电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

学士学位论文

BACHELOR THESIS

论文题目 面向多芯粒集成的封装级网络设计与验证

学　　院 机械与电气工程学院

专　　业 电气工程及其自动化

学　　号 **2017080201019**

作者姓名 曹文旭

指导教师 姜书艳

摘　要

伴随着集成电路工艺的进步，单个硅片上集成的晶体管数量越来越多，芯片面积越来越大，导致其良率下降、成本升高。Chiplet（芯粒）概念的诞生为超大规模集成电路的可持续发展提出了解决方案，它将一个完整的芯片系统拆分为多个功能模块，每个模块进行独立的验证和封装，并最终组合成芯片整体。Chiplet 模式凭借其强大的成本和利润优势得到了众多芯片厂商的青睐，但这些厂商在构建其 Chiplet 产品时采用的设计架构各不相同，其技术封闭、通用性差，Chiplet 的互连和通信结构有待于被标准化。

基于此问题，本文提出了一种基于片上网络的面向多 Chiplet 集成的封装级网络架构，该架构整体上由多个片上网络级联构成，片上网络之间通过扩展端口和片间总线实现数据通信，每个片上网络内通过路由器实现数据交换，每个路由器的本地端口通过标准协议转换接口实现与 Chiplet 的通信。本文首先使用 Verilog HDL 语言实现了封装级网络的设计，然后基于 UVM 仿真验证平台对设计进行了完整的事务传输级的验证测试，并对不同条件下的网络性能进行了细致的量化和分析。最后，本文基于 FPGA 实现了对所设计封装级网络的原型样机，并通过实际数据传输验证了设计的物理可实现性。

关键词： Chiplet，封装级网络，片上网络，UVM

ABSTRACT

With the progress of integrated circuit processing technology, the number of integrated transistors on a single silicon chip is increasing, and the chip area is getting larger and larger, resulting in a decrease in its yield rate and an increase in cost. The birth of the Chiplet provides a solution for the sustainable development of VLSI. It splits a complete chip system into multiple functional modules, and each module is independently verified and packaged, and finally combined as a whole. The Chiplet model has been favored by many chip manufacturers due to its strong cost and profit advantages, but the design architectures adopted by these manufacturers when constructing their Chiplet products are different, with closed technology and poor versatility. Chiplet's interconnection and communication structure remains to be standardized.

Based on this problem, this thesis proposes a Network-in-Package architecture for multi-Chiplet integration based on Network-on-Chip. The architecture is composed of multiple cascaded Network-on-Chips, and data communication between the Network-on-Chips is realized through expansion ports and inter-chip buses. Each network realizes data exchange through routers, and the local port of each router realizes communication with a Chiplet through a standard protocol conversion interface. This article first uses Verilog HDL language to realize the design of the Network-in-Package, and then based on the UVM simulation verification platform to conduct a complete transaction transmission level verification test of the design, and conduct detailed quantification and analysis of the network performance under different conditions. Finally, this thesis implements a prototype of the designed Network-in-Package based on FPGA, and verifies the physical feasibility of the design through actual data transmission.

Keywords: Chiplet, Network-in-Package, Network-on-Chip, UVM

目 录

第一章 绪 论 1
1.1 研究工作的背景与意义 1
1.2 芯粒互连技术研究历史与现状 2
1.2.1 芯粒技术研究现状 2
1.2.2 片上网络与封装级网络研究历史与现状 4
1.3 本文的主要贡献与创新 5
1.4 本论文的结构安排 6
第二章 片上网络的基础研究 7
2.1 片上网络的拓扑结构 7
2.1.1 片上网络直接拓扑 8
2.1.2 片上网络间接拓扑 9
2.2 片上网络的数据包与数据路由 10
2.2.1 片上网络数据包格式 10
2.2.2 片上网络路由算法 11
2.3 片上网络的流控机制 12
2.3.1 基于数据包的流控机制 12
2.3.2 基于数据微片的流控机制 13
2.4 片上网络路由器微架构 14
2.4.1 虚通道技术 14
2.4.2 经典路由器微架构 15
2.4.3 维度切分式路由器微架构 16
2.5 本章小结 17
第三章 封装级网络设计 18
3.1 封装级网络总体架构 18
3.1.1 系统总体结构组成 18
3.1.2 片上网络规格定义 19
3.1.3 数据微片格式定义 21
3.2 路由器设计 22
3.2.1 路由器总体结构设计 22
3.2.2 面向流控的 FIFO 设计与包装 23

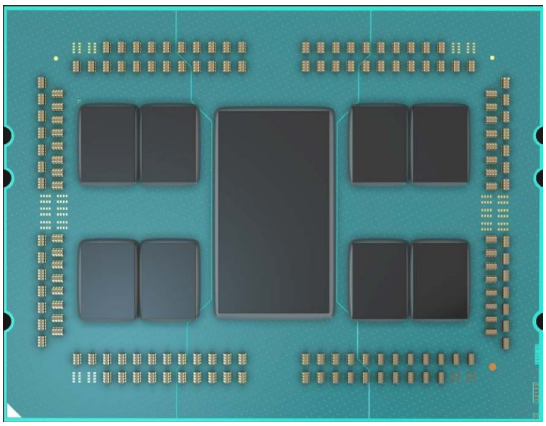
3.2.3 输入分配控制单元设计 25
3.2.4 输出仲裁控制单元设计 26
3.2.5 矩阵仲裁器设计 27
3.3 扩展端口控制器设计 27
3.3.1 旁路控制器设计 28
3.3.2 同步控制器设计 29
3.4 封装级网络的搭建 30
3.5 本章小结 30
第四章 封装级网络验证 32
4.1 仿真级验证 32
4.1.1 UVM 介绍 32
4.1.2 基于 UVM 的系统验证框架 33
4.1.3 基于 UVM 的路由器验证 34
4.1.4 基于 UVM 的片上网络验证 38
4.1.5 基于 UVM 的封装级网络验证 42
4.2 板级验证 44
4.3 本章小结 46
第五章 全文总结与工作展望 48
5.1 全文总结 48
5.2 后续工作展望 48
致 谢 49
参考文献 50

第一章 绪 论

1.1 研究工作的背景与意义

自 20 世纪 50 年代以来，集成电路（Integrated Circuit，IC）产业取得了迅速的发展。随着计算机系统理论的不断完善与成熟，EDA（Electronic Design Automation）工具的不断发展，以及硅晶片加工工艺与封装技术的不断进步，集成电路的规模和性能不断提升。Intel 公司创始人之一 Gordon Moore 曾提出：集成电路上可以容纳的晶体管数目在大约每经过 18 个月便会增加一倍，同时性能也翻一倍。这一规律后来被称为 Moore 定律[1]，它代表着生产厂商和用户对集成电路产品性能提升速度的期望，在一定程度上反映并指导着半导体市场的运作与创新。

随着整个社会对数据计算和信息处理规模和速度的需求不断增加，SoC（System on Chip）几乎已经成为了实现高性能信息处理系统的必要方案[2]，SoC 是指将多个功能模块集成在同一个晶片上，以处理器（Processor）为中心，通过总线互连各个组件实现数据传输与运算。SoC 技术的发展使得生产者能够通过提升处理器的性能和数量实现系统性能的扩张，然而随着单个硅片上集成晶体管数量的膨胀式增长，芯片的面积增加，导致其良率下降、成本提高[3]，再加上硅材料物理性质趋近极限以及加工难度的不断增加，摩尔定律面临着前所未有的挑战。封装级系统（System in Package，SiP）作为一种新的系统集成方式被提出[4]，它依赖于 MCM（Multi-Chip Module）封装技术[5]，顾名思义，SiP 是指将多个裸芯（Die）封装到统一管壳中，从而节约了单个裸芯的面积和成本，保证了良率，在一定程度上延长了摩尔定律。



(a)第二代 EPYC “ROME”;

(b)第三代 Ryzen “Matisse”；

图 1-1 AMD 基于 Chiplet 的处理器

芯粒（Chiplet）的诞生与发展赋予了 SiP 新的概念与新的实践。2014 年，海思（Hisilicon）与台积电（Taiwan Semiconductor Manufacturing Company，TSMC）首次在其晶圆级封装（Chip-on-Wafer-on-Substrate，CoWoS）产品[6]上提出了 Chiplet 的概念。Chiplet 一经提出便得到了国外一些半导体商业巨头的认可与采用，AMD 率先发布了其基于 Chiplet 的处理器产品 EPYC 和 Ryzen（图 1-1），之后众多厂商的 Chiplet 产品也相继问世。Chiplet 是一种设计模式，它指将芯片单独验证并包装成一个 Die，称为一个 Chiplet，通过组合和互连这些 Chiplet 来实现芯片级的知识产权（Intellectual Property，IP）复用。不同 Chiplet 可以采用不同工艺节点、来自不同厂商，这在很大程度上降低了生产成本、缩短了芯片的设计与验证周期、保证了芯片规模与面积能够按照摩尔定律的预期稳定增长。

然而，综观当前国外商业公司的 Chiplet 产品，其采用的互连架构各不相同、且大多使用专用通信协议、技术封闭、通用性差。反观我国当下，受到国际形势影响，我国芯片产业不断受阻。因此，抓住 Chiplet 发展的机遇，制订一套面向 Chiplet 的通用互连架构与通信协议标准，并在此基础上利用我国已有的自主可控的微处理器（Micro Processor Unit，MPU）、数字信号处理器（Digital Signal Processor，DSP）、专用加速器芯片，通过我国目前和国际先进水平差距最小的封装技术实现多 Chiplet 封装，进而构成具有较强运算能力的集成微系统将对我国集成电路产业具有重大意义。这将为我国的集成电路产业补齐短板、迎来全面发展的战略机遇争取宝贵的时间。

1.2 芯粒互连技术研究历史与现状

1.2.1 芯粒技术研究现状

近 6 年来，在国外一些半导体商业巨头的带领下，Chiplet 技术得到了充分的实践，市场上也涌现出了一批优秀的 Chiplet 产品，如 AMD 的服务器处理器 EPYC 和第 3 代桌面级处理器 Ryzen[7]，Intel 的 Lakefield 处理器[8]和 Stratix 10 FPGA[9]（Field Programmable Gate Array），NVIDIA 的 MCM-GPU[10]，Xilinx 的 Vertix 7[11]，法国 CEA 的 6 Die、96 核处理器[12]等。

相比传统的片内 IP 复用，Chiplet 主要具有以下三点优势：

1.不同 Chiplet 可以采用不同工艺节点。如第二代 EPYC[13]中使用先进的 7nm 工艺实现用于运算的处理器 Die，而使用 14nm 工艺实现用于互连的 IO Die，从而更好地实现性能与成本的权衡。

2.不同 Chiplet 可以来自不同厂商。如 Intel 的 i7-8809G 中同时集成了来自本厂商的 SoC、来自 AMD 的 Redeon GPU 和来自内存厂商的 HBM 显存颗粒[14]，每

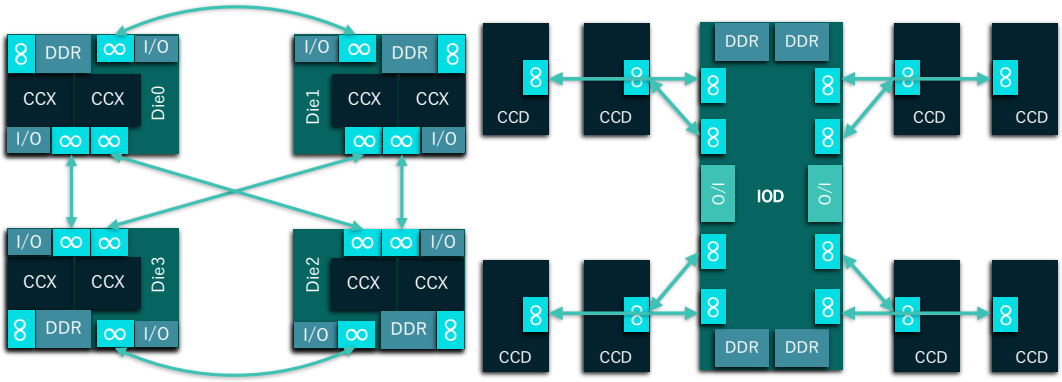
个厂商负责独立验证，相比于片上集成大幅度降低了开发难度、缩短了开发周期。

3.相同的 Chiplet 可以被多次复用。如 AMD 在其服务器产品 ROME 中实现了 8 片相同 Die 的复用，并将其复用至其桌面级产品 Matisse 中，这种芯片级的 IP 复用极大地节省了成本。

Chiplet 的发展面临着两大挑战：先进封装技术与 Chiplet 互连技术。

在 Chiplet 封装技术上，各大芯片厂商均取得了一定的突破与创新，这些封装技术为 Chiplet 的互连而生，其大致可以分为 2D 结构、2.5D 结构和 3D 结构三类，其中，2D 结构属于传统的 MCM 结构，以 AMD 的 EPYC 所采用的 Infinity Fabric 结构[15]为代表，另外，2.5D 结构与 3D 结构分别以 Intel 在 Stratix 10 FPGA 中采用的 EMIB 结构[9]和在 Lakefield 中采用的 Foveros 结构[8]最具代表性。

在 Chiplet 互连技术上，各厂商均采用特定架构与专用协议，市场上缺少统一的互连标准。例如，AMD 在其第一代 EPYC 产品中采用完全图的拓扑互连四片 Zeppelin Die，而在其第二代 EPYC 产品中引入了一个专门用于互连的 IO Die 来集中管理 8 片 Zepplin Die，在通信结构上采用自主定义的 SDF（Scalable Data Fabric）和 SCF（Scalable Control Fabric）实现数据的一致性传输和系统的配置与管理[15]。Intel 在其 Strtix 10 FPGA 中以 FPGA Core Die 为中心，使用 AIB（Advanced Interface Bus）互连其他 Die 组件。Xilinx 在其 Virtex7 2000 FPGA 产品中使用 SSI（Stacked Silicon Interface）技术实现多个 Die 之间 MAC 至物理层的连接[16]，其具体通信结构与总线协议被厂商保护而不予公开。



(a)第一代 EPYC 互连结构;　　(b)第二代 EPYC 互连结构;

图 1-2 AMD EPYC 处理器互连结构

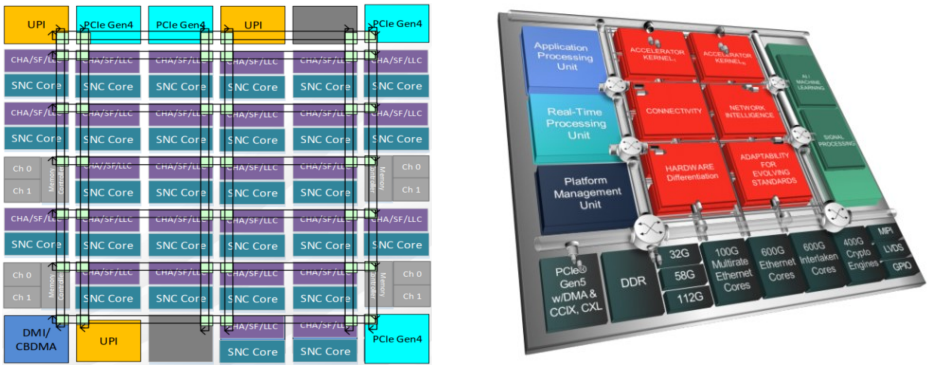
能否实现高效的互连与通信是决定 Chiplet 芯片性能的关键因素之一，然而综管当下，各大芯片厂商虽然赋予了 Chiplet 模式以充分的实践并取得了巨大进展，但其产品采用的互连拓扑却各不相同，通信结构或带有强烈的企业特色而缺乏通

用性，或技术封闭、由一家厂商垄断控制，或侧重于简单的 MAC 层和物理层，缺乏完整的数据链路层的定义和对组网的支持。

1.2.2 片上网络与封装级网络研究历史与现状

在 SoC 发展初期，数字集成电路大多使用总线式通信结构连接各个组件，随着加工工艺的进步以及单个硅片上集成的晶体管的数量不断增长，总线的互连结构复杂程度以及互连线的总长度在不算增加，随着互连线的增长，一方面，互连延迟与系统组合逻辑关键路径增加，另一方面，由互连结构引入的串扰增强，危及系统运行。NoC（Network on Chip）的概念最早作为一种十亿晶体管数量级集成电路的互连解决方案在 2000 年由瑞典皇家理工学院的 Axel 等人提出[17]，Axel 等人将 NoC 定义为一个计算、存储和 I/O（输入/输出）资源的网络，由交换开关（路由器）进行互连。交叉开关的引入以流水级寄存器的形式打断了传统总线结构中的较长的总线，从而克服了超大规模集成电路（Very Large Scale Integration，VLSI）中总线延迟大的缺陷，提升了系统时钟频率，减少了串扰问题。同时，通过采用特定的拓扑可以避免传统总线结构中的交叉布线，降低了芯片后端设计难度。另外，NoC 在可扩展性和吞吐量等指标上也展现出了优越的性能。

之后，越来越多的国外科研机构参与进了片上网络的研究，并取得了优秀的研究成果：从最初网络拓扑、路由算法、流控机制和路由器微架构方面的基础研究，到后来随着多核处理器的发展，服务于两大片上多核系统架构 MPSoC（Muti-Processor System-on-Chip）和 CMP（Chip MutiProcessors）的 Chache 一致性协议以及多核系统中网络死锁的避免与恢复等先进研究，还有一直伴随着 NoC 发展的性能、低功耗和可靠性的优化研究等，片上网络理论体系被不断补完善并趋于成熟。



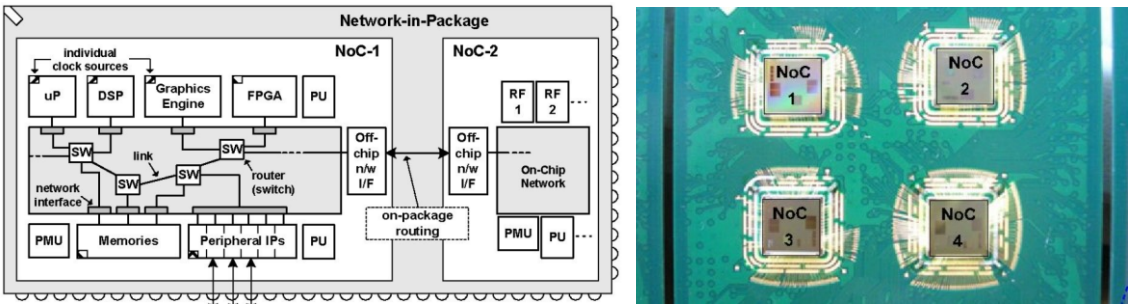
(a)Intel Xeon Ice Lake SP 28 核架构;　　(b)Xilinx Versal Premium;

图 1-3 基于 NoC 的芯片产品

同时也有越来越多的芯片厂商将 NoC 的解决方案应用到其产品中，例如，Intel 在其服务器处理器 Xeon[18]产品中使用 NoC 互连 28 个 CPU 核心和其他功能组件，验证了 NoC 强大的多核适应性；Xilinx 在其高端 FPGA 产品 Versal Premium[19]中使用 NoC 互连 FPGA 核心、DSP 引擎、4 个 Arm 处理器核心以及其他外设，构成了拥有 350 亿个晶体管的在当时全球单颗容量最大的 FPGA。

国内对于片上网络的研究起步时间较晚，但目前电子科技大学、国防科技大学、中国科学院、西安电子科技大学等研究单位已经在 NoC 的路由算法、低功耗设计、可测性设计以及容错设计等领域取得了一定的研究进展。

继 SiP 和 NoC 被提出后，封装级网络（Network in Package，NiP）的概念也于 2005 年被韩国科学技术院的 Kangmin Lee 等人提出[20]，他们将 NiP 定义为一个面向 MCM 的封装级互连结构，由多个 NoC 子系统通过封装级总线互连而成，每个 NoC 子系统包括一个 NoC 和多个与 NoC 相连的功能设备。



(a)NiP 结构;　　(b)NiP 原型实现;

图 1-4 Kangmin Lee 等人提出的 NiP

目前关于 NiP 的研究一直处于一种缺乏活力的状态，只有之前少量的研究致力于 NiP 的低功耗通信结构设计 [21]或基于有权图的 NiP 理论建模与分析[22]，然而这些研究都具有较浓的学术性质，更多停留在概念和理论的层面，缺乏实际生产的实践支撑。Chiplet 的出现和发展对 NiP 的研究工作提出了新的要求，市场上亟需一种统一的 Chiplet 互连与通信架构，这种架构属于 NiP 的范畴，但又超越 NiP 的历史概念，它将在未来成为支撑 Chiplet 发展的强大基石。

