technische universität
dortmund

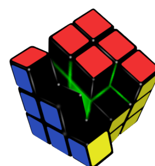
Bachelorarbeit

Entwicklung eines konfigurierbaren NoCs für das LavA-Framework

Mark Breddemann
27. August 2013

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Matthias Meier

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de



Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 27. August 2013

Mark Breddemann

Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Entwurf und der Implementierung eines Network on Chips für das LavA-Projekt, mit dem mehrere auf einem FPGA liegende CPUs Daten austauschen können. Die besondere Aufmerksamkeit gilt der Parallelität sowie der Performanz bei der Übertragung großer Datenmengen, da im LavA-Framework bisher noch keine Kommunikationsstruktur integriert ist, die für diesen Anwendungsfall gut geeignet ist. Nachdem die in LavA bereits vorhandenen Mittel zur Kommunikation zwischen mehreren CPUs analysiert worden sind, werden verschiedene Network on Chip-Strukturen betrachtet. Es wird eingeschätzt ob diese geeignet sind, die Lücke der Anwendungsfälle die bisher mit LavA aufgrund der eingeschränkten Möglichkeiten zur Interprozessorkommunikation nicht realisierbar sind, zu füllen. Im Laufe der Arbeit wird dann ein Crossbar-NoC entworfen, welches in vielerlei Hinsicht konfigurierbar ist und daher speziell für verschiedene Anwendungsfälle maßgeschneidert werden kann. Es besitzt zur Steigerung der Performanz eine direkte Anbindung an den Datenspeicher der CPUs, damit diese sich während der Datenübertragung anderen Aufgaben widmen können. Außerdem kann das NoC dadurch in jedem Takt ein Wort übertragen, wobei die Wortbreite des NoCs konfigurierbar ist. Voneinander unabhängige Sender und Empfänger können ohne Geschwindigkeitseinbußen parallel über ein Netzwerk miteinander kommunizieren.

Es stellt sich heraus, dass sich der Platzverbrauch des entwickelten Network on Chips auf dem FPGA im Rahmen anderer, vergleichbarer Implementierungen bewegt und die Übertragung nicht nur in der Theorie, sondern auch in praktischen Tests sehr performant ist. So gibt es bei einer verteilten Matrixmultiplikation mit 16 CPUs, welche die Daten mittels des NoCs austauschen, einen Speedup von 15,6 gegenüber einer Single-Core Variante. Zudem wird evaluiert, wie sich das Netzwerk unter Last verhält.

Zusammen mit den anderen Kommunikationsstrukturen bietet das LavA-Framework dank des im Rahmen dieser Arbeit entwickelten Network on Chips umfassende Möglichkeiten zum Datenaustausch zwischen mehreren auf einem FPGA existierenden CPUs.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Gliederung der Arbeit 2

2 Grundlagen des LavA-Frameworks **3**
2.1 LavA System on Chip (SoC) 3
2.2 IPC-Bus 4
2.3 Shared Memory 5

3 Network on Chip **7**
3.1 Crossbar 7
3.2 Virtual Channel 9
3.3 Simple Virtual Channel 9
3.4 Routerbasiertes NoC 10
3.5 Geeignete NoC-Konzepte für das LavA-Framework 11

4 Entwurf eines NoCs für das LavA Framework **13**
4.1 Struktur des Crossbar-Netzwerkes 13
4.2 Anbindung an den Speicher mittels Direct Memory Access 14
4.3 Konfigurierbarkeit des NoCs 15
4.4 NoC im LavA-Metamodell 18

5 Implementierung des LavA-NoCs **21**
5.1 Hierarchie der Komponenten 21
5.2 NoC-Controller 21
5.3 NoC Empfangseinheit (RX-Unit) 23
5.4 NoC Sendeeinheit (TX-Unit) 24
5.5 Arbiter 26
5.6 Verschaltung der Komponenten 28

6 Evaluation des Platzverbrauches **29**
6.1 Gesamtverbrauch des NoCs 30
6.2 8 Bit und 32 Bit Busbreite 31
6.3 Platzverbrauch der Arbiter 31
6.4 Sende- und Empfangseinheit 33
6.5 Anzahl der Empfangspuffer 33
6.6 Vergleich mit dem LavA-IPC 34

6.7 Ressourcenmodell 35

7 Evaluation der Performanz **37**

7.1 Theoretischer Durchsatz 37
7.2 Übertragen von Daten 38
7.3 Vergleich mit dem LavA-IPC 40
7.4 Performanz unter Last 41
7.5 Beschleunigung einer Matrixmultiplikation 45

8 Fazit und Ausblick **49**

8.1 Fazit 49
8.2 Ausblick 50

Literaturverzeichnis **52**

Abbildungsverzeichnis **54**

Tabellenverzeichnis **55**

1 Einleitung

Die Möglichkeiten bei der Entwicklung von Eingebetteten Systemen werden immer vielfältiger. Es gelingt durch steigende Integrationsdichten komplexere Systeme zu bauen, die viele Aufgaben übernehmen und dennoch Eigenschaften von Eingebetteten Systemen erfüllen, wie zum Beispiel die Maßschneiderung der Soft- und Hardware an die jeweilige Aufgabe. Das Konzept des System on Chip (SoC), bei dem sich auf einem einzigen Mikrochip nicht nur die CPU, sondern auch viele Peripheriekomponenen befinden, ist heutzutage in vielen Geräten des Alltags zu finden.

Es ist mitlerweile möglich, mehrere dieser Cores auf einem Chip unterzubringen. Ein Core im Multiprocessor System on Chip ist dann speziell an die zu bearbeitende Aufgabe angepasst. Für die allermeisten Szenarien ist es essentiell, dass die Cores miteinander Kommunizieren können. Ein aktuelles Thema in der Forschung stellen daher die dafür geeigneten Kommunikationsstrukturen dar. Klassische Strukturen wie sie bei der Vernetzung von PCs zum Einsatz kommen, eignen sich häufig nicht zur Umsetzung auf Mikrochips, denn für diese Network on Chips (NoC) existieren ganz andere Rahmenbedingungen.

In dieser Arbeit werden verschiedene NoC-Strukturen analysiert, ein ausgewähltes implementiert und dieses dann ausführlich evaluiert. Die Umsetzung des Network on Chips geschieht im Rahmen des LavA-Projektes [1] des Lehrstuhl 12, Arbeitsgruppe ESS der TU-Dortmund.

1.1 Motivation

Das LavA-Framework soll den Entwicklungsprozess von (Multiprocessor) System on Chips vereinfachen und größtenteils automatisieren. Es sollen nur noch die Aufgaben des Systems in Form von C++-Code angegeben werden. Das LavA Projekt soll eine Programmierschnittstelle (API) bereitstellen, die für diese Aufgaben benutzt wird. Eine Design-Space-Exploration wird diese Aufgaben sowie dessen Benutzung der API analysieren und erstellt dann eine oder mehrere angepasste Betriebssystemkonfigurationen des Betriebssystems CiAO [2] sowie maßgeschneiderte Hardware in Form von Hardwarebeschreibung ein System-On-Chip, welches dann auf einem FPGA lauffähig ist.

Werden die gegebenen Aufgaben auf mehrere Cores verteilt, stellt das LavA-Framework zwei verschiedene Kommunikationstrukturen zum Datenaustausch zur Verfügung: Einen Bus zur Interprozessorkommunikation (IPC) sowie einen Shared Memory zur Kommunikation zwischen zwei Cores auf einem SoC. Beide Möglichkeiten haben jeweils verschiedene Schwachpunkte und eingen sich daher nur für bestimmte Anwendungsfälle. Insbesondere der Austausch größerer Datenmengen zwischen beliebigen CPUs gestaltet

sich mit diesen beiden Strukturen sehr umständlich.

Für viele Anwendungsfälle ist es jedoch essentiell, auf umfassende Kommunikationsmöglichkeiten zurückgreifen zu können, insbesondere da das LavA-Framework sehr gute Möglichkeiten bietet, mehrere CPUs auf einem Chip unterzubringen. Um dem Ziel von LavA, Systeme für möglichst viele Anwendungszwecke maßgescheidert erstellen zu können näher zu kommen, wird in dieser Arbeit das Framework um ein Network on Chip erweitert. Es soll möglichst viele Szenarien zur Kommunikation auf einem SoC abdecken und dabei performant hinsichtlich der Kommunikationsgeschwindigkeit sowie minimal bezüglich des Platzverbrauches auf dem FPGA sein.

1.2 Gliederung der Arbeit

Im folgenden Kapitel werden die Grundlagen des LavA-Frameworks erläutert mit besonderem Augenmerk auf die bereits existierenden Kommunikationsstrukturen und deren Schwächen. Anschließend werden in Kapitel 3 verschiedene Konzepte zur Umsetzung eines Network on Chips vorgestellt und es wird analysiert, welches sich davon eignet um die gewünschten Anforderungen zu erfüllen um im LavA-Framework eine Kommunikationsstruktur ohne die zuvor gefundenen Schwächen bereitstellen zu können.

In Kapitel 4 wird dann ein Entwurf für das zu entwickelnde NoC erstellt, insbesondere wird dargestellt, in welchen Aspekten das Netzwerk konfigurierbar ist, um es an verschiedene Anwendungszwecke anzupassen. Dieser Entwurf wird dann implementiert, wobei die groben Details der Implementierung in Kapitel 5 dargestellt werden.

Zur Evaluation wird in Kapitel 6 zunächst ausführlich der Platzverbrauch analysiert, bevor in Kapitel 7 verschiedene Tests zur Performanz des NoCs erläutert und deren Ergebnisse vorgestellt werden.

Abschließend erfolgt in Kapitel 8 eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick auf Erweiterungsmöglichkeiten des entwickelten Network on Chip.

2 Grundlagen des LavA-Frameworks

Im folgenden Abschnitt werden die für das LavA-Framework wichtigsten Begriffe und Aspekte vorgestellt. Anschließend werden die bestehenden Kommunikationsstrukturen erläutert und analysiert, insbesondere um festzustellen, welche Anwendungsfälle mit ihnen nur schlecht realisierbar sind.

2.1 LavA System on Chip (SoC)

Unter dem Begriff "System on Chip" versteht man das Konzept, einen Großteil der Komponenten eines Eingebetteten Systems auf einem einzigen Chip unterzubringen. So gibt es bei einem SoC keine separaten Chips für die CPU, den Speicher sowie Controller für Peripheriegeräte, sondern all diese Funktionen befinden sich auf nur einem Chip. Ein gutes Beispiel für SoCs sind moderne Handys, bei denen unter anderem CPU und GPU, deren Caches sowie Bluetooth, W-LAN, GSM und UTMS-Komponenten auf einem Chip integriert sind. Hierdurch lassen sich die Geräte viel kompakter bauen, da keine Steuerleitungen zwischen vielen verschiedenen Chips notwendig sind. Dies begünstigt wiederum die Geschwindigkeit und den Energieverbrauch, was insbesondere bei batteriebetriebenen Systemen wichtig ist. Möglich wird dies vor allem durch die hohen Integrationsdichten, die heutzutage in der Mikrochipentwicklung möglich sind.

Die LavA-Toolchain ist in der Lage, ein solches System on Chip zusammenzustellen. Es stehen verschiedene CPUs zur Auswahl, beispielsweise der MB-Lite+ (TUMBL) Prozessor [3], ein freier Xilinx-Microblaze Klon oder die ZPU. Hinzu kommen verschiedene Peripheriekomponenten, die über den Wishbone-Bus [4] mit der CPU verbunden sind. Zur Verfügung stehen zum Beispiel Timer, UART, CAN-Controller oder I^2C-Controller. Ein FPGA kann dann mit diesem generierten System konfiguriert werden.

Moderne FPGAs stellen inzwischen so viele Ressourcen bereit, dass es möglich ist, mehrere Cores bestehend aus einer CPU mit Peripheriekomponenten inklusive Bus und Speicheranbindung auf einem einzigen FPGA unterzubringen. LavA ist in der Lage ein solches Multiprocessor-SoC zu erstellen. Die einzelnen Cores arbeiten zunächst komplett unabhängig voneinander und sie können so unterschiedliche Aufgaben erledigen. Schnell gelangt man jedoch an dem Punkt, an dem es sinvoll oder nötig ist, dass die einzelnen CPUs miteinander kommunizieren können. LavA besitzt zu diesem Zweck schon zwei Kommunikationsstrukturen, die im Folgenden erläutert werden.

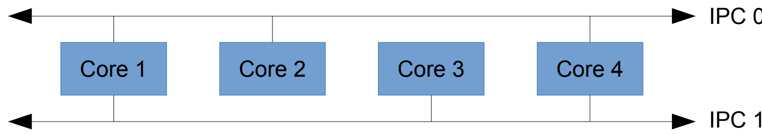


Abbildung 2.1: Beispielhafte Topologie des LavA IPC-Busses

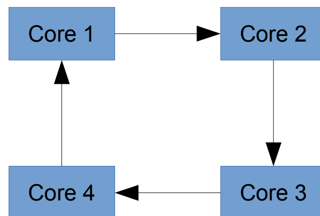


Abbildung 2.2: Topologie des LavA IPC-Bus als Ring

2.2 IPC-Bus

Der in LavA vorhandene IPC-Bus ist einer von zwei zur Zeit zur Verfügung stehenden Möglichkeiten zur Kommunikation zwischen den generierten CPUs. Er ist konfigurierbar und jeder Core kann an einen oder mehrere IPC angeschlossen werden. Dementsprechend können beliebige CPUs miteinander über diesen Nachrichtenkanal kommunizieren. Eine beispielhafte Topologie ist in Abbildung 2.1 dargestellt.

Die Eigenschaften des Busses sind variabel. So kann man beispielsweise die Datenbreite einstellen, wobei das Maximum bei 32 Bit liegt. Zudem ist es möglich eine beliebige Anzahl an Puffern anzugeben, die zum Senden und Empfangen genutzt werden. Es gibt eine Kollisionserkennung, die bei einem erkanntem Konflikt die Anfragen nach der Priorität der zu sendenden Cores abarbeitet. Der Bus kann auch mit einer Ring-Struktur konfiguriert werden, wie in Abbildung 2.2 zu sehen. Es wird keine Kollisionserkennung benötigt und auf dem Bus kann parallel gesendet werden. Allerdings kann es je nach Aufbau des Rings passieren, dass eine Nachricht alle Cores (außer dem Sender und Empfänger) passiert und an diesen weitergeleitet werden muss, bis sie ihr Ziel erreicht. In Abbildung 2.2 geschieht dies unter anderem, wenn eine Nachricht von Core 1 an Core 4 geschickt wird.

Der IPC-Bus hat leider mehrere Nachteile, egal in welcher Konfiguration. Einer besteht darin, dass der Sender alle Daten über den Core-Internen Wishbone-Bus, welcher die CPU mit der Peripherie verbindet, kopieren muss. Auf Empfangsseite muss die CPU ebenfalls über den Wishbone-Bus alle Daten abholen. Außerdem kann es je nach Anwendungsszenario passieren, dass die Empfangs-CPU für jedes empfangene Paket die gesamte Interrupt-Behandlung abarbeiten muss. Es muss daher insgesamt sehr viel Rechenzeit für die Kommunikation mittels des IPC-Busses aufgewendet werden. Ein anderer Nachteil ist die Tatsache, dass auf dem Bus nicht parallel gesendet werden kann. Bei vielen Teilnehmern und hoher Last kann es daher passieren, dass Cores mit geringer Priorität aushungern. Wird der IPC in der Ring-Struktur angelegt, so ist parallele

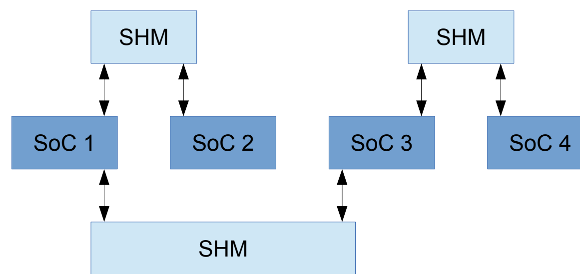


Abbildung 2.3: Beispielhafte Topologie eines SoC mit Shared Memory

Kommunikation zwar möglich, jedoch steigt möglicherweise die Latenz der Daten, wenn die Nachrichten erst mehrmals weitergeleitet werden müssen.

Ein weiterer Aspekt des Busses sind dessen Puffer. Die Anzahl ist variabel, da die Hardwareimplementierung jedoch aus reinen Logik-Schaltungen besteht, verbrauchen sie relativ viel Platz auf dem FPGA. Zudem ist es je nach Anwendung unvorteilhaft, dass die IPC-Bus Controller nicht blockieren, wenn die Puffer voll sind und nicht auf einfache Art festgestellt werden kann, ob alle Daten korrekt übertragen worden sind. Die Daten in den vollen Puffern würden in dem Fall überschrieben. Eine Überprüfung ob dieser Fall eingetreten ist ist nur durch eine in Software impementierte Analyse der empfangenden Daten möglich.

Der IPC-Bus ist daher gut geeignet um kleine Datenmengen zu Übertragen, wobei es vertretbar ist, dass Teile der Daten verloren gehen können. Ein solches Szenario wäre zum Beispiel das Übertragen von kleinen Sensorwerten, die ein Core ermittelt und zur Weiterverarbeitung an einen anderen überträgt. Da in der Regel der nur aktuellste Sensorwert relevant ist, ist es vertretbar wenn ältere Daten überschrieben werden.

2.3 Shared Memory

Der existierende Shared Memory in LavA ist ein Speicher, der direkt im Adressraum der CPU zweier Cores liegt. Dies hat den Vorteil, dass die CPU direkt mit den Daten in diesem Speicher rechnen kann und diese dann zur Übertragung an einen anderen Core nicht erst, wie beim IPC, kopiert werden müssen. Dem Empfänger der Daten muss lediglich mitgeteilt werden, dass die Daten im Shared Memory nun abholbereit sind. Dies kann beispielsweise durch eine IPC-Nachricht geschehen, welche beim Empfänger einen Interrupt auslöst oder von einer Variable im Shared Memory, die von der Ziel-CPU gepollt wird. Der Empfänger kann dann Daten ohne sie erst zu kopieren weiterverarbeiten. Als Speicher werden auf dem FPGA Block-RAM-Zellen verwendet. Diese können relativ viele Daten aufnehmen, ohne dass viel Platz auf dem programmierbaren Teil des FPGAs verbraucht wird. Zwar existieren auf einem FPGA nur eine begrenzte Anzahl dieser Block-RAM-Zellen, aber sie benötigen anders als die Puffer des IPC-Busses lediglich etwas FPGA-Logik zur Ansteuerung der Speicherzellen.

Der Shared Memory eignet sich also gut zur Übertragung großer Datenmengen, lediglich die Synchronisation zwischen Sender und Empfänger muss in Software imple-

mentiert werden. Der große Nachteil des Shared Memorys ist jedoch, dass dieser nur zwei Cores miteinander verbinden kann. Für reine zwei-Core Systeme ist dieser Shared Memory daher vollkommen ausreichend, bei mehreren Cores eignet sich diese Kommunikationsstruktur wie der IPC nur für bestimmte Anwendungsfälle. Als Beispiel sei hier ein Szenario genannt, bei dem die Daten zur Weiterverarbeitung von einem Core zum nächsten gesendet werden, vergleichbar mit einer Pipeline. Hier wird jeder Core an insgesamt zwei Shared Memorys angeschlossen. Bei dieser Struktur ist es allerdings nicht möglich, dass sie Cores Daten mit beliebigen anderen Cores austauschen.

3 Network on Chip

Im vorherigen Kapitel wurde festgestellt, dass die in LavA vorhandenen Kommunikationsstukturen nicht dafür geeignet sind, performant große Datenmengen zwischen beliebigen Cores auszutauschen. Daher wird in diesem Kapitel analysiert, welche Möglichkeiten es gibt um diese Lücke der Kommunikationsmöglichkeiten die LavA anbietet zu füllen. Dazu wird das in der heutigen Microchipentwicklung moderne Konzept des "Network on Chip" aufgegriffen.

