# 摘 要

基于总线互连的多核片上系统（System-on-Chip，SoC），通过层次化总线和多总线等技术在片上系统中实现更多处理核心的集成。然而，随着半导体工艺技术的迅速发展和芯片制造水平的提高，这种总线互连方式出现了许多瓶颈问题。近些年，Network-on-Chip（NoC）的崭新概念被学者们提出。NoC 基于计算机网络通信的思想，核与核之间通过分组路由的方法进行通信，克服了基于总线互连的多核 SoC 遇到的各种瓶颈问题。近年来对 NoC 研究的主要方向包括网络的拓扑结构、映射算法、路由算法、死锁避免、容错机制和低功耗设计等方面。本文的主要研究对象是自适应的路由选择算法，考虑到的性能分析包括网络吞吐率和平均包时延。

本文基于蚁群优化（Ant Colony Optimization, ACO）思想，采用适合于 FPGA 实现的信息素表的更新方法和蚂蚁包的路由选择方法，来设计应用于 NoC 的自适应路由选择算法。本文最后将上述路由选择算法与其它的 NoC 路由选择算法进行性能上的比较和分析。实验结果表明，相较于 Buffer Level 选择算法，文中实现的 ACO 选择算法在 Uniform、Transpose、Hotspot 三种输入模式下能达到的吞吐率最大提升幅度为+23.81%（PIR=0.5）、+16.69%（PIR=0.5）和-0.96%（PIR=0.01），能达到的平均包时延最大降低幅度为+3.90%（PIR=0.2）、+9.73%（PIR=0.01）和+7.18%（PIR=0.6）。本文工作为蚁群优化思想的实际应用以及 NoC 路由算法的 FPGA 实现提供了重要的参考价值。

关键词：片上网络；自适应路由；蚁群优化；选择算法；现场可编程门阵列。

# Abstract

Bus-based multicore SoCs can provide more processing cores by incorporating advanced technologies such as hierarchical buses and multi-buses. However, with the rapid development of semiconductor processes and chip manufacturing technologies, bus-based interconnects encountered performance and energy bottlenecks. Therefore, researchers and industry practitioners proposed and advocated NoCs to implement on-chip interconnects that are inspired by traditional computer networks to enable point-to-point transmission of network packets. Current NoC research areas include network topologies, mapping algorithms, routing algorithms, selection algorithms, deadlock avoidance, fault tolerance and energy saving techniques.

This work focuses on the design and FPGA implementation of ant colony optimization (ACO) based selection algorithms for adaptive routing in NoCs, which consists of pheromone updating and pheromone table based routing. The aforementioned algorithm is then implemented in SystemVerilog and verified by test benches. FPGA Simulation results show that, the aforementioned ACO based selection algorithm can achieve throughput improvement of +23.81% (PIR=0.5), +16.69% (PIR=0.5) and -0.96% (PIR=0.01), and average packet delay reduction of +3.90% (PIR=0.2)、+9.73% (PIR=0.01) 和+7.18% (PIR=0.6), under Uniform, Transpose and Hotspot traffics, as compared to Buffer Level selection algorithm. This work provides great insights on the real-world applications of ant colony optimization and FPGA based implementation of ACO based selection algorithms for adaptive routing on NoCs.

Keywords：network on chip；adaptive routing；ant colony optimization；selection strategy；field programmable gate array.

# 目 录

# 1．绪　论

## 1．1　课题背景及意义

本章第一部分分析了本课题选题背景与意义。第二部分首先阐述了片上网络设计的主要问题和国内外研究现状，然后给出了路由算法的概念、分类以及具体的几种被广泛认可和应用的路由算法的例子，紧接着讲述了被应用到本文的路由选择算法中的蚁群优化思想，之后还介绍了用来实现本文的现场可编程门阵列的发展，最后对本文各章节进行了简单扼要的说明。

### 1．1．1　课题背景

随着半导体工艺技术的迅速发展和芯片制造水平的提高，在单一芯片中能集成的晶体管数量也成倍增长，如何有效地管理数目众多的晶体管成了芯片体系结构必须要面对的新问题。二十世纪九十年代中期，集成电路（IC）设计者们开始在一个芯片上构建功能更加复杂的系统，集成电路开始向集成系统（IS）方向转型，片上系统（System-on-Chip，SoC）在此大趋势下发展而来[1]。

SoC 因其的优势发展得非常迅速，具有功耗较低、面积及体积较小、设计周期较短及成本较低、运行速度较快、系统功能较丰富等优势，因此在工业界和学术界逐渐引起了关注[2]。SoC 具有以下五个方面的优势[3]。

虽然 SoC 相比以前的集成电路具备了许多优势，但是随着用户对产品功能和更新周期等要求的不断提高，SoC 芯片结构的设计也越发复杂，设计计划周期也不断缩短，诸多方面的挑战也随之而来，如测试成本的逐渐增加，生产成品率的开始下降，IP 模块的重用也出现了许多问题。而计算机网络的通信方式可以有效解决基于共享总线的 SoC 通信结构所带来的诸多问题，片上网络结构由此而来。2000 年左右，片上网络（Network on Chip, NoC）的概念被 Axel Jantsch 教授等人首次提出[4]，2002 年，学者 S. Kumar 在 IEEE 上发表了第一篇关于 NoC 的论文[5]。NoC 的基本思想是将计算机网络的实现方式运用到芯片的网络设计中，让若干个结点组成一个网络，而每个结点分成了资源结点和路由器结点两部分，资源结点只能通过网络接口和本地路由结点进行通信，而资源结点间的通信需通过路由器结点间连接的链路间接进行[6]。

NoC 的通信结构设计具有以下四个方面优势[7]。

（1）提高了通信带宽。NoC 具有的全局异步、局部同步的并行通信能力使得各结点同时进行数据传输的情况变成可能，从而在数量级上增加了网络的通信带宽，提高了网络的吞吐量和通信效率。

（2）降低了功耗：随着通信技术的发展，基于总线架构的庞大的时钟网络会大大

消耗网络的功耗，而 NoC 的通信方式在一定程度上简化了时钟控制逻辑，并且将长连线的通信转为短连线的设计进一步缩短了通信距离，从而使功耗得到更好的控制。

（3）良好的可扩展性：由于参考了计算机网络的通信机制，网络拓扑的加入也使得设计具有很好的扩展性。比如若想在二维网格拓扑结构的 NoC 中加入新的资源结点和路由结点，只需要把新加入的结点通过此网络中设计的接口连接到拓扑结构中，就可以完成网络规模的扩展。

（4）高效的可重用性：NoC 中的通信采用分层通信协议、计算和通信完全分离的网格结构设计，使得计算和通信两种模块都具有了良好的可重用性。

由于这些优点，片上网络已经慢慢成为国内外相关学者的研究热点。目前对 NoC 的研究也很广泛，包括网络拓扑结构[8]、路由算法[9]、交换机制[10]、容错机制[11]、映射算法[12]等方面。

## 1．1．2　选题意义

随着时代的发展，人们对片上网络系统功能、网络性能的要求也在逐渐增高，为了满足用户的需求，片上网络系统中所集成的核心数目越来越来，其它部件的数量也在不断增长，内核之间的通信量也随着不断加大。由于受到拓扑结构等的限制，随着点与点之间通信量的剧增，网络的某些地方就有可能会出现拥塞，从而增加了网络时延。路由算法是当前片上网络研究的热点，而低延迟也一直是人们在研究片上网络路由算法时的一个性能目标。