1.3 本文的主要贡献与创新

本文将 NiP 的概念放入 Chiplet 的互连中考虑，结合目前 Chiplet 模式缺少统一架构、技术封闭、可扩展性与通用性较差的缺陷，在 NiP 的研究基础上，重定义了 NiP 的架构概念，并为该架构定义了统一的互连与通信结构。使用 Verilog HDL

语言实现了所提出 NiP 框架的设计，并对该设计进行了基于 UVM 的仿真级验证以及 FPGA 板级原型验证。本文所提出的统一架构主要具有以下几点创新：

1. 提出了“NoC + 标准协议接口”的互连 Die 结构，实现了数据传输的“统一入网、统一协议”，提升了 NiP 的通用性。
2. 采用“总体去中心化，局部中心化”的 NiP 互连结构，保证了 NiP 在各种应用场景下的性能均衡，为大规模 Chiplet 互连奠定了基础。
3. 针对跨 Die 传输的数据进行了架构级性能优化，提出了一种 NoC 旁路结构，提升了数据跨 Die 传输的速度。

1.4 本论文的结构安排

本文共分为五个章节，其结构安排如下：

第一章作为本文绪论，介绍了研究工作的背景，阐述了本文研究工作的意义。对 Chiplet、片上网络和封装级网络的国内外研究历史现状进行了概述，并指出了本论文的主要贡献与创新。

第二章介绍了片上网络的基础研究，主要通过对片上网络的拓扑结构、路由算法、流控机制和经典路由器微架构等方面进行介绍，阐明了片上网络的一些基础概念与原理。

第三章介绍了封装级网络的设计过程。主要介绍了本文所提出的面向多 Chiplet 集成的封装级网络架构，本文设计的片上网络结构、总线协议、数据传输方法和数据包格式的定义，本次设计中路由器、同步控制器和旁路控制器等模块的详细设计细节以及片上网络和封装级网络的搭建。

第四章介绍了封装级网络的验证过程。首先分别介绍了基于 UVM 仿真验证平台的路由器、片上网络和封装级网络的验证过程，并对网络性能进行了深入的量化、统计和分析。然后介绍了基于 FPGA 的封装级网络的硬件原型机的实现以及板级验证过程。

第五章对全文进行了总结并对后续工作进行了展望。

第二章 片上网络的基础研究

一般而言，NoC 由网络节点（Network Node）、数据链路（Link）和网络接口（Network Interface，NI）三部分组成，数据通过数据链路传输在网络节点间传输，每个网络节点处设置一个路由器（Router），用于数据的转发，每个路由器可以挂载一个功能单元，如 MPU（Micro Processing Unit）、DSP 和 RAM（Random Access Memory）等，这些功能单元在系统中执行主要的数据信息处理功能，每个功能单元通过网络接口与路由器连接，网络接口是路由器与功能单元的通信接口。

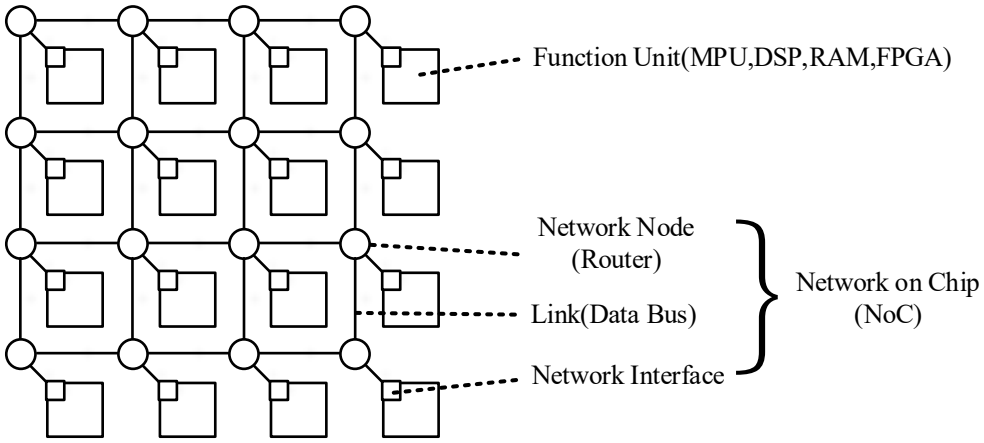


图 2-1 NoC 一般结构组成

NoC 的基础研究大致包括网络拓扑、路由算法、流控机制和路由器微架构四个方面。其中，网络拓扑指 NoC 的拓扑结构形态，它决定了网络节点间的互连方式；路由算法指数据在网络中的传输路径所遵循的算法规律；流控机制决定数据在网络中的分配与调度方式。三者相互依赖，共同决定路由器微架构的实现。此外，为避免网络死锁、解决网络拥塞问题而诞生的虚通道技术也成为了 NoC 基础研究不可或缺的一部分。

2.1 片上网络的拓扑结构

片上网络拓扑决定了网络中节点和通道之间的物理布局和连接。拓扑对整体网络性价比的影响是深远的。拓扑结构决定了消息必须经过的跃点（从一个路由器传输至另一个路由器）数以及跃点之间的互连长度，从而显着影响网络延迟。此外，拓扑结构还影响着网络能耗和网络带宽。

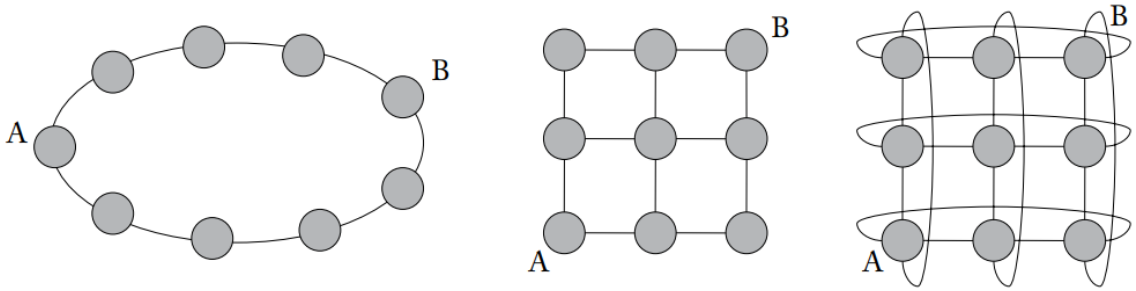
网络的拓扑结构大致可以分为直接、非直接、层次式和非规则式（定制式）四

种类型。在直接拓扑中，每个网络节点都和一个功能单元相连，而非直接拓扑允许一些网络节点不连接功能单元。层次式拓扑将网络划分为几个层次，每个层次内部具有一定的拓扑，不同的层次之间再按照一定的拓扑连接。定制式拓扑受用户的需求决定，主要应用在异质集成系统中。

2.1.1 片上网络直接拓扑

在直接拓扑中，每个网络节点均连接一个功能单元，所有节点既充当流量（Traffic，网络中传输的数据）的源（Source）节点和目的（Destination），又充当来自其他节点的流量的路由单元。迄今为止，大多数 NoC 设计都使用直接拓扑。

常见的直接拓扑有 Ring、Mesh 和 Torus 三种类型，在 Ring 拓扑中，所有的节点在同一个维度上首尾相连构成了一个环，Mesh 拓扑是一个二维的矩形网格结构，而 Torus 拓扑是在 Mesh 拓扑的基础上，将所有同一维度上的节点首尾相连得到的多环式结构。



(a)Ring 拓扑;　　(b)Mesh 拓扑;　　(c)Torus 拓扑;

图 2-2 常见的几种直接拓扑结构

为了衡量不同拓扑的性能，引入跳数（Hop count）、路径多样性（Path diversity）和度（Degree）的概念：跳数指网络中一次数据传输所遍历的链接数，网络中所有可能的数据传输情况的平均跳数可以表征该网络的的网络延迟（Network latency），网络延迟越小，网络消息的传递速度越快；路径多样性指在给定的一个源节点和一个目的节点之间不同路径的个数，较大的路径多样性可以给网络中的数据路由提供更多的可能，从而减轻网络拥塞（Congestion）。网络中与同一个节点相连的数据链路个数称作该节点的度，一个网络中所有节点的度的平均值可以表征网络开销（Cost）的大小，大的度数增加了网络实现的复杂性，并增加了每个路由器的面积和能耗开销。

在上述的三种直接网络拓扑中，相比 Ring 拓扑，Mesh 拓扑的平均跳数更小，这是因为 Mesh 拓扑中增加了更多的连接链路。Torus 拓扑的平均跳数小于 Mesh 拓

扑，这是因为 Torus 拓扑在 Mesh 拓扑的基础上将所有的行和列中首节点和尾结点连接起来，减少了统一行（列）中的平均跳数。因此，在三中拓扑中，Torus 拓扑具有最快的消息传递速度而 Ring 拓扑中的消息传递速度最慢。

Ring 拓扑的路径多样性恒为 2，这是因为 Ring 拓扑中的一次消息传递的路径只有顺时针和逆时针两种可能。相比 Ring 拓扑，Mesh 拓扑和 Torus 拓扑拥有更大的路径多样性，Torus 拓扑由于引入了环状结构而取得了最大的路径多样性。因此，在三种拓扑中，Torus 拓扑给予了路由算法最大的发挥空间，理论上网络拥塞下限最小。而 Ring 拓扑发生网络拥塞的可能性以及拥塞程度在三种拓扑中最大，因此 Ring 拓扑不适合应用在频繁传递数据的复杂信息处理系统中。

由于 Ring 拓扑中每个网络节点仅与其两侧的网络节点有链路连接，所以 Ring 拓扑中每个网络节点的度均为 2。在 Mesh 拓扑中，位于内部的网络节点均与四个方向的节点相连，其度为 4，而位于边界处的网络节点的度小于 4。在 Torus 拓扑中，所有网络节点的度均为 4。因此从 Ring 拓扑到 Torus 拓扑，其实现复杂性和难度依次增大，其面积和功耗也依次增大。

综上，Torus 拓扑拥有最快的消息传递速度和最低的理论拥塞程度，但其实现难度最大，网络开销也最大；Ring 拓扑实现难度和网络开销最小，但其性能在三种拓扑中也最差。Mesh 拓扑的所有指标都处在 Ring 拓扑和 Torus 拓扑之间，是一种性价比较高的折衷方案，因此，Mesh 拓扑往往成为了实际应用中的首要选择。

2.1.2 片上网络间接拓扑

间接网络通过一个或多个中间级的交换节点连接终端节点。在间接网络中，只有终端节点（Terminal Node）是流量的源和目的，中间节点（Intermediate Node）只负责将流量转发到终端节点。

Butterfly 拓扑是一种典型的间接拓扑,这种拓扑类似于快速傅里叶变换（Fast Fourier Transform，FFT）的算法框图中的蝶形结构，它可以被描述为一个k元（ary）n跳（fly）的图，包含$2 \times k^n$个终端节点和n个中间级，每个中间级包含k^{n-1}个$k \times k$的交换节点。Butterfly 拓扑中的源节点和目的节点是分离的，例如，在图 2-3 中，数据只能从一侧的终端节点向另一侧的终端节点传输，而不被允许在同一侧的节点之间传输。

在 Butterfly 拓扑中，当消息从一侧向另一侧传递时，其在每一个中间级必然经过一个节点，因此所有可能的数据传输情况的跳数均为n，所以 Butterfly 拓扑中的平均跳数为n。在给定一组位于不同侧的源节点和目的节点的情况下，连接这两个节点的路径仅有一条，因此 Butterfly 的路径多样性仅为 1。由于每个中间节点都

是一个$k \times k$的交叉开关，用于将k条输入链路上的数据交换至k条输出路径，因此 Butterfly 拓扑中每个中间节点的度均为 4，每个终端节点的度为 1。

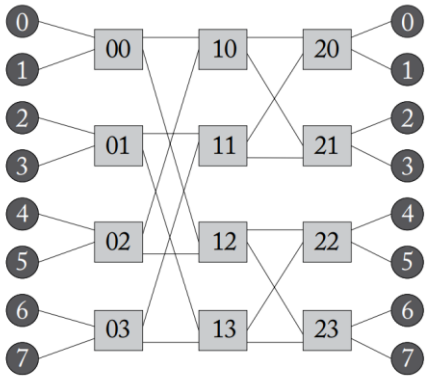


图 2-3 一种 2 元 3 跳的 butterfly 拓扑

由于平均跳数恒定，Butterfly 具有恒定的消息传递速度。但路径多样化的欠缺是 Butterfly 拓扑的一大软肋，在流量大且不均衡的情况下，Butterfly 拓扑将不再具有性能优势。另外，Butterfly 仅支持两侧间数据传输的特点使得它一般仅适用于多主（Master）多从（Slave）式的数据传输，用作对传统总线矩阵（Bus Matrix）的结构补充。

2.2 片上网络的数据包与数据路由

在 NoC 中，数据以数据包（Packet）的形式传输，数据包是按照一定的格式的数据帧排列而成的组合，在时钟（Clock）的驱动下流动于路由节点之间。数据包在被发送时，一般包含有该数据包的被期望的接收节点的 ID（Identity）信息，称作数据包的目的节点 ID，路由算法根据该目的节点 ID 计算确定数据包将要经过的路径。

路由算法的目标是在网络拓扑提供的路径之间平均分配流量，从而避免出现热点（Hot Spot，网络中数据流量较大的区域）并最大程度地减少竞争（Contention，多个数据包争夺同一个物理链路的现象），从而降低网络延迟并提高网络吞吐量（Throughput，单位时间内通过网络的数据量）。

2.2.1 片上网络数据包格式

NoC 的数据帧被称作微片（Flit），其定义为单个时钟周期网路数据总线上传输的有效数据，其位宽与网络的数据位宽相等。数据包由多个微片组成，微片一般包括头（Head）微片、体（Body）微片和尾（Tail）微片三种类型：头微片和尾微片

用以标记数据包的起始位置和终止位置，数据包中的有效数据负载（Payload）以体微片的形式存在。一般以取每个微片中数据的高几位标记该微片的类型。

一般而言，一个完整的数据包包括一个头微片、至少一个体微片和一个尾微片。

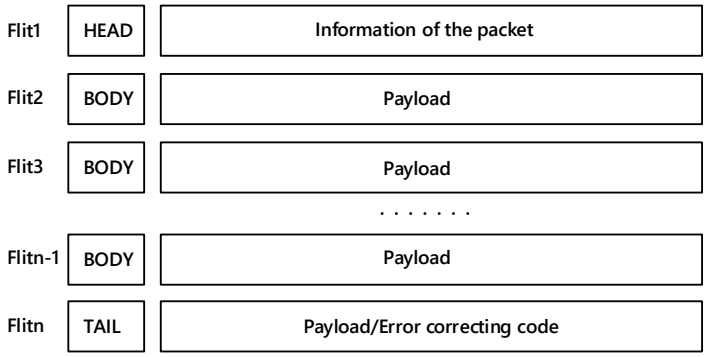


图 2-4 NoC 中数据包的一般格式

2.2.2 片上网络路由算法

NoC 的路由算法与其拓扑形态有关，并直接决定数据传输的跳数，而且间接影响着 NoC 的吞吐量和拥塞程度等性能指标。

NoC 的路由算法大致分为确定性路由和不确定性路由，在给定一个源节点和一个目的节点的情况下，前者指能够给出唯一且确定的传输路径的路由算法；后者指能够根据网络实时的拥塞情况决定数据的传输路径的路由算法。然而不同于以太网（Ethernet）等计算机网络的软件路由算法，NoC 属于硬件网络，其路由算法必须由路由器内部的硬件电路完成，因此路由算法的设计必须要考虑网络的硬件开销、实现难度和功耗等问题。不确定性路由显然能够有效降低网络平均传输延迟，但其复杂的硬件算法会其对增大网络的实现难度和成本开销。

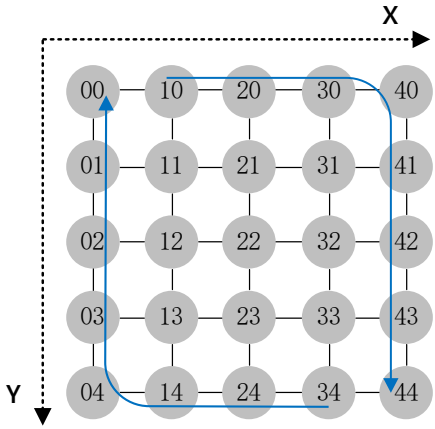


图 2-5 5×5 Mesh 网络中的 XY 维路由方式

在 Mesh 拓扑中，一中典型的性价比较高的路由算法为 XY 维路由算法，该路由算法属于确定性路由中的有序维度路由（Dimension-Orderd Routing，DOR），它将 Mesh 网络看作一个具有 X 和 Y 两个维度的网格坐标系，每个网络节点维护一个独一无二的坐标点(X_N, Y_N)，当一个数据包期望从其源节点(X_S, Y_S)传输至其目的节点(X_D, Y_D)时，规定该数据包首先在 X 维度上进行传输，直到到达与目的节点的 X 坐标相等的节点(X_D, Y_S)，然后在 Y 维度上进行传输，最终到达目的节点。

2.3 片上网络的流控机制

流控（Flow Control）机制是数据的调度机制，负责控制网络缓冲区（Buffer）和数据链路的分配。在传输路径被确定的情况下，流控机制决定数据在路径中的流动规律，即确定何时将缓冲区和链接分配给消息，如何分配缓冲区和数据链路的粒度（Granularity）以及如何使用网络在许多消息之间共享这些资源。优秀的流控机制能够通过提高网络资源占用率实现网络吞吐量和消息传递速度的提高。

根据网络分配数据的粒度的不同，可以将流控机制分为基于消息的流控、基于数据包的流控和基于微片的流控三类。其中基于消息的流控由于其预留链路的特点不利于网络吞吐量的提高，应用范围较窄，本文不再介绍。

2.3.1 基于数据包的流控机制

基于数据包的流控机制以数据包为单位进行网络资源的分配。

Virtual Cut Through（VCT）是一种应用于有缓冲网络（Buffered Network）的典型的基于数据包的流控机制，在有缓冲网络中，所有的路由器的每个输入端口都设有一个至多个缓冲（Buffer），当缓冲内待传出的微片没有被分配到输出链路时，该微片被缓存在缓冲内，这时待传入缓冲的微片可以顺利传入，从而实现缓冲的目的。每个缓冲最多能够存储n个微片，n记为缓冲的深度（Depth）。一般情况下，微片在缓冲中的流动方式遵循先进先出（First In First Out，FIFO）的规律，即缓冲不改变数据顺序的先后性。

VCT 的实现思路为：对于一个数据包的头微片，仅当下游（Downstream）路由器对应的输入端口处的缓存的剩余空间能够存下当前整个数据包时，才允许头微片传输至该输入端口，进入缓冲；否则，强迫头微片继续在上游（Upstream）路由器中等待。对于一个数据包的体微片和尾微片，只要下游路由器缓冲有剩余空间，就允许该微片传至下游。

图 2-6 展示了一个 VCT 的传输案例，其横轴表示时间，每个单位代表一个时钟周期，纵轴表示位置，0 至 8 分别代表数据包传输路径上的多个路由器，其方向

为上游至下游。方格中的字母代表最新传入当前路由器内缓冲的微片，其中 H、B、T 分别代表头微片、体微片和尾微片。从时刻 0 至时刻 2，数据包依次向前传输。在时刻 2 时，路由器 5 的输入缓冲有 2 个空位，由于当前数据包含有 5 个微片，所以该头微片停留在路由器 2 中等待，直到时刻 5 时，路由器 5 的空位增加到了 5 个，路由器 2 中的头微片才得以在下个时刻向路由器 5 传输。

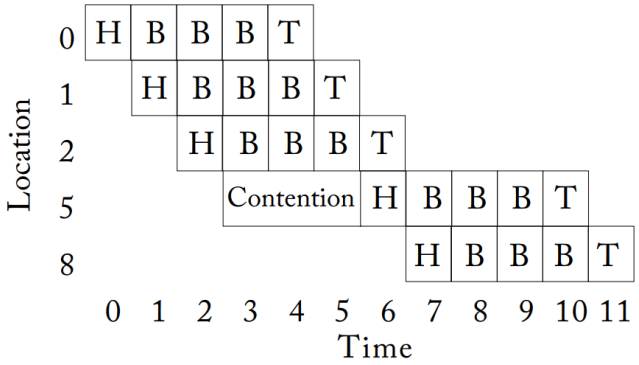


图 2-6 VCT 流控机制

以 VCT 为流控的网络中的数据传输呈现“头探路，体尾跟随”的特点，VCT 流控的特点决定必须保证网络中的每个缓冲能够容下所有数据包，即$n > max_len(packet)$，在这种情况下，当一个头微片由于下游缓冲空间不足而被阻塞在当前路由器时，该数据包的其他微片能够顺利“跟上来”，直到所有微片传入当前路由器内的缓冲，从而保证了上游节点的链路通畅。