Der Begriff Network on Chip beschreibt zunächst nur die Idee, mehrere Knoten innerhalb eines Chips mittels eines Netzwerkes zu verbinden. Es gibt keine konkrete Vorschrift oder Topologie, wie dieses Netzwerk aussehen muss. Auch ist nicht festgelegt, dass sich in den Knoten des Netzwerkes komplette Cores mit CPU und Peripherie befinden. Hinter den Knoten kann sich auch eine einfache Schaltung verbergen, die spezielle Aufgaben erledigt und das Ergebnis wieder zurücksendet. Es ist denkbar, dass ein NoC wie das Ethernet-Netzwerk aufgebaut ist, inklusive Routern, in denen die Daten weiterverwendet werden. Aber auch einfache Topologien, bei denen jeder Knoten mit jedem anderen verbunden ist, ist denkbar. Diese Topologie wäre sogar mit dem existierendem IPC-Bus oder Shared Memory umsetzbar. Dies würde jedoch bezüglich des Platzverbrauches auf dem FPGA sehr schlecht skalieren. Für jeden IPC-Bus müsste es in jedem Core einen eigenen IPC-Controller mit seinen teuren Puffern geben. Außerdem existiert weiterhin das Performanzproblem aufgrund des Kopierens der Daten beim Sender und Empfänger. Beim Shared Memory würde für jede Verbindung mindestens eine Blockram-Zelle verbraucht. Die Blockram-Zellen werden jedoch auch für den Hauptspeicher der CPUs benötigt und werden daher bei sehr vielen Shared Memory-Verbindungen schnell knapp.

Aus diesem Grund ist eine komplette Neuentwicklung einer Kommunikationsstruktur notwendig, damit die CPUs des LavA-Projektes mittels eines NoCs effizient auch größere Datenmengen ohne Einschränkungen untereinander austauschen können. Im folgenden Abschnitt werden nun verschiedene Topologien analysiert und deren Vor- und Nachteile ermittelt, um so eine geeignete Variante zu finden, welche die gewünschten Anforderungen hinsichtlich Performanz und Platzverbrauch erfüllt.

3.1 Crossbar

Ein Crossbar-Netzwerk ist eine sehr einfache Network on Chip Topologie. Die Struktur ist in Abbildung 3.1 dargestellt. Auf der linken Seite finden sich die Sendeeinheiten der Knoten und oben die Empfänger. Die Kommunikation findet zunächst vom Sender ausgehend nur in horizontaler Richtung statt, bis man sich vertikal auf einer Linie mit dem gewünschten Empfänger befindet. Nun erfolgt der weitere Kommunikationsweg in ver-

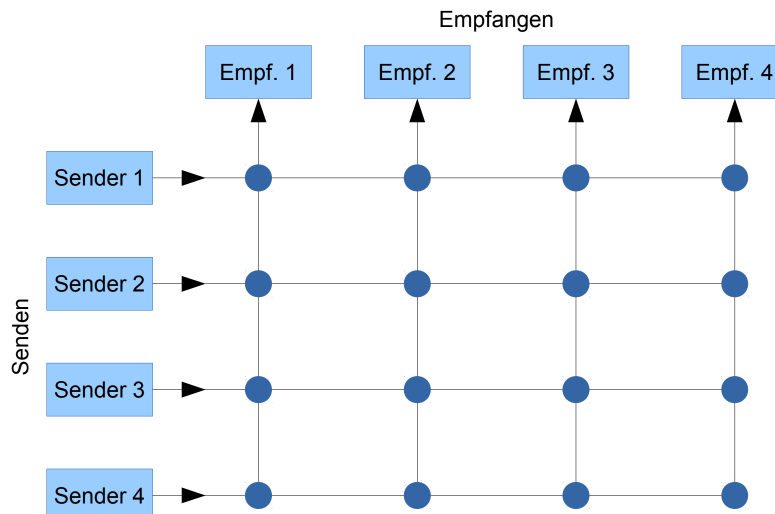


Abbildung 3.1: Struktur eines Crossbar NoCs. Die Kommunikation findet zunächst stets in X-Richtung statt, dann in Y-Richtung. Zwei Kommunikationswege können sich in den Kreisen kreuzen, ohne sich gegenseitig zu blockieren.

tikaler Richtung. Andere Wege sind nicht nicht möglich, es wird auf dem gesamten Weg genau ein mal "links abgebogen". Parallele Kommunikation zwischen vier voneinander unabhängigen Knoten (zwei Sender und zwei Empfänger) ist möglich, da sich die Kommunikationswege in den Kreuzungen überschneiden können. Lediglich das gleichzeitige Empfangen von zwei Sendern durch einen Empfänger ist nicht möglich. Das Senden an mehrere Empfänger ist hingegen denkbar, um beispielsweise einen Broadcast umzusetzen. Die Daten werden dann in den Kreuzungen sowohl in horizontale sowie in vertikale Richtung weitergeleitet.

Bafumba-Lokilo et al. haben in [5] ein solches Konzept eines NoCs entwickelt. Es wird mithilfe der Hardware-Beschreibungssprache CASM [6] beschrieben und implementiert ein Crossbar-NoC mit konfigurierbarer Bitbreite und flexibler Anzahl an Knoten, die über dieses Netzwerk kommunizieren können. Zur Konfliktvermeidung wird das Round-Robin Scheduling-Verfahren benutzt. Dieses NoC benötigt in einer Konfiguration mit 8 Sendern sowie 8 Empfängern und 32 Bit Breite 4956 *6-input LUTs*[1], was 7,1% der LUTs eines Xilinx XC5VLX110T FPGAs [7] entspricht, welches auch von LavA unterstützt wird. Bafumba-Lokilo et al. fassen zusammen, dass ein Crossbar-Netzwerk verglichen mit Punkt-zu-Punkt Verbindungen oder einem Bus eine einfache sowie effiziente Möglichkeit ist, um Knoten performant miteinander zu verbinden.

[1]Lookup-Table, eine zentrale Komponente eines FPGAs

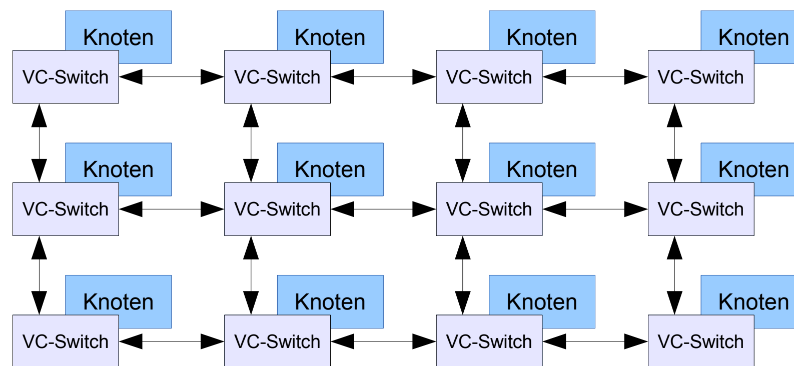


Abbildung 3.2: Struktur eines Virtual-Channel NoCs. Jeder Knoten ist mit einem Switch verbunden, der wiederum mit anderen Switches verbunden ist und so das Netzwerk aufspannt.

3.2 Virtual Channel

Ein komplexeres aber auch flexibleres Konzept ist das Virtual-Channel NoC. Ein beispielhaftes Schema ist in Abbildung 3.2 zu sehen. Bei diesem Netzwerk verschalten die Switches Direktverbindungen (Kanäle) zwischen den Knoten. Ist ein Kanal einmal aufgebaut, können die beiden Endknoten ohne das Konflikte auftreten miteinander kommunizieren, bis dieser wieder abgebaut wird. Die Kanäle müssen nicht auf dem kürzesten Weg geschaltet werden sondern können auch ”Umwege” enthalten. Anhand einer Lastanalyse ist es dadurch möglich, eventuelle Engpässe in dem Netzwerk zu erkennen und zu umgehen. Die zweidimensionale Struktur wie in Abbildung 3.2 kann erweitert werden, sodass die Knoten beispielsweise in einem dreidimensionalen Gitter angeordnet sind. Schelle und Grunwald haben in [8] das Konzept der Virtual-Channel NoCs analysiert. Für eine intelligente Lastverteilung ist es nötig, dass die Status aller Switche bekannt sind, um einen geeigneten Weg zu finden, der eventuelle Lasten umgeht. Eine solche Lastverteilung ist jedoch schwer auf einem FPGA umzusetzen, da dies zu viel Platz auf dem Chip verbrauchen würde. Eine Virtual-Channel Implementierung von Schelle und Grunwald mit 16 Knoten würde 104% der LUTs auf einem Xilinx XC5VLX50 FPGA verbrauchen, doch die Ressourcen des FPGAs reichen nicht aus, um dieses NoC zu verwirklichen. Ein von ihnen entwickeltes Crossbar NoC verbraucht lediglich 18% der LUTs auf demselben FPGA. Abschließend stellen die Autoren fest, dass das Simple NoC teilweise performanter ist. Komplexere Virtual-Channel NoCs mit einer guten Lastverteilung, wie zum Beispiel das von Mullins et al. in [9] vorgestellte, wurde lediglich für eine Simulationsumgebung entwickelt.

3.3 Simple Virtual Channel

Das Konzept des Virtual Channel NoCs lässt sich stark vereinfachen, sodass es dem Crossbar-Netzwerk sehr nahe kommt bzw. einen erweiterten Bus darstellt. Wie in Ab-

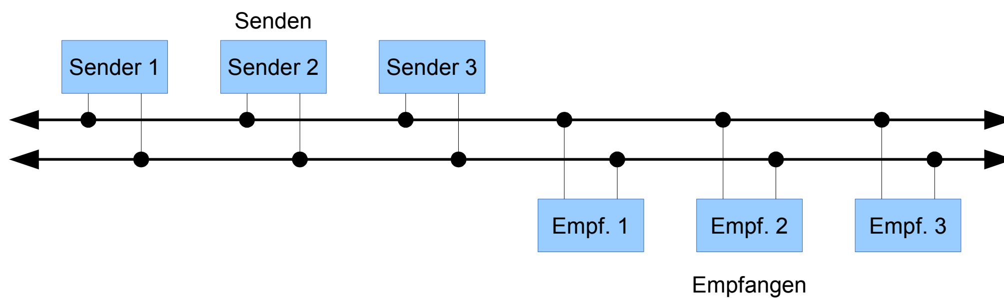


Abbildung 3.3: Struktur eines sehr einfachen Virtual-Channel NoCs. Es stehen dabei weniger Kommunikationswege zur Verfügung als Knoten im NoC vorhanden sind.

bildung 3.3 ersichtlich, gibt es im Gegensatz zum Crossbar nicht für jeden Empfänger bzw. Sender eine eigene Kommunikationleitung. Möchte ein Sender Daten zu einem Empfänger senden, wird eine der zur Verfügung stehenden Leitungen benutzt um eine Direktverbindung zwischen den beiden Teilnehmern zu erstellen. Dazu ist es lediglich notwendig, den Status der wenigen Kommunikationsleitungen zu überwachen und beim Verbindungsaufbau eine Leitung auszuwählen und damit Sender und Empfänger zu verbinden.

Es stehen in diesem Simplen Virtual Channel weniger Datenverbindungen zur Verfügung als bei dem Crossbar-Konzept. Die Performanz bei hoher Last ist dadurch sicher geringer im Vergleich zum Crossbar. Es ist zudem unklar, ob eine Simple Virtual Channel Implementierung überhaupt weniger FPGA-Fläche verbrauchen würde. Die Arbitrierung[2] benötigt eventuell mehr Logik als ein Crossbar, da die Zuteilung der Datenleitungen zu den Knoten dynamisch erfolgt und nicht jeder Sender und Empfänger seine ”eigene” Leitung besitzt, die lediglich an einer Stelle verbunden werden müssen.

3.4 Routerbasiertes NoC

Neben dem Schalten von virtuellen Kanälen durch die zur Verfügung stehenden Datenleitungen gibt es das Konzept von gerouteten Netzwerken, wie beispielsweise beim Ethernet-Netzwerk. Der Aufbau dieses NoCs in Abbildung 3.4 ähnelt sehr dem des Virtual Channel NoC. Der einzige Unterschied ist, dass die Virtual Channel-Switches durch Router ersetzt worden sind. Die Daten werden vom Sender in Pakete verpackt, die dann zu einem Router geschickt werden. Dieser entscheidet dann, an welchen Router dieses Paket weitergeleitet wird. Hier ist es wie beim Virtual Channel theoretisch möglich, Engpässe mit hoher Last zu umgehen. Es existiert aber auch hier das Problem, dass für eine optimale Entscheidung sehr viel Logik nötig ist, sodass dies in einer Hardwareimplementierung kaum umsetzbar ist.

Sethuraman et al. haben in [10] ein solches routerbasiertes NoC implementiert. Die

[2] Zuteilen der Datenleitungen beim Verbindungsaufbau

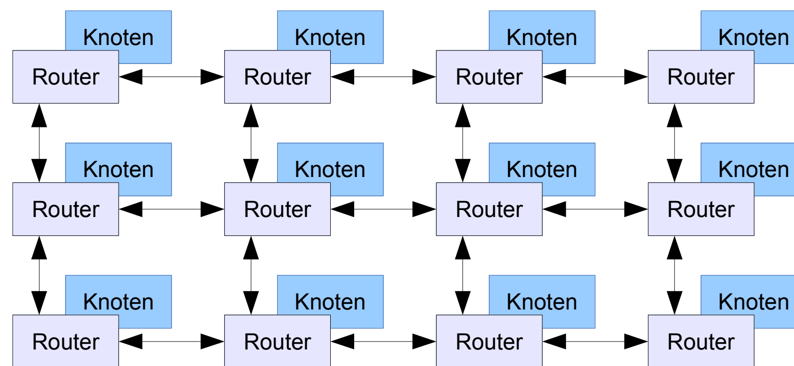


Abbildung 3.4: Struktur eines routerbasierten NoCs. Die Daten werden als Pakete gebündelt und zwischen den Routern weitergeleitet.

Routing-Strategie wurde jedoch sehr stark vereinfacht, denn es gibt nur ein reines XY-Routing, vergleichbar mit dem Crossbar-NoC. Konflikte werden per Round-Robin gelöst. Architekturbedingt unterstützt die Implementation maximal 16 Knoten, wobei der Verbrauch der Slices[3] auf einem Xilinx XC2VP30 FPGA bereits bei 9 Knoten 28% beträgt. Zudem wird ein Paket mit 16 Bit Daten in jedem Router um 10 Takte verzögert, sodass es bei mehreren Hops lange dauern kann, bis das Paket seinen Empfänger erreicht.

3.5 Geeignete NoC-Konzepte für das LavA-Framework

Aus den getätigten Recherchen verschiedener Möglichkeiten der Umsetzung eines NoCs muss entschieden werden, welche Art sich am besten für die Implementierung eines NoCs für das LavA-Framework eignet. Um die Lücken zu füllen, die IPC und der Shared Memory hinterlassen, sind die gewünschten Eigenschaften insbesondere eine gute Performanz bei der Übertragung größerer Datenmengen sowie ein geringer Platzbedarf auf dem FPGA. Das sehr schnelle Verschicken von sehr kleinen Daten kann vernachlässigt werden, da sich hierfür der in LavA bereits vorhandene IPC eignet.

- Crossbar: Das Crossbar Schema eignet sich gut für die Umsetzung. Es erfordert relativ wenig Logik, was dem Platzbedarf und potentiell auch der Performanz zugute kommt, da die Daten nicht wie in einem gerouteten Netzwerk weitergeleitet werden müssen sondern direkt am Empfänger ankommen. Trotz der vielen Direktverbindungen braucht dieses Schema wie beschrieben stets weniger Ressourcen auf einem FPGA als andere NoC-Konzepte.
- Virtual Channel: Die Wahl eines optimalen Weges der Verbindungen durch das NoC ist auf einem FPGA nicht oder nur unter sehr hohem Platzverbrauch möglich, sodass nur vereinfachte Implementierungen möglich sind, wie beispielsweise das folgende Simple Virtual Channel.

[3]Komponente auf einem FPGA, in dem unter anderem mehrere LUTs gebündelt sind

- Simple Virtual Channel: Dieses Konzept benötigt weniger Datenverbindungen als das Crossbar NoC. Da es keine Erfahrungswerte mit diesem Konzept gibt, ist unklar, ob es hinsichtlich des Platzverbrauches besser skaliert als das Crossbar. Zudem ist es nicht möglich, denselben Grad der Parallelität wie beim Crossbar zu erreichen, sodass die Performanz schlechter ist.

- Routerbasiertes NoC: Dieses Konzept eignet sich eher für kleine Datenmengen, weil die Pakete in den Routern zwischengespeichert werden müssen und der Speicher der Router begrenzt ist. Große Datenmengen müssen daher aufgeteilt werden. Die erforderliche Weiterleitung in den Routern kann zudem zu einer großen Latenz der Pakete führen, wenn viele Router zwischen dem Sender und Empfänger liegen. Außerdem erfordern auch einfache Implementierungen relativ viel Logik, sodass viel Chipfläche verbraucht wird.

Für die Entwicklung eines Network on Chip für das LavA Framework ist das Crossbar-Konzept sicher geeignet. Das Simple Virtual Channel könnte ebenfalls geeignet sein, wenn die Anforderung an die Parallelität nicht so hoch ist und dadurch Chipfläche gespart werden kann. Es ist jedoch nicht klar, ob der Platzverbrauch des Simple Virtual Channels wirklich geringer als beim Crossbar ist. Die anderen Konzepte haben verschiedene große Nachteile, die das Crossbar-Netzwerk nicht besitzt.

Um sicher zu stellen, dass das im Rahmen dieser Arbeit entwickelte NoC für LavA die gewünschten Anforderungen erfüllt, wird für die Entwicklung auf das Crossbar Konzept zurückgegriffen, da es potentiell sehr performant ist und dabei wenige FPGA-Ressourcen verbraucht.

4 Entwurf eines NoCs für das LavA Framework

In diesem Kapitel wird der Entwurf eines Network on Chip, welches die gewünschten Anforderungen erfüllt und dessen Integration in das LavA Framework erläutert. Dazu wird zunächst auf die Umsetzung des NoCs und die Anbindung an die existierende LavA-Hardware eingegangen, mit besonderer Beachtung der schnellen Anbindung an die Cores, damit große Datenmengen performant übertragen werden können. Anschließend wird erläutert, welche Konfigurationsaspekte dieses NoC beherrschen soll und wie es in den Hardware-Generierungsprozess von LavA integriert wird.

4.1 Struktur des Crossbar-Netzwerkes

In der von LavA erstellten Hardware kommuniziert die CPU mittels des Wishbone-Busses mit jedem Peripheriegerät. Von diesem Konzept soll auch bei dem Entwurf des NoCs nicht abgewichen werden. Daher wird ein Controller als Bindeglied zwischen der CPU und dessen Wishbone-Bus sowie dem eigentlichen NoC eingeführt.

In Abbildung 4.1 ist ein grober Entwurf für das Crossbar Netzwerk zu sehen. Es gibt vier Teilnehmer an diesem Netzwerk (Core 1-4), wobei jeder Teilnehmer seine eigene, exklusive Sendeleitung besitzt. Soll nun eine Übertragung stattfinden, so muss der Multiplexer am Eingang des Empfängers so eingestellt werden, dass der Empfänger auf der Leitung des Senders "lauscht". Ist diese Verbindung hergestellt, können die Daten übertragen werden. Für diese Aushandlung werden sogenannte Arbiter eingesetzt. Die Arbiter kommunizieren mit den Cores im Netzwerk und stellen die Verbindung her. Sie behandeln auch den Konflikt, wenn mehrere Sender gleichzeitig zu einem Empfänger senden möchten. Nach dem Senden wird diese Direktverbindung wieder abgebaut, sodass andere Übertragungen stattfinden können.

