementert med en statisk prioritetsrekkefølge og et signal som tar vare på hvilken retning som sist har fått tilgang til ruteren. På denne måten kan arbitreringen gjøres generell og den innehar meget gode egenskaper ved 1 eller 2 samtidige forespørsler.

Ved 3, 4 eller 5 samtidige forespørsler, vil løsningen med statisk prioritet gjøre designet sårbart med hensyn på utsulting. Det positive er at en slik utsultning kun oppleves når den normaliserte belastningen på ruteren er meget høy, samt at for hver pakketransaksjon vil ruteren legge request-signalet sitt lavt, slik at faren for utsultning minskes.

I [2] diskuterte Ivar Ersland rundt arbiters kritiske sti. Det ble sagt at den lengste kritiske stien i designet var ved tre samtidige forespørsler, da dette krevde en mer intrikat logikk grunnet implementeringen av Quality of Service. Når støtten for QoS nå er fjernet, vil alle arbitreringssituasjonene gi det samme bidraget til den kritiske stien. Dette medfører at arbiters klokkefrekvens har økt fra 132 MHz til 187 MHz, noe som betyr at modulen som bestemmer systemets opperasjonsfrekvens nå er kontrolleren, med en frekvens på 164 MHz [2]

En mulig løsning for å gjøre arbiter mer rettferdig er å implementere flere minnesteg. På denne måten vil arbiter ha mer bakgrunnsinformasjon tilgjengelig, slik at logikken implementert i arbiteren kan være mer rettferdig. Problemet ved å implementere flere minnesteg er at man da vil få en liste over de N siste som har fått tilgang til ruteren. I tillegg til mer logikk rundt arbitreringen, får man også ved en slik implementering mer logikk for håndteringen av listen. Denne ekstra logikken vil påvirke arealet og ytelsen til kretsen i en negativ retning, noe som ikke er ønskelig.

Det er også vist i kapittel 5.1.4 at det er mulig å gjøre arbiteren mer rettferdig ved å implementere logikk for å endre den statiske prioritetsrekkefølgen på bakgrunn av hvem som fikk tilgang til ruteren sist. Denne løsningen ble vist å kreve 60 % mer areal enn den implementerte arbiteren. Ved å se nærmere på løsningen, vil det mest

sannsynlig være flere måter å optimalisere koden på, dette grunnet at det ble brukt lite tid til å utvikle akkurat denne arbiteren grunnet dens rolle som et ”proof of concept”.

7.2.2 – TrafficMonitor

Ved utviklingen av denne modulen var spørsmålet hvor finkornet informasjonen som ble samlet inn skulle være? Ved å tenke over hvordan overføringen mellom to rutere gjennomføres, er svaret enklere å se. Når ruteren sender en pakke, vil den ved en vellykket synkronisering mellom ruterne leses over i form av at pakken skiftes ut fra sender og inn på mottager. Under hele denne transaksjonen vil den benyttede båndbredden være hundre prosent. Ved å anta at alle pakketransaksjoner er vellykkede, kan man med kunnskapen om oppbygningen av pakken, anta at ingen informasjon går tapt ved å telle pakker i stedet for flits.

Ved å benytte seg av kontrollogikkens CTS signal, kan man enkelt telle antall pakker som traverserer ruteren. Dette fordi at signalet går høyt uavhengig av hvilken inngang som prøver å sende en pakke til ruteren. For å forstå hvorfor kontrollogikken har denne funksjonaliteten, må man forstå litt om oppbygningen til ruteren.

Ruteren har to tilstandsmaskiner, en for å kontrollere innlesning av pakker og en for å kontrollere utlesning. Tilstandsmaskinen som kontrollerer innlesning av pakker vil kun klare å lese inn én pakke om gangen. Siden alle innlesningene gjennomføres på samme måte, trenger ikke kontrollogikken noe viten om hvem som sendte pakken, den trenger kun informasjon om at en pakke skal leses inn. Denne jobben er arbiterens. Denne modulen avgjør hvem som får tilgang til ruteren til en gitt tid. Når arbiteren har kartlagt dette, gir den informasjon videre til to moduler, kontrolleren og readout. Kontrolleren får informasjon om at en pakke ankommer ruteren, mens readoutmodulen får informasjon om hvilken inngang pakken ankommer, slik at kontrollsignalene kan demuxes til korrekt utgang. Grunnet at denne oppbygningen allerede er på plass var det hensiktsmessig å benytte seg av det globalt distribuerte CTS signalet fra kontrollogikken, noe som forenklet utviklingen av denne modulen betraktelig.