2.3.2 基于数据微片的流控机制

基于数据微片的流控机制以微片为单位进行网络资源的分配。

Wormhole 是一种典型的基于数据微片的流控机制，它规定每个微片都根据下游路由器的输入缓冲是否有空位来决定是否传输至下游。图 2-7 展示了一个 Wormhole 的传输案例，假定每个路由器输入端口的缓冲深度为 2，当头微片在时刻 1 至时刻 2 遭到阻塞时，由于路由器 1 中的缓冲空位不足，两个体微片和尾微片只能停滞在路由器 0 中。直到头微片在时刻 5 顺利传输至路由器 2，之后的微片才能进入路由器 1 中。Wormhole 流控进一步减小了数据调度的粒度，从而打破了 VCT 流控中对缓冲容量的要求，使得路由器可以基于更小的缓冲来实现。另外，相比 VCT，Wormhole 提升了头微片的前进速度，但同时增大了数据的分散程度，增大了对网络资源的占用：如果一个较长的数据包在网络中传输并且遭到阻塞时，它可能会阻塞在多个节点和多条链路，使得这些链路的带宽不得被其他数据包使

用，图 2-7 中的灰色区域展示了由于这种链路占用而产生的带宽浪费。

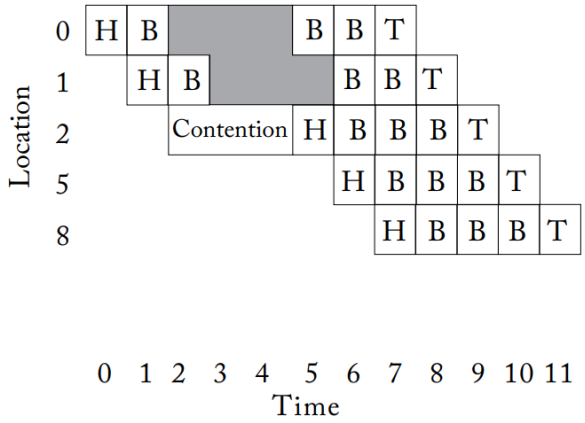


图 2-7 Wormhole 流控机制

2.4 片上网络路由器微架构

路由器是 NoC 的核心部件，负责数据的存储和交换。一个路由器的微架构是由 NoC 的拓扑形态、路由算法和流控机制共同决定的，但一般而言，所有的路由器可以通过一种基础的微架构框架来实现，对于不同的拓扑、路由算法和流控机制，只需修改该架构中相应组件的逻辑功能，在这种架构中，为了保证较高的时钟频率，多级功能的串联需要使用多级流水线打断，每一个流水级只负责一个具体功能，这种路由器称作多周期路由器。

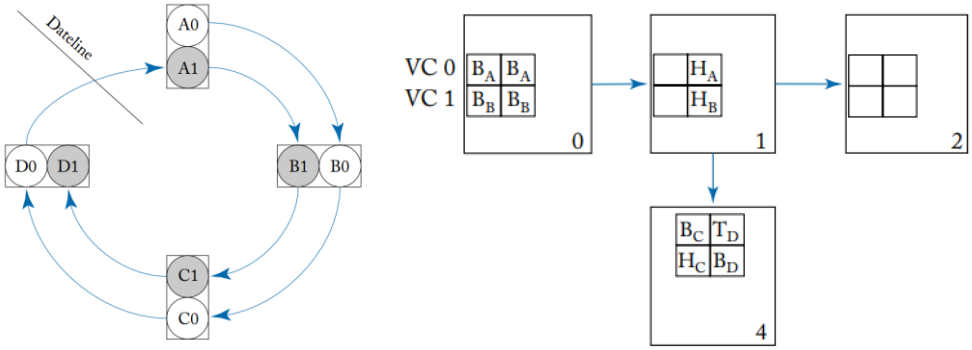
为了实现更高速的数据传输，研究学者们在单周期路由器的实现上付出了巨大的心血，XY 维度切分路由器的出现使得 NoC 的研究进入了一个新的时期，它基于 Mesh 拓扑与 XY 维路由算法，将路由器切分为 X 和 Y 两个维度单元，每个单元负责一个维度的数据传输，实现了 XY 两个维度的解耦以及每个维度上的单周期路由，以更低的面积成本取得了更高性能的数据传输。

2.4.1 虚通道技术

虚通道（Virtual Channel）指网络中虚拟的数据通道，多个虚通道的数据共享同一总线带宽，在路由器中被分配在不同的缓冲中。

虚通道最早是为了解决网络的环形死锁（Dead Lock）问题而被提出，当传输的数据在网络中构成了一个环，形成了环形依赖，就会造成环形死锁，这种死锁会导致网络的瘫痪。图 2-8(a)展示了使用两个虚通道解决环形死锁问题的一个案例，它规定数据只能按照指示线路传输，并在 Dateline 处使数据从一个虚通道（D0）迁移到另一个虚通道（A1），从而解除了数据的环形依赖，避免了环形死锁。

之后，研究学者们发现可以利用虚通道解决 head-of-line blocking 问题，head-of-line blocking 指在网络的流控中，一个数据包的阻塞会导致其后的所有数据包不能行进，即使这些数据包不会阻塞。在这种情况下，可以将数据分布在不同的虚通道内，多个虚通道共享同一输出链路，当一个虚通道的数据包阻塞时，其余虚通道的数据包可以顺利行进。



(a)基于虚通道的环形死锁避免;　　(b)基于虚通道解决 head-of-line blocking 问题;

图 2-8 虚通道技术的常见应用

2.4.2 经典路由器微架构

图 2-10 展示了一种经典的五端口路由器微架构。该路由器将五个输入端口传来的数据分配至五个输出端口。其内部设有输入缓冲（Input Buffers）、路由计算单元（Routing Computation）、虚通道分配（VC Allocator）、交叉开关（Crossbar）和链路分配单元（Switch Allocator）等。

输入缓冲负责输入数据的存储与缓冲，路由计算单元负责实现路由算法并计算数据的输出端口，虚通道分配负责将不同虚通道的数据分配至相应的输入缓冲中，交叉开关中设置了所有可能的输入-输出链接，并在链路分配单元的控制下进行数据的分配与链路的仲裁。

图 2-9 经典路由器流水线结构

一个经典的路由器流水线包括路由计算、链路分配、交叉传输和链路传输四个流水级，由于在一个数据包的传输过程中，路由器可以将根据头微片中的信息计算

得到的路由结果和链路分配结果保存下来，因此体微片和尾微片的传输只需要交叉传输和链路传输两个过程。

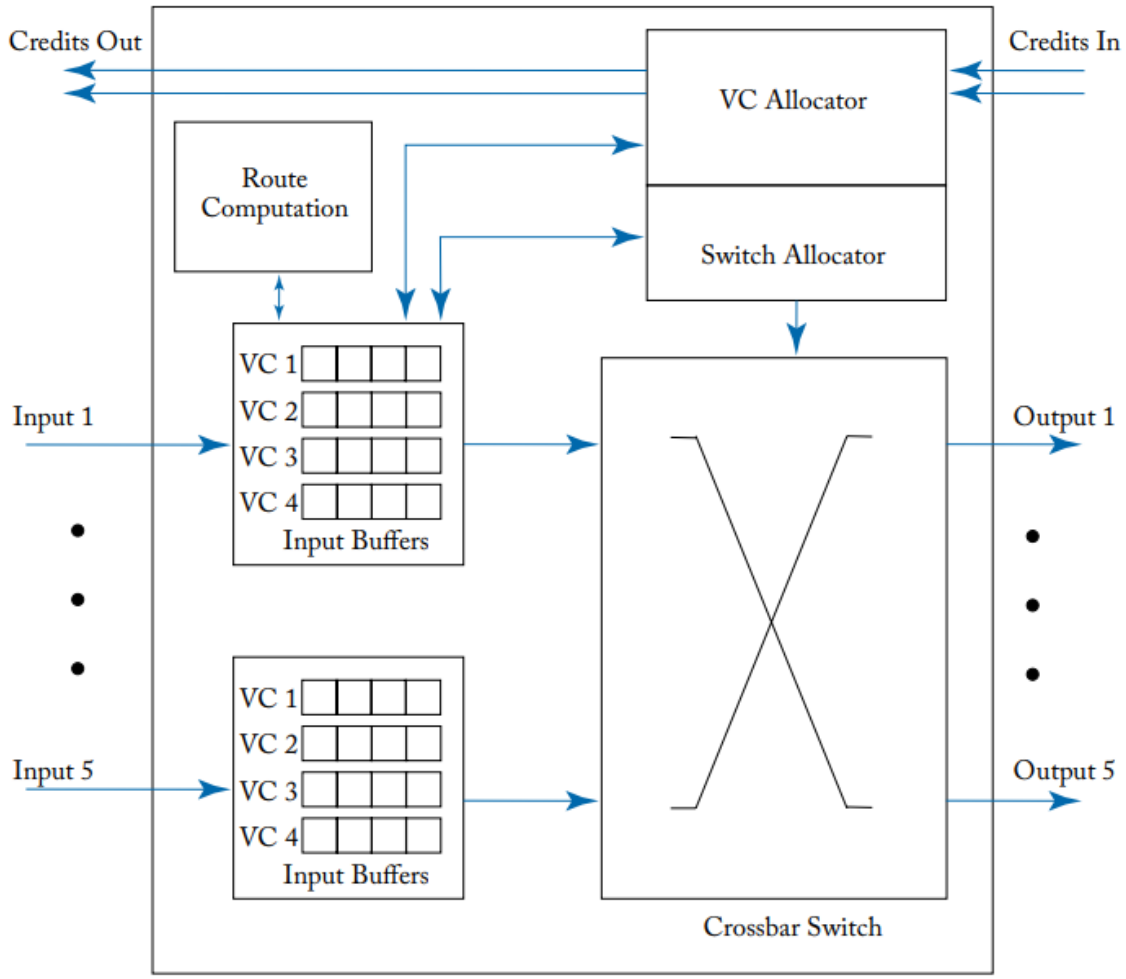


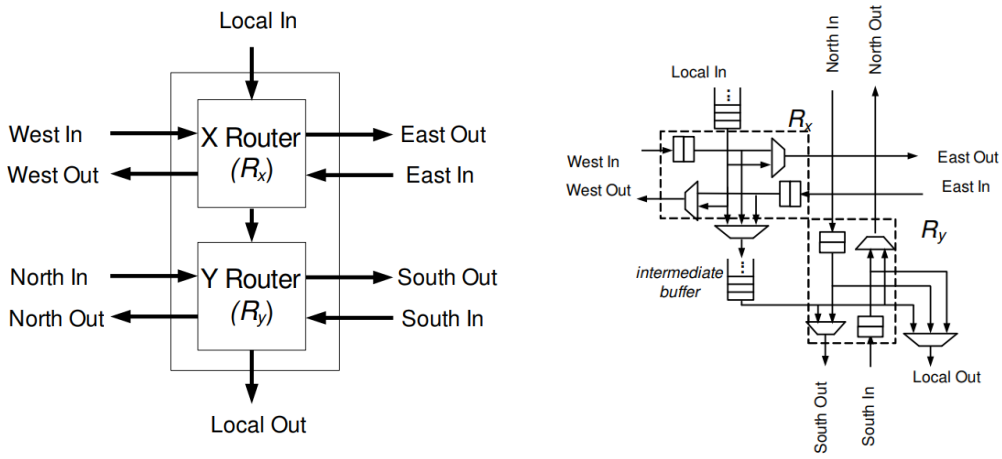
图 2-10 经典路由器微架构

2.4.3 维度切分式路由器微架构

维度切分式路由器是一种针对 Mesh 网络拓扑和 XY 维路由算法的特定应用的路由器，由 John Kim 在“Low-Cost Router Microarchitecture for On-Chip Networks”一文[27]中提出，它由 X 路由器和 Y 路由器两个子路由器构成。

一个完整的五端口路由器包括东、西、南、北和本地五个端口，每个端口包含一组输入总线和一组输出总线。将完整路由器划分的依据是将不同的维度划分开，即将东西方向的 X 维度划分至子 X 路由器管理，而将南北方向的 Y 维度划分至 Y 子路由器管理，另外，在 X 子路由器和 Y 子路由器之间设置一条单向的数据通路，以允许数据从 X 维度传输至 Y 维度。这种对功能切分降低了每个子路由器中输入

缓冲、数据分配与链路仲裁的逻辑复杂度，实现了在较高频率下每个维度的单周期运行，极大地降低了数据传输延迟、提升了数据传输速度。



(a)维度切分式路由器大致结构;　　(b)维度切分式路由器详细结构;

图 2-11 维度切分式路由器

图 2-11(b)展示了维度切分式路由器的详细结构。对于 X 子路由器，它将本地、西和东三个输入端口的数据分配至中间、东和西三个输出端口，其中，中间输出端口对本地、东和西三个输入端口的数据进行仲裁，西输出端口对本地和东两个输入端口的数据进行仲裁，东输出端口对本地和西两个输入端口的数据进行仲裁。Y 子路由器与之类似。当数据从 X 维度传输至 Y 维度时会耗费一个额外时钟周期，但这种情况在一次基于 XY 维路由算法的路由过程中只出现一次，对系统性能影响甚微。

2.5 本章小结

本章首先介绍了 NoC 的拓扑结构，主要讨论了以 Mesh 为代表的直接拓扑和以 Butterfly 为代表的的间接拓扑，并采用一定的性能指标对所提到的拓扑结构进行了衡量和对比。然后指出数据包是 NoC 数据交换的最小单位，并介绍了数据包的一般格式，并以 XY 维路由为例简要介绍了 NoC 的路由算法。接着介绍了 NoC 的流控机制，主要讨论了基于数据包的流控机制 Virtual Cut Through 与基于数据微片的流控机制 Wormhole，并对两者性能与成本进行了简要对比。最后介绍了 NoC 的路由器微架构，主要讨论了常见的虚通道技术、经典路由器微架构以及单周期的维度切分式路由器微架构，并指出维度切分式路由器相比传统经典路由器具有更低的数据传输延迟与成本开销。

第三章 封装级网络设计

实现面向 Chiplet 集成的 NiP 的首要目标是实现多个 Chiplet 间通信的统一、高效以及较强的系统可扩展性。Kangmin Lee 等人在其提出的 NiP 架构中使用 NoC 互连多个功能单元，然后将多个 NoC Die 通过片间总线互连起来，实现了数据的统一传输。然而，他们将功能单元定义为 NoC Die 的内部模块，使其与 NoC 形成了封闭且固定的 SoC，限制了系统的可扩展性；其次，他们没有针对数据的跨 Die 传输作出架构级的优化。

本文在上述 NiP 架构的基础上做出改良，提出了一种面向 Chiplet 集成的新概念 NiP 架构，在该架构中，多个基于 Mesh 拓扑的 NoC 互连构成 NiP 网络框架，NoC 的每个本地端口通过协议转换电路转换为标准协议接口，功能单元以外部 Chiplet 的形式与标准协议接口连接，系统的通用性和可扩展性得到显著提升。另外，考虑到数据跨 Die 传输时由 NoC 内部多级路由引入的高网络延迟，本文提出了一种旁路通路结构，提升了数据跨 Die 的传输速度。

3.1 封装级网络总体架构

本文提出的 NiP 整体上由多个互连 Die 组成，每个互连 Die 由 NoC、扩展端口控制器和标准协议接口三部分构成，多个互连 Die 之间通过片间总线连接。每个 NoC 均采用 Mesh 拓扑结构、XY 维路由算法和 Wormhole 流控机制，并使用维度切分式路由器实现网络的数据路由。为了实现数据的统一入网、统一传输，本文为 NoC 定义了统一的通信协议，并提出了一种基于实时目的路由器 ID 的路由寻址策略以解决数据的跨片传输问题。

3.1.1 系统总体结构组成

图 3-1 展示了本文提出的由四个互连 Die 组成的 NiP 结构，每个互连 Die 的内部是一个用于数据交换的 NoC，扩展端口控制器（Port Controller）用于互连 Die 的横向扩展和级联，它的一侧与 NoC 的两个网络节点连接，另一侧连接到互连 Die 的外部，通过片间总线，如 LVDS（Low-Voltage Differential Signaling）等将多个互连 Die 串联起来。NoC 的其他网络节点均分别与一个协议转换电路连接，协议转换电路既充当 NoC 的网络接口，又负责实现 NoC 协议与一种专用标准协议（Dedicated Standard Protocol）的协议转换，从而具备连接多种以专用标准协议进行通信的功能设备。

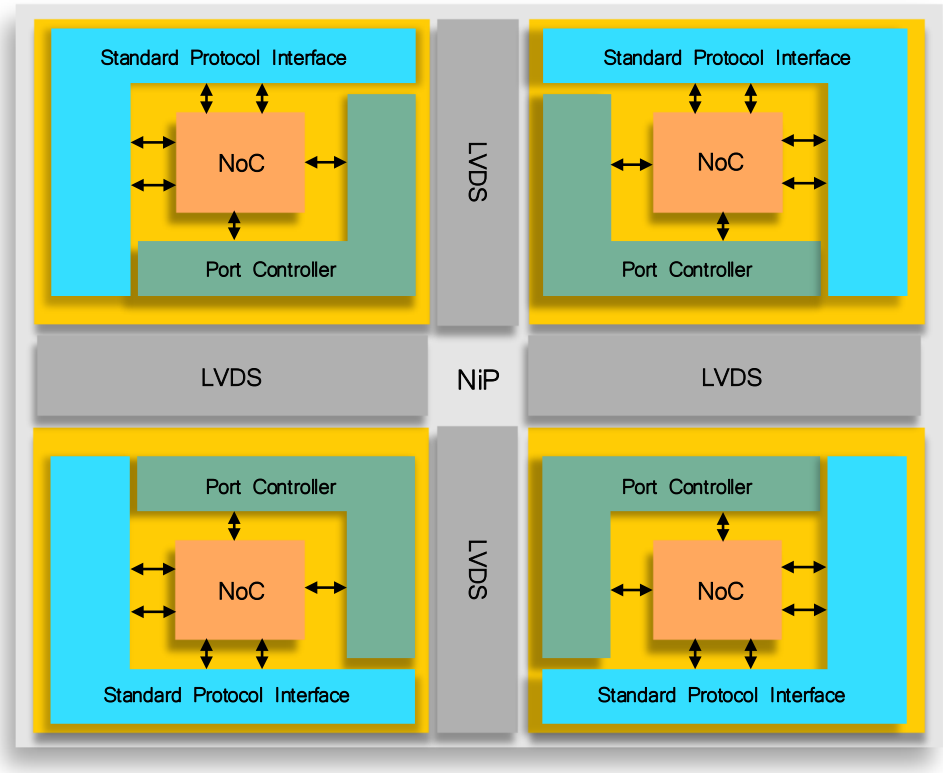


图 3-1 本文提出的 NiP 的总体结构

本文提出的 NiP 结构的核心在于互连，包括互连 Die 与 Chiplet 间的互连和多个互连 Die 之间的互连。

协议转换电路与标准协议接口用于互连 Die 与 Chiplet 之间的互连。为了尽可能保证 NiP 的通用性和可扩展性，可以使协议转换电路覆盖尽可能多的主流的标准协议，如 AXI（Advanced eXtensible Interface）、PCIe（Peripheral Component Interconnect express）、RapidIO、SPI（Serial Peripheral Interface）等，以对外提供丰富的标准协议接口，从而更方便地互连 MPU、DSP、AI（Artificial Intelligence）加速引擎和存储器等功能设备。

扩展端口控制器用于多个互连 Die 之间的互连。片上的 NoC 与片间总线一般具有不同的时钟频率、处在不同的时钟域，设置扩展端口控制器的目的之一是连接不同时钟域，实现数据信号的同步。扩展端口控制器内另设有一条旁路通路，它为需要跨互连 Die 传输的数据提供了一条旁路链路，保证了 Die 间互连的高效性。

3.1.2 片上网络规格定义

本文所实现的 NoC 是一个 5×5 的 Mesh 网络，网络中的所有数据链路包含一个输入通道和一个输出通道，每个通道包含一个数据信号和一组握手信号，数据信

号位宽为 130bit（2bit 微片类型＋128bit 数据负载），握手信号包括上游设备发出的 valid 信号和下游设备返回的 ready 信号。valid 信号为 1 表示上游设备发出的当前数据总线 data 上的数据为有效数据，反之为无效数据，从而告知下游设备不予接收；ready 信号为 1 表示下游设备已经准备好接收当前数据总线 data 上的数据，反之表示未准备完毕，从而告知上游设备在下一个时钟周期保持当前数据总线 data 上的数据不变。