Ein Rückkanal vom Empfänger zum Sender ist nicht vorgesehen, sodass der Empfänger nicht in der Lage ist, dem Sender mitzuteilen, dass dieser gerade nicht empfangsbereit ist. Je nach Konfiguration des NoCs blockiert der Arbiter dann den Sender, bis der Empfänger wieder bereit ist, oder der Sender sendet die Daten zum nicht empfangsbereiten Empfänger, der die Daten dann verwirft.

Die Daten im NoC werden als Pakete versendet, die eine bestimmte Länge haben. Um zusätzliche Logik und dementsprechend Chipfläche zu sparen, werden die Pakete immer als Ganzes versendet, es findet also keine Verdrängung statt, wenn die Arbiter einmal die Verbindung aufgebaut haben. Da der Empfänger erfahren muss, von wem

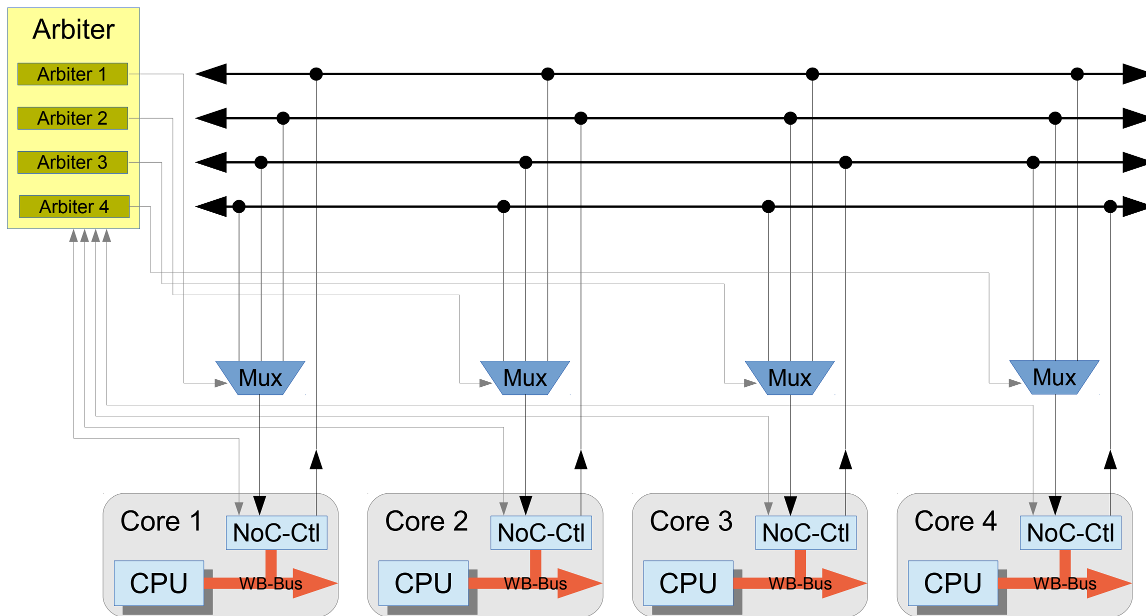


Abbildung 4.1: Konzept einer Implementierung des Crossbar-NoCs für LavA.

ein Paket stammt und wie lang es ist, wird bei der Übertragung ein Header den Daten vorangestellt, in dem diese Information kodiert ist.

Die Übertragung der Pakete findet in Wörtern statt, die so groß wie die Busbreite sind, wobei in jedem Takt ein Wort übertragen wird. Die Performanz des NoCs ist daher stark von der gewählten Wort- beziehungsweise Busbreite abhängig.

4.2 Anbindung an den Speicher mittels Direct Memory Access

Wie bereits in Kapitel 2.2 erwähnt, besteht ein großer Nachteil des existierenden IPC-Busses darin, dass alle Daten vom Sender über den Wishbone-Bus zur IPC-Hardware hin und beim Empfänger wieder zurück kopiert werden müssen. Dies ist aufgrund vieler Sprünge im Programmablauf der CPU relativ langsam und die CPU ist währenddessen komplett ausgelastet. Der Ansatz des LavA Shared-Memory eignet sich daher insbesondere für große Datenmengen viel besser, denn der geteilte Speicher befindet sich im Adressbereich der CPU, sodass die Daten direkt auf diesem bearbeitet werden können und sofort dem Kommunikationspartner zur Verfügung stehen.

Dieses Konzept des Direct Memory Access soll auch auf das LavA NoC übertragen werden. Eine Möglichkeit wäre es, Blockram-Zellen als Puffer zu benutzen, die im Adressbereich der CPU liegen. Dies würde natürlich zusätzliche Blockrams verbrauchen, jedoch sind diese auch ohne Verwendung als NoC-Puffer häufig der begrenzende Faktor, wie viele CPUs auf einem FPGA untergebracht werden können. Zudem ist die Datenmenge,

die ohne weitere Kopieraktionen der CPU mit dem NoC versendet werden kann, auf die Größe des Puffers beschränkt.

Besser wäre es daher, wenn der NoC-Controller die Daten direkt aus dem Speicher der CPU liest. Dies ist mittels Dualport-Blockrams möglich, die es erlauben, gleichzeitig von zwei verschiedenen Anschlüssen auf denselben Speicher lesend oder schreibend zuzugreifen. Manche von LavA unterstützte CPU-Architekturen benutzen solche Dualport-Blockrams jedoch bereits, um den Instruktions- und Datenspeicher in einem Speicherblock unterzubringen. Da jedoch Geschwindigkeit sowie Stabilität bei der CPU-Auswahl eine große Rolle spielen, wird bei Benutzung des LavA-Frameworks häufig der TUMBL-Prozessor gewählt. Dieser benutzt einen getrennten Instruktions- und Datenspeicher, die jeweils in normalen (Singleport) Blockrams untergebracht sind. Der Datenspeicher des TUMBL-Prozessors kann daher umgebaut und erweitert werden, sodass auch der NoC-Controller auf diesen zugreifen kann. Dann ist nur ein Austausch von Metainformationen zwischen der CPU und dem Controller über den Wishbone Bus notwendig, es müssen aber keine Daten aktiv von der CPU über diesen Bus kopiert werden.

Der Controller speichert lediglich die Adressen der Speicherbereiche, von denen er die Daten aus dem RAM lesen soll oder die der Controller als Empfangspuffer nutzen kann. Zudem ist die Speicherung der Längenangaben der Puffer wichtig, damit keine Speicherbereiche die über den reservierten Platz hinausgehen überschrieben werden. Außerdem muss sich der Controller merken, ob der Puffer frei oder belegt ist.

Ein Nachteil dieser Methode ist, dass eine CPU mit nur einem NoC verbunden werden kann, da sonst mehr Ports an den Speicherzellen benötigt würden. Ein einziges Network on Chip bietet bereits eine allgemeine und umfassende Möglichkeit, parallel zwischen beliebigen CPUs Daten auszutauschen, sodass die Anbindung einer CPU an mehrere auf dem Chip existierenden NoCs nicht sinnvoll ist.

4.3 Konfigurierbarkeit des NoCs

Nun steht bereits der grobe Aufbau des NoCs fest. Bevor es implementiert werden kann, muss noch festgelegt werden, in welchen Aspekten das Network on Chip konfigurierbar sein soll. Abbildung 4.2 zeigt das Merkmaldiagramm des NoCs, welches nun erläutert wird.

- **Busbreite:** Zunächst einmal bietet sich eine maximale Busbreite von 32 Bit an, da die Daten in 32 Bit Wörtern aus dem Datenspeicher gelesen werden. Um Chipfläche zu sparen, soll es möglich sein auch geringere Breiten anzugeben, damit die Multiplexer welche die Datensignale der Teilnehmer verschalten kleiner ausfallen. Hier bieten sich sowohl 8 als auch 16 Bit Breite an, da dies ganzzahlige Teiler von der Wortbreite des Speichers sind. Allerdings benötigen Implementierungen mit weniger als 32 Bit Breite etwas mehr Steuerlogik in den Controllern, da für den Bytezugriff auf den Speicher zusätzliche Logik und Zähler benötigt werden. Daher beschränkt sich die Implementierung auf 32 Bit sowie 8 Bit Breite.

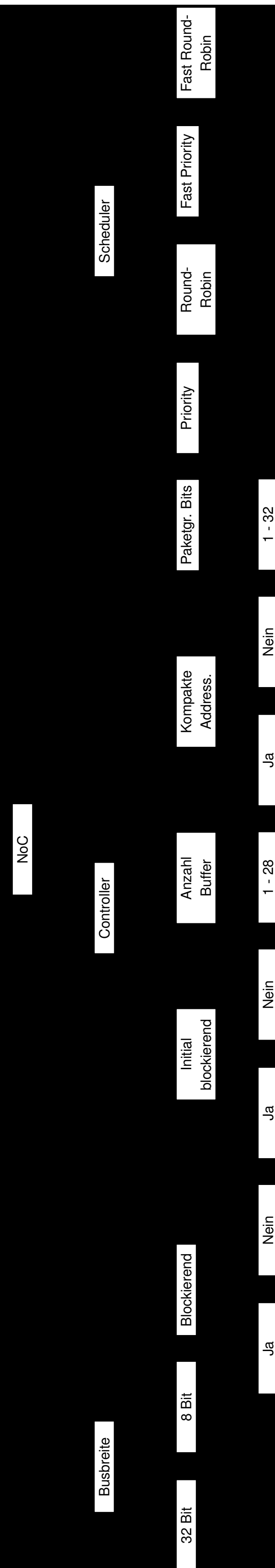


Abbildung 4.2: Merkmaldiagramm des NoCs

- **Anzahl Empfangspuffer:** Je nach Szenario ist es sinnvoll, mehrere Empfangspuffer bereitzustellen. Wenn zwei Sender gleichzeitig zu einem Empfänger senden, muss der zweite Sender warten, bis im Empfänger ein neuer Empfangspuffer bereit steht. Wenn die CPU von vornherein mehrere Puffer festlegen kann, so kann nach einer Übertragung direkt der nächste Puffer befüllt werden, ohne dass zusätzlich darauf gewartet werden muss, dass die CPU beispielsweise in der Interrupt-Routine einen neuen Empfangspuffer bereitstellt oder den bestehenden als "frei" markiert. Außerdem können beide eingegangenen Datenpakete bei rechtzeitiger Ankunft in einer gemeinsamen Interrupt-Routine behandelt werden und die CPU spart sich so den Overhead des mehrmaligen Sprungs in die Routine.
- **Blockierend:** Der Nutzer des NoCs soll zur Entwicklungszeit wählen können, ob das NoC blockiert, wenn die Empfangspuffer belegt sind. Werden beispielsweise Sensordaten zur Weiterverarbeitung durch das NoC gesendet, so ist häufig nur der aktuellste Wert relevant. Ältere, nicht verarbeitete Werte in den Puffern können dann überschrieben werden. Andererseits ist es für viele Anwendungszwecke wichtig, dass alle Daten vollständig den Empfänger erreichen. Dann muss der Sender so lange blockieren, bis beim Empfänger freie Puffer vorhanden sind.
- **Initial blockierend:** Diese Einstellung legt fest, wie sich das NoC verhält, wenn beim Empfänger nach dem Startup des SoCs noch keine Empfangspuffer gesetzt worden sind. Dieser Zustand unterscheidet sich von den typischen Zuständen "frei" oder "belegt" eines Puffers, da er zu diesem Zeitpunkt quasi nicht vorhanden ist. Ist das NoC initial blockierend, dann wartet der Sender so lange, bis Empfangspuffer bereit stehen, ansonsten werden die Daten des Senders verworfen.
- **Arbiter Scheduler:** Der Arbiter soll das Round-Robin sowie Priority-Verfahren zur Konfliktlösung ermöglichen. Diese beiden Verfahren werden auf zwei verschiedene Arten implementiert, einmal als eine schnelle Variante, die in maximal zwei Takten den nächsten Sender ermittelt sowie als platzsparende Variante, die aber in der Regel langsamer ist.
- **Paketgrößen-Bits:** Die Anzahl der Bits die zur Kodierung der Paketgröße dienen, soll variabel sein. Werden stets nur kleine Pakete versendet, so kann die Anzahl der Bits gering gehalten werden. Da in der Hardware mehrere Zähler erforderlich sind, die die Paketgröße mitzählen, kann auf diese Art Logik und Chipfläche gespart werden. Sollen dagegen sehr große Pakete übertragen werden, müssen die Zähler entsprechend viele Bits besitzen. Die maximale Paketgröße bei gegebener Bitzahl ist $2^{Paketgr.-Bits} * \frac{Busbreite}{8} - 1$ Bytes.
- **Kompakte Adressierung:** Bevor die Daten eines Paketes übertragen werden, muss der Empfänger erfahren, wer der Sender ist und wie groß das Paket ist. Diese Informationen können gemeinsam in einem Takt übertragen werden oder in zwei getrennten Takten. Ein gemeinsamer Takt hat neben einem kleinen Geschwindigkeitsvorteil die Eigenschaft, dass etwas weniger Logik zur Implementierung benötigt wird. Allerdings tritt bei kompakter Adressierung insbesondere bei 8 Bit

Busbreite das Problem auf, dass nur eine geringe Anzahl Absender sowie nur eine sehr kleine Paketgröße codiert werden kann. Bei 20 Cores werden beispielsweise 5 Bits benötigt, um den Sender zu kodieren, sodass bei 8 Bit Breite lediglich 3 Bits zur Kodierung der Paketlänge übrig bleiben, was maximal 7 Bytes Daten entspricht. Daher gibt es die Möglichkeit, diese Daten in zwei verschiedenen Takten zu übertragen, sodass ein 8 Bit NoC eine Paketlänge von maximal 255 Bytes unterstützt (bei 32 Bit $2^{32} - 1$ Bytes).

- **Speicheradressierungs-Bits:** Ähnlich zu der Anzahl der Paketgrößen-Bits gibt dieser Konfigurationspunkt an, wie viele Bits zur Adressierung des Datenspeichers benutzt werden. Dieser Wert wird im LavA-Framework jedoch nicht vom Benutzer gesetzt, sondern aus der Größe des Datenspeichers der CPUs abgeleitet. Es werden stets so viele Bits benutzt, dass der gesamte zur Verfügung stehende Speicherbereich adressiert werden kann.

4.4 NoC im LavA-Metamodell

LavA erstellt die Hardwarebeschreibung mittels eines Modells des Eclipse Modelling Frameworks (EMF), in dem alle möglichen Konfigurationen der Hardware beschrieben sind. In dieses Modell muss nun auch das neu entwickelte Network on Chip integriert werden. In Abbildung 4.3 ist ein Ausschnitt dieses Metamodells zu sehen, welches bereits um die Elemente für die Beschreibung des NoCs erweitert wurde. Im Folgenden wird nun die Erweiterung des Metamodells erläutert.

Jedes Peripheriegerät welches an die CPU über den Wishbone-Bus angebunden wird erbt im LavA Metamodell vom *AbstractDevice*. Da das NoC zur Kommunikation mit der CPU ebenfalls den Wishbone-Bus benutzen soll, wird ein Element für den NoC-Controller (*NocCtrl*) eingeführt, welcher die Schnittstelle zwischen der CPU, dessen Peripheriebus und dem NoC darstellt. Dieses Element erbt dementsprechend auch von *AbstractDevice* und übernimmt damit das Konzept der LavA-Peripheriegeräte. Ein Core kann dem Metamodell nach zunächst einmal beliebig viele NoC-Controller besitzen, aufgrund der Speicheranbindung über Dualport-Blockrams ist es jedoch nicht möglich, dass eine CPU an mehr als einem NoC angeschlossen ist.

Da das Network on Chip Core-übergreifend im gesamten Hardwaresystem vorhanden ist, wird im Metamodell ein allgemeines Element für das NoC an der Komponente *MPSoC* angefügt, an der auch die einzelnen Cores angebunden sind. Die *MPSoC*-Komponente kann beliebig viele NoCs enthalten, sodass auf einem Chip auch mehrere voneinander unabhängige Netzwerke existieren können. In dem Element für das NoC wird die Konfiguration des Netzwerkes gespeichert und eine ID vergeben, welche jede Instanz des NoCs eindeutig beschreibt.

Im Unterschied zum bereits existierendem IPC ist die Eingliederung des NoCs in das Metamodell etwas einfacher gehalten, obwohl beide eine Kommunikationsmöglichkeit zwischen mehreren Cores darstellen. Der Controller für den IPC (*IPC*) ist ebenso wie der NoC-Contoller ein *AbstractDevice* und genau gleich an das *SOC*-Element an-

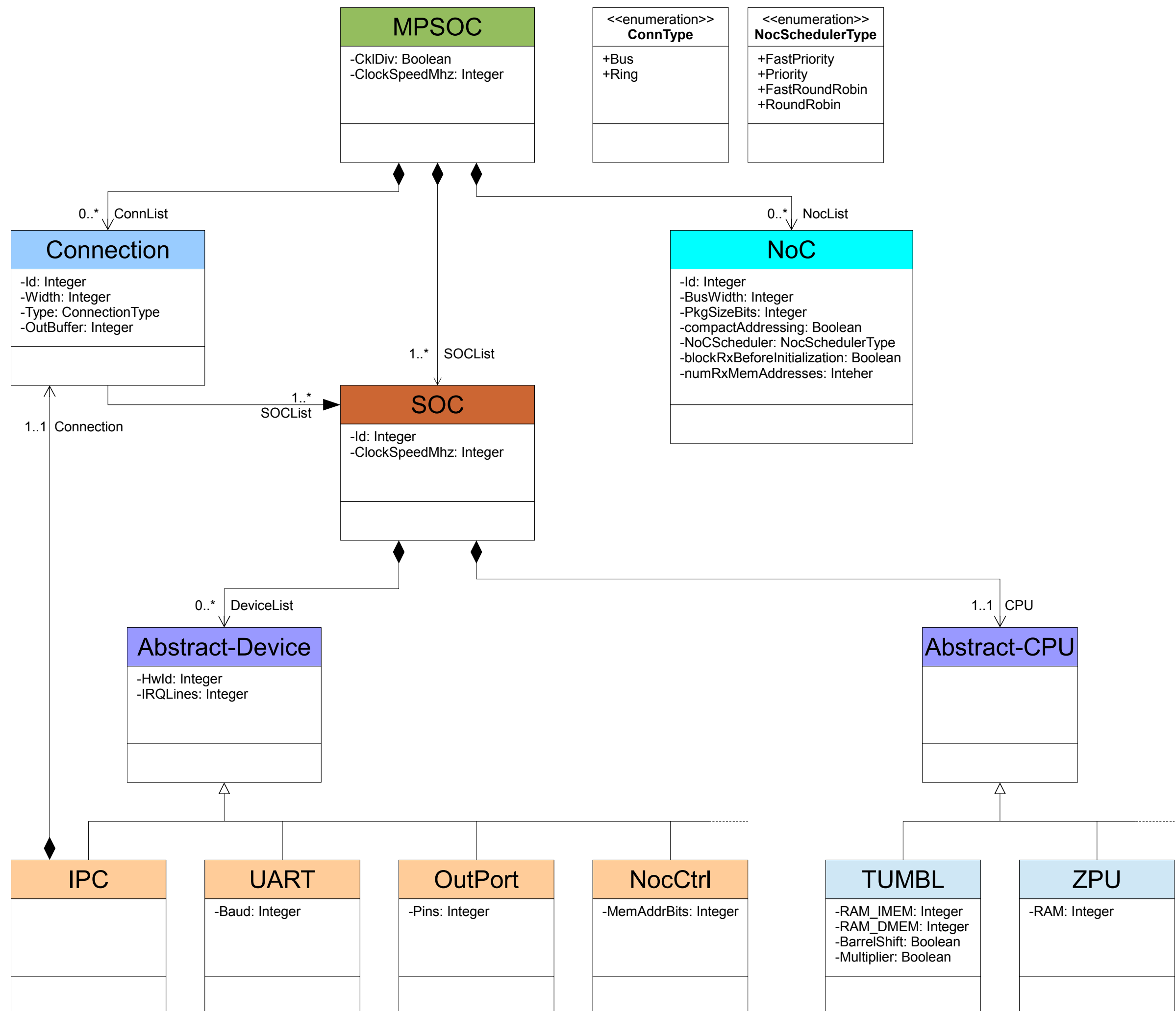


Abbildung 4.3: Auschnitt aus dem LavA Metamodell, aus dem die Hardwarekonfiguration erstellt wird. Darstellung übernommen von [11], aktualisiert/modifiziert und um die Elemente des NoCs erweitert.

gebunden. Das *MPSoC*-weite Äquivalent zum *NoC* ist beim IPC die *Connection*. Der IPC-Controller kennt genau eine Instanz einer *Connection*, während der NoC-Controller keine Referenz auf das *NoC* kennt. Die Zuordnung geschieht hier über die *HwId* des *NocCtrl*. Diese *HwId* beschreibt, zu welcher Instanz des *NoC* der Controller gehört. Ein NoC-Controller mit der *HwId* 1 ist also stets dem NoC mit der ID 1 zugeordnet, ein Controller mit der *HwId* 2 dem NoC mit der ID 2 usw. Eine alternative Zuordnung ist mit dem existierenden IPC möglich, sodass ein IPC-Controller mit der ID 2 auch der *Connection* mit der ID 4 zugewiesen werden kann. Dies ist jedoch in der Praxis eher verwirrend und es entsteht dadurch kein Vorteil, sodass auf diese beliebige Zuordnung bei dem Entwurf des Network on Chip verzichtet wird.