路由算法是片上网络研究中的一项关键技术，也是当前研究的重要方向。路由算法的目标不仅是将消息包正确无误地发送到目的节点，还需在网络拓扑提供的路径中均匀地分配流量，以避免热点，减少争用，从而改善网络延迟和吞吐量（翻译中后面还有一些）。路由算法可以按照路由路径是否单一，分成确定性路由和自适应路由，而自适应路由算法又可以分为路由计算和路由选择两部分。确定性路由算法，顾名思义，在同一种网络拓扑中，进行转发消息包时的路由计算后，由它所得的路径是确定的，即是单一的。每个源结点和目的地结点之间都会存在一个路径，且也只会存在一条路径。这样的算法缺少了路径的多样性，不能随网络的状态而更改路径从而提高网络性能。而自适应路由算法在每次转发消息包时可以提供大于等于一的输出选择，并能根据网络状态选择比较合适的下一跳输出，从而能在一定程度上提高网络性能。反之，应用某种路由算法的网络所体现出的性能也可以用来衡量此路由算法的优劣。所以自适应路由算法的研究和平均包延迟等性能的比较都是很有意义的。

本文的设计对象是自适应路由选择算法，并且通过与其它选择算法的性能进行比较分析，尝试寻找更优的自适应路由选择算法，从而提高多核架构的整体运行效率。

## 1．2　NoC 及 NoC 路由算法

### 1．2．1　NoC 研究现状

半导体集成电路技术的发展极大推动 NoC 研究的发展，目前已经有超过 90 个组织机构加入到 NoC 研究的领域，现在已经积累了丰富的研究成果。为了促进 NoC 的研究，一些国际知名期刊为 NoC 的研究设立了专题，并且已多次召开了关于 NoC 研究的国际会议。同时 NoC 的研究也受到了 Intel、IBM 等科技公司的重视。在 NoC 的研究方面，国外的研究水平与研究成果明显领先于国内，国外对 NoC 的研究投入了大量人力、财力，而中国只是刚刚起步。下文将详细介绍 NoC 研究的国外现状。

NoC 发展的十几年里，法国成立了 NoC 技术公司 Arteris，国外不仅在学术上而且在工业上都得到快速发展。2000 年 5 月，法国的 Pierre Guerrier 在设计片上系统通信时采用了基于分组交换(packet-switched)的通信技术，并提出可编程定制的通信网络的概念。2000 年 11 月，瑞典皇家技术学院给出 NoC 的定义，随后斯坦福大学的 Giovanni de Micheli 与博洛尼亚大学的 Luca Benini 对 SoC 低功耗互联进行了研究，并提出“NoC 是 SoC 设计的新范例”的论断[13]。斯坦福大学的 W.J.Dally 在发表相关论文指出 NoC 中的通信应是基于包交换的通信，确定了 NoC 的研究重点。在这些 NoC 研究团队中，Jingcao Hu、R.Marculescu 等人组成的团队和另一个 Jiong Luo、N.K.Jha 等人组成的团队对 NoC 的基本理论进行了广泛深入的研究，且这两个团队提出了多数的关于 NoC 的大多数重要的基础理论[14]。国外有很多的研究起到了推动作用例如 Li 等人提出一种动态无死锁容错路由算法 Dy XY[15]，该算法是为数据包选择最短路径，如果存在多条最短路径时，则根据拥塞情况，选择一条最优路径。HU 和 Lin 等人提出了一种改进的无死锁转弯模型路由算法 Odd-Even[16]，此算法根据当前结点所在的位置限制某些转弯，使得算法无死锁，具有很好的自适应性。Hu 和 Marchculescu 提出动态路由算法 Dy AD[17]，此算法将确定性路由算法和自适应路由算法的优点相结合，更大程度上提高了算法的自适应性。Jiong Luo、N.K. Jha 的团队提出了使用电压分级技术来降低功耗，Jingcao Hu、R.Marculescu 的团队开创性的提出了 NoC 的映射问题。

### 1．2．2　NoC 路由算法介绍

在确定网络拓扑后，将由路由算法（Routing Algorithms）决定一个消息使用何种路径通过网络到达目的地。路由算法通常分为三类：确定性（Deterministic）、无关性（Oblivious）和自适应性（Adaptive）[18]。确定性的维序路由算法和无关性的路由算法简单、易实现，但是具有一定的盲目性，自适应的路由算法能够根据网络拥堵等情况灵活地选择路由路径，但是在实现时需要复杂的控制逻辑和硬件电路。

路由算法也可以被归类为最短路径路由算法（Minimal Routing Algorithms）和非最短路径路由算法（Non-Minimal Routing Algorithms）。最短路由算法要求只能选择从源

结点到目的地结点间跳数最少的路径。非最短路由算法允许选择可能增加从源结点到目的地结点间跳数的路径。在没有拥堵的情况下，非最短路由增加了延迟和功耗，因为消息需经过额外的路由器和链接。交通拥堵时，非最短路由的选择能避免拥堵的链接，所以可能会降低数据包的延迟。

下面分类介绍几种应用广泛的能进行路由计算的路由算法。

### 1．2．2．1　维序路由算法

虽然大量的路由算法已经提出，但由于维序路由的简单性，在片上网络中最常用的路由算法是维序路由（Dimension-Ordered Routing，DOR）。维序路由是一个确定性路由算法的例子，从结点 A 传送到结点 B 的所有消息总是会经过相同的路径。在维序路由中，消息逐维遍历网络，在切换到下一维度前，当前维度需先到达吻合目的地结点的坐标。

维序路由既简单又不会产生死锁，使用维序路由，每个源结点和目的地结点之间都会存在一个路径。然而，维序路由也消除了在网格网络中路径的多样性，从而降低了吞吐量；由于缺乏路径多样性，路由算法也无法绕过网络中故障或拥堵的区域，路由限制的结果使得维序路由在平衡网络负载方面表现得较差。

（a）所有转弯　　（b）X-Y 转弯

图 1.1 在 2D 网格中可能产生的路由顺序

X-Y 路由、Y-X 路由都是确定性维序路由的典型例子。图 1.1（a）说明了一个二维网状网络的所有可能的转弯路径，而图 1.1（b）说明了被 X-Y 路由允许的、被更多条件限制后的可能的转弯路径，具体地说，一个正向东或向西前进的消息可以向北或南转弯；然而，一个正向北或向南前进的消息却不允许转弯。允许所有的转弯路径的结果是形成了循环资源依赖关系，这可能会导致网络发生死锁。为了防止这些循环依赖，有些转弯不能被采用。