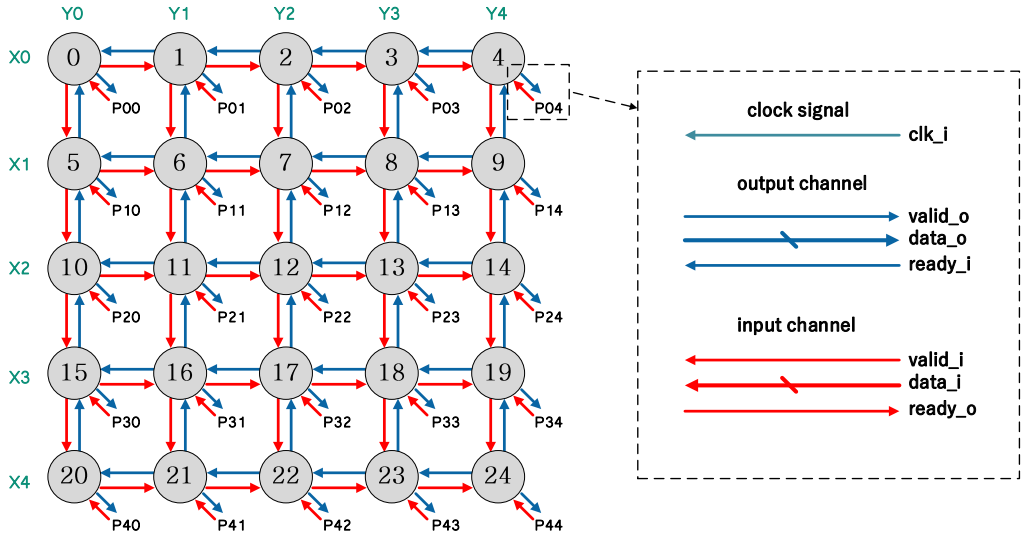


图 3-2 NoC 结构设计及总线协议定义

图 3-3 展示了上述握手信号的工作机制。在 0 时钟周期，valid 为 1，指示 data 为有效数据，同时 ready 为 1，指示数据被准备好接收。在下一个时钟沿,旧的 data（D0）被写入下游设备中，同时上游设备将新的数据（D1）表现在 data 上。在 2 时钟周期，valid 为 0，指示当前 data 为无效数据，下游设备即使返回的 ready 为 1，也对 data 不予接收。在时刻 4 至 5，valid 为 1 而 ready 为 0，指示下游设备未准备好接收 data，于是上游设备保持 data 为 D3 不变，直到在 6 时钟周期 ready 为 1，在下一个时钟沿该数据完成传输。之后的情况与之类似。

总而言之，数据完成一次传输的充要条件为 valid 和 ready 同时为 1。

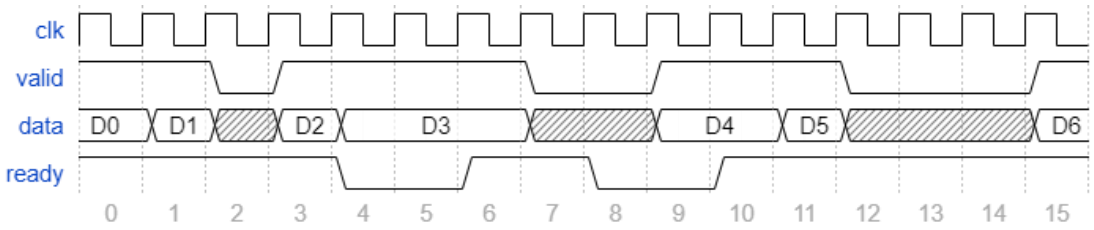


图 3-3 valid 和 ready 握手信号工作机制

之前的讨论表明 XY 维路由算法具有简洁高效的特点，而且可以使用低开销的单周期维度切分路由器来实现这一路由算法。而 Wormhole 流控机制相比基于数据包的流控机制具有缓冲深度不受数据包长度制约的优势，因此也可以降低网络的硬件开销，适合在本次设计的原型验证平台——资源有限的 FPGA 上实现。基于上述考虑，本次 NoC 设计采用 XY 维路由算法和 Wormhole 流控机制，并设计维度切分路由器来实现 XY 维路由功能，利用上述 valid 和 ready 握手信号实现路由器间的数据流控。

3.1.3 数据微片格式定义

一般情况下，在对 NoC 数据微片格式进行定义时，通常把与路由计算有关的数据包源节点和目的节点 ID 信息存储在头微片中，路由器的路由计算单元只对头微片进行路由计算，在对头微片分配链路的同时将计算结果寄存至寄存器中，之后的体微片和尾微片直接由该计算结果分配链路。然而，在本次设计中，由于 NiP 包含多个互连的 NoC，因此一个单独的网络的源节点 ID 和目的节点 ID 已经不足以用来表征数据包的起始位置和终止位置，因此这里引入了 NoC ID（CID）用于区分不同的 NoC，并提出实时路由器 ID（RTRID）保证数据的实时传输。使用 SCID 和 SRID 分别表示数据发送节点从属的网络和路由器，DCID 和 DRID 分别表示数据接收节点从属的网络和路由器，并使用 RTRID 表示数据包在当前 NoC 中的实时目的节点，它始终被用于网络中路由器的路由计算。

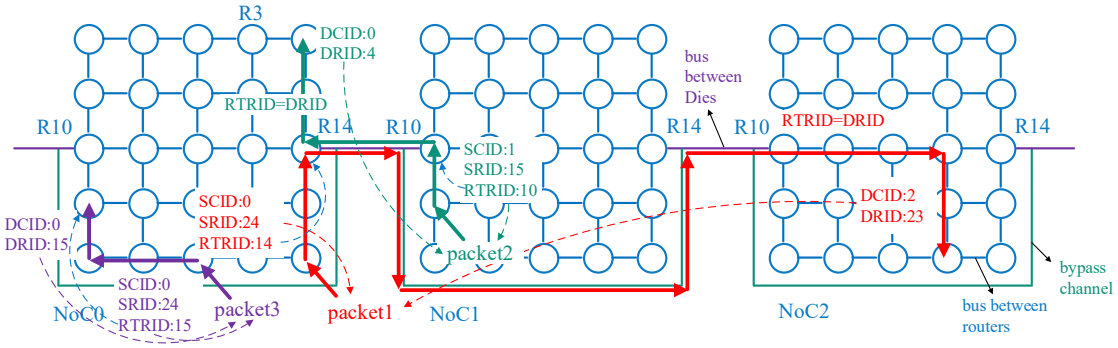


图 3-4 数据包跨 Die 传输方式

图 3-4 展示了数据在 3 个级联的 NoC 中的传输方式。数据包 1 从 NoC0 的 24 号路由器出发（R24），期望传输至 NoC2 中的 R23。当数据包 1 被发送时，NoC0 中的 R24 的网络接口将相应的源节点 ID（SCID 和 SRID）和目的节点 ID（DCID 和 DRID）存放至该数据包的头微片中，另外，由于该数据包需要跨 Die 传输，为了告知该数据包应从源 NoC（NoC0）的哪个路由器节点离开，需要提前将 R14 的

ID 赋值给 RTRID。数据包从 NoC0 的 R14 离开后进入 NoC1，由于该数据包的目的 NoC 并非 NoC1，所以它通过 NoC1 的旁路通路直接传输至 NoC2，当 NoC2 的扩展端口检测到该数据包携带的 DCID 与当前网络 ID 相等时，允许数据包入网，同时将数据包头微片中的 DRID 赋值给其 RTRID，以供数据包在目的 NoC（NoC2）中路由使用。当一个数据包在 NoC 内部传输时，发送节点的网络接口在设置其源/目的节点 ID 信息时需要将其 DRID 赋值给 RTRID，这时 DRID 不起实际作用。

基于以上的讨论，本次设计定义数据包微片格式如图 3-5。在该格式中，每个微片使用其高 2bit 编码其微片类型：如 00 为头微片、01 为体微片、10 为尾微片。头微片中存放有所有与数据传输和路由有关的 ID 信息，其中路由器 ID 的高位和低位分别表示其 X 坐标和 Y 坐标。体微片中除高 2bit 之外，全部用于存放有效数据负载。尾微片中除高 2bit 之外，其余位均为无效数据，即尾微片仅用于标记数据包的终止位置。

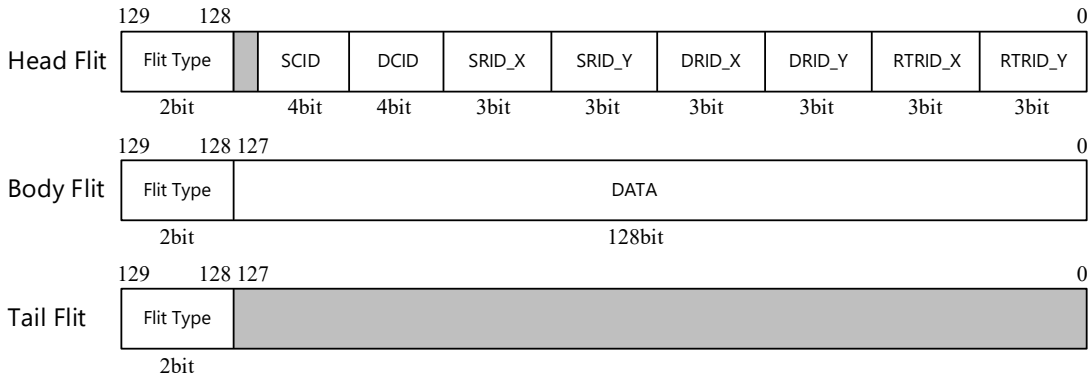


图 3-5 数据微片格式定义

3.2 路由器设计

本次设计采用维度切分式路由器，在 John Kim 提出的路由器微架构的基础上，重新对其结构进行了功能划分，将路由器划分为多个输入分配控制单元和多个输出仲裁控制单元的组合，并在输出仲裁控制单元中使用矩阵仲裁器（Matrix Arbiter）实现多路输入的仲裁。使用先入先出缓存（FIFO）实现输入端口的缓冲，并提出了一种面向 valid，ready 握手流控的 FIFO 包装方法。

3.2.1 路由器总体结构设计

路由器由 X 路由器和 Y 路由器两个子路由器组成，X 路由器包含西输入、东输入和本地输入三个输入端口以及西输出、东输出和中间输出三个输出端口，Y 路由器包含北输入、南输入和中间输入三个输入端口以及北输出、南输出和本地输出

三个输出端口，X 路由器的中间输出端与 Y 路由器的中间输入端相连。输入分配控制单元（alloc）负责缓存输入数据，并将缓存的数据分配至多路；输出仲裁控制单元（Link）负责将多路输入仲裁至一路输出。所有的输入分配控制单元和输出仲裁控制单元之间通过握手信号进行数据确认。

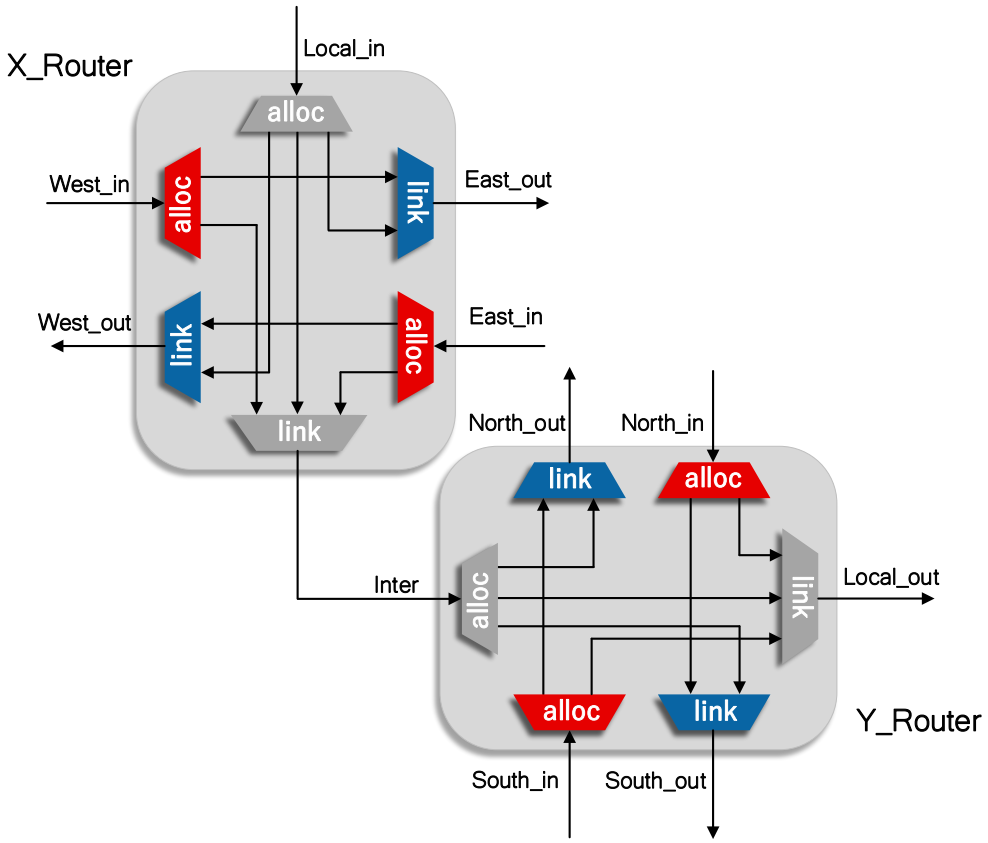


图 3-6 本次设计采用的路由器结构

3.2.2 面向流控的 FIFO 设计与包装

FIFO，即先入先出缓存，在数字系统中常用作数据缓冲，用于连接流量不同或不可控的系统区域，或连接不同的时钟域。它的一端在写使能信号的控制下接受数据输入并加以存储，另一端在读使能信号的控制下读出数据，实现数据的暂存与缓冲。FIFO 一般向外界提供满标志（full）和空标志（empty）用于指示其内部的存储状态。FIFO 有同步（Synchronous）和异步（Asynchronous）之分，前者的写接口和读接口共享同一个时钟，后者的写接口和读接口处在不同的时钟域。

在本次设计中，路由器的输入端口缓存需要使用同步 FIFO 来完成，而互连 Die 的片间互连需要使用异步 FIFO 来连接片内和片间两个时钟域，然而，由于异步 FIFO 也可以被当做同步 FIFO 使用，并且 FIFO 的实现并不会成为本次 NiP 设计的

主要硬件开销，因此，为了提升代码的复用性，本文仅设计了异步 FIFO 原型。

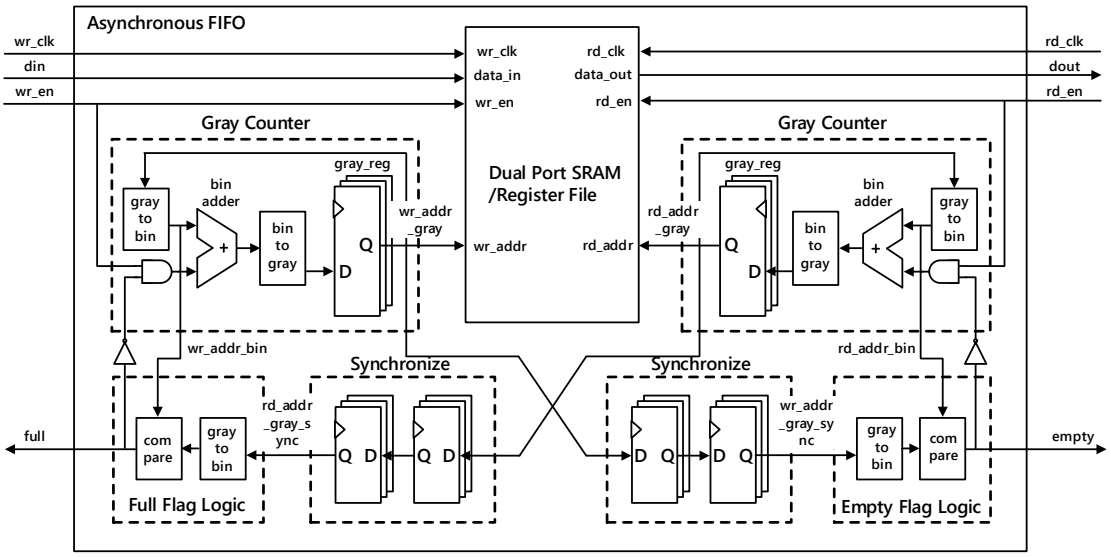
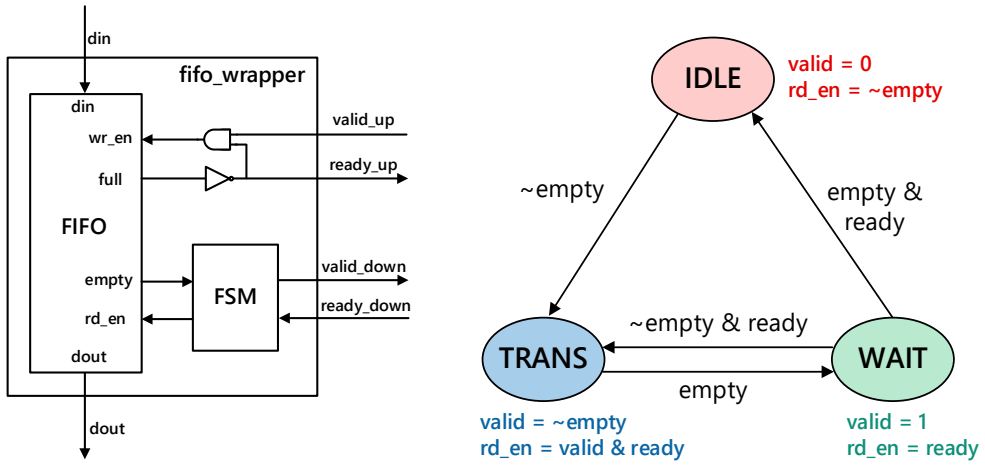


图 3-7 异步 FIFO 设计框图

在该异步 FIFO 中，使用一个双端口 SRAM（Static Random Access Memory）或寄存器堆（Register File）作为存储器件，每当写使能有效时，SRAM 的写指针自增 1，独使能有效时，SRAM 的读指针自增 1，任何指针计满时，进入下一个轮次，从 0 开始计数。通过比较读写指针的轮次 ID 与值的大小可以生成相应的空满标志：当两指针值相等时，若两指针处在同一轮次，使能空标志；若处在不同轮次，使能满标志。然而，读指针和写指针处在不同的时钟域，不能直接进行比较，需要将一方通过同步电路同步后再与另一方比较，本设计中的同步电路采用双寄存器结构实现。另外，由于多 bit 信号跨时钟域容易引发时序问题，本设计中采用格雷码（Gray Code）对地址编码，并设计相应的格雷码计数器以实现指针的自增。

由于基于 SRAM 实现，FIFO 的读逻辑存在一个时钟周期的延迟，即当读使能有效时，在下一个时钟周期期望读出的数据才表现在输出总线上，本文提出了一种基于 FSM（Finite State Machine，有限状态机）的方法用于解决该问题。然而 FSM 的引入会使 FIFO 的例化方式变得非常繁琐，为了方便设计，本文又提出了一种 FIFO 包装方法，即将上述 FSM 以及上游流控逻辑包装在一个 wrapper 内，使其对上游和下游只表现数据信号 data 以及两个握手信号 valid 和 ready，这种面向握手流控的 FIFO 包装方法既解决了 FIFO 输出时钟周期延迟问题，又方便了 FIFO 的例化。本文之后的模块设计均使用上述 wrapper 实现 FIFO 的调用。



(a) FIFO 包装结构; (b)使用 FSM 解决 FIFO 输出时钟周期延迟问题;

图 3-8 面向握手流控的 FIFO 包装方法

3.2.3 输入分配控制单元设计

输入分配控制单元负责将一路输入根据路由计算结果分配至n个输出端口。主要由 FIFO、路由计算单元和选择逻辑三部分组成，三者在 FSM 的控制下共同工作。FIFO 负责缓冲输入数据，路由计算单元根据 FIFO 输出总线上数据头微片中的 RTRID 进行路由计算，计算结果以 n bit 独热码的形式表现在 dst_0 至 dst_n-1 n 个信号上，用于表示当前激活哪个输出端口。由于体微片和尾微片需要根据头微片的计算结果进行端口分配，因此需要使用 FSM 控制头微片路由计算结果的寄存和刷新。FSM 将最终的端口使能信号 en_0 至 en_n-1 输出给选择逻辑，选择逻辑对输出端口的 n 路 valid 输出和 n 路 ready 输入进行分配与选择。