Außerdem besitzt die IPC-*Connection* eine Liste mit den an diesem Kommunikationskanal angeschlossenen Cores (*SOCList*). Auch hier wird auf ein entsprechendes Äquivalent beim NoC verzichtet. Um bei der Hardware-Generierung zu ermitteln, welche Cores an ein NoC angeschlossen sind, wird eine Liste der *SoC*s erstellt, die ein *NocCtrl* mit der entsprechenden *HwId* haben. Dies stellt mit der im EMF enthaltenden Generierungssprache Xpand kein Problem dar. Dadurch wird vermieden, dass für die Generierung der Hardwarebeschreibung aus dem Metamodell heraus redundante Informationen für das NoC angegeben werden müssen.

5 Implementierung des LavA-NoCs

LavA erstellt zur Hardwarebeschreibung aus dem Metamodell mithilfe des Eclipse Modeling Frameworks VHDL[1]-Code. Dazu wird auf die Generierungssprache Xpand [12] und deren Erweiterungsschnittstelle Xtend zurückgegriffen. Xpand ist in der Lage, abhängig von den gewünschten Features unterschiedlichen VHDL-Code zu erzeugen. Dadurch wird zugleich das Problem gelöst, dass es umständlich ist, alleine mit VHDL konfigurierbare Hardware zu erzeugen. Auf eine genauere Betrachtung dieser Aspekte wird hier nicht eingegangen, dies ist bereits in Kapitel 3 in [13] beschrieben und kann dort nachgelesen werden. Details zur Codegenerierung mittels Xpand/Xtend finden sich in derselben Arbeit in Kapitel 6.2.

Im Folgenden wird nun auf die Implementierung der einzelnen Komponenten des NoCs eingegangen.

5.1 Hierarchie der Komponenten

In Abbildung 5.1 ist zu sehen, wie das Network on Chip in die Hierarchie der generierten LavA Hardware eingefügt wird. Gut erkennbar ist die Unterscheidung zwischen den Komponenten Arbiter (*arbiter*) sowie NoC-Controller (*noc_0_ctrl*), wobei der Controller nochmals zwei Unterkomponenten enthält, eine zum Empfangen (*noc_0_rx_unit*) sowie eine zum Senden (*noc_0_tx_unit*). Die Multiplexer werden in die *mpsoc_top* Komponenten integriert und der Datenspeicher (*dmem4_dp*) wurde zu einem Dualport Speicher erweitert. Die existierenden Komponenten *soc_top* sowie *wb_core* wurden lediglich um die für das NoC benötigten Signale erweitert, wobei diese Signale nur durch die Komponenten durchgeleitet werden, ohne dass sie dort in irgendeiner Art benutzt werden.

5.2 NoC-Controller

Der NoC-Controller verbindet die CPU, den Speicher und das eigentliche Netzwerk. Die Wishbone-Schnittstelle für die Anbindung der CPU stellt insgesamt zehn Register zur Verfügung, auf die die CPU entweder lesend oder schreibend zugreifen kann. Eine genaue Beschreibung der Register ist in Tabelle 5.1 zu finden.

Außerdem beherbergt der Controller die benötigten Informationen für die gesetzte Anzahl von Empfangspuffern in einem Array. Die Whishbone-Kommunikation bezieht sich immer auf genau einen Empfangspuffer. Welcher dies ist, kann mittels des Registers

[1] Very High Speed Integrated Circuit Hardware Description Language

Offset	Bezeichnung	lesen/schr.	Beschreibung
0x00	revc_addr	schreiben	Setzt die erste Speicheradresse des aktuellen Empfangspuffers, unter der eingehende Daten gespeichert werden. Nach diesem Zugriff wird dieser Puffer als "frei" markiert und vom NoC beschrieben
0x04	recv_max_size	schreiben	Setzt die Länge des freien Speichers des aktuellen Empfangspuffers.
0x08	–	–	Reserviert
0x0C	recv_last_sender	lesen	Liefert die CPU-ID des Absenders der Daten im aktuellen Empfangspuffer.
0x10	revc_length_sould	lesen	Die vom Absender gegebene Länge des Paketes
0x14	recv_length_is	lesen	Die Empfangene Paketlänge ($\neq$ revc_length_sould bei Puffer-Overflow)
0x18	send_addr	schreiben	Setzt die Speicheradresse, an der die zu sendenden Daten beginnen
0x1C	send_length	schreiben	Setzt die Länge des zu sendenden Paketes
0x20	send_target	schreiben	Setzt den Empfänger des zu sendenden Paketes und beginnt mit dem Senden
0x24	status	lesend	Bit 0 = 1, wenn TX-Interrupt aktiv Bit 1 = 1, wenn RX-Interrupt aktiv Bit 2 = 1, wenn gesendet wird Bit 3 = 1, wenn empfangen wird Bit 4...31 = 1, wenn Empfangspuffer 1...28 beschrieben worden sind Zugriff auf dieses Register löscht eventuelle Interrupts.
0x28	set_recv_buffer	schreibend	Setzt den aktiven Empfangspuffer, auf den sich die Wishbone-Zugriffe beziehen

Tabelle 5.1: Beschreibung der Hardwareregister des NoC-Controllers

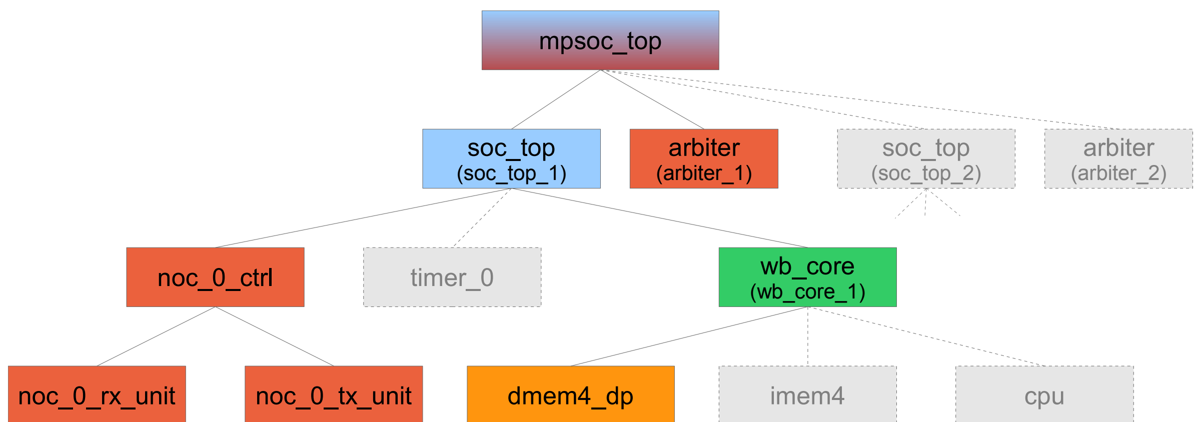


Abbildung 5.1: Hierarchische Darstellung der für das NoC relevanten Komponenten. Die zu implementierenden Komponenten sind rot hinterlegt. Sofern die Instanznamen von den Komponentennamen abweichen, sind die Instanznamen in Klammern angegeben.

set_recv_buffer gesetzt werden. Werden Pakete empfangen, so werden die Empfangspuffer reihum gefüllt. Trifft die Empfangseinheit auf einen bereits beschriebenen Puffer, so wird gewartet, bis dieser wieder als "frei" markiert wird (mittels Wishbone-Zugriff auf das Register *revc_addr*). Es werden von der Empfangseinheit keine belegten Puffer übersprungen, sodass die dafür notwendige Logik gespart werden kann. Zudem ist es ohnehin sinnvoll, dass belegte Empfangspuffer schnell von der CPU neu gesetzt werden, um das NoC nicht auszubremsen. Wurde ein Paket komplett empfangen oder gesendet, löst der Controller einen Interrupt in der CPU aus.

Die Signale zum NoC wurden so ausgewählt, dass es zu einem späteren Zeitpunkt möglich ist, neben dem Crossbar eine andere NoC-Struktur an den Controller anzubinden. Der Controller mit seinen Empfangs- und Sendeeinheiten sowie die Speicheranbindung und -verwaltung kann dann wiederverwendet werden, sodass lediglich die Logik zur Herstellung der Direktverbindungen neu entwickelt werden muss.

5.3 NoC Empfangseinheit (RX-Unit)

Die Empfangseinheit des NoCs besteht im wesentlichen aus einer State-Machine, wie in Abbildung 5.2 gezeigt wird.

Zunächst wartet die RX-Einheit, dass ein freier Empfangspuffer zur Verfügung steht. Signalisiert der NoC-Controller, dass dies der Fall ist, wechselt der Empfänger in den Status *Idle*. Wenn nicht alle Bits der Eingangsleitung "0" sind, signalisiert dies, dass ein eingehendes Paket empfangen werden soll. Die ersten Daten, die empfangen werden, kodieren bei kompakter Adressierung den Absender sowie die Länge des Paketes. Werden diese Daten getrennt übertragen, besitzt die RX-Einheit einen weiteren Zustand zwischen *Idle* und *Receive*, um die getrennt voneinander gesendeten Daten zu verarbeiten. Nun werden die eingehenden Daten so lange in den Speicher geschrieben, bis entweder

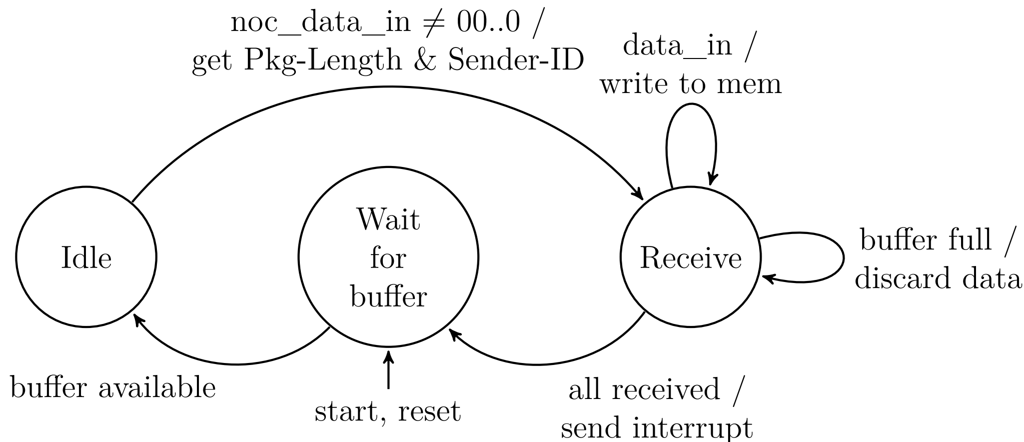


Abbildung 5.2: State-Machine der RX-Einheit mit kompakter Adressierung. Dies ist eine vereinfachte Darstellung, insbesondere sind die Zustandsübergänge der besseren Übersicht wegen nur qualitativ angegeben und sehr stark vereinfacht.

alles empfangen wurde oder das Ende des Puffers erreicht ist. Ist der Puffer voll, werden die Daten zwar weiterhin empfangen, dann aber direkt verworfen und nicht in den Speicher geschrieben. Anschließend wechselt der Empfänger wieder in den Zustand *Wait for buffer* und löst dabei den RX-Interrupt aus.

5.4 NoC Sendeeinheit (TX-Unit)

Auch die Sendeeinheit des NoCs besteht aus einer State-Machine. Sie ist in Abbildung 5.3 zu sehen. Wird nichts gesendet, befindet sich der Sender im Zustand *idle*. Wird nun aufgrund des Zugriffs auf das *send_target* Register im NoC-Controller das Senden gestartet, so wird im Status *Wait for Ctl 1* zunächst darauf gewartet, bis eventuelle Empfangsvorgänge abgeschlossen sind. Da das NoC nicht zur selben Zeit mehrfach auf den Datenspeicher zugreifen kann, ist paralleles Senden und Empfangen eines Cores nicht möglich.

Wird gerade nichts empfangen, wird der Controller als "sendend" markiert und vom Zustand *Wait for Ctrl 1* zu *Wait for Ctl 2* übergegangen. Da alle Einheiten im NoC getaktet sind, kann es passieren, dass zum selben Zeitpunkt zu dem der Controller als "sendend" markiert wird, Daten zur Empfangseinheit geschickt werden sollen und daher die Empfangseinheit in den Zustand "Receiving" wechselt. Dies wird im Zustand *Wait for Ctl 2* festgestellt und bei Auftreten dieses Falles wird dem Empfangen Vorzug gegeben und im Zustand *Wait for Ctl 1* gewartet, bis der Empfangsvorgang abgeschlossen ist.

Anderenfalls wird fortgefahren und dem Arbiter der Sendewunsch mitgeteilt sowie der (erste) Paketheader auf die Datenleitungen gelegt. Nun wird im Zustand *Wait for NoC* darauf gewartet, dass der Empfänger frei ist und der Arbiter die Multiplexer schaltet. Anschließend wird bei kompakter Adressierung direkt mit dem Senden der Daten ange-

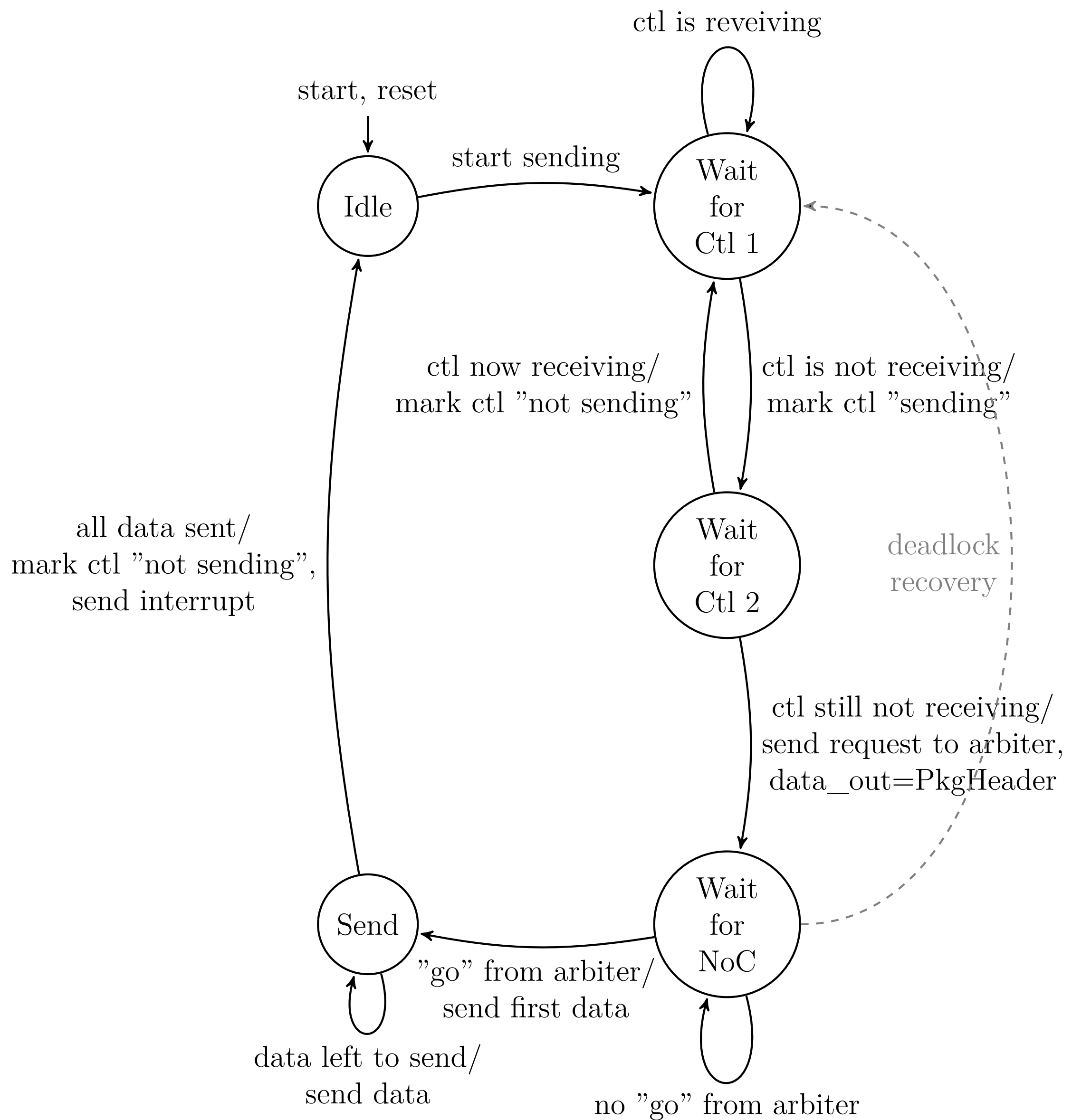


Abbildung 5.3: State-Machine der TX-Einheit mit kompakter Adressierung. Wie Abbildung 5.2 ist auch dieser Graph der Übersicht wegen stark vereinfacht

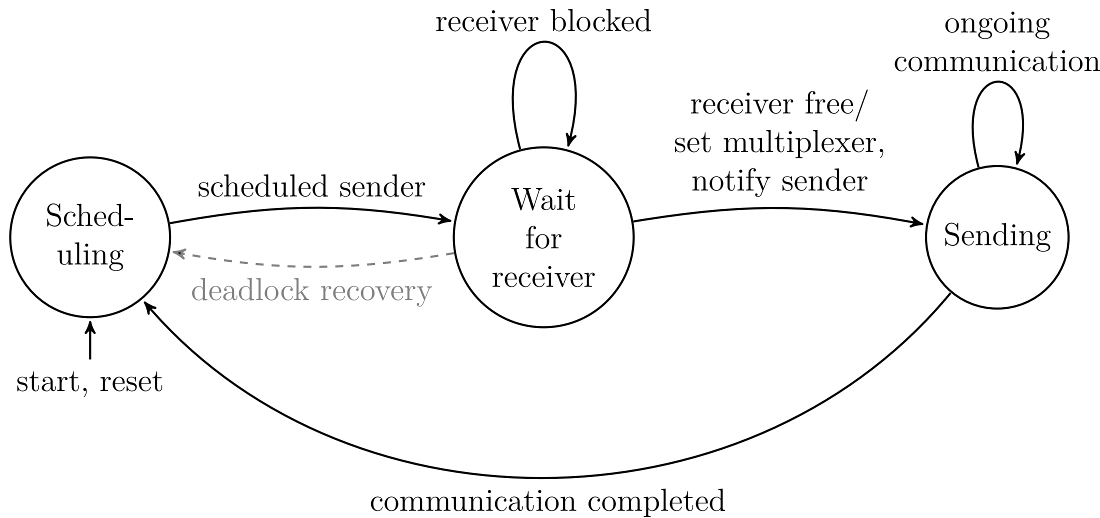


Abbildung 5.4: Vereinfachte State-Machine des Arbiters

fangen. Ansonsten existiert zwischen *Wait for NoC* und *Send* noch ein weiterer Zustand, welcher den zweiten Teil des Headers überträgt. In bestimmten, seltenen Fällen wird der Controller vom Arbiter angewiesen, den anstehenden Sendevorgang abzubrechen und später zu senden, um zu verhindern dass eventuell ein Deadlock auftritt. Würden zwei CPUs zum selben Zeitpunkt Sendevorgänge zu der jeweiligen anderen CPU starten, träte ohne diese Maßnahme ein Deadlock auf.