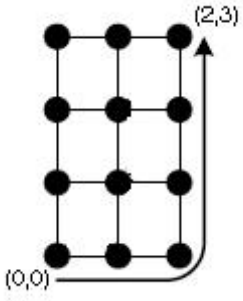


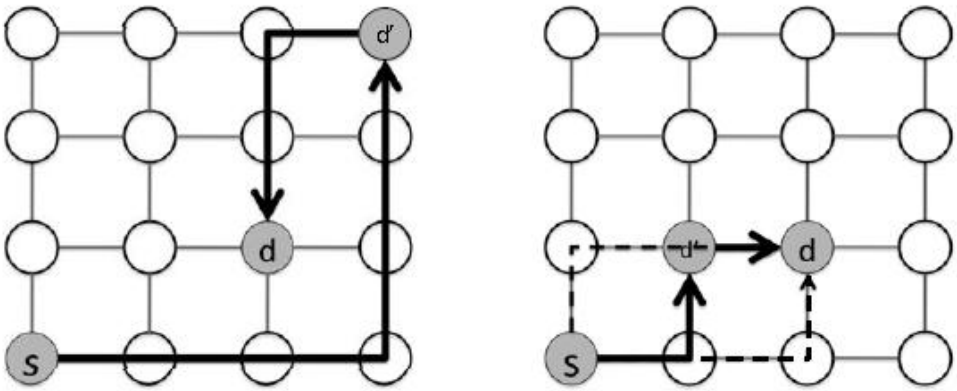

图 1.2 从结点（0,0）到（2,3）的 DOR 的路由情况

图 1.2 显示了 X-Y 维序路由的一个具体例子。在一个二维拓扑中，由 X-Y 维序路由计算发送的消息包，首先沿着 x 维传送，其次再沿着 y 维传送。一个要从结点（0,0）发送到结点（2,3）的消息包，将首先沿着 x 维经过 2 跳，到达结点（2,0），然后才沿 y 维经过 3 跳到达其目的地。

### 1．2．2．2　无关路由算法

第二类路由算法是无关路由，一个路由器可以在发送消息之前随机选择可选路径，但路由路径的选择不考虑网络拥塞情况。例如，从结点 A 到达结点 B 的消息在途中随机选择 Y-X 路由或 X-Y 路由来计算下一跳路由结点，因此，从结点 A 到达结点 B 可以有不同的路径。确定性路由是无关路由的一个子集。不考虑网络状态信息进行路由选择，所以无关路由算法也可以保持简单。

Valiant 的随机路由算法（randomized routing algorithm）[19]和最短无关路由算法都是无关路由算法。




（a）Valiant 的路由算法　　（b）最短无关路由算法

图 1.3 无关路由例子

如图 1.3（a），使用 Valiant 的算法将消息包从源结点 s 路由到目的地 d，会先随机选择一个中间目的地结点 d’。数据包首先从结点 s 路由到结点 d’，然后从结点 d ' 路由到结点 d。在路由到最终目的地之前先路由到一个随机选择的中间目的地的方法，使得任何交通格局的出现都是均匀随机的，因此 Valiant 的算法能够平衡网络中的流量负载。但是 Valiant 的算法的负载平衡是以牺牲局部性为代价的；例如，通过路由到一个中间目的地，网格中近邻通信上的局部性被破坏。跳数增加，这又增加了网络中的平均数据包延迟和数据包所消耗的平均能量。跳数从最小的 3 跳增加到了 9 跳。

图 1.3（b）描述了最短无关路由算法，用实线标识了从结点 s 到结点 d 的一种可能路由选择，用虚线标识了另外两种可能路径。最短无关路由算法是保持了局部性的 Valiant 的算法。即在一个 k 元 n 维立方体拓扑结构中，中间节点 d'必须位于最小象限（以

源结点和目的地结点作为对角结点的最小 n 维子网络）内。中间目的地结点 d'只能在由源结点 s 和目的地结点 d 所形成的最小象限内选择，以此路径长度保持在了最小的 3 跳。

在使用 Valiant 的随机路由和最短路径自适应路由的过程中，维序路由可用于从 s 到 d'和从 d'至 d 的路由。如果使用 DOR，并不是所有的路径都能被利用，但能实现比从 s 直接前往 d 的确定性路由更好的负载平衡。

### 1．2．2．3　自适应路由算法

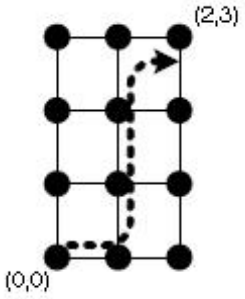



图 1.4 从结点（0,0）到（2,3）的 Adaptive 路由的一种可能情况

一种更复杂的路由算法是可以自适应的，即一个消息可以根据本地或全局信息（如缓存队列占用率、消息排队延迟等网络流量情况）、当前或历史信息来选择从结点 A 到结点 B 的路径。如图 1.4 所示，一个消息最初遵循 X-Y 路由，在结点（1,0）发现东输出端口链接的路由器堵塞，于是在考虑这个拥塞后，该消息将修改选择，将消息发送给北输出端口链接的路由器。

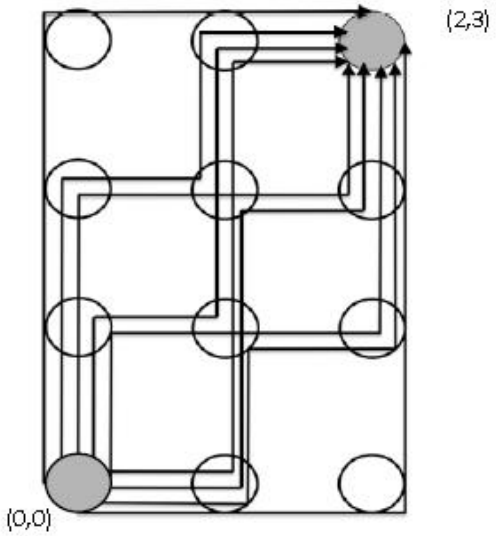
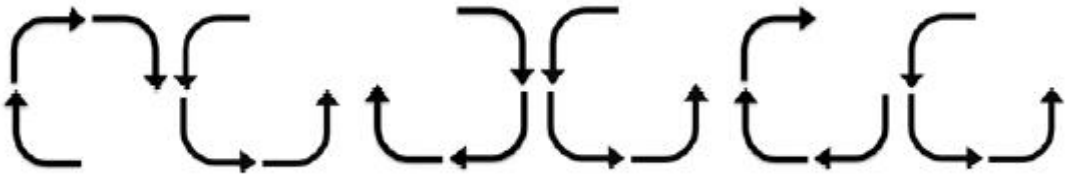



图 1.5 自适应路由例子

图 1.5 显示了消息从结点（0,0）到结点（2,3）的所有可能（最短）路径。总共有九条可能路径。只利用这些最短路径的自适应路由算法，可以在很大程度上利用路径多样性来提供负载平衡和容错。

使用全自适应路由算法时就会存在潜在死锁，如何避免这些潜在死锁的发生是自适应路由的一个挑战。自适应路由的另一个挑战是保持一致性协议可能需要的消息间的顺序。如果消息必须以和源结点发送的相同的顺序到达目的地，自适应路由就会成为问题。

自适应转弯模型路由（Adaptive Turn Model Routing）是自适应路由算法。为了实现无死锁，自适应转弯模型路由（Adaptive Turn Model Routing）剔除了必要的最小范围上的一组转弯，保留了较多的路径的多样性和自适应性的潜力。具体的说，在一个二维网格中，维序路由只允许 8 种所有可能转弯中的 4 种。而转弯模型路由允许 8 种所有可能转弯中的 6 种，即在每个环中，都只有一个转弯被剔除了，从而增加了算法的灵活性。