Funksjonaliteten til TM-modulen har en teller som basis. Hver gang en pakke ankommer ruteren, inkrementeres en intern teller, for så å leses ut hver gang TM-control initierer en utlesning. det var også nødvendig å implementere en flankedetektor for CTS signalet grunnet at det holdes høyt gjennom hele pakketransaksjonen.

7.2.3 – TrafficGenerator

Trafikkgeneratoren har som hensikt å tilby muligheten for sluttbruker å simulere et bredt spekter av forskjellig datatrafikk. Dette gjør modulen ved å inneholde logikk som kan konfigureres på bakgrunn av informasjon mottatt gjennom konfigurasjonspakker. Modulen har muligheten for å produsere data med en pseudotilfeldig bitrate, eller med en fast bitrate.

Ved å benytte seg av den pseudotilfeldige bitraten, vil man oppleve at hver utlesning leverer det samme pakkeantallet hver gang, dette er grunnet den periodiske oppførselen ved bruk av LFSR-moduler og at perioden for utlesningen er et multiplum av perioden til LFSR8 modulen. Ved å benytte seg av pseudotilfeldig bitrate i flere trafikkgeneratorer, vil tiden for konfigurasjon, samt plassering og antall skape et unikt trafikkbilde, som rettferdiggjør implementeringen av denne funksjonaliteten.

Hvis man benytter seg av en fast bitrate, har man med denne trafikkgeneratoren mulighet til å produsere datatrafikk som har en normalisert belastning mellom 0,29 % og 100 %. Med systemets throughput gitt av ligning 8, vil dette representere en bitrate mellom 3.39 Mbit/s og 1160 Mbit/s. Hvis det er ønskelig å representere en lavere bitrate i systemet, er det bare å øke bitbredden på delayverdien. Utregningen for den laveste bitraten systemet klarer å gjengi er gitt av ligning 18.

Trafikkgeneratoren inneholder to tilstandsmaskiner som tar seg av innlesning og utlesning av pakker. Denne funksjonaliteten er implementert på bakgrunn av at ruteren ikke inneholder noen spesifikk logikk for behandling av pakker som har nådd sin destinasjon. Dette innebærer at hver lokale modul må selv inneholde denne logikken, for å kunne svare på handtrykksignalene til ruteren. Denne løsningen fører med seg to problemer.

- Hver modul som skal kobles på en ruter, må inneholde logikk for mottak og sending av pakker, noe som fordrer bruken av modulen kun på det spesifikke systemet.
- Hvis en feil oppstår i adresseringen av en pakke, vil denne pakken stå og vente på svar fra en lokal modul som ikke eksisterer, og på denne måten oppta en av bufferne i den gjeldende ruteren.

Ved å føre logikken for overføringen av pakker inn i ruteren, er det enklere å implementere moduler på systemet, samt at nettverket ikke er avhengig av at modulene er korrekt designet med hensyn på pakketransaksjoner for å unngå eventuelle vranglåser. For trafikkgeneratoren ville dette betydd mindre resursforbruk, samt mindre utviklingstid for å få tilstandsmaskinene til å tilpasse seg ruterens kontrollogikk.

Trafikkgeneratoren har også støtte for tidsstempling av pakkene, på denne måten kan man observere tidsforsinkelsen den gitte pakken har gjennom nettverket. Denne informasjonen er meget verdifull for utviklingen av plasseringsalgoritmene som benyttes i et Hardware Operating System, som er den modulen som tar seg av dynamisk rekonfigurering av systemet. Informasjonen kan også benyttes til å oppdage urettferdig arbitrering og hjørnetilfeller som gir unormalt høy tidsforsinkelse.