Nachdem alle Daten übertragen worden sind, wird der Controller wieder freigegeben, ein Interrupt bei der CPU ausgelöst und wieder in den Zustand *Idle* gewechselt.

5.5 Arbiter

Für jeden Teilnehmer im NoC gibt es eine Instanz des Arbiters. Der Arbiter regelt, wann ein Teilnehmer im Netzwerk zu "seinem" Empfänger senden darf. Der Arbiter ist wie die vorherigen Komponenten als State-Machine aufgebaut, die in Abbildung 5.4 dargestellt ist.

Wurde vom Scheduler ein Teilnehmer ausgewählt, der zu dem zum Arbiter zugeordneten Empfänger senden darf, wird im Zustand *Wait for receiver* zunächst darauf gewartet, dass der Empfänger empfangsbereit ist, dieser also nicht empfängt und je nach Konfiguration freie Puffer zur Verfügung stehen hat. Anschließend wird der Eingangsmultiplexer geschaltet und dem Sender mitgeteilt, dass er anfangen soll zu senden. Ist der Sendevorgang vorüber, wird im Zustand *Scheduling* der nächste Sender ermittelt. Zudem gibt es wie in der Empfangseinheit auch im Arbiter einen weiteren Zustandsübergang um Deadlocks zu lösen.

In einem Bitvektor ist kodiert, welcher Sender zu dem zum jeweiligen Arbiter zugeordneten Empfänger senden möchte. Anhand dieses Bitvektors wird im Zustand *Scheduling*

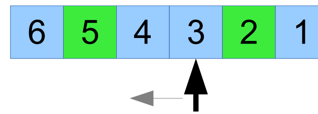


Abbildung 5.5: Darstellung des Priority Schedulers. Die grün eingefärbten Sender signalisieren ihre Sendeabsicht, der Zeiger befindet sich bei Sender 3.

entschieden, welcher Sender senden darf. Es stehen dafür verschiedene Schedulingverfahren zur Verfügung:

- **Priority:** Für das Priority-Verfahren werden wie in Abbildung 5.5 die einzelnen Bits des Vektors Takt für Takt abgearbeitet, wobei mit Core 1 begonnen wird, welcher entsprechend die höchste Priorität hat. In dem Beispiel in Abbildung 5.5 ist jedoch eine Ungenauigkeit des Schedulers erkennbar: Befindet sich der Zeiger bei Sender 3 und signalisieren Sender 2 und 5 dann ihre Sendeabsicht, wird Sender 5 ausgewählt und nicht erkannt, dass der höher priorisierte Sender 2 ebenfalls bereit steht. Ein weiterer Schwachpunkt ist die Zeit die vergehen kann, bis der Zeiger auf einen Sender mit Sendeabsicht trifft, obwohl der Sender schon länger bereit steht, da der Zeiger in jedem Takt nur eine Position betrachtet.

- **Round-Robin:** Dieser Scheduler arbeitet exakt wie der Priority-Scheduler, nur dass nach einem Sendevorgang der Zeiger nicht auf den höchstpriorisierten Sender 1 zurückgesetzt wird, sondern dieser lediglich einen Sender weiter nach links wandert. Auch hier existiert das Problem der eventuellen Verzögerung bis ein Sender gefunden wird.

- **Fast-Priority:** Dieser Scheduler betrachtet stets alle Sender zur gleichen Zeit. Das Prinzip wird in Abbildung 5.6 anhand von zwei Beispielen erläutert: Der Bitvektor welcher die Sendeabsicht der Sender codiert, wird als Unsigned-Integer-Zahl interpretiert und um eins dekrementiert. Der resultierende Vektor wird mit dem Ursprungsvektor per XOR verknüpft. Das Resultat besteht stets links aus Nullen, rechts aus Einsen. Die Position der Grenze zwischen den Nullen und Einsen gibt an, welcher der sendewilligen Teilnehmer die niedrigste Priorität hat. Im linken Beispiel in Abbildung 5.6 ist dies Sender 2, im rechten Beispiel Sender 3. Dieses Prinzip funktioniert nur, wenn es mindestens eine Sendeanfrage gibt. Daher wird ebenfalls überprüft, ob der Bitvektor mindestens eine Eins enthält. Dieser Scheduler entscheidet ohne Verzögerung in einem Takt und ist stets korrekt, enthält aber gegenüber dem *Priority* Scheduler wesentlich mehr Logik.

- **Fast-Round-Robin:** Dieser Scheduler übernimmt das Prinzip des *Fast-Priority* Schedulers, nullt jedoch nach einem Sendevorgang mittels einer Maske die rechten Bits des Bitvektors für einen Takt. Ein Fall, der auftreten kann nachdem zuletzt von Sender 3 empfangen wurde, ist in Abbildung 5.7 gezeigt. Steht im ersten Takt nach einem Empfangsvorgang kein Sender zur Verfügung, werden alle Bits

Bitvector:	0	1	0	0	1	0	0	1	0	1	0	0
Bitvector - 1:	0	1	0	0	0	1	0	1	0	0	1	1
XOR:	0	0	0	0	1	1	0	0	0	1	1	1

Abbildung 5.6: Zwei Beispiele des Prinzips des Fast-Priority Schedulers. Die gtrün eingefärbten Sender signalisieren ihre Sendeabsicht und sind im Bitvektor mit einer 1 kodiert.

Bitvector:	0	1	0	0	1	0
Maske nach dem Empfangen von 3:	1	1	1	0	0	0
AND:	0	1	0	0	0	0
- 1:	0	0	1	1	1	1
XOR:	0	1	1	1	1	1

Abbildung 5.7: Beispiel des Prinzips des Fast-Round-Robin Schedulers. Es wurde zuletzt von Sender 3 empfangen und daher eventuell bereitstehende Sender 1-3 ausgeblendet, was hier auf Sender 2 zutrifft.

eingeblendet. Dadurch ergibt sich dasselbe Verhalten des *Fast-Priority* Schedulers bei einer maximalen Verzögerung von zwei Takten.

5.6 Verschaltung der Komponenten

Nachdem alle Komponenten entwickelt wurden, müssen deren Signale noch miteinander verbunden werden. Dazu sind viele Anpassungen in den bereits existierenden Komponenten notwendig, wobei in den meisten Fällen die Signale lediglich durch die Komponenten durchgereicht werden. Eine Ausnahme stellt die *mpsoc_top* Komponente dar. Dort werden die Multiplexer, die in Abbildung 4.1 zu sehen sind, implementiert. Darüber hinaus werden in dieser Komponente die Bitvektoren für den Arbiter zusammengestellt. Sehr viele Signale passieren also die *mpsoc_top* Komponente, sie enthält jedoch abgesehen von den Multiplexern keine Logik für das NoC.

6 Evaluation des Platzverbrauches

Zur Evaluation des Platzverbrauches des Network on Chip auf dem FPGA wird die Anzahl der Look-up-Tables (LUT) betrachtet. Die LUTs stellen ein Kernstück bei der programmierbaren Logik dar, denn sie sind in der Lage, je nach Programmierung logische Gleichungen wie *AND*, *OR*, *XOR* usw. darzustellen. Es existieren auf einem FPGA auch andere Größen die den Platzverbrauch repräsentieren, häufig findet man in der Literatur eine Angabe des *Slice*-Verbrauchs. Slices fassen u.a. mehrere LUTs zusammen, sind jedoch häufig nicht vollständig ausgelastet, sodass sich der Slice-Verbrauch nur bedingt zur Angabe des Platzverbrauches eignet.

Für die folgende Evaluation wurden verschiedene Konfigurationen des NoCs erstellt und mittels der Xilinx ISE 14.6 Entwicklungsumgebung für das Virtex XC5VLX110T FPGA synthetisiert. Nach der Synthese bietet die Entwicklungsumgebung die Möglichkeit, den LUT-Verbrauch einzelner Instanzen von Komponenten auszugeben. Dieses FPGA stellt insgesamt 69120 LUTs [7] zur Verfügung, die programmiert werden können.

Bei der Synthese werden unterschiedliche Hardwarekonfigurationen verschieden stark optimiert bzw. deren Logik minimiert, sodass der Platzverbrauch schwankt und auf den ersten Blick nicht immer ganz logisch erscheint. Dies hat den Hintergrund, dass stets ein Kompromiss zwischen dem Platzverbrauch, der maximalen Taktrate sowie der Laufzeit der Synthese gefunden werden muss. Für die Evaluation werden ausgewogene Standardeinstellungen benutzt, die in der Xilinx Entwicklungsumgebung mitgeliefert werden.

Wird der Platzverbrauch einzelner Komponenten untersucht, bezieht sich die Angabe stets auf den gemittelten LUT-Verbrauch einer Instanz der Komponente im Gesamtsystem. Wenn im Folgenden vom Gesamtplatz des NoCs die Rede ist, ist damit der LUT-Verbrauch aller Instanzen der NoC-Komponenten (NoC-Controller, RX-/TX-Einheit & Arbiter) sowie der Verbrauch der Komponente *mpsoc_top* gemeint. In der *mpsoc_top* befindet sich zwar auch wenig andere Logik, jedoch überwiegt die für das NoC benötigte Logik insbesondere bei vielen angeschlossenen Cores. Vernachlässigt wird bei dem Gesamtverbrauch der zusätzliche Aufwand zur Speicheranbindung, da die dafür zuständigen Komponenten *wb_core* sowie *dmem4_dp* überwiegend andere Logik enthalten.

Im weiteren Verlauf dieses Kapitels wird zunächst der Gesamtverbrauch des NoCs analysiert und mit anderen Implementierungen verglichen. Anschließend wird der Platzverbrauch der einzelnen Komponenten in ihren verschiedenen Konfigurationen analysiert und verglichen. Konfigurationspunkte wie zum Beispiel die Optionen zur Blockierung werden nicht im einzelnen vorgestellt, da deren Einfluss auf den Platzverbrauch minimal ist.

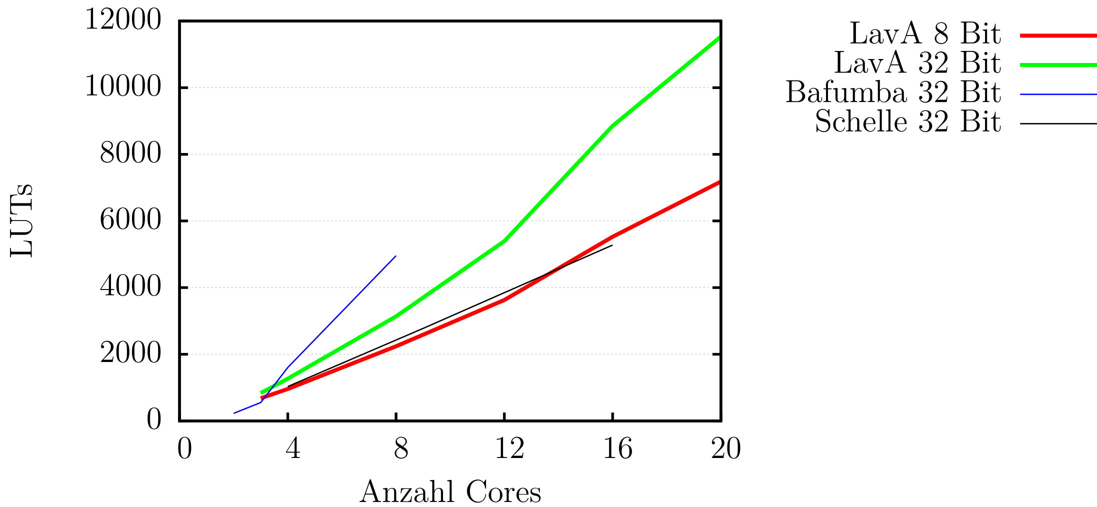


Abbildung 6.1: Vergleich des gesamten LUT-Verbrauchs von NoCs mit 8 Bit Busbreite ohne kompakte Adressierung und 32 Bit Busbreite und mit kompakter Adressierung, bei verschiedener Anzahl an das NoC angeschlossenen Cores. Der Datensatz "Bafumba" ist aus [5] entnommen, "Schelle" aus [8].

6.1 Gesamtverbrauch des NoCs

Zunächst wird der gesamte LUT-Verbrauch des NoCs anhand von zwei typischen Konfigurationen erläutert: 32 Bit Datenbreite mit kompakter Adressierung und 8 Bit Datenbreite ohne kompakte Adressierung. Typisch ist diese Kombination aus Datenbreite und Einstellung der Adressierung, da Absender sowie größere Datenpaketlängen nicht in 8 Bit kodierbar sind, während dies bei 32 Bit kein Problem darstellt. Als Scheduler wurde das *Priority*-Verfahren gewählt, 8 Bits zur Paketlängenkodierung, 13 Bits zur Datenspeicheradressierung sowie jeweils ein Empfangspuffer.

In Abbildung 6.1 ist der LUT-Verbrauch bei einer unterschiedlichem Anzahl von Teilnehmern im NoC dargestellt. Der Verlauf des Graphen verwundert nicht. Zwar wird bei der 8 Bit Konfiguration etwas mehr Logik aufgrund der getrennten Übertragung von Absender und Paketlänge benötigt, insgesamt wird dennoch wesentlich weniger Logik im Vergleich zur 32 Bit Konfiguration benötigt. Dass der LUT Verbrauch bei steigender Core-Anzahl steigt, ist zu erwarten.

Bei 20 Teilnehmern im NoC werden von dem XC5VLX110T FPGA in der 8 Bit Konfiguration 7184 LUTs verbraucht, was ca 10% der gesamten auf dem FPGA vorhandenden LUTs entspricht. Bei 32 Bit Datenbreite sind es 11531 LUTs beziehungsweise ca. 17%.

Zur Einordnung des im Rahmen dieser Arbeit entwickelten NoCs ist in Abbildung 6.1 zudem der LUT-Verbrauch anderer NoC Implementierungen angegeben. Die verwendete FPGA-Technologie ist bei allen Datensätzen dieselbe, die Zahlen sind also vergleichbar.

Bafumba et al. haben in [5] ein Crossbar Network on Chip entwickelt, welches das Round-Robin Scheduling-Verfahren beherrscht und aus einer API generiert wird. Es

ist gut sichtbar, dass deren NoC weniger Platz benötigt, wenn es sehr klein ist. Ab 4 Teilnehmern benötigt jedoch das in dieser Arbeit entwickelte NoC weniger LUTs. Viele Details zu dem NoC gibt es nicht, da der Fokus eher bei der Art der Generierung liegt.

Schelle und Grunwald haben in [8] sowohl ein recht komplexes Virtual Channel NoC sowie ein sehr simples Crossbar entwickelt. Das Virtual Channel NoC wird in diesem Vergleich vernachlässigt, da es extrem viel Platz verbraucht. Das implementierte Crossbar hingegen eignet sich zum Vergleich: Der Platzverbrauch des 32 Bit Datenbreite Crossbars von Schelle und Grunwald verhält sich annähernd genau so wie das in dieser Arbeit entwickelte NoC mit 8 Bit Datenbreite. Das NoC von Schelle und Grunwald verbraucht also weniger LUTs als das hier implementierte, jedoch hat dieses NoC einige Einschränkungen: Es ist nicht in der Lage, größere Datenmengen als Pakete zu bündeln, sondern es besitzt lediglich einen FIFO Puffer, welcher 32 Bit aufnehmen kann. Dadurch entfällt gegenüber der in dieser Arbeit entwickelten Lösung viel Logik, insbesondere sind keine Zähler mehr nötig um die Paketgröße zu verfolgen. Darunter leidet jedoch bei großen zu übertragenden Datenmengen die Performanz. In den Endknoten gibt es außerdem keine Anbindung des NoCs an einen Bus wie beispielsweise dem Wishbone-Bus oder eine Speicheranbindung, es eignet sich eher für die Anbindung von reinen Logikschaltungen in den Knoten. Dadurch ist es offensichtlich, dass die Implementierung wesentlich weniger Platz benötigt als die, die in dieser Arbeit entwickelt wurde.

6.2 8 Bit und 32 Bit Busbreite

Im weiteren Verlauf dieses Kapitels wird nun der Platzverbrauch der einzelnen Komponenten beziehungsweise Konfigurationen evaluiert.

Die Busbreite beeinflusst viele Komponenten des NoCs, da fast jede Komponente mit den Datensignalen agiert. Wie schon in Abbildung 6.1 gezeigt, sind die Unterschiede bezüglich des Platzverbrauches bei NoCs mit vielen Teilnehmern bei verschieden breiten Bussen relativ groß. Dies wird nun genauer untersucht.

In Abbildung 6.2 ist der Platzverbrauch der Komponenten eines NoCs mit 20 Teilnehmern angegeben, wobei die LUT-Anzahl aller Instanzen eines Typs aufaddiert wurden. Der Grafik kann man entnehmen, dass bei 32-Bit Busbreite insbesondere in der *mpsoc_top* Komponente viel mehr LUTs benötigt werden gegenüber dem 8-Bit NoC. Dies ist damit zu erklären, dass sich dort die Multiplexer befinden, die die Datensignale schalten um so die Verbindung zwischen den einzelnen Cores herzustellen. Außer bei den Arbitern ist auch in jeder anderen Komponente ein Zuwachs bei 32 bittiger Busbreite erkennbar. Da die Arbiter lediglich die Kommunikation verwalten, aber mit den zu übertragenden Daten selbst nichts zu tun haben, ist deren LUT-Verbrauch konstant.

6.3 Platzverbrauch der Arbiter

Die Arbiter-Komponente ist konfigurierbar bezüglich der Anzahl der an das NoC angeschlossenen Cores sowie des Scheduling Verfahrens. In Abbildung 6.3 ist der LUT-

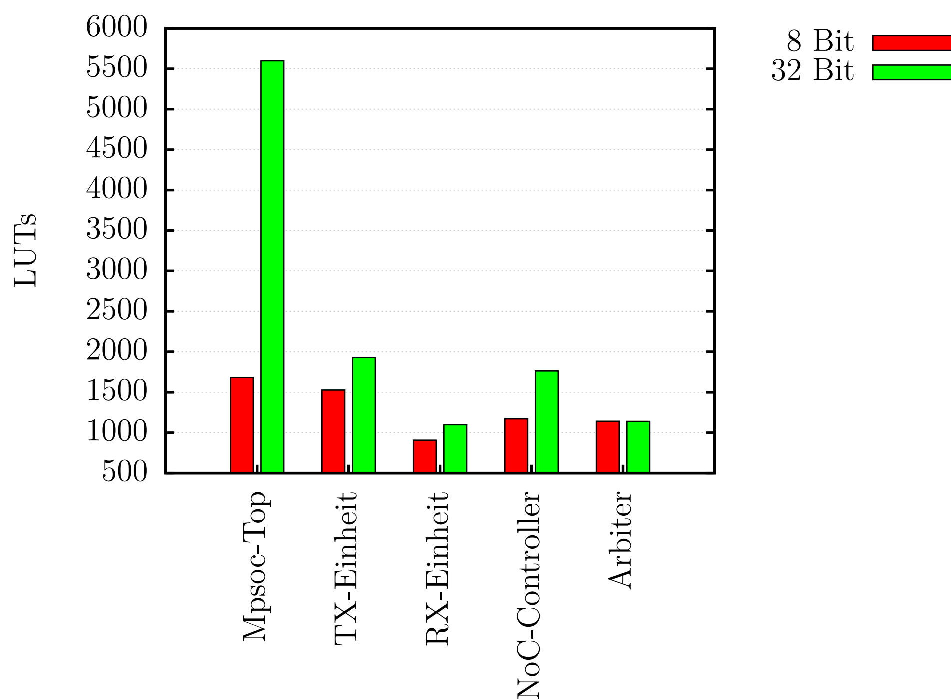


Abbildung 6.2: Vergleich des LUT-Verbrauchs zweier NoC-Konfigurationen mit jeweils 20 Teilnehmern und 8 oder 32 Bit Busbreite. Der Platzverbrauch der jeweiligen Instanzen eines Typs sind aggregiert.