（a）西向优先转弯　　（b）北向最后转弯　　（c）负向优先转弯

图 1.6 转弯模型路由

在图 1.6 中说明了三种可能的路由算法。如图 1.6 所示，从所有可能的转弯开始，剔除由北向西的转弯，在这个剔除完成后，可以得到三种路由算法。在图 1.6（a）中显示了西向优先算法（the West-First Algorithm）；在剔除了由北向西的转弯之外，还剔除了由南向西的转弯。换句话说，一个消息在向其它方向前行时必须先向西方向前行。北向最后算法（the North-Last Algorithm，如图 1.6（b）所示）剔除了由北向西和由北向东的转弯。一旦消息转北方向，就不再允许其进一步的转弯；因此，向北方向的转弯必须在最后进行。最后，图 1.6（c）移除了由北向西和由东向南的转弯，构成了负向优先（Negative-First）算法。一个消息向负向（西向和南向）的传播，必须是在其允许向正方向（东向和北向）传播之前。所有这三个转弯模型路由算法都是无死锁的。转弯模型路由提供了比维序路由更多的灵活性和自适应性，但其还是受到些限制。

奇偶转弯模型路由（Odd-Even Turn Model Routing）[20]提出了是否消除一组两个的转弯将根据取决于当前节点是在奇数列还是偶数列。例如，当一个数据包经过一个在偶数列的节点时，由东转向北和由北转向西的转弯被禁止。由于这种的限制，奇偶转弯模型通过禁止 180°转弯达到了无死锁。奇偶转弯模型比其他转弯模型算法如西向优先算法提供了更好的自适应性。使用西向优先算法，若从在目的地西向的源地址出发，就没有了灵活性；使用奇偶路由，就可以灵活性地根据给定的列所允许的转弯。

### 1．2．3　NoC 路由选择算法介绍

一般地，自适应路由算法可分为路由算法和选择算法两部分。路由算法部分将计算出可行的输出端口队列，之后再根据选择算法从可行的输出端口队列中选择最终的（选择算法认为的最优的）端口。

高效率和无死锁的路由对改善片上网络的性能来说至关重要。每一个自适应路由算法的效率都依赖于和其所结合的选择算法（Selection Algorithms）。路由算法可以计算出所有可选择的下一跳路由链接的本地输出端口，而选择算法可以在这些可选择的输出端口队列中选择一个它认为最合适的输出端口，从而确定消息包最终的前进方向。

选择算法在进行端口选择时的依据可分为两种：一种是从空间上分析，即通过分析相邻结点的状态信息来选择合适的下一跳结点；一种是从时间上分析，即通过在本地存储历史上从当前结点发往其它结点的延时等情况，来选择合适的下一跳结点。当然，两种依据可以相结合，比如可以分析相邻结点在历史上从其发往其它结点的延时等情况来选择合适的下一跳结点。下面介绍两种通过在空间上分析来进行选择的策略。

#### 1．2．3．1　输出缓冲水平

输出缓冲水平（Output Buffer Level，OBL）选择算法[21]的基本思想是，当一个路由器在转发一个消息包时，对由路由计算算法得到的各个可行输出端口链接的结点，分析它们输入输出缓冲区的占用率，并将其作为可以成功接收并转发当前消息包而不产生拥塞的概率，概率最大的结点链接的端口将被 OBL 选择算法选为当前消息包最终的输出端口。

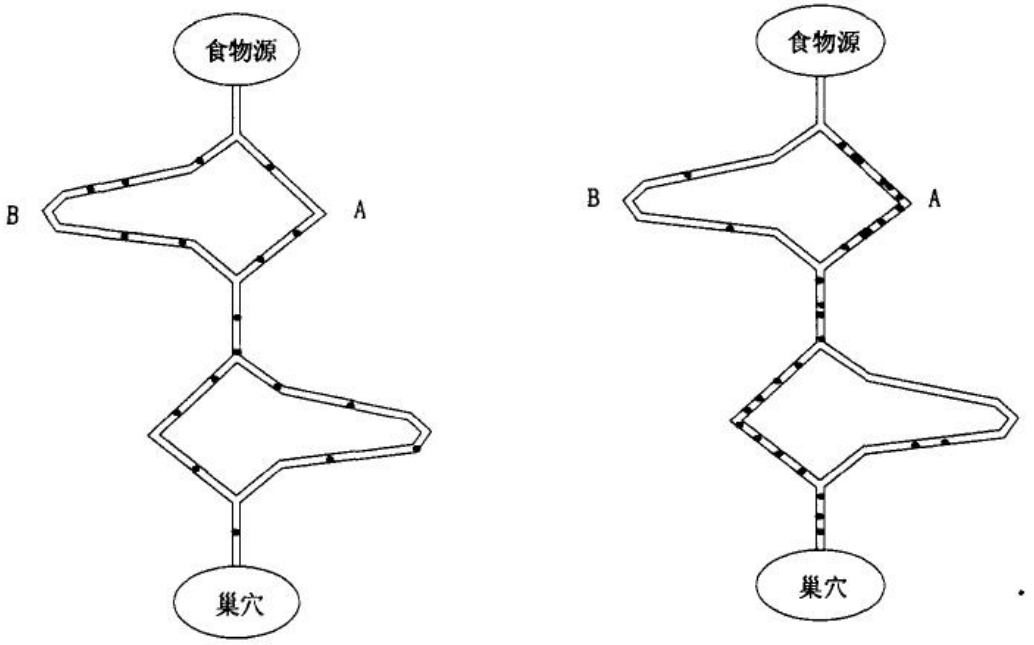
#### 1．2．3．2　路径邻居策略

路径邻居（Neighbors-on-Path，NoP）[22]选择算法的基本思想和 OBL 相似，只是将拓展分析的空间，即不仅分析直接相链接的结点，还会结合考虑与直接链接结点相邻的结点的当前状态。所以其基本思想可概括为，当一个路由器在转发一个消息包时，对由路由计算算法得到的各个可行输出端口链接的结点，分析它们可以成功接收并转发当前消息包而不产生拥塞的概率，并将其数值化，概率最大的结点链接的端口将被 NoP 选择算法选为当前消息包最终的输出端口。

## 1．3　蚁群优化及其研究现状

通过大量的观察研究，生物学家发现，蚂蚁的个体之间的信息交流与协作，是通过一种称为信息素的化学激素进行的[23]。蚂蚁在寻找食物的过程中，能够在它们走过的道路上留下这种信息素，且蚂蚁都能够感知这种化学物质。当蚂蚁在寻找食物或返回时，哪一条道路上的信息素量越大，对其的吸引力就越大，其选择走这条路的概率就越高，

从而又进一步增强了该条路的信息素浓度，表现出对信息的正向反馈的现象。通过这种间接的通讯机制，蚂蚁群体就可在寻找食物的过程中发现并选择最短路径行走了。Goss 等人在 1989 年公布的进行“双桥”实验[24]的结果也验证了这个生物学现象。如图 1.7 所示，图 1.7（a）为实验开始 4 分钟后双桥上蚂蚁的路径选择情况，图 1.7（b）为实验开始 8 分钟后双桥上蚂蚁的分布情况，可以看出，随着时间推移，越来越多的蚂蚁选择了最短的路径。




（a）4 分钟　　（b）8 分钟

图 1.7 非对称双桥实验

蚁群优化算法最初用于解决旅行商问题，自从在解决一的旅行商问题上取得成效以来，已经陆续渗透到其它诸多领域中。可以将这些应用分为两类，一类应用于静态组合优化问题，其典型代表有、二次分配问题、生产调度问题、车辆路由问题等另外一类应用于离散式动态组合优化问题中，其典型代表就是网络路由问题。在网络路由处理中，网络的流量分布不断变化，网络链路或结点也会随机地失效或重新加入。蚁群的自身催化与正向反馈机制正好符合了这类问题的求解特点，因此，蚁群算法在网络领域得到广泛应用。而在多核片上网络环境下，网络的流量分布也会受程序阶段影响而不断发生变化，且蚁群觅食行为所呈现出的并行与分布特性使得算法特别适合于并行化处理，因而，使用基于蚁群优化算法的思想进行片上网络路由选择算法的设计是具有可行性的。