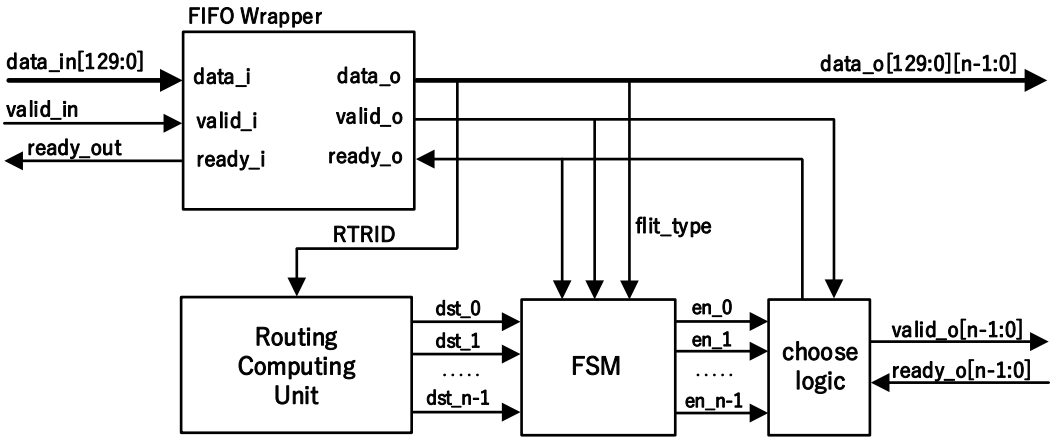


图 3-9 输入分配控制单元设计框图

路由计算单元是 XY 维路由算法的执行单元，它将头微片中的 RTRID 和本地路由器的 ID（LRID）进行比较，根据比较结果为输入数据分配输出端口。

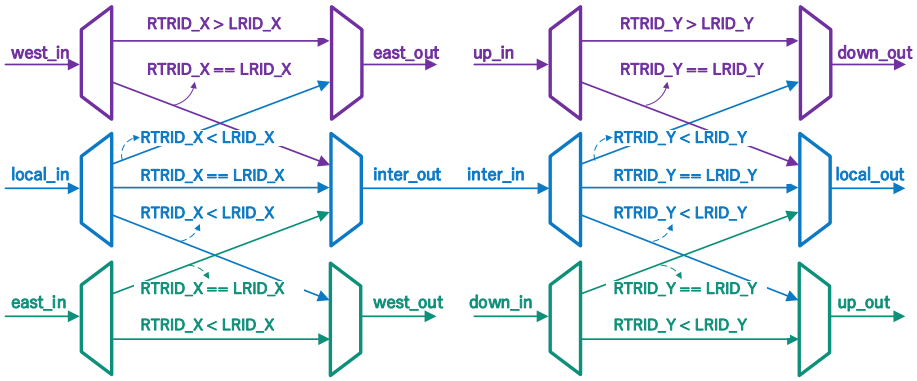


图 3-10 路由计算单元计算方法

3.2.4 输出仲裁控制单元设计

输出仲裁控制单元将 n 路输入仲裁至一路输出，其核心功能组件是一个 n 阶的矩阵仲裁器。n 路输入的 valid 信号作为各输入端口的传输请求（req）输入至矩阵仲裁器，得到矩阵仲裁返回的 n 个 grt 信号作为对每个请求的允许响应。由于一个数据包的传输不允许被打断，因此需要使用 FSM 对头微片仲裁所得的允许响应进行寄存和刷新，从而得出多路复用器的选择信号 CHO，CHO 负责所有输入端口的 valid、ready 和 data 信号的选择和分配，并在每个数据包传输过程中保持不变。

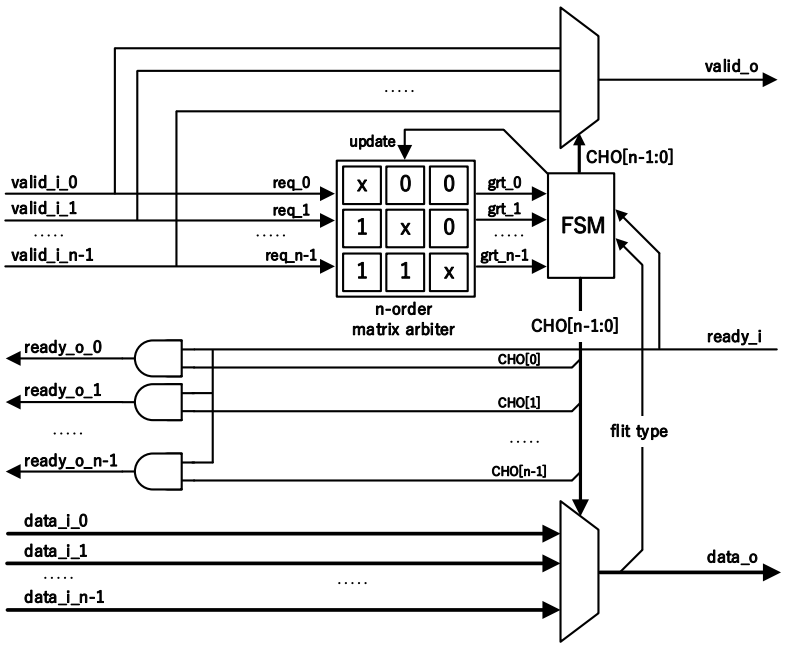


图 3-11 输出仲裁控制单元设计框图

3.2.5 矩阵仲裁器设计

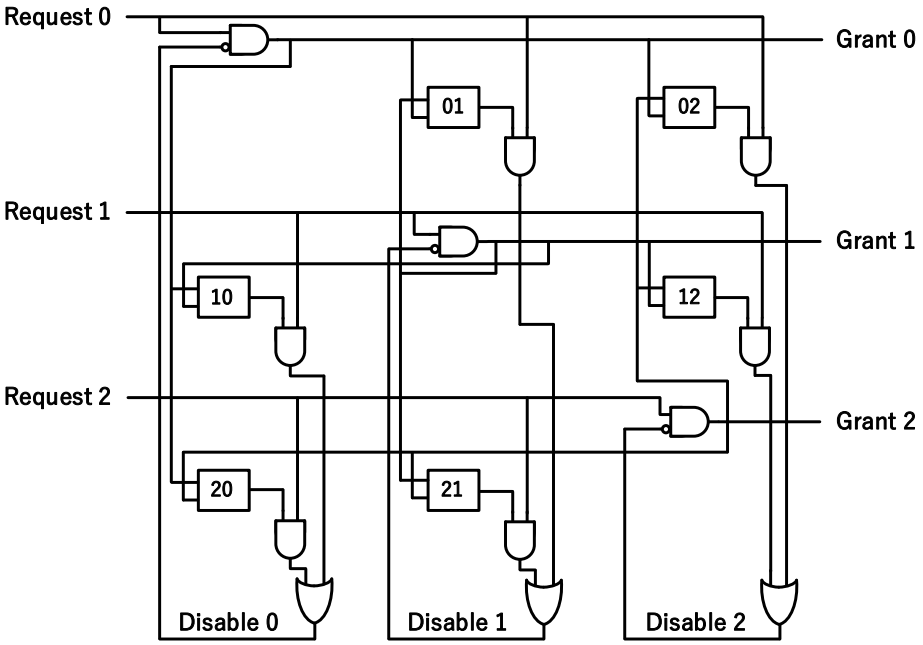


图 3-12 3 阶矩阵仲裁器设计电路图

本文设计的矩阵仲裁器用于 n 路请求的仲裁，它接受可配置位宽的请求向量（即 n 可配置），并输出相应的允许向量用以指示仲裁胜出者。在矩阵仲裁器中，Request 0-n-1 为 n 路输入请求信号。并设有优先级权重矩阵 w(00)-w(n-1 n-1)，以寄存器的形式存储着当前的优先级信息，w(ij)=1 表示第 i 路请求的优先级高于第 j 路请求，wij=0 表示第 i 路请求的优先级低于第 j 路请求，当 i=j 时，此位被省略。Grant 0-n-1 为仲裁器输出的 n 路允许信号。对于第 k 路的允许输出 Grant k，其产生机制为 Request k 和（~Disable k）的逻辑与。Disable k 的逻辑表达式为：

$$
\begin{aligned}
Disable\ k \quad = \quad & Request\ 0\ \&\ w(0)(k) \quad | \\
& \dots\dots \quad | \\
& Request\ k-1\ \&\ w(k-1)(k) \quad | \\
& Request\ k+1\ \&\ w(k+1)(k) \quad | \\
& \dots\dots \quad | \\
& Request\ n-1\ \&\ w(n-1)(k)
\end{aligned}
$$

当仲裁器使能第 k 路允许信号后，更新仲裁优先级，更新依据为将第 k 路的优先级设置为最低，即复位 w(kj)(0 <= j < n)并置位 w(ik)(0 <= i < n)，如此反复，实现轮询仲裁。由此可以看出矩阵仲裁器逻辑简洁、易于控制和扩展的优势。

3.3 扩展端口控制器设计

片间总线支撑着互连 Die 之间的互连，而 NiP 内部的 NoC 与外部的片间互连总线通常处在不同的时钟域、具有不同的总线位宽。片内集成程度高、时钟频率高、位宽较大，而片间集成程度低、线路较长、时钟频率相对较低，另外，为了减少并行总线传输的串扰、节省互连 Die 的 I/O 引脚，片间一般采用较小的位宽。

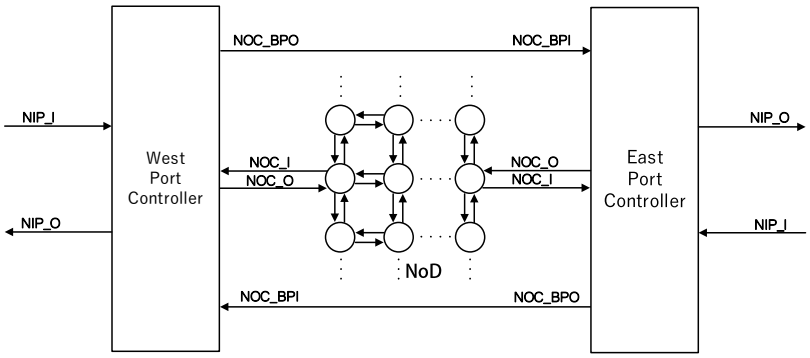


图 3-13 扩展端口控制器与 NoC 的连接方式

本次设计将片外位宽定为片内的 1/2，即 65bit，并设计扩展端口控制器（Port Controller）用于连接片内和片外两个系统区域。扩展端口控制器不仅管理着 NoC 与外界的通信，还管理着互连 Die 内部的两条旁路（Bypass）通路。扩展端口控制器由旁路控制器（Bypass Controller）和同步控制器（Synchronization Controller）构成，其结构关系如图 3-14 所示。

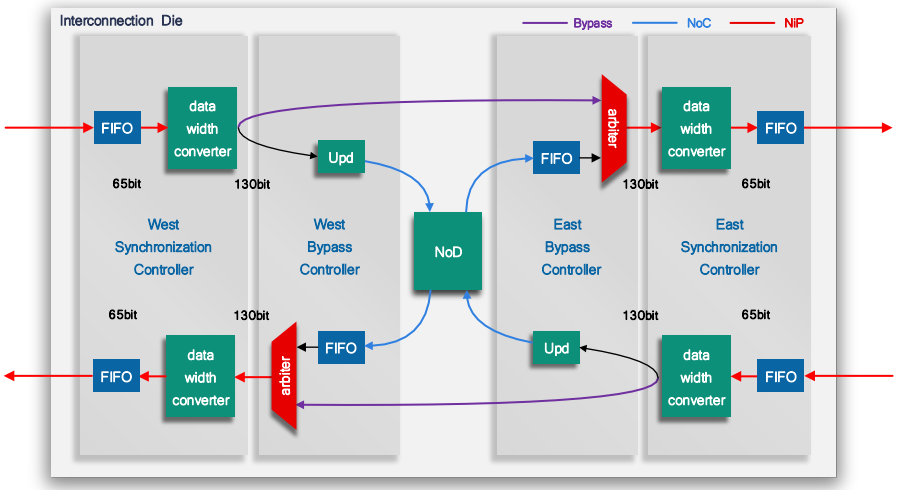


图 3-14 互连 Die 中各模块间的连接关系

3.3.1 旁路控制器设计

旁路控制器负责管理互连 Die 内的两条旁路通路。旁路控制器的外部输入通

道接收来自互连 Die 外部的数据包，并通过比较该数据包的 DCID 与本地 NoC 的 ID，决定将该数据包送入 NoC 还是旁路通路：当 DCID 与本地 NoC 的 ID 不相等时，说明该数据包需要跨过本地 NoC 传输，则为数据包分配通往旁路通路的输出端口，反之，则分配通往 NoC 的输出端口。

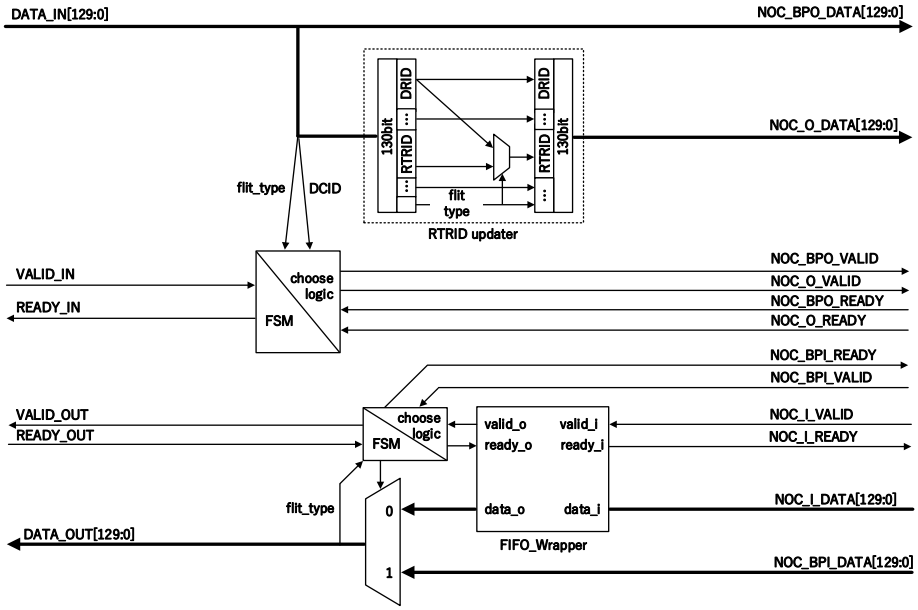


图 3-15 旁路控制器设计框图

之前的讨论提到，当数据包进入目的 NoC 时，需要将其头微片中的 RTRID 替换为头微片中的 DRID，因此，在所设计的旁路控制器的外部输入通道的 NoC 数据输出端口处增加一个 RTRID 更新单元用于 RTRID 的替换。旁路控制器的外部输出通道接收来自 NoC 和旁路通路的数据，并将其仲裁到一路输出上。为了进一步提升数据跨片传输效率，规定旁路输入的优先级高于 NoC 输入，即当两个输入端同时有数据请求时，优先传输旁路通路中的数据，而将 NoC 的输入数据存储至 FIFO 中等待，当旁路通路数据传输结束时，再对存储 FIFO 中的数据进行传输。

3.3.2 同步控制器设计

本文设计的同步控制器设有异步 FIFO 和位宽转换电路两个功能部分，异步 FIFO 用于连接片外和片内两个时钟域，而位宽转换电路用于 130bit 和 65bit 位宽之间的转换。当片内数据向片外传输时，位宽转换电路将 130bit 的数据切分成 65bit 分两次传输，当片外数据向片内传输时，位宽转换电路将连续两次传输的 65bit 数据拼接成 130bit。由 FSM 统一协调位宽转换过程和握手信号的生成。

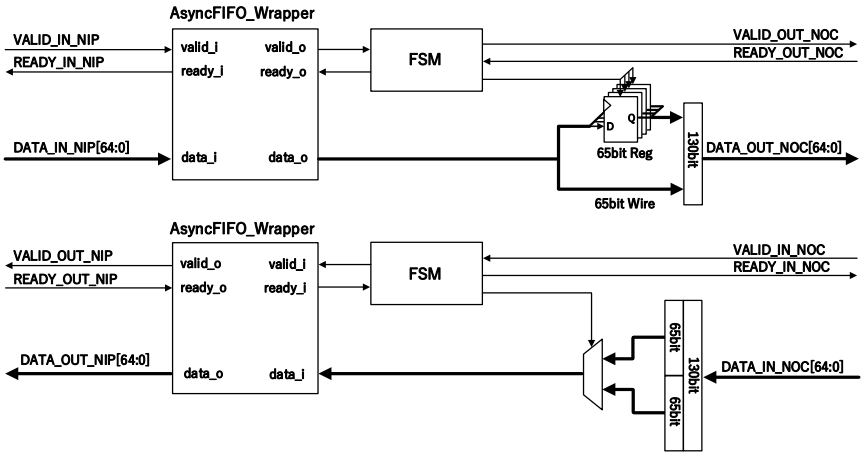


图 3-16 同步控制器设计框图

3.4 封装级网络的搭建

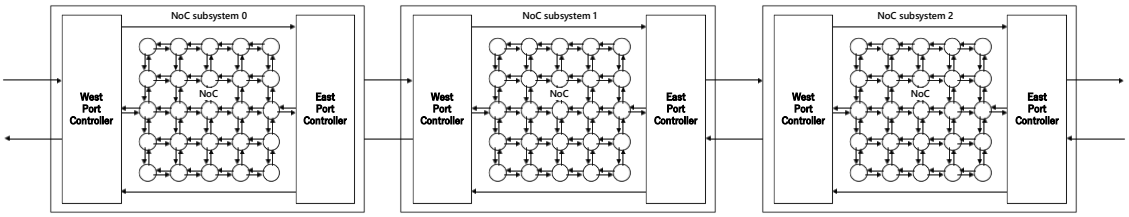


图 3-17 封装级网络系统框图

将 NoC 和扩展端口控制器结合起来，构成一个 NoC 子系统，再把 3 个 NoC 子系统按照图 3-17 所示的方式级联起来，便完成了本次设计的封装级网络的寄存器传输级（Register-Transfer Level，RTL）原型的搭建。在该原型中，每个 NoC 的 10 号和 14 号路由器的本地端口分别与西、东两个扩展端口控制器连接，两个扩展端口控制器的旁路通路的输入、输出端口彼此交错连接，3 个相同的 NoC 子系统通过片间总线相连。

3.5 本章小结

本章主要介绍了 NiP 的设计思路和设计过程。本章首先介绍了本次设计采用的 NiP 的总体架构，接着定义了 NiP 中 NoC 的具体规格以及一种针对于数据跨片传输的数据微片格式。随后介绍了路由器设计的总体结构，并深入讨论了 FIFO、输入分配控制单元、输出仲裁控制单元和矩阵仲裁器的设计细节，指明了本次设计面向代码复用、方便扩展、逻辑简洁的特点。然后具体讨论了扩展端口控制器的设计细节，其中，旁路控制器实现了片内旁路通路的有效管理，同步控制器实现了片内和片外系统区域时钟域和位宽的连接和转换。最后基于以上的模块化设计，进行

模块之间的组合和连接，实现了由 3 个 NoC 子系统组成的 NiP 原型的搭建。

第四章 封装级网络验证

验证（Verification）是数字系统设计流程中不可或缺的一个环节。为了尽最大可能查出和修改本次设计中的逻辑错误、证实设计与预期的相符性、量化分析所设计网络的性能，本文主要从仿真级和板级两个层面对所设计的NiP执行验证工作。

其中，仿真级验证是基于相应的计算机软件、使用特定的编程语言对 NiP 的数据传输进行仿真、对比和量化分析，这种验证方式功能灵活、覆盖面较广，但是只能在 RTL 的逻辑层面进行系统分析；板级验证是将所编写的 HDL（Hardware Description Language）代码下载到可编程逻辑器件，实现系统硬件原型，在此基础上进行实际的数据传输和逻辑控制，这种验证方式将系统的布局布线和晶体管等器件的实际电气特性考虑在内，能够准确地测试设计方案所对应的实际电路的可行性和可靠性，然而由于实际测试案例种类有限、场景单一，这种验证方式很难全面覆盖设计的每一个功能细节。本章将仿真级验证和板级验证相结合，互作补充，对设计进行了充分的分析与验证。