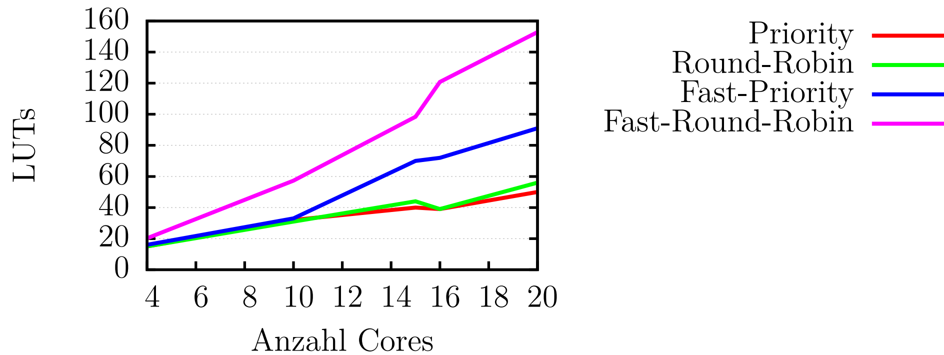


Abbildung 6.3: LUT-Verbrauch von Arbitern mit verschiedenen Scheduling-Konfigurationen und unterschiedlicher Gesamtanzahl der ans NoC angeschlossenen Cores

Verbrauch in Abhängigkeit zu diesen Parametern aufgetragen. Der Verbrauch bezieht sich jeweils auf eine Instanz des Arbiters im NoC, denn für jede angebundene CPU gibt es einen Arbiter. Der hier sichtbare Größenzuwachs bei steigender Anzahl an Cores erklärt sich dadurch, dass die Bitvektoren der Signale zur Kommunikation mit den einzelnen Controllern wachsen und so mehr Daten im Scheduler verarbeitet werden müssen. Da bei den schnellen Scheduling-Verfahren *Fast-Priority* und *Fast-Round-Robin* die Daten stets parallel verarbeitet werden, ist hier der LUT-Zuwachs besonders hoch.

Bei dem Graphen des *Round-Robin*-Schedulers ist erkennbar, dass eine Konfiguration mit 15 Cores mehr Platz verbraucht als mit 16 an das NoC angeschlossenen Cores. Die einzige Erklärung für diese Beobachtung sind die oben beschriebenen Optimierungen bei der Synthese.

6.4 Sende- und Empfangseinheit

Die Sende- und Empfangseinheiten des NoCs sind abhängig von der Wahl der kompakten Adressierung, der Anzahl der Teilnehmer im NoC sowie von der Busbreite. Bei größerer Busbreite steigt die Anzahl der Signale in der Komponente und ohne kompakte Adressierung wird zusätzliche Logik benötigt, um die Absenderadresse und die Paketlänge in getrennten Takten zu übertragen. Bei steigender Anzahl der an das NoC angeschlossenen Teilnehmer müssen mehr Bits für die Kodierung des Senders bzw. Empfängers bereitgestellt werden. Der in Abbildung 6.4 dargestellte LUT-Verbrauch der beiden Komponenten in verschiedenen Konfigurationen entspricht diesen Erwartungen.

6.5 Anzahl der Empfangspuffer

Im NoC-Controller werden die Daten gespeichert, auf die die CPU über den Wishbone-Bus zugreifen kann. Die gespeicherten Informationen über eingehende Pakete sind:

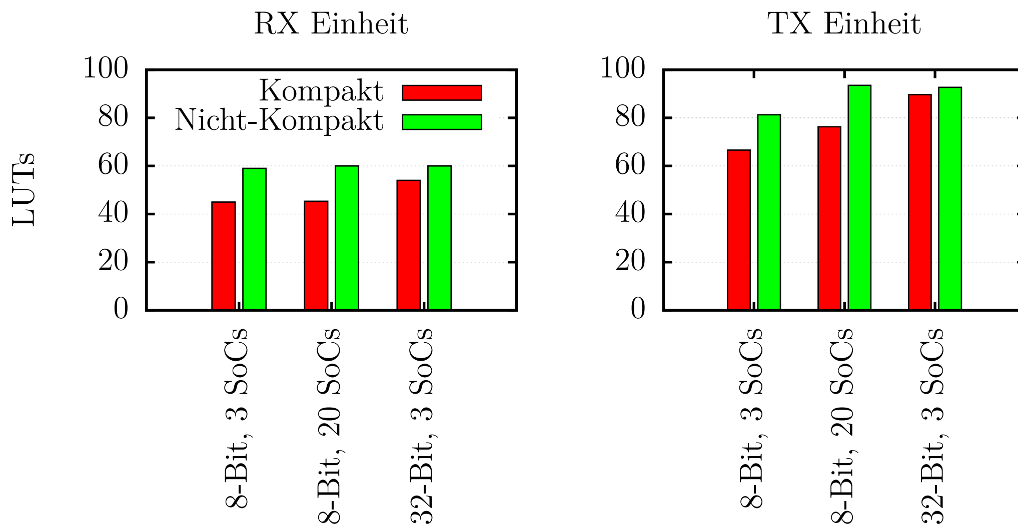


Abbildung 6.4: Darstellung des LUT-Verbrauchs von einzelnen Instanzen der RX- und TX-Komponente mit verschiedenen Konfigurationen.

- Der Ort und die Länge des Empfangspuffers im Speicher
- Die Länge des Paketes wie es der Sender vorgibt
- Die Länge des empfangenen Paketes
- Die Information ob der Puffer beschreibbar ist

Werden nun mehrere Empfangspuffer bereitgestellt, müssen diese Informationen entsprechend der Anzahl der Puffer gespeichert werden. Bei vielen Puffern führt dies zu einem beachtlichen Anstieg der Größe des Controllers, wie Abbildung 6.5 zeigt. Eine Instanz des Controllers mit 10 Empfangspuffern benötigt 3 mal so viele LUTs wie ein Controller mit nur einem Puffer.

6.6 Vergleich mit dem LavA-IPC

Zur Gegenüberstellung des Platzverbrauches des NoCs mit dem LavA-IPC wurde ein System mit 20 Cores synthetisiert. Die 20 CPUs sind mit einem 32 Bit NoC verbunden, welches den Priority-Scheduler sowie kompakte Adressierung benutzt und jeweils nur ein Empfangspuffer besitzt. Zudem sind die 20 CPUs auch mittels einem 32 Bit breitem IPC-Bus verbunden, der in jedem Knoten 5 Empfangs- und Sende FIFO-Puffer bereitstellt. In Tabelle 6.1 ist der LUT-Verbrauch der verschiedenen Kommunikationsstrukturen angegeben.

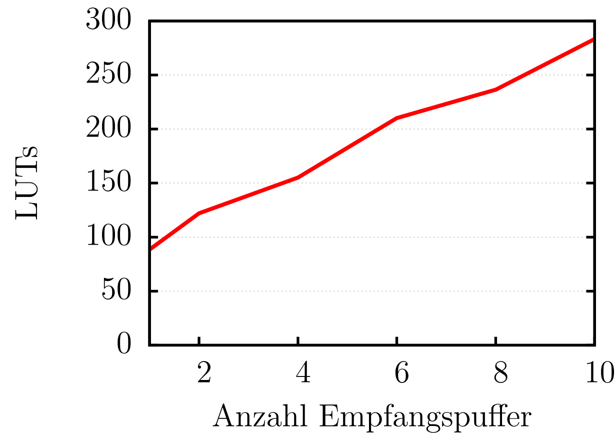


Abbildung 6.5: LUT-Verbrauch einer Instanz des NoC-Controllers bei unterschiedlicher Anzahl von Empfangspuffern

	NoC	LavA-IPC
mpsoc_top	5600	1919
Controller, Empfangs-/Sendeeinheit, Arbiter (gesamt)	6720	9500
Gesamt	12320	11419

Tabelle 6.1: Gegenüberstellung des LUT-Verbrauchs des NoCs mit dem LavA-IPC in einem System mit 20 Cores, mit typischen Konfigurationen der beiden Kommunikationsstukturen

Es ist zu sehen, dass die Multiplexer des NoCs in der *mpsoc_top*-Komponente wesentlich mehr Platz benötigen als die dort erforderliche Logik für den IPC. Die Komponenten, welche die CPU an den IPC bzw. an das NoC anbinden, benötigen beim IPC hingegen mehr LUTs als die entsprechenden Teile des NoCs. Insgesamt führt dies dazu, dass das NoC gerade einmal einen um 8% größeren LUT-Verbrauch hat als der existierende IPC-Bus.

An dieser Stelle muss beachtet werden, dass zu erwarten ist, dass der IPC bei einer Reduzierung der Anzahl der FIFO-Puffer weniger Platz benötigt. Die konfigurierten 5 Ein- und Ausgabepuffer ist jedoch die Standardkonfiguration.

6.7 Ressourcenmodell

Anhand der ermittelten Werte wurde in Tabelle 6.2 ein Ressourcenmodell erstellt, welches bei beliebigen Konfigurationen den LUT-Verbrauch des Network on Chip ohne Synthese abschätzt. Als Grundlage dieses Modells dient der LUT-Verbrauch verschiedener Konfigurationen. Diese Daten werden dann mittels einer linearen oder quadratischen Funktion interpoliert. Aufgrund von Optimierungen bei der Synthese weicht das Ergebnis dieses Modells jedoch von dem wirklichem LUT-Verbrauch ab.

$$
\begin{aligned}
LUT_{Arbiter} &= \begin{cases} 2.19 * Anz.\ Cores + 6.25 & \text{Priority} \\ 2.56 * Anz.\ Cores + 4.75 & \text{Round-Robin} \\ 4.68 * Anz.\ Cores - 2.75 & \text{Fast-Priority} \\ 8.29 * Anz.\ Cores - 12.91 & \text{Fast-Round-Robin} \end{cases} \\
LUT_{Controller} &= \begin{cases} 58 + 19.5 * Anz.\ RX\text{-}Puffer & \text{8 Bit Busbreite} \\ 84.5 + 19.5 * Anz.\ RX\text{-}Puffer & \text{32 Bit Busbreite} \end{cases} \\
LUT_{RX-Einheit} &= \begin{cases} 45 & \text{8 Bit Busbreite, ohne kompakte Adressierung} \\ 54 & \text{32 Bit Busbreite, ohne kompakte Adressierung} \\ 60 & \text{8 Bit Busbreite, mit kompakter Adressierung} \\ 68 & \text{32 Bit Busbreite, mit kompakter Adressierung} \end{cases} \\
LUT_{TX-Einheit} &= \begin{cases} 0.5 * Anz.\ Cores + 65 & \text{8 Bit Busbreite, ohne kompakte Adr.} \\ 0.4 * Anz.\ Cores + 88 & \text{32 Bit Busbreite, ohne komp. Adr.} \\ 0.5 * Anz.\ Cores + 76 & \text{8 Bit Busbreite, mit kompakter Adr.} \\ 0.4 * Anz.\ Cores + 99 & \text{32 Bit Busbreite, mit komp. Adr.} \end{cases} \\
LUT_{mpsoc_top} &= \begin{cases} 3.5 * Anz.\ Cores^2 + 16.2 * Anz.\ Cores - 40 & \text{8 Bit Busbreite} \\ 12 * Anz.\ Cores^2 + 40 * Anz.\ Cores - 114 & \text{32 Bit Busb.} \end{cases} \\
LUT_{Gesamt} &= Anz.\ Cores * (LUT_{Arbiter} + LUT_{Controller} + LUT_{RX-Einheit} + \\ &\quad LUT_{TX-Einheit}) + LUT_{mpsoc_top}
\end{aligned}
$$

Tabelle 6.2: Allgemeine Abschätzung zum LUT-Verbrauch des NoCs

Eine 8 Bit NoC-Konfiguration mit 8 Teilnehmern, je einem Empfangspuffer und dem Priority Schedulingverfahren und ohne kompakte Adressierung benötigt nach der Synthese 2242 LUTs. Laut dem Modell benötigt es hingegen 2200 LUTs, es ist somit eine relativ gute Näherung. Eine andere Konfiguration des NoCs mit 32 Bit Busbreite, Fast-Round-Robin Scheduler, kompakter Adressierung und 15 Teilnehmern mit je 10 Empfangspuffern benötigt laut dem Modell 12216 LUTs, während dieses NoC nach einer Synthese lediglich 9978 LUTs des FPGAs verbraucht. Dies entspricht einer Abweichung von ca. 22%.

Um dieses Modell zu verbessern, müsste der Platzverbrauch wesentlich intensiver und systematisch untersucht werden, eine so umfassende Analyse geht jedoch über den Umfang dieser Arbeit hinaus.

7 Evaluation der Performanz

Nachdem der Platzverbrauch des im Rahmen dieser Arbeit implementierten NoCs ausführlich evaluiert wurde, wird in diesem Kapitel nun die Performanz untersucht.

Zunächst wird das NoC in seiner Struktur analysiert und daran die theoretische Performanz ermittelt. Anschließend werden verschiedene Tests mit dem NoC in einem für die Tests maßgeschneiderten LavA System on Chip durchgeführt und deren Ergebnisse erläutert. Die Tests beziehen sich stets auf die Anzahl der Takte bzw. Zeit, die für die Übertragung einer gewissen Datenmenge benötigt wird. Dabei wird zunächst nur das NoC betrachtet und anschließend mit dem LavA-IPC verglichen. Dann folgt ein längerer Test, in dem das Verhalten des NoCs unter Last ermittelt wird sowie eine kurze Analyse einer parallelen Matrixmultiplikation.

7.1 Theoretischer Durchsatz

Wie schon in Kapitel 4.1 erwähnt, überträgt das NoC pro Takt ein Wort, das heißt je nach Konfiguration 8 oder 32 Bit. Wenn man davon ausgeht, dass das FPGA mit einer Frequenz von 100MHz getaktet ist, werden die Daten zwischen zwei CPUs mit 100MB/s bei 8 Bit Busbreite beziehungsweise 400MB/s bei 32 Bit Busbreite übertragen, sobald der Übertragungskanal aufgebaut ist. Die von LavA generierte Hardware kann in der Regel noch etwas schneller getaktet werden, sodass noch höhere Übertragungsraten möglich sind. Das Aufbauen des Kanals benötigt einige zusätzliche Takte, wie viele hängt stark von der gewählten Konfiguration des NoCs ab. Die folgende Auflistung gibt Aufschluss über die zum Aufbau der Verbindung benötigten Takte, unter der Voraussetzung, dass das NoC vollständig unausgelastet ist und somit nicht auf andere Sendevorgänge gewartet werden muss:

- 3 Takte vergehen vom Zugriff der CPU auf das *send_target*-Register bis zum Ende der Kollisionserkennung des NoC-Controllers.
- Mindestens 2 Takte vergehen zwischen der Sendeanfrage am Arbiter und dem Herstellen der Verbindung zwischen dem Sender und Empfänger. Werden die langsamen, platzsparenden Arbiter eingesetzt, müssen im schlechtesten Fall zusätzlich so viele Takte gewartet werden, wie das NoC Teilnehmer hat.
- Weitere 3 oder 4 Takte dauert es, bis die Paketlänge sowie der Absender übertragen worden sind. Die genaue Anzahl der benötigten Takte hängt davon ab, ob die beiden Daten in einem gemeinsamen oder zwei getrennten Takten übertragen werden. Anschließend beginnt der eigentliche Sendevorgang.

- Nach dem Senden der Daten benötigt es einen weiteren Takt, um die Interrupts auf Sende- und Empfangsseite auszulösen. Stehen beim Empfänger mehrere Empfangspuffer zur Verfügung, dauert es noch einen weiteren Takt, bis ein neuer Puffer der Empfangseinheit zugewiesen wird und das nächste Paket empfangen werden kann.

Zusammenfassend heißt dies, dass zur Taktzahl der Datenübertragung mindestens 10 weitere Takte benötigt werden, bis das NoC ein Paket versendet hat und wieder bereit für weitere Übertragungen ist.

7.2 Übertragen von Daten

Die im vorherigen Abschnitt ermittelten Werte sollen nun praktisch überprüft werden. Dazu werden verschieden große Pakete von einem Sender-Core zu einem Empfänger gesendet und die Anzahl der Takte gemessen, die zur Übertragung benötigt werden. Der Empfänger verarbeitet die Pakete nicht, er stellt lediglich sicher, dass es zu jedem Zeitpunkt freie Empfangspuffer gibt, sodass der Sender niemals auf den Empfänger warten muss. Neben dem Test findet auf dem NoC keine weitere Kommunikation statt.

In Abbildung 7.1 ist der grobe Programmablauf der sendenden CPU angegeben. Es fällt auf, dass in diesem Test komplett auf Interrupts verzichtet und stattdessen das Status-Register gepollt wird. Der Aufruf der Interruptroutinen würde zu lange dauern und dadurch das Ergebnis stark verfälschen. Doch leider ist es auch mittels Pollen nicht möglich, die Anzahl der benötigten Takte genau zu messen. Nach Beginn des Sendevorgangs überprüft die CPU den Wert des Statusregisters und stellt anhand dessen fest, ob das NoC noch sendet. Die CPU muss zwischen den Überprüfungen des Statusregisters jedoch stets einige Recheninstruktionen sowie einen Sprungbefehl abarbeiten, um im Programmcode zurückzuspringen. Dies dauert einige wenige Takte, sodass die Dauer der Übertragung nicht ganz genau festgestellt werden kann. Außerdem ist nicht sichergestellt, dass zwischen den Zugriffen der CPU auf die NoC-Register und dem Timer in den Zeilen 4/7 und 10/13 in beiden Fällen exakt dieselbe Zeit vergeht. Auch an dieser Stelle können Ungenauigkeiten bei der Zeitnahme auftreten.

Für die Messung der Übertragungszeit wird nun ein 32 Bit NoC mit 16 Teilnehmern, kompakter Adressierung und verschiedenen Arbitern benutzt. Der Test wurde für jede Paketgröße 1000 mal durchgeführt und die Ergebnisse dann gemittelt. In Abbildung 7.2 ist die benötigte Taktzahl zur Übertragung von 4 bis 4096 Bytes zu sehen. Der lineare Verlauf entspricht den Erwartungen.