## 1．4　FPGA 介绍及发展现状

现场可编程门阵列（Field Programmable Gate Array，FPGA）采用了逻辑单元阵列（Logic Cell Array, LCA）这样一个概念，内部包括可配置逻辑模块（Configurable Logic Block,

CLB）、输入输出模块（Input Output Block, IOB）和内部连线等单元模块。FPGA 编程是通过重新连接芯片本身的门电路来开发用户所需的功能，而不是像 PC 中的处理器一样运行一个软件应用。FPGA 是在可编程阵列逻辑（Programmable Array Logic，PAL）、通用阵列逻辑（Generic Array Logic，GAL）、复杂可编程逻辑器件（Complex Programmable Logic Devices，CPLD）等可编程器件的基础上进一步发展的产物。FPGA 是专用集成电路（ASIC）领域中的一种半定制电路，原有可编程器件门具有电路数有限等的缺点，FPGA 的出现既解决了定制电路的不足，又克服了原有可编程器件的缺点，具有体系结构和逻辑单元灵活、集成度高以及适用范围宽等特点。在 20 世纪 80 年代初，Xilinx 公司开发了第一块 FPGA，并于 1984 年投入市场。目前 FPGA 是数字系统设计的主要硬件平台，而同为 FPGA 厂商的两大巨头 Altera 公司和 Xilinx 公司，它们推出的大规模 FPGA 芯片内部的逻辑门数量已经达到了数百万门甚至更多[25]。

随着 FPGA 和 NoC 的逐渐活跃，也出现了越来越多的基于 FPGA 的 NoC 设计，使用硬件描述语言（ Hardware Description Language，HDL）设计硬件的优势在于容易理解、容易维护、电路的调试速度快，而且有许多易于掌握的仿真、综合和布局布线工具。目前也已经有很多研究机构公布了自己设计的基于 FPGA 的 NoC 核心路由器。使用 FPGA 实现 NoC 并进行仿真测试，对进一步研究 NoC 具有重要意义。

## 1．5　论文内容与结构

论文内容与章节安排如下。

第 1 章主要详细介绍了 NoC 研究背景以及国内外研究现状，同时指出了选题意义与选题目的；介绍了几种应用广泛的路由算法，作为之后路由算法设计描述的铺垫；介绍了蚁群优化思想，分析了基于蚁群优化思想设计路由选择算法的可能性；并简单介绍了 FPGA 的发展。

第 2 章详细介绍了本文中基于蚁群优化思想的自适应路由选择算法的设计。包括了网络消息包的格式设计、信息素表的相关设计以及具体的路由选择算法设计说明。

第 3 章主要说明本文基于 FPGA 的具体实现。本章首先简单介绍了 Altera Quartus Prime 15.1 FPGA 设计工具以及 FPGA 开发流程的四个主要步骤。之后详细说明了在具体实现中的 NoC 网络通信链接、路由器的总体架构以及网络模块、路由器模块和路由器中各个子模块的功能和实现。

第 4 章主要针对 XY+Random 、 Odd-Even+Random 、 Odd-Even+Buffer Level 、 Odd-Even+ACO 四种组合的路由算法在 Uniform、Transpose、Hotspot 三种输入下的功能仿真的测试结果进行统计分析，分析的性能指标包括网络平均吞吐率和平均包时延。

# 2．基于蚁群优化的自适应路由选择算法设计

## 2．1　总体设计

一般地，自适应路由算法可分为路由算法和选择算法两部分。本文选择奇偶转弯模型路由作为路由算法，采用基于蚁群优化思想的自适应路由选择算法作为选择算法。目前已被提出的基于蚁群思想的路由算法包括 Ant Based Control（ABC） Routing[26]，Ant Colony Based Routing Algorithm（ARA）[27]，Probabilistic Emergent Routing Algorithm（PERA）[28]，AntHocNet[29]，AntNet[30,31]等。经过分析，此类算法具有如下共同特征：

（1）在网络中多了蚂蚁包的设计。在除了载有有效载荷的消息包外，还有专门用于探测网络状态的蚂蚁包。在有些设计中，两种消息包是分开独立的；而在有些设计中会将有效载荷消息与蚂蚁包相结合，即将某些有效载荷包赋予蚂蚁包的功能。

（2）在每个路由器结点中都保存有一张信息素表。此信息素表对应着从此结点到网络中其它结点的所有可选输出端口的概率，也可以说，此信息素表记录着历史上蚂蚁包经过此结点时所留下的信息素信息（信息素浓度）。

（3）每当蚂蚁包（或从目的地返回的蚂蚁包）到达一个结点时，将会根据一定比例系数在信息素表已有值的基础上更新信息素表的值。在有的算法中，会考虑更多的因素来进行信息素表的更新，如当前蚂蚁包时延等。

（4）当路由器结点转发一个消息包时，将会根据信息素表上的信息素信息来选择下一跳结点。由于信息素表代表的是历史上的路由网络情况，所以在选择下一跳结点时，可再加上对当前网络状态的考虑，如邻居结点的输入输出缓冲水平等。几种考虑因素按一定比例系数分配各自的比重。

（5）信息素表上的值或将信息素表上的值与其它因素综合考虑得出的值，代表的是普通包选择该值对应的下一跳结点的概率，而并不是谁的值最大就选谁。也就是说，有较少的可能会出现选择信息素值较低的下一跳结点的情况。

本文工作中设计和实现的 ACO 路由选择算法是在 AntNet 算法基础上进行改造的，针对 FPGA 硬件实现的特点做了一定的调整。AntNet 算法采用以下公式基于信息素浓度和输入输出缓冲水平进行路由选择：

$$P'(j,d)=\frac{P(j,d)+\alpha L_i}{1+\alpha(|N_k|\text{-}1)},$$

其中 P(j,d)是信息素浓度，$L_i$ 是输入输出缓冲水平，$N_k$ 是当前结点的邻居结点数，$\alpha$ 是系数，其取值在 0.2-0.5 之间，但在文工作中实现的路由选择算法暂没有考虑输入输出缓冲水平，因此，$\alpha$ 取值为 0。P’(j,d)是路由选择依据的概率，本文工作中暂用该值直接选择下一跳。另外，对于信息素浓度的更新，其中，若结点在蚂蚁包的 memory 中，