4.1 仿真级验证

数字系统的仿真级验证是一个由特定编程语言（或HDL）、对应方法学以及相应的计算机软件平台组成的集合。随着数字系统的复杂程度不断增大，仿真级验证的语言也不断完善和更迭，从简单的基于 Verilog Testbench（测试平台）的验证、到基于软件编程的 SystemC 的验证、再到基于面向对象（Object Oriented Programming）的 System Verilog 的验证，从硬件逻辑底层到应用处理层，验证的层次不断提高，其功能也在不断扩充、可扩展性不断提升。本节将使用基于 System Verilog 语言的 UVM（Unified Verification Methodology，统一验证方法论）、结合 questasim 仿真软件对所设计的NiP进行功能验证和性能分析。

4.1.1 UVM 介绍

UVM 是一个由 System Verilog 语言、UVM 官方库和统一仿真平台架构方法学组成的集合，它由早期的 OVM（Open Verification Methodology）补充完善而来、于 2011 年 2 月由 Accellera 正式推出，得到了 Synopsys、Mentor 和 Cadence 三家 EDA 厂商的联合支持，是当今芯片市场上最年轻、最富有活力、最流行的芯片验证手段和平台。

UVM 平台是基于 System Verilog 语言编写和运行的，System Verilog 语言为芯

片验证而生，融合了多种计算机编程语言的优秀特征。在硬件层面，System Verilog 进一步扩充了 Verilog 中的数据类型，引入了数组、队列、链表等数据类型，实现了硬件数据的灵活处理以及软硬件接口的完美融合；引入了接口（Interface）类型，实现了对大规模数字系统中复杂接口的统一、简化的管理。在软件层面，System Verilog 完全支持 OOP 编程，用户可以通过类的继承、多态和封装实现测试代码的扩展和复用；提供丰富的随机化及约束函数用于产生测试激励；提供多线程并发和通信机制用于多路信息的并行处理等等。

在 System Verilog 的支持下，UVM 建立了一套完整清晰的验证平台架构，它将验证平台划分为 driver、monitor、agent、scoreboard 和 reference model 等功能组件，每个组件负责一定范围内的工作，并使用一个测试类例化每个功能组件，验证人员将验证需求抽象为一个个具体的程序下发到每一个功能组件中。其次，UVM 提供 phase 运行机制，它为验证流程的每个时间段分配一个 phase，实现了高效的流程管理和多线程并发。此外，UVM 将测试用例中的测试序列抽象为 sequence 类，并将其从测试平台中脱离出来，使验证人员能够做到在不增减功能组件的情况下方便快速地修改测试序列，提高了验证效率。最后，UVM 还提供事务级建模（Transaction Level Modeling，TLM）机制用于简化各组件间的事务通信、提供寄存器模型以方便待测设计（Design Under Test，DUT）的灵活配置等等。

4.1.2 基于 UVM 的系统验证框架

UVM 实现验证灵活性的关键在于将对硬件信号的处理转化为对软件的编程，而在 UVM 定义的验证框架中，实现软硬件连接的功能组件就是 driver 和 monitor，其中，driver 负责把上层软件下发的事务转化为逻辑信号发送至 DUT 的输入端口，以实现对 DUT 的驱动；monitor 负责从 DUT 的输出端口采集逻辑信号并将其转化成事务提交给上层软件，实现对 DUT 的监测。事务（transaction）以类的形式存在，一般来讲，一种总线协议对应一种特定的事务。sequence 是测试事务序列的产生者和发起者，它将产生的事务转交给 sequencer，sequencer 再将事务下发到 driver。在 DUT 的输入端和输出端，分别设置 agent 用于统一管理 sequencer、driver 和 monitor 等组件，根据执行功能的不同将 agent 划分为 active 和 passive 两类，active agent 用于管理 DUT 的驱动，而 passive agent 用于管理 DUT 的监测。

为了判断 monitor 采集的输出事务是否符合预期，需要将其与正确的输出结果作对比。reference model 是 DUT 的软件功能模型，它接受 active agent 的输入事务并计算产生预期的输出事务，该预期输出事务与 passive agent 监测得到的实际输出事务一同递交至 scoreboard，scoreboard 将两者进行对比、量化和分析，并打印出

验证报告，以告知验证人员验证结果。

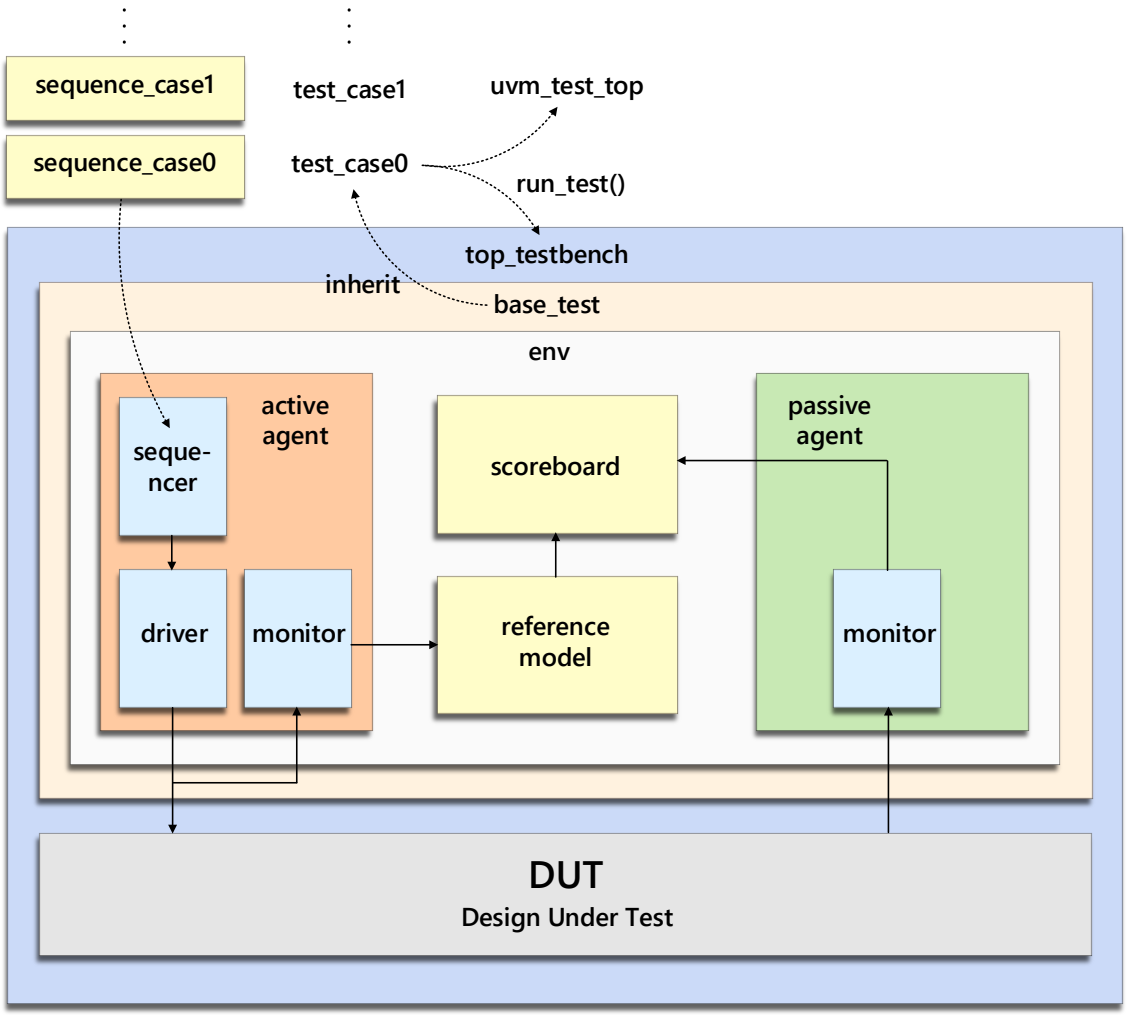


图 4-1 基于 UVM 的系统验证框架

为了统一管理 agent、scoreboard 和 reference model，UVM 定义了 env（environment）组件，每个 env 可以包含多个 agent、scoreboard 和 reference model，UVM 又定义测试基类 base test 用于实现基础的测试需求，每个 base test 中可以包含多个 env。当用户需要使用多个不同的测试案例（test case）测试 DUT 时，这些不同的案例只需从 base test 类继承即可，每个测试案例绑定一个或多个 sequence，实现了代码的高度可扩展。最后，验证人员自定义顶层模块 top_testbench 连接测试环境与 DUT，并调用 UVM 的 run_test 函数开启某个测试案例以启动系统测试。

4.1.3 基于 UVM 的路由器验证

为了验证本次设计的 XY 维度切分式路由器的路由功能，搭建路由器的 UVM 验证环境如图 4-2。

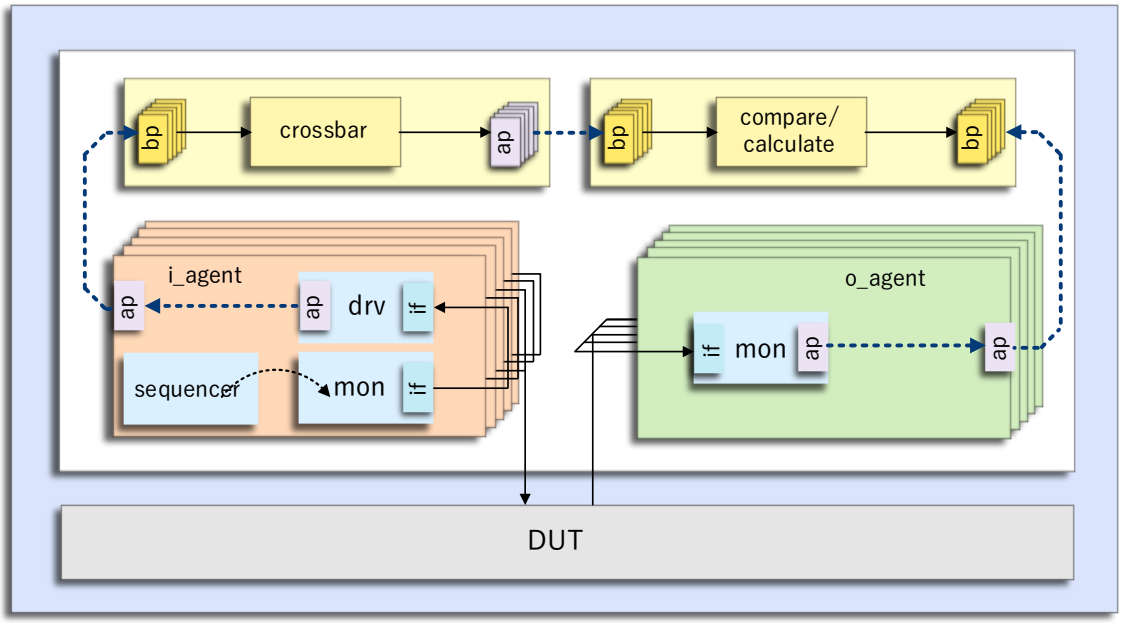


图 4-2 路由器的 UVM 验证环境

本次设计的维度切分式路由器对外表现 5 个输入端口和 5 个输出端口，因此本次 UVM 环境中例化了 5 个相同的 i_agent 和 5 个相同的 o_agent，每个 i_agent 包含一个 sequencer、一个 driver 和一个 monitor，每个 o_agent 中包含一个 monitor，driver、monitor 与 DUT 间的通信基于 system verilog 的 interface 实现。分别设置一个 reference model（下称 model）和一个 scoreboard，5 个 i_agent 分别将其输入事务传输至 model，model 中开启 5 个线程接收事务并将 5 路事务通过一个软件模拟的 5 端口 crossbar 根据其 RTRID 分配到 5 个输出端口。scoreboard 中开启 10 个线程用于分别接收 5 路来自 model 的的期望输出事务和 5 路来自 o_agent 的实际输出事务，并将所有事务压入期望队列和实际队列中。其中，agent、model 和 scoreboard 间的通信均基于 TLM 机制、通过 TLM 中的 analysis port（ap）和 blocking get export（bp）接口实现。测试结束后，在 scoreboard 的 report phase 中将其期望队列和实际队列中的事务一一比较，并将比较结果打印至相应文件。

在本次验证中，设定待测路由器的坐标为（2,2），每个事务在创建时调用随机化函数随机化其 RTRID，然而由于维度切分路由器的链路分配是维度有序的，所以需要对每个输入端口的驱动事务做二次约束，使其 RTRID 只能在有效路径对应的坐标范围内随机，每个输入端口的所有有效路径如图 4-3。

driver 和 monitor 中实现软件事务与逻辑信号之间的相互转化，其信号驱动和信号监测函数均在 UVM 的 main_phase 中实现和执行，多个组件间的 main_phase 通过多个线程并行执行。图 4-4 展示了 driver 和 monitor 的 main_phase 执行流程。

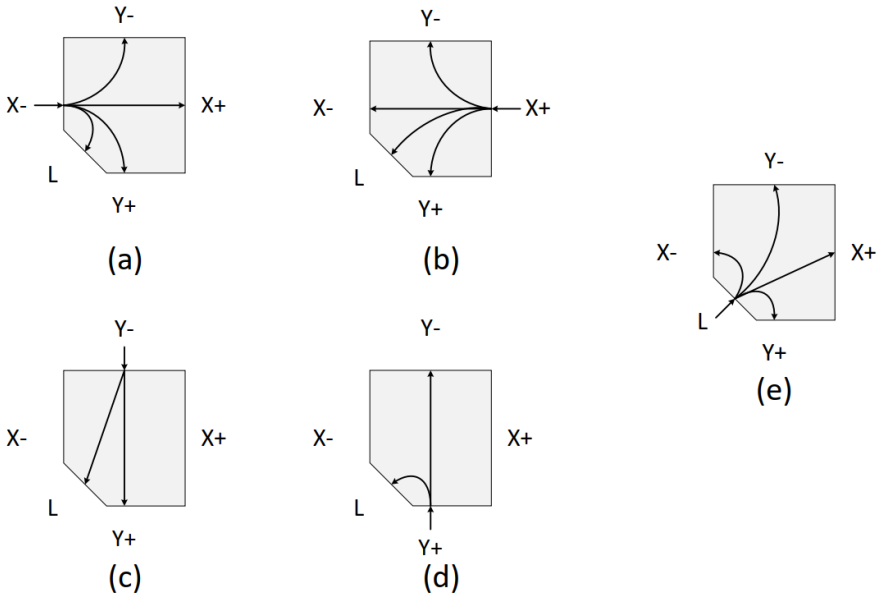


图 4-3 路由器各输入端口有效路径

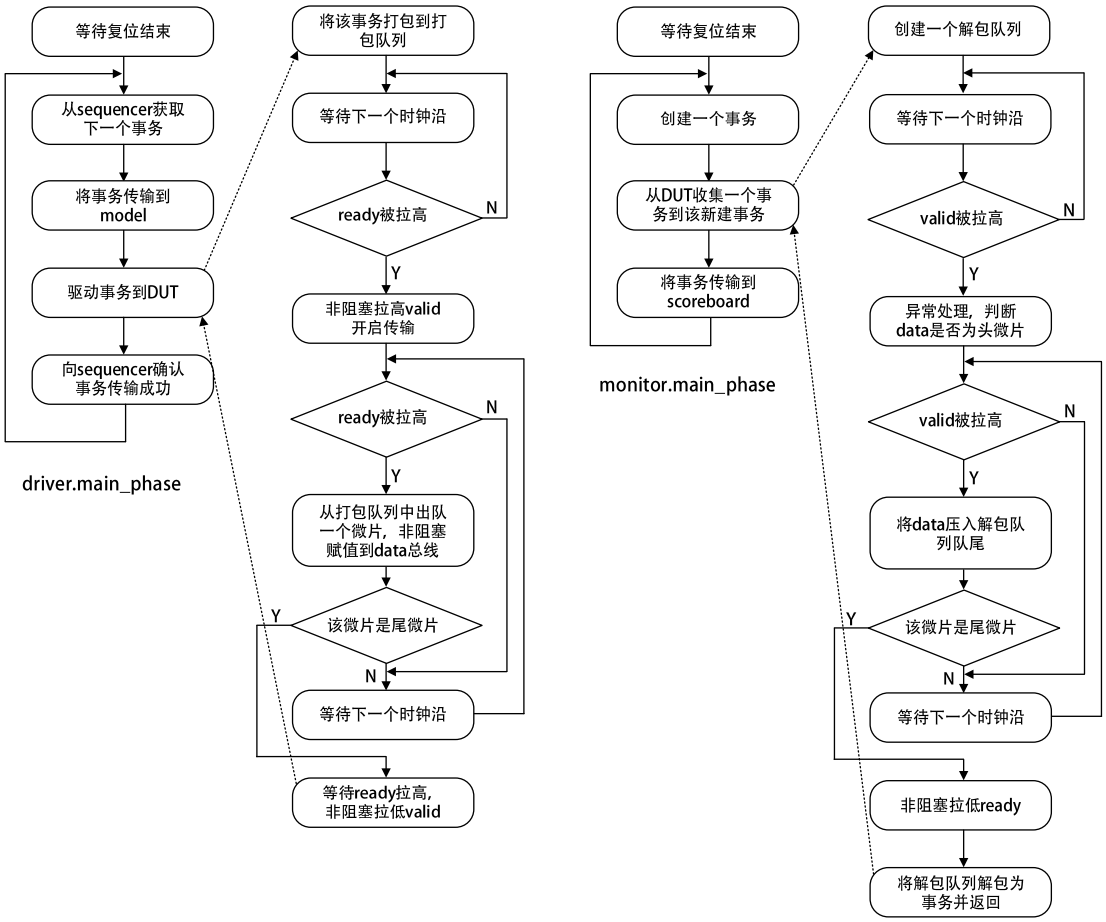


图 4-4 driver 和 monitor 的 main_phase 执行流程

```
`uvm_info("router_scoreboard","\n\n\nstarting west port verification",UVM_LOW)
super.report_phase(phase);
if(actual_queue_w.size() == expect_queue_w.size()) begin
    while(actual_queue_w.size() > 0)
    begin
        temp_trans = actual_queue_w.pop_front();
        for(i = 0; i < expect_queue_w.size(); i++) begin
            if(temp_trans.compare(expect_queue_w[i])) begin
                flag = 1;
                break;
            end
        end
        if(!flag) begin
            `uvm_error("router_scoreboard","packet transfer error")
        end
        else begin
            `uvm_info("router_scoreboard","packet transfer successfully",UVM_LOW)
            flag = 0;
        end
    end
end
else begin
    `uvm_error("router_scoreboard","size not equal")
end
```

(a)scoreborad 事务对比程序;

```
starting west port verification
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
UVM_INFO .\uvm\router_scoreboard.sv(140) @ 8530000: uvm_test_top.env.sb [router_scoreboard] packet transfer successfully
```

(b)scoreboard 事务对比报告;

图 4-5 事务对比验证

验证平台搭建完毕后，编写测试序列，规定路由器每个输入端口循环发送 1000 个受约束的随机数据包。测试结果（图 4-5）显示，所有数据包均能够按照预期完成传输，即路由器能够正常执行路由功能。

图 4-6 展示了路由器的部分测试时序波形，1 处所指为一个由路由器南输入端口传输至北输出端口的数据包，2 处所指为一个由东输入端口传输至西输出端口的数据包，上述两个数据包均在同一个维度内传输，且传输过程均经历了两个时钟周期的延迟，这是因为输入端口处 FIFO 的输入和输出分别需要耗费 1 个时钟周期。3，4 处所指为一个由东输入端口传输至南输出端口的数据包，根据维度切分式路由器的结构特点，该数据包需要从 X 维度的子路由器传输至 Y 维度的子路由器，于是，该数据包首先通过 X 路由器的中间输出仲裁控制单元传输至 Y 路由器的中间输入端口，此过程耗费两个时钟周期（如 3 处所指），之后该数据包在 Y 路由器的南输出仲裁控制单元内与其他数据包产生竞争，竞争失败后等待了一段时间（如 4 处所指）才得以从南输出端口输出。