Bei einer Übertragung von 4096 Bytes werden je nach Konfiguration des Arbiters 1027 bis 1037 Takte gemessen. Dies entspricht den Erwartungen nach der theoretischen Analyse in Kapitel 7.1, laut der für die reine Datenübertragung von 4096 Bytes 1024 Takte benötigt werden sowie einige weitere für die Verwaltung des NoCs. Die gemessene Taktzahl von 1027 Takten ist definitiv etwas zu gering, da sicher mehr als 3 Takte zum Aufbau des Kommunikationskanals zwischen Sender und Empfänger benötigt werden. An dieser Stelle äußern sich die oben beschriebenen Probleme mit der genauen Zeitmessung. Die Größenordnung des gemessenen Wertes ist allerdings vollkommen in Ordnung.

```
1  // Zugriff auf die NoC-Register zum Senden der Daten
2  *SEND_ADDR = &dummy_data;
3  *SEND_LENGTH = test_length;
4  *SEND_TARGET = 2;
5
6  // Timer Nullen
7  *TIMER = RESET;
8
9  // Warten, bis das NoC nicht mehr sendet
10 while (*STATUS & NOC_IS_SENDING_MASK);
11
12 // Timerwert merken
13 int time = *TIMER
```

Abbildung 7.1: Pseudo-Code zum Feststellen der Dauer einer Übertragung im NoC

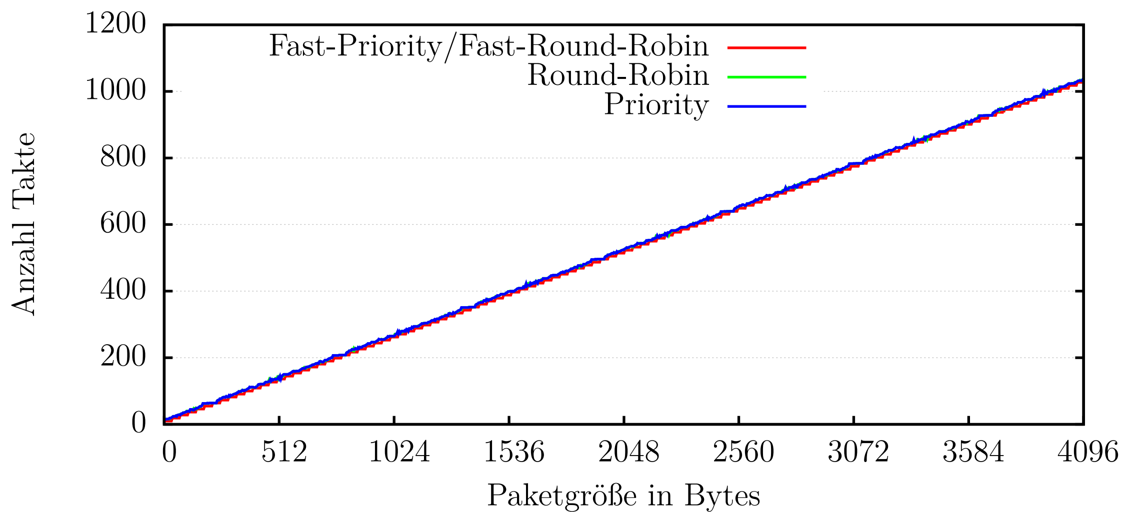


Abbildung 7.2: Dauer der Datenübertragung verschiedener Paketgrößen in einem NoC mit 32 Bit Busbreite und kompakter Adressierung

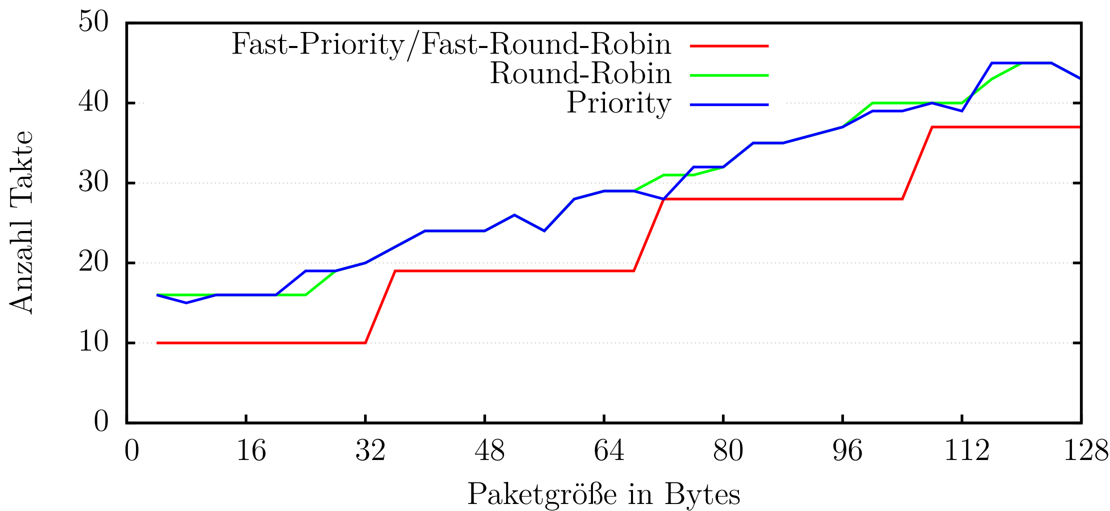


Abbildung 7.3: Ausschnitt aus Abbildung 7.2, in dem die Unterschiede zwischen den Arbitern deutlich werden

Abbildung 7.3 zeigt einen Ausschnitt aus der vorherigen Grafik, auf der die Unterschiede zwischen den Arbitern deutlicher werden. Die Arbiter mit Fast-Priority und Fast-Round-Robin sind exakt gleich schnell, die anderen beiden Konfigurationen benötigen im Schnitt etwas mehr Zeit. Die Treppenstufen bei den schnellen Arbitern entstehen dadurch, dass die CPU nicht in jedem Takt überprüft, ob das NoC noch sendet. Sie stellen daher eine Messungenauigkeit dar. Das Schwanken der langsamen Arbiter hingegen ist keine Messungenauigkeit. Dies liegt an der variablen Position des Zeigers auf den Bitvektor des Arbiters, welcher die Sendebereitschaft der verschiedenen Cores angibt. Da das NoC im Test insgesamt 16 Teilnehmer hat (von denen 14 nicht mit dem NoC interagieren), dauert es, wie in Abbildung 7.3 deutlich wird, im Schnitt einige Zeit, bis der Arbiter auf eine Sendeanfrage reagiert.

7.3 Vergleich mit dem LavA-IPC

Das Hauptziel dieser Arbeit ist es, eine flexible Kommunikationsstruktur zu implementieren, welche insbesondere große Datenmengen effizienter übertragen kann als der LavA-IPC. Um nun das entwickelte NoC mit dem existierenden IPC zu vergleichen, wird ein Test erstellt, bei dem Daten mit variabler Größe mittels NoC sowie IPC zu einem anderem Core geschickt werden und der Erhalt aller Daten vom Empfänger durch ein kurzes Paket mit einer Länge von 4 Bytes quittiert wird.

Ein Test wie im vorherigen Abschnitt, bei dem ausschließlich gesendet wird, ist mit dem LavA-IPC nicht umsetzbar, da die Sende-CPU nicht feststellen kann, ob der IPC die Daten versendet hat und ob beim Empfänger eventuell ein Überlauf aufgetreten ist. Wie zuvor wird auf Interrupt-Routinen verzichtet und es findet keine weitere Kommunikation auf dem NoC bzw. IPC-Bus statt. Beide Kommunikationsstrukturen haben eine

Busbreite von 32 Bit und der IPC besitzt 5 FIFO-Puffer, während die CPU beim NoC-Empfänger stets sicherstellt, dass freie Empfangspuffer vorhanden sind. Die Tests mit den verschiedenen Paketgrößen wurden wie zuvor 1000 mal ausgeführt und die Ergebnisse dann gemittelt.

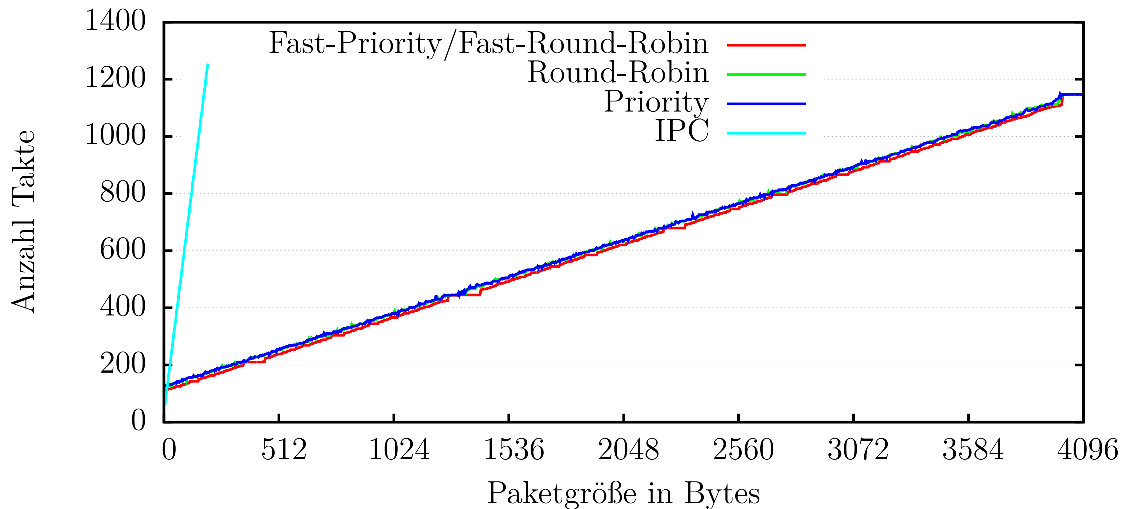


Abbildung 7.4: Vergleich des NoCs mit dem LavA-IPC bei großen Datenmengen

In Abbildung 7.4 ist gut zu sehen, dass das NoC bei großen Datenmengen wesentlich schneller ist als der LavA-IPC-Bus. In Abbildung 7.5 sind die Ergebnisse dieses Tests für kleine Datenmengen detaillierter dargestellt. Trotz der bei kleinen Datenmengen relativ langen Zeit, die das NoC vor und nach dem eigentlichen Datentransport benötigt, ist der IPC-Bus lediglich bei der Übertragung von bis zu 12 Bytes schneller. Bereits ab 16 zu übertragenden Bytes benötigt das NoC weniger Takte zur Übertragung und Quittierung der Daten. Das in dieser Arbeit entwickelte NoC eignet sich also sogar auch für die Übertragung von relativ kleinen Datenmengen.

7.4 Performanz unter Last

Bisher wurden alle Tests unter der Annahme gemacht, dass außer der Kommunikation im Rahmen des Tests das NoC komplett frei ist und kein weiterer Datenaustausch stattfindet. Nun soll das Verhalten untersucht werden, wenn die Kommunikation zwischen mehreren Teilnehmern auf dem NoC stattfindet.

Das Network on Chip ist in der Lage, Kommunikation zwischen 4, 6, 8, . . . Teilnehmern ohne Geschwindigkeitseinbußen durchzuführen, solange nicht mehrere Sender zu einem Empfänger senden. In diesem Fall sind alle beteiligten Komponenten unabhängig voneinander. Wenn ein Sender ein Datenpaket zu mehreren (empfangsbereiten) Teilnehmern senden möchte, ist das Verhalten auch klar: Das Datenpaket muss separat zu jedem Empfänger gesendet werden, da das NoC keine Multi- oder Broadcasts unterstützt. Die

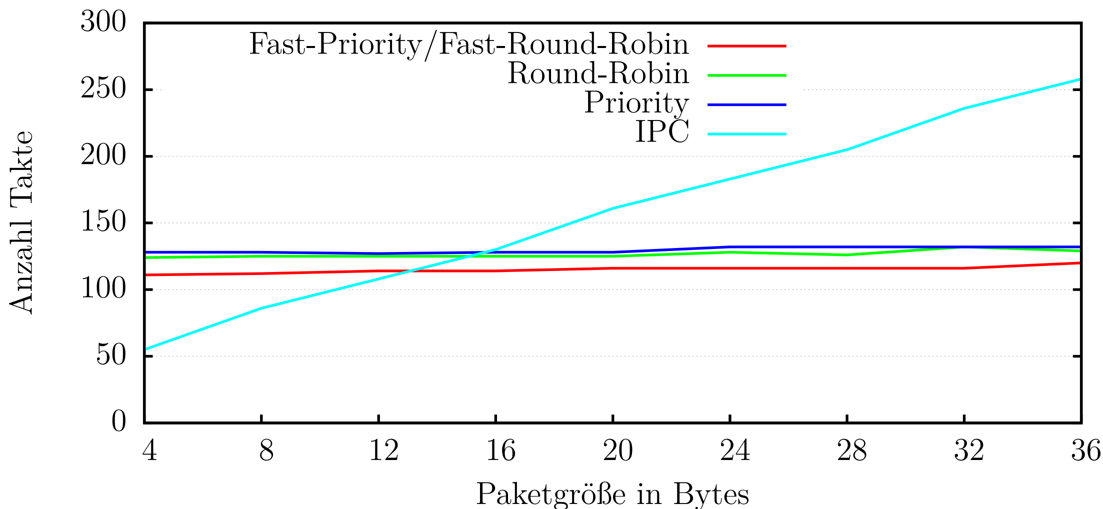


Abbildung 7.5: Vergleich des NoCs mit dem LavA-IPC beim Übertragen von kleinen Datenmengen

Gesamtdauer der Übertragung des Paketes entspricht der Übertragungsdauer wie im vorherigen Kapitel multipliziert mit der Anzahl der Empfänger.

Interessanter ist das Verhalten des NoCs, wenn mehrere Sender zu einem einzigen Empfänger senden möchten. Der Empfänger kann zu einem Zeitpunkt nur ein Paket von genau einem Sender empfangen, nicht von mehreren gleichzeitig. Die Art, wie die wartenden Sender abgearbeitet werden, bestimmt die Konfiguration des Arbiters. Im nun folgenden Test wird ermittelt, wie sich das NoC in einem solchen Fall verhält.

In Abbildung 7.6 sind die Funktionen der einzelnen CPUs im Test schematisch dargestellt. 14 Cores dienen zur Erzeugung einer bestimmten Last am Empfänger, indem sie in bestimmten Abständen Pakete versenden. Der Test-Core verschickt ebenfalls Pakete zum Empfänger und misst die Zeit die vergeht bis ein Paket unter einer bestimmten Last den Empfänger erreicht. Die CPU beim Empfänger stellt sicher, dass stets freie Empfangspuffer zur Verfügung stehen. Alle Cores sind mit einem 32 Bit breiten Network on Chip mit verschiedenen Arbiter-Konfigurationen miteinander verbunden.

Zur Lasterzeugung werden von den Sendern jeweils 4 KB große Pakete über das NoC an den Empfänger verschickt. Nach dem Versenden wartet die CPU eines Senders eine ganz bestimmte Zeit, sodass der Empfänger zu einem bestimmten Prozentsatz ausgelastet ist. An einem Beispiel ist diese Lasterzeugung in Abbildung 7.7 dargestellt. Bei der Lasterzeugung gibt es dieselben Probleme bei der Zeitnahme wie schon in Kapitel 7.2 beschrieben. Daher ist es nicht exakt möglich, genau die gewünschte Last zu erzeugen. Die erzeugte Last kann aber kalibriert werden, denn für die beiden Punkte "keine Last" sowie "100% Last" gibt es besondere Anhaltspunkte: Keine Last tritt natürlich auf, wenn die Sender nichts senden. 100% Last erkennt man daran, dass die Sendezeit eines Paketes im NoC mit einem Round-Robin Arbiter die obere Zeitschranke erreicht. Daher wird die erzeugte Last anhand dieser beiden Punkte kalibriert. Dazwischen steigt die Wartezeit

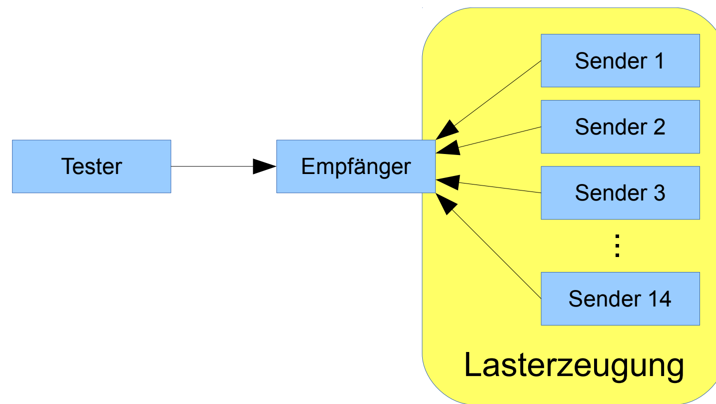


Abbildung 7.6: Schematische Darstellung der einzelnen Cores im Lasttests

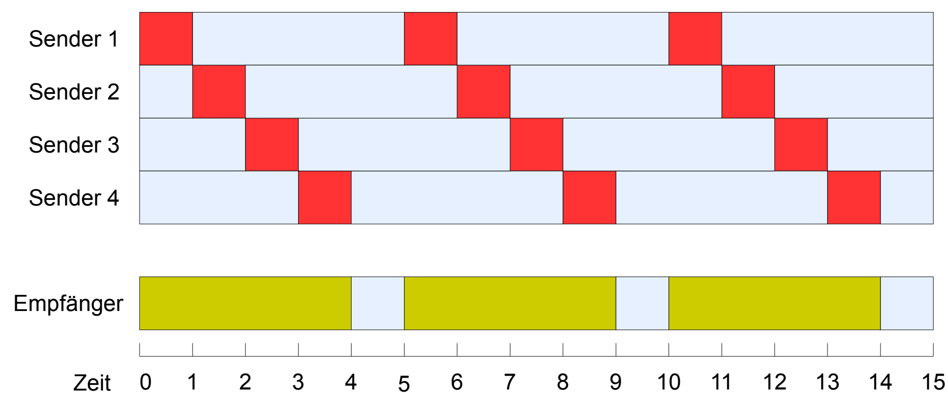


Abbildung 7.7: Beispiel für die Erzeugung von 80% Last für den Lasttest mit 4 Sendern. Die Sender senden eine bestimmte Zeit lang (rot) und warten dann (hellblau). Der Empfänger erhält dann 80% der Zeit Pakete (gelb).

der Sender linear, sodass auch die Last dazwischen gut approximiert werden kann.

Zunächst sendet der Tester 4 KB große Pakete ohne Unterbrechung an den Empfänger, lastet diesen also nahezu komplett aus. Die Auslastung ist aufgrund von CPU-Operationen zwischen den Sendevorgängen etwas geringer. Dadurch muss der Tester bei hohen Lasten meistens auf andere Übertragungen warten. Abbildung 7.8 verdeutlicht dies am Beispiel von 100% Last bei Verwendung des Round-Robin Schedulers. Nach dem Versenden eines Paketes folgt im Tester eine kurze Zeit von CPU-Operationen, anschließend soll das nächste Paket versendet werden. Bevor das Paket verschickt werden kann, empfängt der Empfänger Pakete der Sender, sodass eine lange Zeit vergeht, bis das Paket verschickt worden ist.

Bei der Ausführung des Tests versendet der Tester 100000 Pakete, misst deren Übertragungszeit und mittelt die Ergebnisse, wobei der Tester durch unterschiedliche Arbiter-Konfigurationen verschiedene Prioritäten im NoC besitzt.