那么该算法采用以下公式计算得出信息素浓度的新值：

$$P(\mathrm{i}) = P(i) + r \times (1 - P(i)),$$

否则采用以下公式得出信息素浓度的新值：

$$P(\mathrm{i}) = P(i) - r \times P(i)。$$

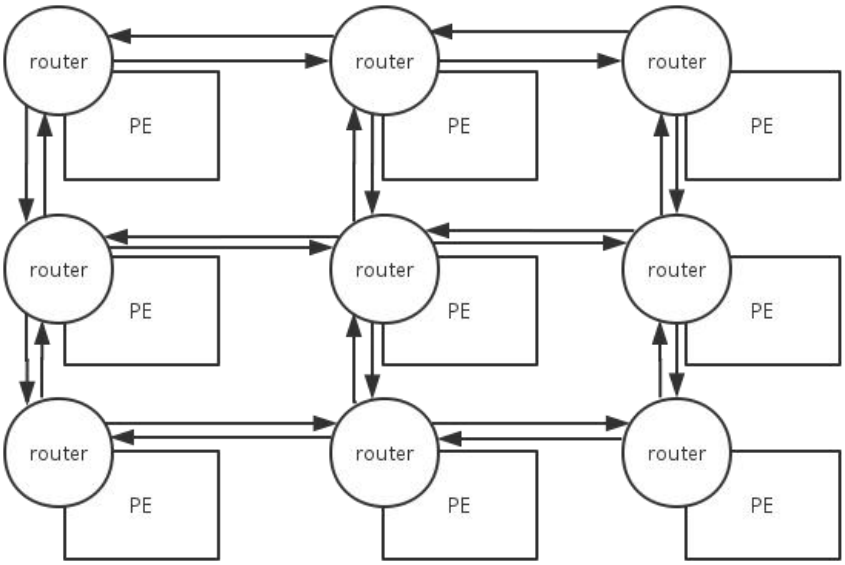



图 2.1 基于 2D-Mesh 拓扑的 3 x 3 网络结构

本章在之后将详细描述本文中的选择算法部分的设计。该算法的设计基于二维网格（2D-mesh）拓扑结，定义了 PE（Processing Element）结点和路由器（Router）结点之间的网络连接。如图 2.1 所示，是 3 x 3 的 2D-Mesh 拓扑结构的一个例子。每个 PE 结点都拥有一个路由器，且 PE 结点只能和自己的路由器进行直接交流，而路由结点不仅可与 PE 结点进行通信，还可与片上网络中的其它路由器结点进行通信，因此，PE 结点间只能通过路由结点进行通信。

## 2．2 消息包的格式设计

表 2.1 中列出了网络中消息包结构体的各个字段说明。

表 2.1 网络中消息包结构体字段

| 类型 | 名称 | 描述 |
|---|---|---|
| Logic [7:0] | id | 仿真时记录消息包的 id 号 |
| logic [$clog2(`X_NODES)-1:0] | x_source | 该消息的源结点地址 x 坐标 |
| logic [$clog2(`X_NODES)-1:0] | y_source | 该消息的源结点地址 y 坐标 |
| logic [$clog2(`X_NODES)-1:0] | x_dest | 该消息的目的地结点地址 x 坐标 |
| logic [$clog2(`X_NODES)-1:0] | y_dest | 该消息的目的地结点地址 y 坐标 |
| logic | ant | 标记蚂蚁包的类型。1: ant packet; 0: normal packet |
| logic | backward | 标记蚂蚁包的类型（仅为蚂蚁包时有效）。1: backward packet; 0: forward packet |
| logic [0:`NODES-1][$clog2(`X_NODES)-1:0] | x_memory | 蚂蚁包走过的路径队列（X 坐标） |
| logic [0:`NODES-1][$clog2(`Y_NODES)-1:0] | y_memory | 蚂蚁包走过的路径队列（Y 坐标） |
| logic [$clog2(`NODES)-1:0] | num_memories | memory 中记录的路径的结点数目 |
| logic | measure | 在仿真时记录测试状态 |
| logic [`TIME_STAMP_SIZE-1:0] | timestamp | 记录数据包进入网络时的时间 |

本文中的消息包和蚂蚁包各自独立。消息包通过 packet_t.ant 标记区分为两种数据包。每个结点在每个周期都根据概率产生消息包，而间隔一定周期产生蚂蚁包。

本文中将蚂蚁包路由过程设定为两个阶段：第一阶段（forward 阶段），从源结点出发，通过路由算法到达目的地结点，并记录路径中的每个结点和路径时延；第二阶段（backward 阶段），从目的结点返回，通过查找 forward 阶段记录的路径队列，从队列最后一个结点开始直到到达源结点；并在返回过程中更新信息素表。

## 2．3　信息素表的相关设计

每一个路由结点都使用一个信息素表，存储此结点根据蚂蚁包所形成的信息素的信息，使得所有进入此路由器的消息包都能够根据此信息素表选择能较快到达指定目的结点的输出链接，从而实现自适应路由算法。

（1）信息素表结构设计

表 2.2 信息素表设计结构

| Destination\Neighbor | 北 | 东 | 南 | 西 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| :<br>:<br>: | | | | |
| 结点数-2 | | | | |
| 结点数-1 | | | | |

由于在 FPGA 设计中无法动态地添加信息素表的条目，所以本文中将信息素表结构设计为固定的行数和列数。信息表的设计结构如表 2.2 所示，各行对应编号为 Destination 的目的结点，行数等于结点数，各列对应名称为 Neighbor 的输出端口，因为网络拓扑基于 2D-Mesh 结构，所以有北、东、南、西 4 列，因此可将信息素表中的某一格的值表示为 ph[Destination][Neighbor]。

（2）信息素表的更新

本文中的信息素表更新算法考虑到了每个蚂蚁包在网络中消耗的时延。每个蚂蚁都带最短路径（min_l）的 n 倍的信息素（total_ph）。ant packet 在 forward 路途中，自身携带的信息素每经过一个周期就减少 1（在 fifo 的 memory 里完成）。当 ant packet 开始 backward 阶段时便不再减少自身信息素量，并开始根据自身所剩的信息素量更新信息素表。具体算法如 2.1 所示。

```
if（有更新请求）begin
    if (8*实际路径长度 < 此蚂蚁包所剩的信息素量) 信息素更新量=6;
    else if (7*实际路径长度 < 此蚂蚁包所剩的信息素量 <= 8*实际路径长度)
信息素更新量=5;
    else if (6*实际路径长度 < 此蚂蚁包所剩的信息素量 <= 7*实际路径长度)
信息素更新量=4;
    else if (5*实际路径长度 < 此蚂蚁包所剩的信息素量 <= 6*实际路径长度)
信息素更新量=3;
    else if (3*实际路径长度 < 此蚂蚁包所剩的信息素量 <= 5*实际路径长度)
信息素更新量=2;
    else if (0*实际路径长度 < 此蚂蚁包所剩的信息素量 <= 3*实际路径长度)
信息素更新量=1;
    else 信息素更新量=0;

    If（当前行中申请更新的对应信息素值 < （最大值 - 信息素更新量））
        此信息素值 = 此信息素值 + 信息素更新量;
    else
        此信息素值 = 最大值;
end else begin
    for(int d=0;d<`NODES;d++)begin
        if(信息素表某一行的某个值达到最大) begin
            本行所有的信息素值除以 2;
        end
    end
end
```

算法 2.1 信息素表的更新算法

## 2．4 路由算法详细设计

因为基于蚁群优化的路由选择算法中涉及到对消息包数据的修改，且在将路由算法部分和选择算法部分相结合时，也需要进行相应控制。在此考虑设计一个代理模块，从输入缓冲区出来的消息包先进入代理模块，进行对消息包的过滤操作、路由计算和选择的操作，然后再交给交换模块传输到输出端口。因此，路由算法部分设计成三个模块，消息包处理（Agent）模块、路由算法（Routing）模块、选择算法（Selection）模块。