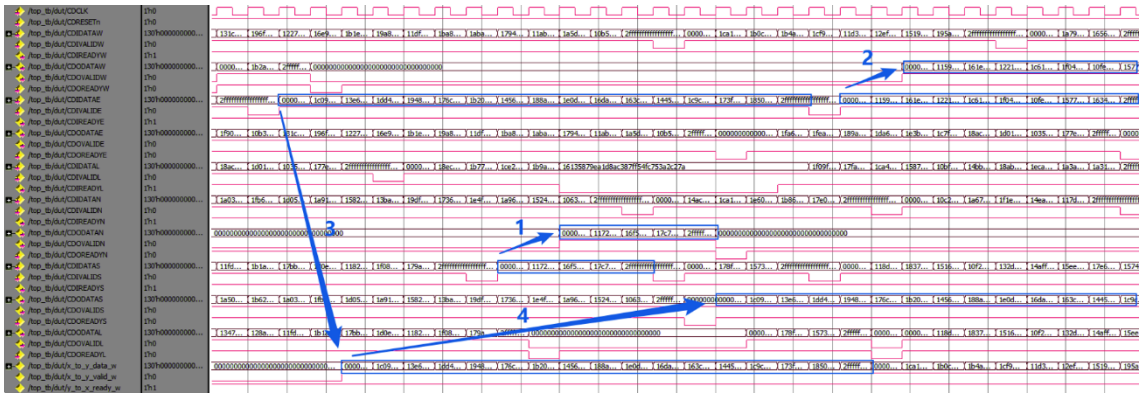


图 4-6 路由器测试部分时序波形图

4.1.4 基于 UVM 的片上网络验证

在上一节中，路由器的数据交换功能得到了验证，本节将对本次设计的由 25 个路由器构成的 5×5 的 NoC 进行验证，其验证目标主要分为以下 3 点：

1. 对 NoC 的数据传输功能进行验证，即验证 NoC 是否具有将数据包从任意一个本地端口转发到任意的另外一个本地端口的能力；
2. 统计分析不同的网络流量模型对 NoC 数据传输性能的影响；
3. 统计分析不同数据包长度和不同缓存大小对 NoC 数据传输性能的影响；

实现正确的数据传输功能是对 NoC 的基本要求，为了对该功能进行全面的验证，引入功能覆盖率（Functional Coverage）的概念，功能覆盖率是指在数字系统的验证中已得到验证的功能点占所设计模块所有功能点的比率，是衡量验证满足

计划要求程度的重要指标。对于本次设计的 NoC 的验证，为了达到 100%的功能覆盖率，需要使测试激励遍历输入-输出节点的所有可能的组合。另外，为了模拟不同的网络流量模型，引入网络注入率（Injecting Rate）的概念，注入率是网络某个输入端口发送数据包所占用的时间与总时间的比率，是衡量网络流量大小的指标，注入率越高，网络流量越大，拥塞程度也越大。最后，为了衡量网络传输性能，引入数据包的传输延迟（Latency）的概念，传输延迟指数据包从开始注入网络的输入端口开始，直到完全离开网络的输出端口所耗费的时钟周期数，网络平均传输延迟越大，消息整体传递速度越慢，网络的整体传输性能越差。

为了得到所有数据包平均传输延迟的计算方法，首先给出第 i 个端口的第 j 个数据包的传输延迟计算公式：

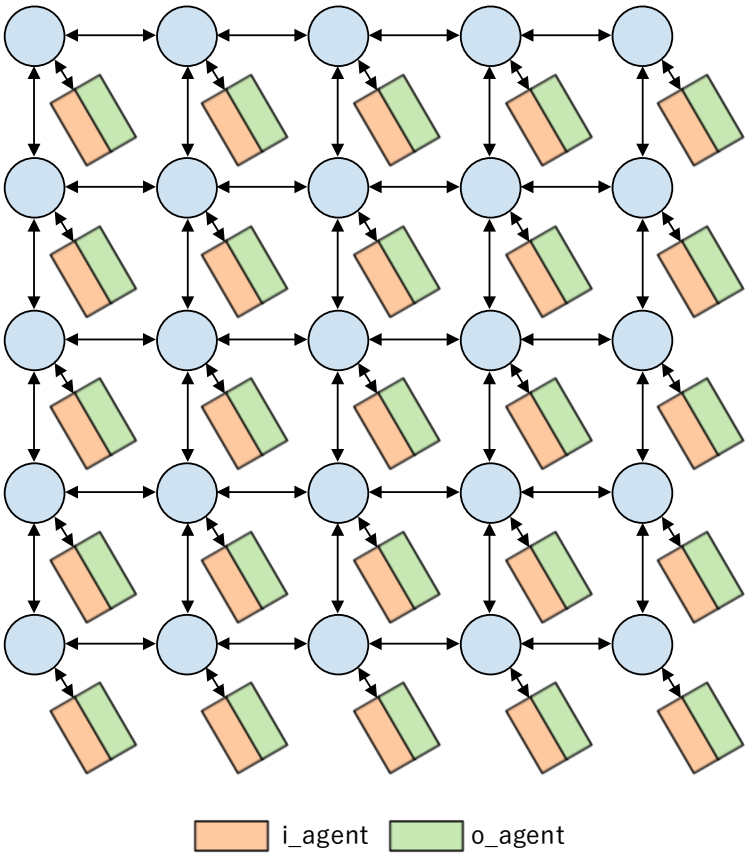
$$\mathrm{latency}^{\mathrm{i,j}} = \frac{\mathrm{T}_{\mathrm{eject}}^{\mathrm{i,j}} - \mathrm{T}_{\mathrm{inject}}^{\mathrm{i,j}}}{\mathrm{T}_{\mathrm{cycle}}} \tag{4-1}$$

其中$\mathrm{latency}^{\mathrm{i,j}}$表示第 i 个端口的第 j 个数据包的传输延迟，$\mathrm{T}_{\mathrm{inject}}^{\mathrm{i,j}}$表示该数据包注入网络时的时间，$\mathrm{T}_{\mathrm{eject}}^{\mathrm{i,j}}$表示该数据包离开网络时的时间，$\mathrm{T}_{\mathrm{cycle}}$为时钟周期。则 n 个端口的 m 个数据包的平均传输延迟可以由下式给出：

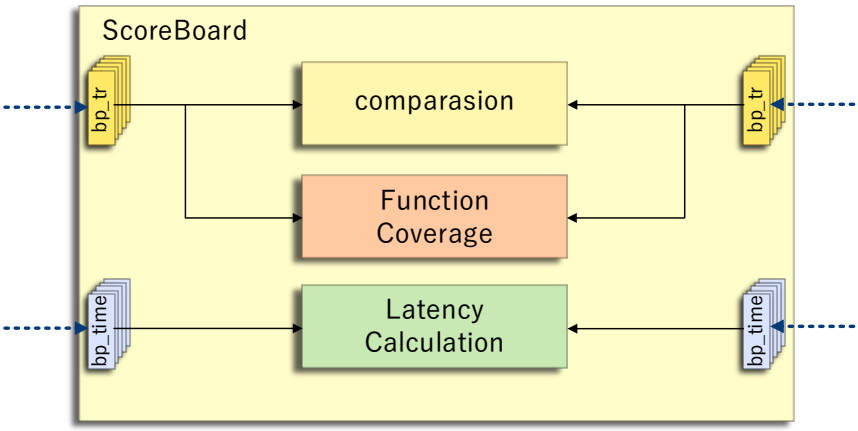
$$\begin{aligned} latency^{av} &= \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{\mathrm{T}_{\mathrm{eject}}^{\mathrm{i,j}} - \mathrm{T}_{\mathrm{inject}}^{\mathrm{i,j}}}{\mathrm{T}_{\mathrm{cycle}}} \\ &= \frac{1}{nmT_{cycle}}\sum_{i=1}^{n}\sum_{j=1}^{m}\mathrm{T}_{\mathrm{eject}}^{\mathrm{i,j}} - \sum_{i=1}^{n}\sum_{j=1}^{m}\mathrm{T}_{\mathrm{inject}}^{\mathrm{i,j}} \end{aligned} \tag{4-2}$$

通过式$(4-2)$可以看出，所有数据包的平均传输延迟等于所有数据包的离开时间之和与所有数据包的注入时间之和的差除以数据包总量和时钟周期的积。

基于上述分析，本次对 NoC 的验证框架在上一节的基础上做出了改进。首先将 i_agent 和 o_agent 应用到 NoC 的每个本地端口，分别用于驱动和监测所在端口的随机事务，所有驱动的检测的事务均递交至统一的 model 和 scoreboard，并在 i_agent 的 driver 中加入驱动等待机制用于改变网络的注入率。然后在 Score Board 组件中加入了功能覆盖率和传输延迟的计算单元，功能覆盖率计算单元用于记录已验证功能点数量，传输延迟计算单元分别接收来自 i_agent 的数据包注入时间和来自 o_agent 的数据包离开时间，并利用式$(4-2)$计算平均传输延迟。

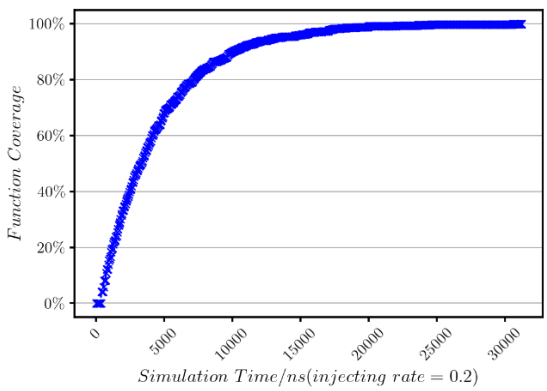


(a)每个本地端口连接一个 i_agent 和一个 o_agent;

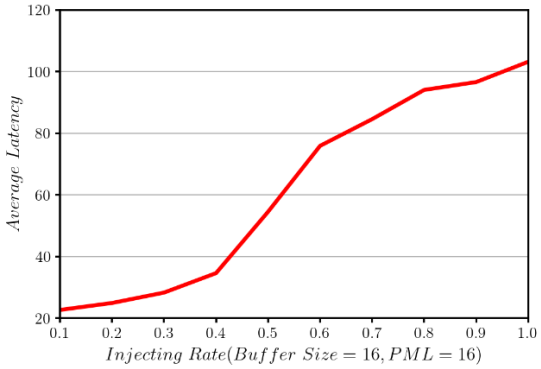


(b)Score Board 结构;

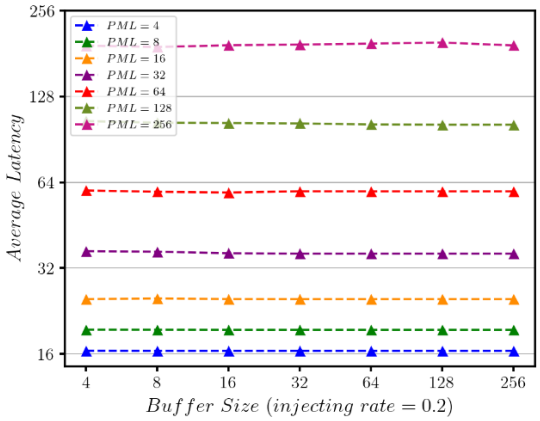
图 4-7 NoC 验证框架结构



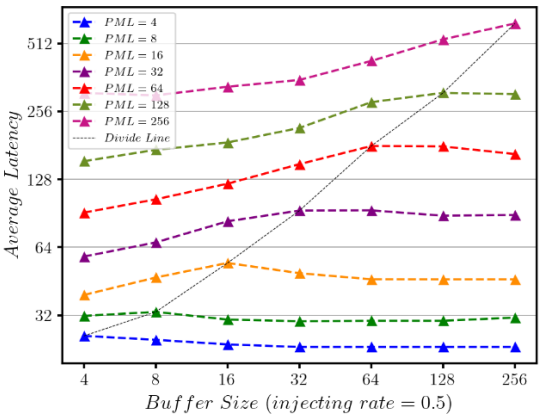
(a)功能覆盖率收敛曲线；



(b)平均传输延迟随注入率变化曲线；



(c)注入率为 0.2 时的传输延迟；



(d)注入率为 0.5 时的传输延迟；

图 4-8 基于 UVM 的 NoC 验证分析结果

图 4-8(a)展示了 NoC 功能覆盖率随仿真时间的收敛曲线，设置时钟周期为 10ns，在仿真时间小于 10000ns 时，功能覆盖率收敛速度较快；当仿真时间大于 10000ns 时，功能覆盖率收敛速度减缓；直到仿真 30000ns 时，功能覆盖率达到 100%，且所有数据包均能够正确完成传输，即 NoC 能够正确执行事务传输功能。

在 NoC 每个输入端口发送 1000 个随机数据包的情况下，图 4-8(b)展示了当网络缓存深度为 16、随机数据包最大长度（Packet-Max-Length，PML）为 16 时平均传输延迟随注入率的变化趋势。随着注入率的不断提高，平均传输延迟不断增大，这是因为注入率的提高增大了网络流量和网络拥塞程度，数据包的竞争概率增大，而由竞争引起的竞争失败必然增加数据包的传输延迟。

图 4-8(c)和图 4-8(d)分别展示了当注入率分别为 0.2 和 0.5 时，平均传输延迟随缓存深度和数据包最大长度的变化趋势。当注入率为 0.2 时，由于网络流量较小，网络拥塞程度较低，可以保证几乎所有数据包传输路径的畅通，网络缓存起到的作

用相对较小，所以该情况下，平均传输延迟与缓存深度几乎无关。当注入率提高到0.5时，网络具有了一定的拥塞程度，数据包之间的竞争程度增大。当缓存深度小于数据包最大长度时，意味着一个数据包的阻塞可能同时跨占多个网络节点，这种情况下传输延迟一部分由输出端口的竞争导致，还有一部分由数据包跨占链路、浪费带宽导致，这时增大缓存深度会使网络的容量增大，由于网络整体吞吐量有限，而输入端在driver的激励下流量是不可控的，所以网络会始终以最大负载运行，这导致网络的拥塞程度随缓存深度的增大而增大，同时平均包延迟增大；当缓存深度增加到大于数据包最大长度时，意味着所有数据包都能够被一个网络节点容纳，网络中的带宽浪费得到了缓解，尤其是随着缓存深度的进一步增大，单个网络节点能够容纳更多的数据包，进一步减少了链路跨占造成的带宽浪费，这种有利因素在传输延迟的决定因素中起到主导作用，使平均传输延迟随缓存深度的增大而减小。图4-8(d)中的Divide Line是缓存深度与数据包最大长度相等所表征的分界线，在该分界线的两侧，平均传输延迟随缓存深度的变化呈现不同的变化趋势。

4.1.5 基于UVM的封装级网络验证

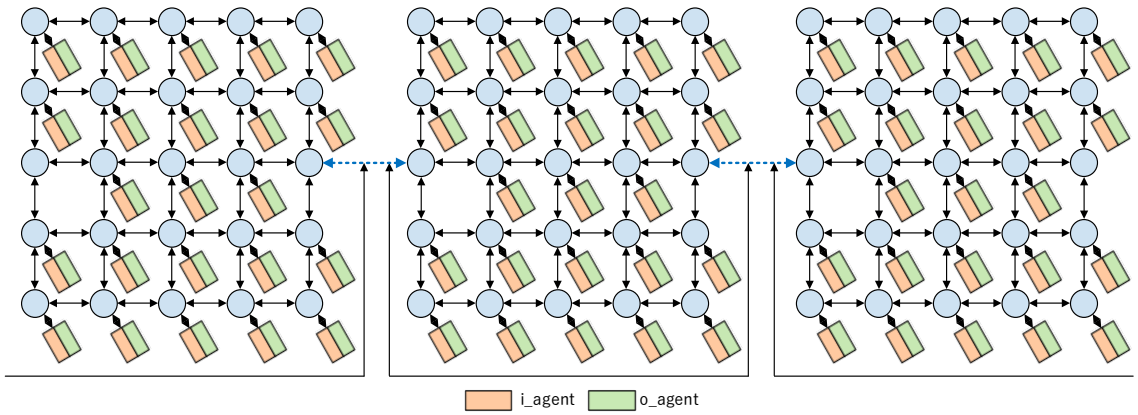
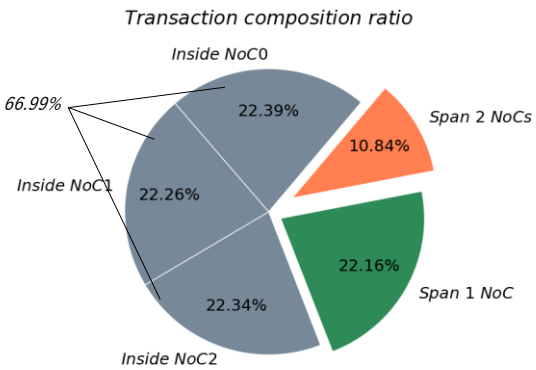


图4-9 基于UVM的NiP的验证框架

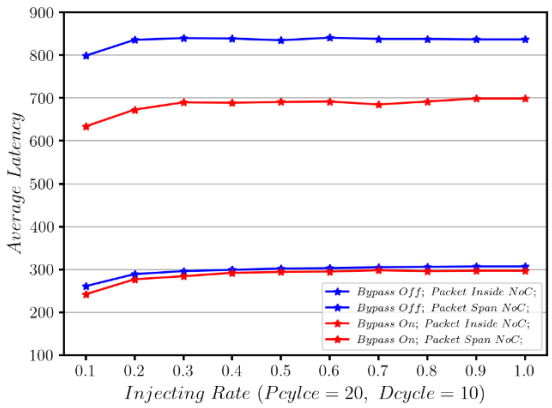
在对所设计NiP的UVM验证中，沿用了之前的验证框架结构，即为NiP中每个路由器本地端口（挂载扩展端口控制器的端口除外）设置一个i_agent和一个o_agent,并使用全局的model和scoreboard对所有agent的数据进行处理和分析。对NiP的验证目标主要分为以下4点：

1. 验证NiP是否能够执行正确的数据路由和传输功能；
2. 统计分析所设计旁路通路是否能够提升系统性能；
3. 统计分析片内/片外时钟频率比对数据传输性能的影响；
4. 统计分析片间异步FIFO深度对数据数据传输性能的影响；

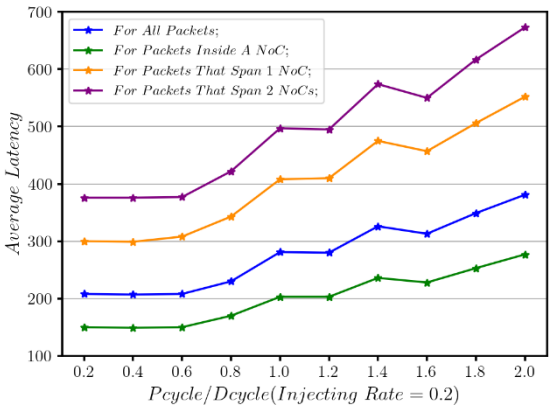
考虑到实际应用场景中数据包应当更多集中在片内传输，而片间传输的数据包所占比重较低，为了尽可能模拟真实的应用场景，控制每个端口 sequence 产生的测试序列中片间/片内传输的事务数量比为 1/3，最终得到的所有事务的类型构成比例如图 4-10(a)所示，即在所有事务中，片内传输占 66.99%，邻网传输占 22.16%，隔网传输占 10.84%。另外，上一小节的分析表明，NoC 内 FIFO 深度越小、数据包长度越小，网络传输性能越好，因此为了降低数据传输延迟，本节的验证中固定 NoC 内部 FIFO 深度为 4、数据包最大长度为 4。



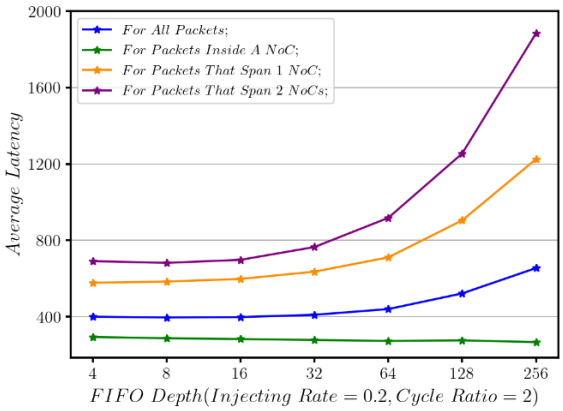
(a)事务类型构成比例；



(b)旁路通路有/无情况下的传输延迟；



(c)传输延迟随时钟周期比的变化曲线；



(d)传输延迟随片间 FIFO 深度的变化曲线；

图 4-10 基于 UVM 的 NiP 验证分析结果

首先，在每个输入端口发送 1000 个随机测试序列的情况下，所有事务均能够正确完成对比，即 NiP 的数据传输功能得到了验证。

然后，在片内时钟周期为 10ns、片间时钟周期为 20ns 的条件下，通过使能和禁用 NoC 子系统中的旁路通路，得到了两种情况下平均传输延迟随注入率的变化

关系曲线（图 4-10(b)）。结果表明，旁路通路的设置显著提升了跨片传输性能，通过使能旁路通路，跨片传输延迟能够降低超过 20%。而跨片传输性能的提升缓解了每个 NoC 内部的拥塞，使片内传输延迟也得到了 7.3%的降低。