7.3 – Testing

Hoveddelen av denne oppgaven har vært selve utviklingen av testmiljøet. Med et velfungerende design i ryggen, har testingen vært enkel, hurtig og behagelig å gjennomføre. I oppgaven er det gjennomført totalt fem tester, der hver test hadde som mål å avduke relasjonene mellom kretsens ytelse og modulenes antall og plassering. Underveis ble systemets funksjonalitet bevist ved at en vranglås i ruterdesignet ble oppdaget.

7.3.1 – Test #1

Målet med denne testen var å se hvordan en datastrøm med høy throughput ble påvirket av en kryssende datastrøm med middels throughput. Det ble i denne testen antydet av resultatet, at ruteren vil fordele resursene sine mellom to datastrømmer etter beste evne. Det ble observert at datastrømmen som benyttet seg av 100 % av ruterens resurser, måtte senke resursforbruket sitt ned til 70 % grunnet den kryssende datastrømmen som forespurte 30 % av resursene. Disse resultatene ga utgangspunktet for neste test, der det var ønskelig å se hvordan denne fordelingen ville utspille seg for en datastrøm som kontinuerlig økte resursforbruket sitt.

7.3.2 – Test #2

Denne testen inneholdt som i test #1, to datastrømmer. I denne testen ble en av datastrømmene kjørt på en normalisert belastning på 100 % (datastrøm A), mens den andre steg fra 0 % til 100 % av NB (datastrøm B). Resultatene fra denne testen bekreftet antagelsene fra den første testen, nettopp at ruteren vil fordele resursene sine jevnt over begge inngangene. Etter hvert som resurskravet fra B øker, vil resursene tilgjengelige for A synke helt til begge har allokert halvparten av resursene hver.

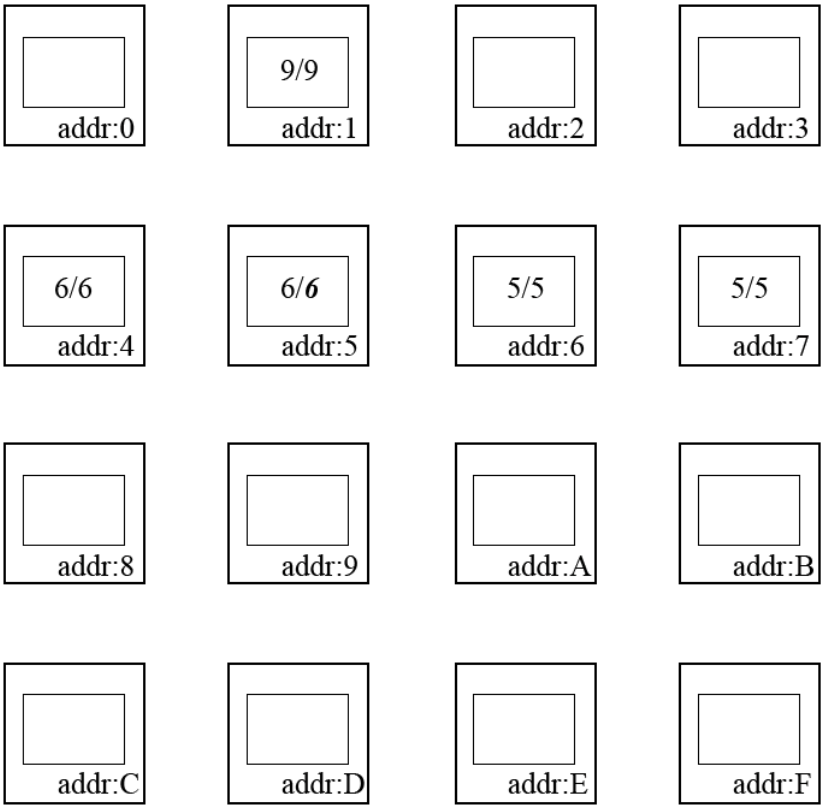
7.3.3 – Test #3 – Vranglås i ruterdesignet

Etter test nummer tre, ble det avdekket flere svakheter i ruterdesignet som ikke har dukket opp under tidligere testing. En av hovedgrunnene til dette er at fram til dette punktet har testingen bestått av å sende pakker *til* ruterne, mens nå genererer lokale trafikkgeneratorer trafikk som sendes *fra* ruterne. Dette resulterer i at det interne trafikkbildet har en høyere kompleksitet.