### 2．4．1　消息包处理流程设计

消息包处理（Agent）模块的具体流程设计如算法 2.2 所示。

```
if（有数据输入） begin
    if（是普通消息包） begin
        if（消息未到达目的地） begin
            给 routing 模块发送 路由计算 信号;
        end else begin // 到达目的地
            直接输出给 PE 结点;
        end
    end else begin // 是人工蚂蚁包
        if（是 forward 阶段的蚂蚁包） begin
            记录本地结点;
            if（消息包未到达目的地） begin
                给 routing 模块发送 路由计算 信号;
            end else begin // 消息包到达目的地
                将 forward 阶段的蚂蚁包 标为 backward 阶段;
                交换蚂蚁包的源地址和目的地地址;
                if（消息包未到达目的地） begin
                    根据 forward 阶段记录的路径往回发送蚂蚁包;
                end else begin
                    直接输出给 PE 结点;
                end
            end
        end else begin // 是 backward 阶段的蚂蚁包
            给 selection 模块发送更新表信号; // 请求更新信息素表
            if（消息包未到达目的地） begin
                根据 forward 阶段记录的路径往回发送蚂蚁包;
            end else begin
                直接输出给 PE 结点;
            end
        end
    end
end
```

算法 2.2 消息包处理流程算法

## 2．4．2　路由算法设计

在本文中的路由计算部分将采用奇偶转弯模型路由（Odd-Even Turn Model Routing）[32]。前面提到 2D-Mesh 拓扑中每个消息包都有 8 种转弯，而奇偶路由算法对消息包的转弯方向进行了以下限制：当数据包位于偶数列的结点时，不能进行由东向北和由东向南的转弯；当数据包位于奇数列的结点时，不能进行由北向西和由南向西的转弯。

通过路由算法，计算出可选择的输出端口队列。路由算法可包括 X-Y 路由、奇偶转弯模型路由等，根据具体的路由算法进行路由计算时需要的具体信息条件，可再创建更多的输入端口以输入需要的信息。

表 2.3 中列出了路由算法（Routing Algorithms）模块的各个端口设计。

表 2.3 路由算法模块端口设计

| 输入参数类型 | 输入参数名称 | 输入参数描述 |
|---|---|---|
| integer | X_LOC | 当前结点的 X 坐标 |
| integer | Y_LOC | 当前结点的 Y 坐标 |

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic [0:`N-1] | i_routing_calculate | 路由计算使能信号 |
| logic [0:`N-1][$clog2(`X_NODES)-1:0] | i_x_dest | 输入数据包目的地的 x 坐标 |
| logic [0:`N-1][$clog2(`Y_NODES)-1:0] | i_y_dest | 输入数据包目的地的 y 坐标 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| logic [0:`N-1] | o_select_neighbor | 输出选择使能信号 |
| logic [0:`N-1][0:`M-1][1:0] | o_avail_directions | 输出可选择的输出端口队列 |

本文中的奇偶转弯路由算法如算法 2.2 所示。该算法是在消息包未到目的地的情况下执行，输出可选择的输出端口队列，并发送选择信号。

```
X_LOC=当前结点 x 坐标; Y_LOC=当前结点 y 坐标; i_x_source=消息包源结点 x 坐标;
i_x_dest=消息包目的地 x 坐标; i_y_dest=消息包目的地 y 坐标;
if(路由计算信号)begin
    if(i_x_dest == X_LOC) begin //  到达目的地 x 维
        if(i_y_dest > Y_LOC)   //  目的地 y 坐标  >  当前 y 坐标
            将北输出端口放入可行输出端口队列;
        else //  目的地 y 坐标  <  当前 y 坐标
            将南输出端口放入可行输出端口队列;
    end else begin
        if(i_x_dest > X_LOC) begin //  目的地 x 坐标  >  当前 x 坐标
            if(i_y_dest == Y_LOC) begin //  到达目的地 y 维
                将东输出端口放入可行输出端口队列;
            end else begin
                if(X_LOC % 2 == 1 || X_LOC == i_x_source) begin //  当前 x 坐标值为奇
数  或  当前 x 坐标==源结点 x 坐标）
                    if(i_y_dest > Y_LOC)   //  目的地 y 坐标  >  当前 y 坐标
                        将南输出端口放入可行输出端口队列;
                    else   //  目的地 y 坐标  <  当前 y 坐标
                        将东输出端口放入可行输出端口队列;
                end
                if(i_x_dest % 2==1||((i_x_dest - X_LOC != 1)&&(X_LOC - i_x_dest != 1)))
                    将东输出端口放入可行输出端口队列;
            end
        end else begin// else（目的地 x 坐标  <  当前坐标）
            将西输出端口放入可行输出端口队列;
            if(X_LOC % 2 == 0)   // if（当前 x 坐标为偶)
                if(i_y_dest > Y_LOC) // if（目的地 y 坐标  >  当前 y 坐标）
                    将南输出端口放入可行输出端口队列;
                if(i_y_dest < Y_LOC) // if（目的地 y 坐标  <  当前 y 坐标）
                    将南输出端口放入可行输出端口队列;
        end
    end
    发送选择信号; //  路由计算结束就发送选择信号
end
```

算法 2.2  奇偶转弯路由算法

## 2．4．3　选择算法设计

本文中基于蚁群优化的思想，设计了路由选择算法，在可选择的输出端口队列中选择一最优输出端口，从而实现自适应、无死锁的路由。

表 2.3 中列出了选择算法（Selection Strategies）模块的各个端口设计。

表 2.4 选择算法模块端口设计

| 输入参数类型 | 输入参数名称 | 输入参数描述 |
|---|---|---|
| integer | X_LOC | 当前结点的 X 坐标 |
| integer | Y_LOC | 当前结点的 Y 坐标 |

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic | clk | 时钟信号 |
| logic | reset_n | 复位信号 |
| logic [0:`N-1] | i_select_neighbor | 进行输出端口选择的使能信号 |
| logic [0:`N-1][0:`M-1][1:0] | i_avail_directions | 可选择的输出端口队列 |
| logic [0:`N-1][$clog2(`X_NODES)-1:0] | i_x_dest | 输入数据包的目的地的 x 坐标 |
| logic [0:`N-1][$clog2(`Y_NODES)-1:0] | i_y_dest | 输入数据包的目的地的 y 坐标 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| logic [0:`N-1][0:`M-1] | o_output_req | 通过选择算法分别得到的对应 N 个输入数据请求的输出端口 |

本文中采用的 ACO 选择算法如算法 2.3 所示。

```
max value =  最小值; //  最大信息素值，初始化为最小
max column = '0; //  最大信息素值对应的输出端口
d =  消息包目的地

if（选择信号）
  if（第 i 个可行输出端口！ = parent 端口）begin
    //  判断最大信息素值，并选择对应的端口
     if（max value < ph[d][第 i 个可行输出端口]）  begin
        max value = ph[d][第 i 个可行输出端口];
        max column =  第 i 个可行输出端口;
      end
    end
    If（此消息包是数据包）begin
          输出端口  = max column; // ACO 选择
    end else begin
      if（可选择输出端口队列的 n 个端口在信息素表中对应的信息素有为 0 的）
begin
          max column =  该可行输出端口;
       end else begin
          输出端口  = max column; // ACO 选择
       end
    end
  end

  if（更新表信号）begin
    更新信息素表;
  end
```

算法 2.3　ACO 选择算法

算法 2.3 主要包括了选择最优输出端口和更新信息素两部分。在选择最优输出端口时，通过比较各个可选择端口对应的信息素浓度，选择浓度最高的端口作为最终的输出端口。但若发现有可选择端口对应的信息素浓度为 0，则在这些信息素浓度为 0 的端口中随机选择一个端口作为最终的输出端口。在更新信息素表时，会调用信息素表更新算法。