片间时钟和片内时钟的频率关系同样影响着 NiP 的传输性能，图 4-10(c)展示了 NiP 传输延迟与片间/内时钟周期比（Pcycle/Dcycle）间的关系曲线，该曲线表明，时钟周期比对三种类型的传输性能均有统一且相当的影响，随着片间时钟频率的降低，传输延迟增大，传输性能降低。其中，受时钟频率比直接影响的片间传输延迟变化幅度最大。当片间时钟频率高于片内时钟频率的 2 倍时，该影响几乎为 0，这是因为本次设计中固定片内总线的位宽为两倍的片间总线位宽，当片内时钟频率为 1/2 倍的片间时钟频率时，片内外带宽一致，当继续提升片间时钟频率时，片内带宽成为了系统带宽的瓶颈，从而将系统传输性能限制在了某一特定值。然而，在实际场景中，片间的时钟频率往往低于片内的时钟频率，因此为了提升系统性能，应当尽可能提升片间时钟频率或增大片间数据位宽。

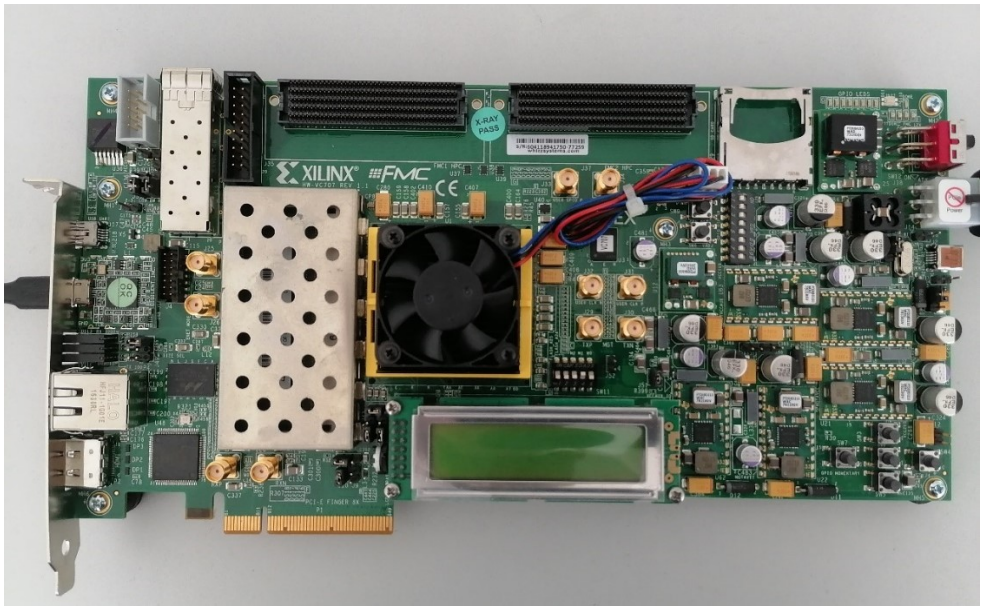
网络的传输性能还受到片间 FIFO，即扩展端口控制器中同步控制器中的异步 FIFO 的深度的影响。图 4-10(d)表明，片间 FIFO 的深度的变化对不同类型的传输延迟具有不同的影响趋势，对于片内传输，其传输延迟随 FIFO 深度的增加而降低，这是因为 FIFO 深度的增加允许扩展端口容纳更多的数据包，从而减少了跨片传输的数据对片内资源的占用，缓解了 NoC 内部的拥塞，提升了片内传输性能；但对于片间传输，其传输延迟随 FIFO 深度的增加而增大，且跨片数量越多，增大幅度越大。这是因为在输入端口流量不可控的情况下，所注入的数据包会尽可能地“塞满”片间 FIFO，但是网络整体吞吐量有限，所以 FIFO 容纳的数据包越多，被阻塞在 NiP 中的跨片传输的数据包就越多，从而导致传输延迟的增大。因此，为了获得更佳的数据传输性能，应当尽可能减小片间 FIFO 的深度。

4.2 板级验证

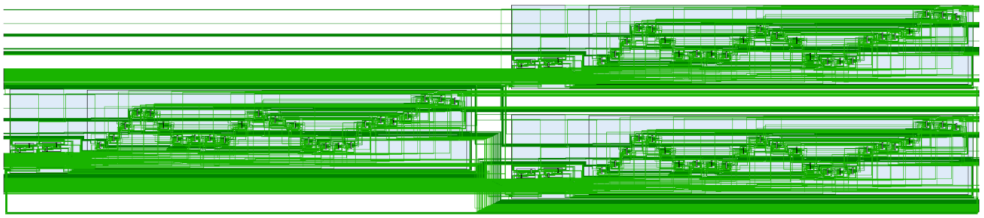
可编程逻辑器件（Progranmmable Logic Device，PLD）是实现数字系统验证的必然之选，它允许用户通过使用上位机配置其内部的逻辑阵列实现任意数字逻辑，从而构成任意的数字系统。综管当下的可编程逻辑器件，FPGA 的发展最为先进和成熟，因此本文中使用 Xilinx 的 VC707 FPGA 开发板实现所设计 NiP 的板级验证，其搭载的 FPGA 芯片型号为 Virtex-7 系列的 XC7VX485TFFG1157-1。

使用 Vivado 工具对所设计的 NiP 进行 RTL 分析和逻辑综合（Synthesis），得到的逻辑综合结果如图 4-11 所示。图 4-11(b)展示了 NiP 的逻辑综合的原理框图，从图中可以看出 NiP 包含的 3 个 NoC 子系统以及每个子系统中的 25 个路由器单

元。图 4-11(c)展示了 NiP 逻辑综合的资源使用情况，设计占用了 99504 个查找表（Look Up Table，LUT）、34920 个分布式 RAM（LUTRAM）、10 个块 RAM（BRAM）和 60146 个触发器（Flip Flop，FF），由于该 FPGA 芯片资源丰富，NiP 的资源占用率不高，不会对之后的布局布线造成较大影响，能够达到良好的实现效果。



(a)NiP 的板级验证平台——VC707 FPGA 开发板；



(b)NiP 逻辑综合原理框图；

Resource	Utilization	Available	Utilization %
LUT	99504	303600	32.77
LUTRAM	34920	130800	26.70
FF	60146	607200	9.91
BRAM	10	1030	0.97
IO	18219	600	3036.50

(c)NiP 逻辑综合资源占用情况；

图 4-11 基于 FPGA 的 NiP 板级验证

由于之前基于 UVM 的仿真级验证已经在逻辑控制与事务传输的层面对所设计 NiP 进行了充分的验证与分析，所以本次板级验证仅需验证 NiP 的物理电路的可实现性，而不再涉及大规模的事务传输。为了在尽可能简化操作、提高验证结果的可观测性的同时增大验证的功能覆盖面，本次板级验证在 NiP 中设置了两条点对点的数据传输路线，分别为 NoC0 至 NoC1 和 NoC0 至 NoC2，在这种情况下，NiP 中 NoC 的路由功能、扩展端口控制器的互连功能和旁路控制器的旁路功能均能够得到有效验证。在所搭建的验证环境中，使用 BRAM 存储和产生测试激励，并在发送端和接收端设置 ILA（Integrated Logic Analyzer，在线逻辑分析仪）用于实际信号的采集和显示。

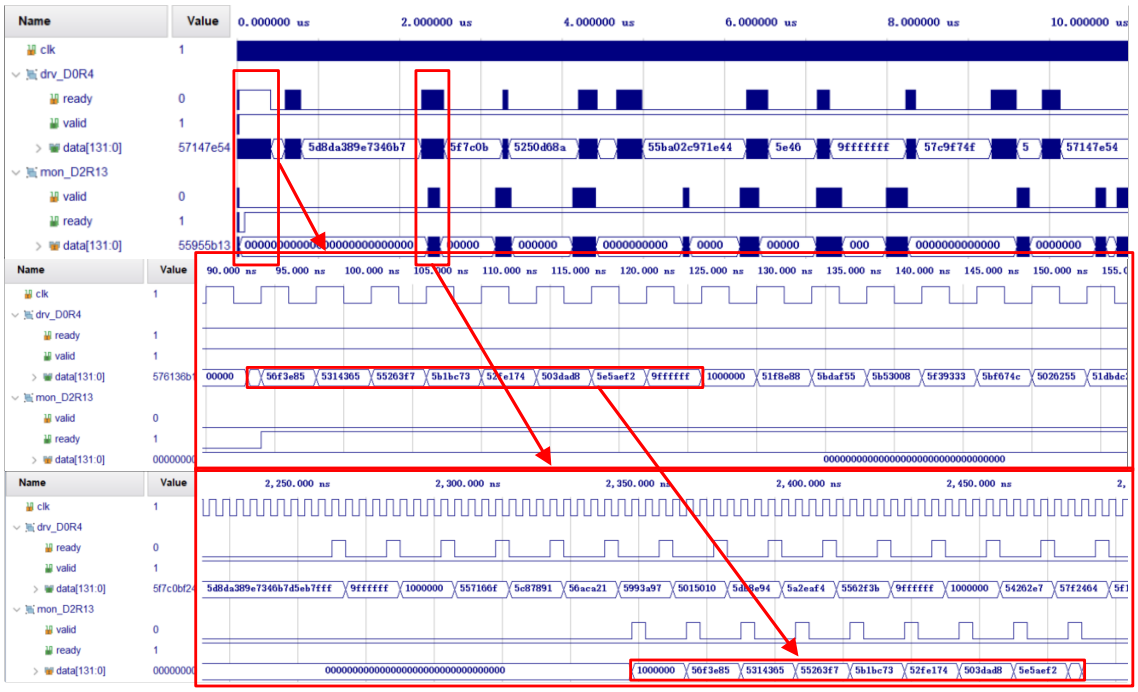


图 4-12 ILA 波形采集结果

上板测试结果表明，两条点对点传输路线均能够进行正常的数据传输。图 4-12 展示了 ILA 采集到的 NoC0 至 NoC2 传输路线中驱动端和采集端的数据传输时序波形，从图中可以看到，在驱动端将发送数据驱动到数据总线上一段时间后，数据能够正常地传输到采集端的数据总线上，并产生正确的握手信号，这些结果表明 NiP 能够正确执行其路由、互连和旁路功能。至此，NiP 的板级验证完毕。

4.3 本章小结

本章主要介绍了对所设计 NiP 的验证过程。本章执行的验证工作主要分为仿真级验证和板级验证两个方面。本章首先介绍了 UVM 统一验证平台以及其验证框

架结构，然后介绍了基于 UVM 的 NiP 仿真级验证，其过程分为路由器验证、NoC 验证和 NiP 验证三步，三个步骤循序渐进，从数据交换底层到片内事务传输层，再到片间事务传输层，全面验证了所设计 NiP 的逻辑功能的正确性。另外，在此基础上，本章还对 NoC 和 NiP 的传输性能进行了量化分析，统计并分析了 FIFO 深度、数据包长度、片内/片间时钟频率比以及旁路通路的有无等因素对各种类型事务平均传输延迟的影响趋势，统计结果表明，所设计的旁路通路能够提高 20%的网络性能，且合适的 FIFO 深度和片间时钟频率能够使网路性能得到最大程度的发挥。最后，本章对所设计 NiP 进行了基于 FPGA 的板级实物验证，实现了两条点对点线路的事务传输，验证了 NiP 的物理可实现性。

第五章 全文总结与工作展望

5.1 全文总结

Chiplet 技术的兴起为片间互连通信的标准化提出了新的要求。

本文提出了一种面向多 Chiplet 集成的互连与通信结构，该结构由多个片上网络组成，片上网络中的每个路由器端口都通过一个标准协议转换接口实现与一个 Chiplet 的通信，多个片上网络通过扩展端口和片间总线模块级联构成封装级网络，并在片上网络的两个扩展端口之间设置旁路通路用于提升数据跨片传输速度。本文对该互连和通信结构的框架部分，即由多个片上网络、扩展端口和旁路通路构成的封装级网络进行了设计和验证。本文完成的工作主要包括以下几点：

1．设计片上网络的核心单元——维度切分式单周期路由器，在路由器微架构设计中运用了 XY 维路由算法、Wormhole 流控机制、FIFO 设计和矩阵仲裁器设计等多种设计方法。将所设计的路由器按照一定的拓扑互连构成片上网络。

2. 设计用于片上网络级联的扩展端口控制器。设计同步控制器用于实现片内外时钟域和数据位宽的转换，设计旁路控制器用于数据的旁路控制。将所设计的片上网络和扩展端口控制器互连起来，构成封装级网络。

3. 对设计进行了仿真级验证，基于 UVM 平台、编写软件测试环境依次对所设计路由器、片上网络和封装级网络进行验证分析，验证了所设计模块的逻辑功能的正确性，并量化分析了不同的 FIFO 深度、数据包长度、时钟频率比以及旁路通路的有无等因素对所设计片上网络和封装级网络传输性能的影响。

4. 基于 FPGA 实现了所设计 NiP 的板级验证，通过 ILA 观测波形，实现了两条点对点线路的事务传输，验证了所设计 NiP 的物理可实现性。

5.2 后续工作展望

后续的工作分为硬件和软件两条研究线路。在硬件方面，所设计封装级网络的片间拓扑有待于进一步完善，如基于 Mesh、Ring 等拓扑互连多个片上网络、构成二级网络。在完善封装级网络拓扑的过程中，关于新网络路由算法、流控机制以及死锁避免的研究也同等重要，可能需要利用虚通道技术和流水线技术，结合 credit 等机制进一步完善网络中每个模块的设计。在软件方面，后续工作主要集中在从网络事务类型定义、地址映射、接口配置到整个传输协议栈的研究和设计。具备了这些条件，本设计才能够获得较好的完整性，从而真正应用于 Chiplet 的互连。

致　谢

感谢姜书艳老师和黄乐天老师的悉心指导！
感谢赵天津学长和陈颖芃学长带来的帮助！
感谢我的大学同学、我的朋友和家人的一路陪伴！

参考文献

[1] G. E. Moore. Cramming more components onto integrated circuits[G]. Electronics, 1965, 38(8): 114-117.

[2] Saleh R, Wilton S, Mirabbasi S, et al. System-on-chip: Reuse andintegration[J]. Proceedings of the IEEE, 2006, 94(6):1050-1069.

[3] G. Kim et al. Networks-in-Package; Design, Analysis and Implementation[C]. 2006 IEEE Electrical Performane of Electronic Packaging, Scottsdale, AZ, USA, 2006:221-224.

[4] Tai K L. System-In-Package (SIP): Challenges and opportunities[C]. Proc of IEEE ACPDAC'2000, Piscataway, 2000:191-196.

[5] A. Blodgett. A Multilayer Ceramic Multichip Module[J]. in IEEE Transactions on Components, Hybrids, and Manufacturing Technology, 1980, 3(4): 634-637.

[6] Lin L, Yeh T C, Wu J L, et al. Reliability characterization of Chip-on-Wafer-on-Substrate (CoWoS) 3D IC integration technology[C]. Proc of the 63rd Electronic Components and Technology Conf, Piscataway, 2013: 366-371.

[7] Naffziger S, Lepak K, Paraschou M, et al. 2.2 AMD chiplet architecture for high-performance server and desktop products[C]. Proc of IEEE ISSCC'2020, Piscataway, 2020: 6-7.

[8] Gomes W, Khushu S, Ingerly D B, et al. 8.1 lakefield and mobility compute: A 3D stacked 10nm and 22FFL hybrid processor system in 12×12mm^2, 1mm package-on-package[C]. Proc of IEEE ISSCC'2020, Piscataway, 2020: 144-146.

[9] M. Hutton. Stratix® 10: 14nm FPGA delivering 1GHz[C]. 2015 IEEE Hot Chips 27 Symposium (HCS), Cupertino, CA, USA, 2015: 1-24.

[10] Arunkumar A, Bolotin E, Cho B, et al. MCM-GPU: Multi-chip-module GPUs for continued performance scalability[C]. Proc of the 44th Annual Int Symp, Piscataway, 2017: 320-332.

[11] Jack G. Xilinx's Virtex-7 2000T FPGAs [EB/OL]. (2011-11-14) [2020-02-11]. https://www.embedded.com/xilinxs-virtex-7-2000t-fpgas/.

[12] Vivet P, Guthmuller E, Thonnart Y, et al. 2.3 A 220GOPS 96-core processor with 6 chiplets 3D-stacked on an active interposer offering 0.6ns/mm latency, 3Tb/s/mm2 inter-chiplet interconnects and 156mW/mm2@ 82%-peak-efficiency DC-DC converters[C]. Proc of IEEE ISSCC'2020, Piscataway, 2020:8-9.

[13] T. Singh et al. 2.1 Zen 2: The AMD 7nm Energy-Efficient High-Performance x86-64 Microprocessor Core[C]. 2020 IEEE International Solid- State Circuits Conference - (ISSCC),

San Francisco, CA, USA 2020: 42-44.

[14] Intel Corp. New Intel core processor combines high performance CPU with custom discrete graphics from AMD to enable sleeker, thinner devices [EB/OL]. (2017-11-06) [2020-01-09]. http://newsroom.intel.com.

[15] N. Beck, et al. 'Zeppelin': An SoC for multichip architectures[C]. 2018 IEEE International Solid - State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2018: 40-42.

[16] A. Wong, G. Tsui, et al. High bandwidth low power interposer interconnect challenge, design, and validation in 3D silicon stacked interconnect (SSI) technology[C]. 2016 IEEE International Symposium on Electromagnetic Compatibility (EMC), Ottawa, ON, Canada, 2016: 434-439.

[17] A. Hemani, A. Jantsch, S. Kumar, et al. Network on chip: An architecture for billion transistor era[C]. Proceeding of the IEEE NorChip Conference, Oslo, Norway, 2000.

[18] I. E. Papazian. New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP)[C]. 2020 IEEE Hot Chips 32 Symposium (HCS), Palo Alto, CA, USA, 2020: 1-22.

[19] M. Voogel, et al. Xilinx Versal™ Premium[C]. 2020 IEEE Hot Chips 32 Symposium (HCS), Palo Alto, CA, USA, 2020: 1-46.

[20] K. Lee, et al. Networks-on-chip and Networks-in-Package for High-Performance SoC Platforms[C]. 2005 IEEE Asian Solid-State Circuits Conference, Hsinchu, Taiwan, 2005: 485-488.

[21] M. M. Ahmed, N. Mansoor, A. Ganguly. An Asymmetric, Energy Efficient One-to-Many Traffic-Aware Wireless Network-in-Package Interconnection Architecture for Multichip Systems[C]. 2018 Ninth International Green and Sustainable Computing Conference (IGSC), Pittsburgh, PA, USA, 2018: 1-8.

[22] M. Abdellah, A. M. Kouadri M. et al. Networks-In-Package: Performances management and design methodology[C]. 2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT), Hsinchu, Taiwan, 2008, 140-143.

[23] I. Swarbrick, et al. Versal Network-on-Chip (NoC)[C]. 2019 IEEE Symposium on High-Performance Interconnects (HOTI), Santa Clara, CA, USA, 2019: 13-17.

[24] M. Hutner, R. Sethuram, et al. Special Session: Test Challenges in a Chiplet Marketplace[C]. 2020 IEEE 38th VLSI Test Symposium (VTS), San Diego, CA, USA, 2020: 1-12.

[25] W. J. Dally, B. Towles. Principles and Practices of Interconnection Networks[M]. Morgan Kaufmann Publishers Inc, 2004.

[26] N. E. Jerger,T. Krishna, L. -S. Peh. On-Chip Networks,Second Edition[M]. Morgan & Claypool Publishers Inc, 2017.

[27] J. Kim. Low-cost router microarchitecture for on-chip networks[C]. 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, USA, 2009: 255-266.

[28] G.-M. Chiu. The odd-even turn model for adaptive routing[J]. IEEE Transactions on Parallel and Distributed Systems, 2000,11(7): 729-738.

[29] B. Q. Wang. High-Performance Network-on-Chip Design for Many-Core Processors[D]. Stockholm: KTH Royal Institude of Technology, 2020.

[30] 陈桂林，王观武，胡健，等. Chiplets 封装结构与通信结构的研究综述[J/OL]. 计算机研究与发展. http://kns.cnki.net/kcms/detail/11.1777.TP.20210320.1642.002.html.

[31] 天野英晴. FPGA 原理和结构[M]. 赵谦，译. 北京：人民邮电出版社, 2019.

[32] K. Mishra. Verilog 高级数字系统设计技术与实例分析[M]. 乔庐峰，尹廷辉，于倩，等，译. 北京：电子工业出版社, 2018.

[33] 刘斌. 芯片验证漫游指南：从系统理论到 UVM 的验证全视野[M]. 北京：电子工业出版社, 2018.

[34] C. Spear. SystemVerilog 验证[M]. 张春，麦宋平，赵益新，译. 北京：科学出版社, 2009.

[35] 张强. UVM 实战[M]. 北京：机械工业出版社, 2014.