I figur 7.1 kan man observere ruternettverket der ruter #1,4 og 7 er konfigurert til å sende pakker til henholdsvis ruter #9, 6 og 5. Figuren gjenspeiler adressene på de pakkene som ligger i de interne bufferne i hver ruter. Slik som kontroll-logikken er implementert i ruterne, vil den lese inn pakke nr en i buffer en, så vil tilstandsmaskinen som behandler utgangen observere at det ene bufferet inneholder en pakke og be tilstandsmaskinen på inngangen om å bruke det andre bufferet, for så å lese ut pakken i buffer en. Problemet oppstår når man har et scenario som i figuren under, der ruter #5 inneholder to pakker. En som skal til den lokale modulen og en som skal sendes til ruter #6. Hvis det nå er bufferet som inneholder pakken til ruter #6 som skal leses ut neste gang, vil ruternettverket gå i vranglås. Dette fordi den eneste måten å løse problemet på er å lese ut pakken som skal til den lokale modulen.

Figur 7.1 – Vranglås i ruternettverket

En måte å løse denne vranglåsen på er at ved en sending uten svar, alternerer man pakken som skal leses ut, og dermed leser ut det andre bufferet i stedet. Ved å gjøre dette kommer man ett hakk lengre, men kan da risikere å bli fanget i en ny vranglås som vist i figuren under.



Figur 7.2 – Total vranglås i ruternettverket

Her vil det ikke ha noen innvirkning om utlesningen av pakker alterneres ved sending uten svar, da begge pakkene skal til samme destinasjon.

For å løse dette problemet ble logikken for innlesning av pakker endret. Endringen bestod av at man definerte buffer A for bruk ved pakker som traverserer nord og vestover og buffer B til pakker som traverserer sør og østover. Det er viktig å observere at med denne endringen halverer man den effektive båndbredden grunnet begrensningene i samtidig inn og utlesning. Simulering av denne endringen ga en ny hold-and-wait deadlock. Denne oppstod da arbiter kunne velge å gi grant til en pakke der destinasjonsbufferet var i bruk. Ved å måtte skifte grantsignalet over til retningen som løste problemet oppstod det flere nye problemer.

- Hvordan skal man detektere en vranglås?

- Hvordan skal man vite hvilken retning som skal ha grant for å løse vranglåsen?
- Hvordan skal dette implementeres uten at det påvirker ytelsen til kretsen?

Grunnet dette problemets komplekse oppbygning og tidsbegrensninger, er det ikke gjennomført videre endringer i designet. Det er tatt høyde for muligheten for vranglåser i testingen av systemet.

7.3.3.1 – Endringer i ruterdesignet for å løse vranglåssituasjonen

Network on Chip designet har utviklet seg over tid, da det har kommet inn nye studenter med nye ideer og nye oppgaver. På denne måten vil designet endres for å tilpasse seg de nye funksjonene og at tidsbegrensninger gjør det umulig å levere fra seg en feilfri kildekode. På denne måten vil designet møte på slike humper i veien, der uforutsette vranglåser og bugs kommer til overflaten.

I dette tilfellet dreier det seg om en vranglås som oppstår når tre datastrømmer benytter seg av samme ruter. Med tanke på at normal datatrafikk gjerne opptrer reaktivt og i bursts [6], er det tenkelig at denne oppførselen legger til rette for nettopp dette trafikkscenarioet, som i sin tur kan forårsake en vranglås. Det vil derfor være nødvendig å endre ruterdesignet for å kunne takle disse situasjonene.