In Abbildung 7.9 ist das Ergebnis dieses Tests dargestellt. Die obere Schranke der

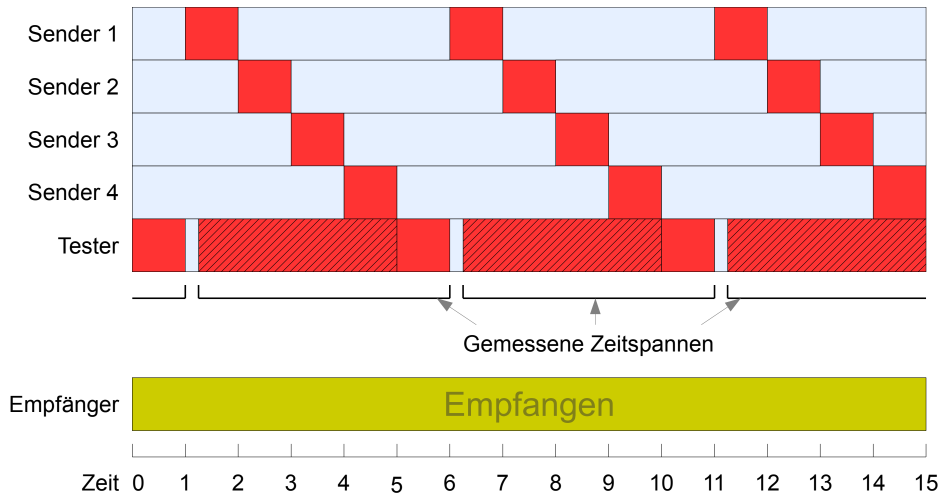


Abbildung 7.8: Verhalten des Testers bei maximaler Last unter Verwendung des Round-Robin Schedulers. Der Core muss stets lange warten (rot schraffiert), bis er das Paket versenden kann (rot)

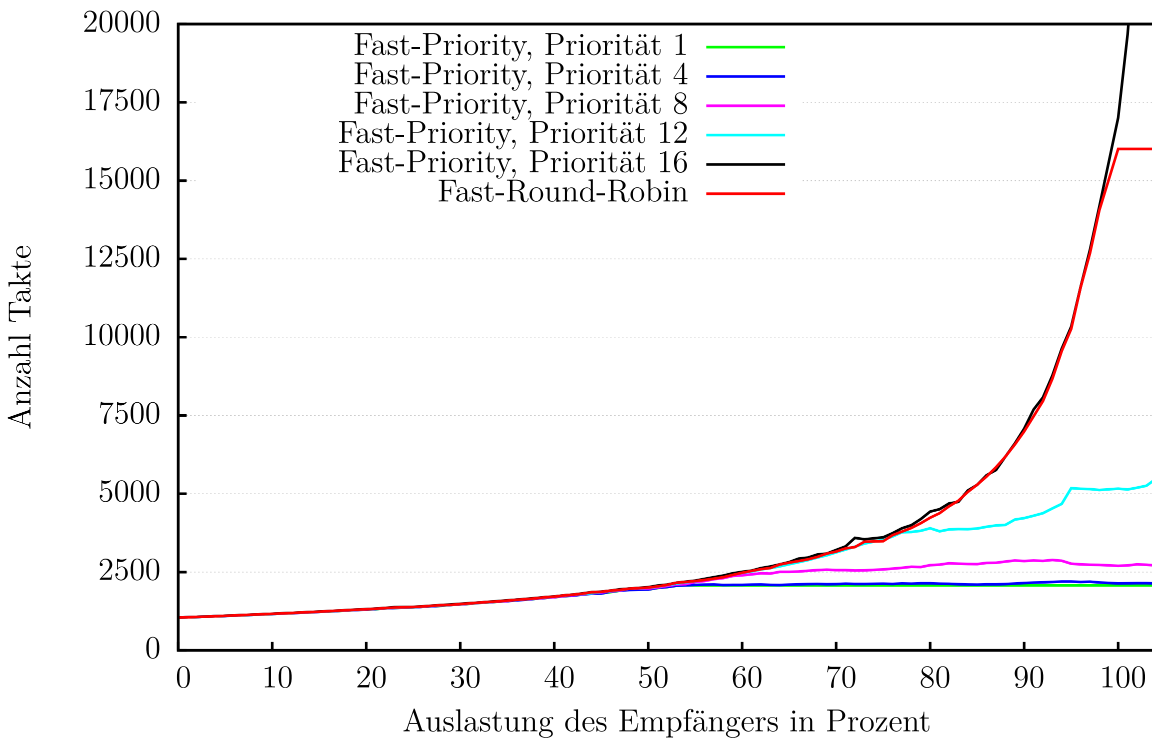


Abbildung 7.9: Zeitmessung der Übertragungszeit eines Paketes im NoC unter Last

Übertragungszeit eines Paketes vom Tester in einem NoC mit Round-Robin Scheduler ist sehr gut erkennbar, ebenso die Verhungerung bei hoher Last, wenn der Tester die geringste Priorität (16) besitzt. Es ist auch zu sehen, dass die Laufzeit bei höchster Priorität (1) bis auf das Doppelte anwächst. Dies ist dadurch zu erklären, dass in der kurzen Zeit zwischen zwei Sendevorgängen des Testers das NoC frei ist und ein anderes Paket empfangen wird. Da im NoC keine gestarteten Übertragungen verdrängt werden können, muss diese Übertragung erst abgeschlossen werden, bevor der Core mit höchster Priorität senden darf. Je öfter dies vorkommt, desto länger dauert die Übertragung im Schnitt. Damit ist der Verlauf des Testers mit der Priorität 1 zwischen 0% und 50% geklärt.

Hat der Tester eine der mittleren Prioritäten, liegt die durchschnittliche Übertragungszeit wie erwartet zwischen den hohen und niedrigen Prioritäten, wobei das Ergebnis etwas schwankt.

Dass der Graphenverlauf des Testers mit der niedrigsten Priorität (16) dem Graph des Round-Robin-Testers bis zum Erreichen der oberen Round-Robin-Schranke gleicht, liegt daran, dass die Tester nach einem Sendevorgang direkt die nächsten Sendevorgänge anstoßen und somit immer warten müssen, bis alle Sender ihre bereitstehenden Pakete versendet haben.

Um zu zeigen, dass dies nicht dem Durchschnittsfall entspricht, wurde dieser Test leicht abgewandelt und nochmals durchgeführt. Nach dem Versenden eines Paketes wartet der Tester eine zufällig lange Zeit, bis er das nächste Paket verschickt. Das Ergebnis dieses Versuches ist in Abbildung 7.10 zu sehen. Die Graphen schwanken relativ stark, die Ursache ist wahrscheinlich die relativ schlechte Verteilung der "zufälligen" Wartezeit. Diese hängt lediglich von gewissen Bits eines Timerwertes ab. Für eine bessere Generierung einer Wahrscheinlichkeitsverteilung, wie mit der *rand()*-Funktion aus der C-Library, ist der Instruktionsspeicher zu klein. Im Ergebnis ist aber gut zu sehen, dass der Tester im NoC mit Round-Robin-Scheduler bei hohen Lasten die Pakete nun wesentlich schneller verschickt. Dies war auch zu erwarten, da der Tester nun nicht mehr jedes Mal die maximale Wartezeit des Round-Robin-Schedulers abwarten muss. Außerdem ist hier gut sichtbar, dass der Anstieg der Sendedauer bei Testern mit hoher Priorität auch bei sehr hoher Last geringer ist als im vorherigen Versuch.

7.5 Beschleunigung einer Matrixmultiplikation

Abschließend wird anhand einer parallelen Matrixmultiplikation überprüft, ob sich das NoC auch für reale Anwendungen eignet. Für diesen Test wird ein System mit 17 Cores erstellt, das zwei 16x16 Floating-Point Matrizen miteinander multipliziert. Dabei dienen 16 Cores zum Rechnen, der 17. Core übernimmt die "Verwaltung", indem dieser die Daten zu den einzelnen Cores sendet und diese dann wieder empfängt. Jeder der 16 Rechencores berechnet jeweils eine Zeile der Ergebnismatrix. Zum Vergleich wird diese Ergebnismatrix zudem allein von einem Core berechnet.

Das Resultat dieses Tests ist eine 15,2-fache Beschleunigung der parallelen Matrixmultiplikation gegenüber der Single-Core Variante. Dieses Ergebnis ist nah an der theo-

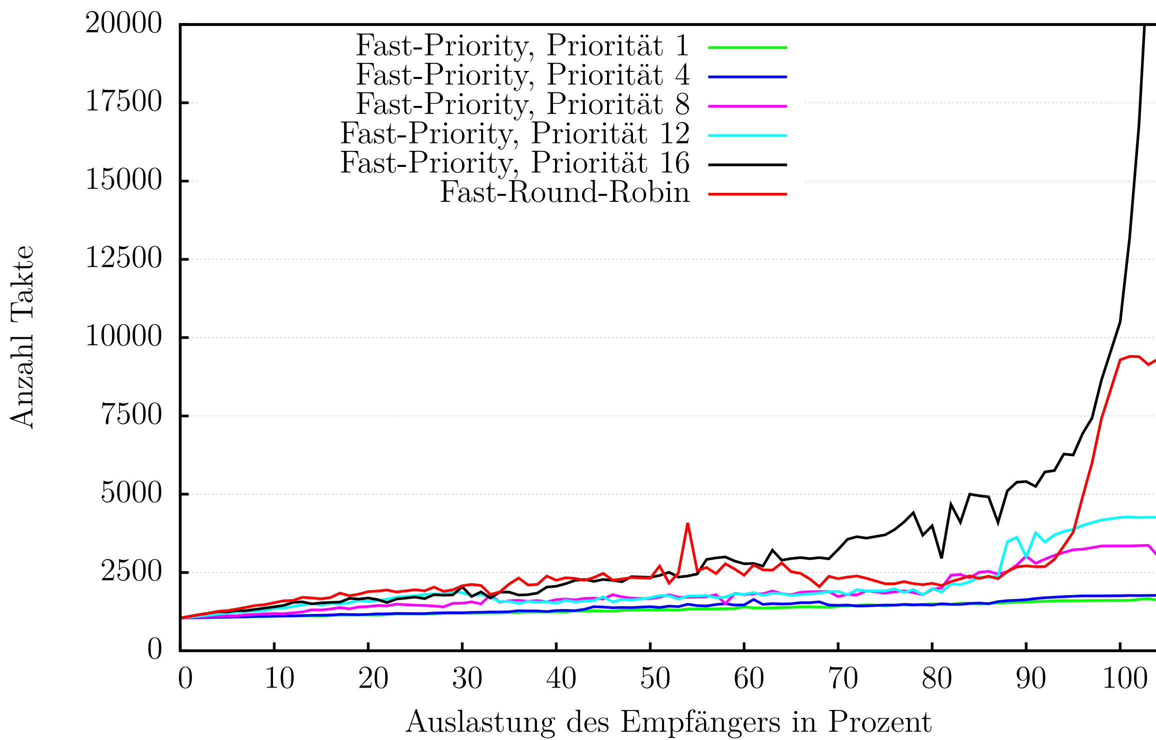


Abbildung 7.10: Abwandlung des voherigen Tests, in dem der Tester eine zufällige Zeit zwischen zwei Sendevorgängen wartet

retisch maximal erreichbaren 16-fachen Beschleunigung, welche nur möglich wäre, wenn die Daten nicht zwischen den Cores verteilt werden müssten. Der Overhead der Datenübertragung mit dem NoC ist somit sehr gering und es kann sicher gesagt werden, dass das in dieser Arbeit entwickelte Network on Chip als Kommunikationsstruktur für einen solchen Anwendungsfall bestens geeignet ist.

8 Fazit und Ausblick

Nach der ausführlichen Evaluation des in dieser Arbeit entwickelten Netwok on Chips folgt nun als letztes eine kurze Zusammenfassung der erreichten Ziele sowie ein Ausblick, wie das NoC weiterentwickelt werden kann.

8.1 Fazit

In Kapitel 2 wurde festgestellt, dass es noch starke Einschränkungen hinsichtlich der Kommunikationsmöglichkeiten zwischen mehreren CPUs bei der mit dem LavA-Framework erstellbaren Hardware gibt. Der bestehende IPC-Bus ist für die Übertragung großer Datenmengen nicht geeignet, da die CPU aktiv die Daten senden und empfangen muss und der IPC außerdem keine Möglichkeiten zur Vermeidung von Pufferüberläufen bietet. Außerdem ist es designbedingt nicht möglich, parallel mehrere Datenübertragungen gleichzeitig zu tätigen. Als Alternative zum IPC bietet LavA noch den Shared-Memory, mit dem jedoch lediglich genau zwei CPUs miteinander verbunden werden können.

Anhand dieser Analyse wurden die Anforderungen ermittelt, um die vorhandenen Lücken der Kommunikationsmöglichkeiten zu füllen. Diese sind insbesondere der Austausch großer Datenmengen zwischen beliebigen CPUs ohne diese stark zu belasten und die Möglichkeit der zeitgleichen Kommunikation zwischen mehreren Sender und Empfänger-Paaren. Hinzu kommt noch die Berücksichtigung des Platzverbrauches der resultierenden Kommunikationsstruktur, ein Aspekt, der für alle Entwicklungen für FPGAs oder Microchips relevant ist.

Nachdem die Anforderungen an die zu entwickelnde Kommunikationsstruktur klar definiert worden sind, fiel nach der Analyse verschiedener Ansätze in Kapitel 3 die Wahl auf ein Crossbar Network on Chip, welches anschließend unter Berücksichtigung der Gegebenheiten des LavA-Frameworks in Kapitel 4 entworfen und anschließend wie in Kapitel 5 beschrieben implementiert wurde.

Entwickelt wurde ein Crossbar Network on Chip welches unter anderem hinsichtlich der Busbreite, des Schedulers und der Anzahl der Empfangspuffer konfigurierbar ist. Es kann direkt auf den Datenspeicher der CPU zugreifen, sodass der Prozessor lediglich Metainformationen über die Datenübertragung mit dem NoC austauschen muss und während der Datenübertragung anderen Aufgaben nachgehen kann.

Bei der Evaluation in Kapitel 6 wurde festgestellt, dass die entwickelte Kommunikationsstruktur zwar einen beachtlichen Anteil der Chipfläche des FPGAs benötigt, der Verbrauch aber insgesamt vergleichbar mit anderen Implementierungen eines Crossbar NoCs ist. Bei der Untersuchung der Performanz wurde in Kapitel 7 abschließend an verschiedenen Beispielen gezeigt, dass das in dieser Arbeit entwickelte NoC sehr per-

formant und schon bei sehr kleinen zu übertragenden Datenmengen schneller als der LavA-IPC Bus ist. Die Anforderungen an das zu implementierende NoC wurden erfüllt, sodass das LavA-Framework nun umfassende Kommunikationsmöglichkeiten zwischen mehreren CPUs für alle Anwendungsfälle bereit stellt.

8.2 Ausblick

Das in dieser Arbeit entwickelte NoC kann in vielerlei Hinsicht weiterentwickelt werden. Eine sinnvolle Erweiterung der bestehenden Struktur wäre die Bereitstellung von Multi- oder Broadcasts, um die Gesamtdauer der Datenübertragung zu beschleunigen, wenn dieselben Daten an mehrere Empfänger gesendet werden. Das Beispiel der parallelen Matrixmultiplikation in Kapitel 7.5 könnte dadurch weiter beschleunigt werden.

Weiter ist es denkbar, eine alternative NoC-Struktur bereit zu stellen, wie beispielsweise das Simple Virtual Channel aus Kapitel 3.3. Eine mögliche Weiterentwicklung dieser Art wurde bereits beim Entwurf des Network on Chips in dieser Arbeit berücksichtigt, indem klare Schnittstellen zwischen den Komponenten geschaffen wurden, sodass im Idealfall keine Anpassung an den NoC-Controllern nötig ist.

Denkbar ist auch eine Anbindung spezialisierter Logikschaltungen ohne CPU an das Netzwerk. Der NoC-Controller, der im Wesentlichen zur Wishbone-Anbindung dient, fiele dann weg, dafür ließen sich die Sende- und Empfangseinheiten wiederverwenden.

Literaturverzeichnis

[1] TU-Dortmund, Informatik 12 Arbeitsgruppe Eingebettete Systemsoftware: *Laufzeitplattform für anwendungsspezifische verteilte Architekturen*. http://ess.cs.uni-dortmund.de/DE/Research/Projects/LavA/index.html. Version: Juli 2013

[2] Universität Erlangen, Informatik Lehrstuhl 4.: *CiAO: An Aspect-Oriented Operating-System Family*. https://www4.cs.fau.de/Research/CiAO/. Version: Juli 2013

[3] TU Delft, H.J. Lincklaen A.: *MB-Lite+*. http://ens.ewi.tudelft.nl/~huib/vhdl/mb-lite_plus.php. Version: Juli 2013

[4] OpenCores: *SoC Interconnection: Wishbone*. http://opencores.org/opencores,wishbone. Version: Juli 2013

[5] Bafumba-Lokilo, David ; Savaria, Yvon ; David, J-P: Generic crossbar network on chip for FPGA MPSoCs. In: *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on* IEEE, 2008, S. 269–272

[6] David, Jean-Pierre ; Bergeron, Etienne: An Intermediate Level HDL for System Level Design. In: *FDL*, ECSI, 2004, 526-536

[7] Xilinx Inc.: *Virtex-5 Family Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[8] Schelle, Graham ; Grunwald, Dirk: Exploring FPGA network on chip implementations across various application and network loads. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on* IEEE, 2008, S. 41–46

[9] Mullins, Robert ; West, Andrew ; Moore, Simon: Low-latency virtual-channel routers for on-chip networks. In: *ACM SIGARCH Computer Architecture News* 32 (2004), Nr. 2, S. 188

[10] Sethuraman, Balasubramanian ; Bhattacharya, Prasun ; Khan, Jawad ; Vemuri, Ranga: LiPaR: A light-weight parallel router for FPGA-based networks-on-chip. In: *Proceedings of the 15th ACM Great Lakes symposium on VLSI*. New York, NY, USA : ACM, 2005 (GLSVLSI '05). – ISBN 1–59593–057–4, 452–457

[11] *Hardware-Produktlinien.* http://ess.cs.uni-dortmund.de/DE/Research/Projects/LavA/areas/productlines.html. Version: Juli 2013

[12] The Eclipse Foundation: *Xpand.* http://wiki.eclipse.org/Xpand. Version: August 2013

[13] Dworschak, Jan M.: Beispielhafte Entwicklung einer Hardwareproduktlinie anhand einer UART-Familie TU-Dortmund, Informatik Lehrstuhl 12 Arbeitsgruppe Eingebettete Systemsoftware, 2011

Abbildungsverzeichnis

2.1 Beispielhafte Topologie des LavA IPC-Busses 4
2.2 Topologie des LavA IPC-Bus als Ring 4
2.3 Beispielhafte Topologie eines SoC mit Shared Memory 5

3.1 Struktur eines Crossbar NoCs 8
3.2 Struktur eines Virtual-Channel NoCs 9
3.3 Struktur eines sehr einfachen Virtual-Channel NoCs 10
3.4 Struktur eines routerbasierten NoCs 11

4.1 Konzept einer Implementierung des Crossbar-NoCs 14
4.2 Feature-Diagramm des NoCs 16
4.3 Auschnitt aus dem LavA Metamodell 19

5.1 Hierarchische Darstellung der für das NoC relevanten Komponenten . . . 23
5.2 State-Machine der RX-Einheit 24
5.3 State-Machine der TX-Einheit 25
5.4 State-Machine des Arbiters 26
5.5 Darstellung des Priority Schedulers 27
5.6 Beispiel des Prinzips des Fast-Priority Schedulers 28
5.7 Beispiel des Prinzips des Fast-Round-Robin Schedulers 28

6.1 Vergleich des gesamten LUT-Verbrauchs von NoCs mit 8- und 32 Bit Busbreite 30
6.2 Vergleich des LUT-Verbrauchs von NoCs mit 20 Teilnehmern und 8 oder 32 Bit Busbreite 32
6.3 LUT-Verbrauch verschiedener Arbiter Konfigurationen 33
6.4 Darstellung des LUT-Verbrauchs der RX- und TX-Komponente 34
6.5 LUT-Verbrauch des Controllers bei unterschiedlicher Anzahl von Empfangspuffern 35

7.1 Pseudo-Code zum Feststellen der Dauer einer Übertragung im NoC . . . 39
7.2 Dauer der Datenübertragung verschiedener Paketgrößen 39
7.3 Ausschnitt aus Abbildung 7.2 40
7.4 Vergleich des NoCs mit dem LavA-IPC bei großen Datenmengen 41
7.5 Vergleich des NoCs mit dem LavA-IPC bei kleinen Datenmengen 42
7.6 Schematische Darstellung des Lasttests 43
7.7 Beispiel für die Erzeugung von 80% Last 43

7.8 Verhalten des Testers bei maximaler Last 44
7.9 Zeitmessung der Datenübertragung in einem NoCs unter Last 44
7.10 Abwandlung des voherigen Tests, mit zufälliger Wartezeit im Tester . . . 46

Tabellenverzeichnis

5.1 Beschreibung der Hardwareregister des NoC-Controllers 22

6.1 Gegenüberstellung des LUT-Verbrauchs des NoCs mit dem LavA-IPC . . 35
6.2 Allgemeine Abschätzung zum LUT-Verbrauch des NoCs 36