Ivar konkluderte i [2] med at Store-and-Forward svitsjing var den mest egnete svitsjeteknikken for å støtte QoS. Ved å se bort ifra QoS, eller akseptere en vanskeligere implementering, vil man kunne se på andre svitsjestrategier slik som virutal-cut-through, wormhole osv. Hver med sine enge fordeler, bakdeler og muligheter for vranglåser. Hvis det eksisterende ruterdesignet er ønskelig å beholde, må det endres slik at det ikke lengre oppstår vranglåser. En mulig løsning er å implementere minst et buffer per retning i ruteren. For vårt design vil dette bety minst fem buffere. På denne måten sikrer man seg at det alltid er ledig plass til å lese inn en pakke, uansett hvilken retning den kommer fra. Ved å øke antall buffere til 5 per ruter, vil man effektivt øke det totale arealforbruket til bufferne med 250 %. Dette vil bety at bufferne i et 4 ganger 4 meshoppsett alene vil bruke ~37.5 % av spartan3 brikkens tilgjengelige areal, mot det nåværende forbruket på ~15 %. Videre krever en slik modifikasjon en utvidelse av kontrollogikken og at man må ta hensyn til at båndbredden halveres.

7.4 – Trafikkplanlegging av on-the-fly videoskalerer

Hensikten med denne testen var å se om systemet kunne benyttes til å trafikkplanlegge en større applikasjon. I dette tilfellet var applikasjonen en videoskalerer. I den første av to tester med dette oppsettet, ble en fordelaktig plassering av modulene gjennomført, før de ble tilordnet en bitrate ut ifra antagelsene gjort i kapittel 6.4.4.

Resultatene fra den første testen viste at to av ruterne ikke taklet den høye båndbredden som ble krevd, med avvik på 5,8 og 6,1 % i antall pakketransaksjoner. Disse resultatene gav noen nye spørsmål med hensyn på ytelsen til ruteren og designet til trafikkgeneratoren.

- Hvorfor et så høyt avvik, når ruteren teoretisk skal klare 512 pakketransaksjoner?
- Ligger feilen i trafikkgeneratorene, eller er det en begrensning i ruteren?

Ved å simulere hele testen i activeHDL, var det mulig å fastslå at de samme resultatene ble gjengitt ved simulering. Dette innebar at det nå var mulig å søke etter svaret på disse spørsmålene ved hjelp av simuleringsdataene. Først ble alle trafikkgeneratorens verdier dobbelsjekket for å se at de stemte overens med spesifikasjonene for testen, noe de gjorde. Videre ble tilstandsmaskinen som styrer produksjonen av pakker gått etter i sømmene, dette ble gjort ved å ta flere stikkprøver av tiden mellom hver inkrementering av send_packet variabelen. For så å se om denne samsvarte med verdien gitt i trafikkgeneratoren. Ved alle stikkprøvene så var resultatet entydig, tellerne ble inkrementert etter spesifikasjon. Simuleringen ble så kjørt helt fram til avbrudd, der alle send_packet variablene ble avlest. Send_packet variabelen til IDCT modulen hadde en verdi på 14 og pakketelleren en verdi på 495, noe som ville gitt et totalt antall pakker på 509. Ved å benytte seg av denne informasjonen reduserer avvikene til 2,1 og 2,4 %, noe som antyder at det er ruteren som er den begrensende faktoren.

For å utforske dette litt videre ble det satt opp en enkel test som hadde to datastrømmer, en på 100 % av NB inn til ruteren og en datastrøm på 100 % NB ut av ruteren. Dette ga resultatet i at den totale pakketraverseringen i ruteren var 534 pakker, noe som er over den teoretisk maksimale på 512. Ut ifra disse resultatene er det ikke mulig å trekke noen konklusjon. Grunnet tidsbegrensninger antas det at designet av trafikkgeneratoren er korrekt, men at det er et komplekst trafikkbilde og mange dataavhengigheter som er årsaken til den lavere ytelsen til ruterne.

De andre avvikene i kretsen er ubetydelige og kan avskrives på flere små feilkilder i designet. Dette vil være slik som oppløsningen på delayverdien og trafikkmonitorenes pakketellermekanisme som kun kan inkrementeres i heltall. Ved å

summere og midle verdiene over flere utlesninger, vil avvikene bli mindre grunnet heltallsforskjellene feilkildene gir mellom hver utlesning.

7.5 – Trafikkplanlegging av on-the-fly videoskalerer med kryssende datavei

For å kunne avgjøre om systemet har muligheten til å skille mellom en god og en mindre god plassering av modulene, ble ikke modulene i denne testen gitt den samme fordelaktige plasseringen. Dette innebar at dataveien til videoskalereren krysset seg selv i henhold til figur 6.11. Resultatene fra denne testen ga både noen forventede og noen uforventede resultater.

Testmiljøet ga et klart svar i sine resultater om at dette oppsettet var mindre fordelaktig enn det forrige, dette ble vist ved at avvikene var større for de to ruterne med mest trafikk, i forhold til den forrige testen. Avvikene var nå 11,9 og 10,8 % i forhold til 5,8 og 6,1 % i forrige test.

Ruter fem i denne testen var ruteren som opplevde kryssende trafikk i henhold til figur 6.11. Siden ingen av datastrømmene hadde en høyere NB enn 57 %, var det ikke forventet noen utsultning av inngangene. Det spesielle med resultatene var at trafikken som kom fra den vestlige inngangen, var den eneste retningen som ikke leverte alle pakkene sine. Dette antyder at det er retningen med den høyeste normaliserte belastningen som er mest utsatt for resursmangel.

Kapittel 8 – Konklusjon

For å kunne kartlegge relasjonene mellom systemets ytelse og modulenes plassering og antall er det utviklet et testmiljø for AHEAD prosjektets Network on Chip. Resultatet er et testmiljø som er skreddersydd for dette nettverket.

Systemet tilbyr gjennom bruken av konfigurasjonspakker, produksjon av trafikk med en pseudotilfeldig eller fast bitrate. Videre tilbyr også systemet overvåkning av tidsforsinkelsen til hver eneste pakke. På denne måten er det mulig å konstruere testoppsett i henhold til ønsket oppførsel, samt at man ved hjelp av trafikkmonitoren kan samle inn data om antall pakketraverseringer og tidsforsinkelsene i systemet.

Ved å bruke dette systemet er det også mulig å trafikkplanlegge større applikasjoner, som krever bruken av flere moduler. Systemet vil som vist i kapittel 6.4 og 6.5 også hjelpe til med å skille mellom modulplasseringer av fordelaktig, og mindre fordelaktig type. Dette gjør at systemet er et yndet verktøy for utviklingen og optimaliseringen av plasseringsalgoritmen for et framtidig tenkt Hardware Operating System og for videreutvikling av rutermodulen for å sikre et vranglåsfritt design.

Utviklingen av testmiljøet er preget av arealbegrensningene i utviklingsplattformen, noe som har medført at alle ønskelige testfasiliteter ikke var realiserbare. Ved å benytte en utviklingsplattform med flere resurser vil både systemets ytelse og funksjonalitet kunne heves.

.

Kapittel 9 – Videre arbeid

Gjennom utviklingen av testmiljøet har det dukket opp flere områder som kan være fordelaktig å se litt nøyere på.

- Løse vranglåssituasjonen som oppstår når ruteren mottar data fra tre retninger samtidig.
- Gjøre ruterdesignet ansvarlig for håndtering av pakker som sendes til og fra de lokale modulene, slik at disse vil være uavhengig av ruterdesignet.
- Se hvorfor ruterdesignet minker i ytelse ved spesielle trafikksituasjoner, slik som i test nummer fire og fem.
- Gjennomføre ytterlige tester for å se om arbiteren yter en rettferdig nok tjeneste.
- Ved større arealresurser på målplattformen er det mulig å implementere flere av de kartlagte testfasilitetene.

Referanser

[1] – A. Hepsø. Ferdigstilling, implementering og testing av NoC for AHEAD prosjektet. NTNU, 2009.

[2] – I. Ersland. Quality of Service for Network on Chip. NTNU, 2009.

[3] – Wikipedia, (2 februar, 2010) Observer Effect (information technology) [Online] URL: http://en.wikipedia.org/wiki/Observer_effect_(information_technology)

[4] – Xilinx, (3 Mars 2010), Spartan-3 FPGA Family Data Sheet [Online] URL: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

[5] – J. Krohn, J. Linnerud. MPEG Transcoder for Xilinx Spartan. NTNU, 2008.

[6] – S. Mahadevan, F. Angiolini m.f. A Network Traffic Generator Model for Fast Network-on-Chip Simulation.