# 3．FPGA 实现

## 3．1　开发环境与开发流程

### 3．1．1　开发环境

本课题计划在 Ubuntu Linux 15.04 操作系统上，基于 Altera Quartus Prime 15.1 FPGA 设计工具，设计和实现基于蚁群优化的片上网络自适应路由算法。在设计输入中采用的语言为 SystemVerilog。Quartus Prime 软件是 altera 公司提供的综合性 FPGA 开发软件，支持的设计输入形式有原理图、Verilog、SystemVerilog、VHDL 等；软件内部嵌有综合器和仿真器；支持 Altera 的 IP 核，包含了 LPM/MegaFunction 宏功能模块库，使开发者可以充分利用成熟的模块，从而简化设计的复杂性，加快设计速度；软件也提高了设计流程中各个阶段要使用的工具，利用这些工具可以独立完成具有 Altera 特色的、从设计输入到硬件配置的完整的 FPGA 开发流程。

### 3．1．2　FPGA 开发流程

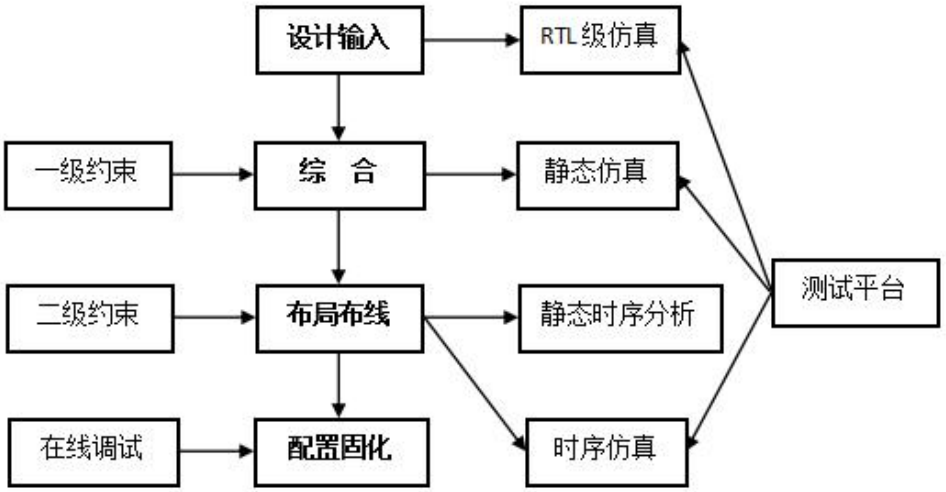



图 3.1 FPGA 的开发流程

如图 3.1 所示，FPGA 的开发流程可分为如下四个主要步骤[29]：

（1） 设计输入。设计输入方式有 3 种：原理图、HDL 和 IP 核。其中 HDL 语言具有不同层次上的抽象，这些抽象层包含开关级、逻辑门级、RTL 级（Register Transfer Level，寄存器传输级）、行为级和系统级。因为 Verilog 拥有较广泛的设计群体和其与 C 语言的相似性，且考虑到 Verilog 虽然在仿真方面具有高层建模能力不足的缺陷，但 SystemVerilog 可以在系统级和行为级上为 Verilog 做补充，所以本次设计选用的 HDL 语

言 SystemVerilog。其中，RTL 级仿真（或功能仿真）属于第一道测试，对工程在寄存器描述时进行测试，查看其在 RTL 级描述功能的正确性。

（2） 综合。对设计输入进行综合，得到一个可以和 FPGA 硬件资源相匹配的描述。假设 FPGA 是基于 LUT 结构的，那么就得到一个基于 LUT 结构的门级网表。其中，一级约束即综合约束，用来指导综合过程，是小范围内实现运行速度和资源消耗平衡的一种方式。不同的约束，将会产生性能不同的电路。另外，静态仿真（或门级仿真）：是综合后 LUT 门级网表的仿真，目的是当工程用 LUT 门级描述时，从功能上验证工程的正确性。

（3） 布局布线。布局考虑的问题是如何将这些逻辑上已连接的 LUT 及其他元素合理地放到现有的 FPGA 里，并且达到功能要求的同时保证质量。布线考虑的问题就是线路最优问题，具体来说就是如何让各部分连接起来，如何让输入输出信号到达相应的位置，且保证电路连接后的整体性能。其中，二级约束：即布局布线约束，可分为位置约束和时序约束。位置约束指布局策略，根据所选择的 FPGA 平台现有的硬件资源分布来决定布局。时序约束在很大程度上和布线有关，但是是先用软件默认的原则布线，然后对其结果进行静态时序分析，不满足时序要求的，再对具体的问题路径做一些指导约束。另外，布线时时延问题的截获就可以通过时序仿真完成。将工程下载到 FPGA 芯片上，可通过在线调试（或板级调试）分析代码运行的情况。

（4） 配置及固化。在配置模式和初始化模式下，FPGA 的用户 I/O 处于高阻态（或内部弱上拉状态），这两个模式相继结束后，进入用户模式，此时用户 I/O 就能够按照用户设计的功能工作。固化既是将程序固化到存储器中。

## 3．2 具体实现

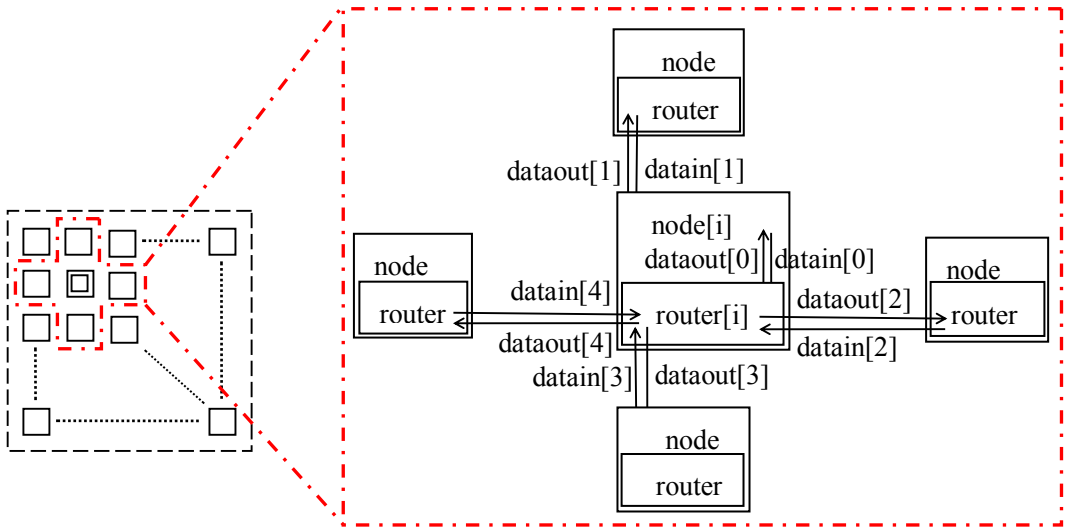



图 3.2 基于 SystemVerilog 的片上网络结点间的详细链接图

如图 3.2 所示，我们可以在 FPGA 中模拟片上网络及其路由算法。基于 2D-Mesh 拓扑结构（图 3.3 为 4x4 2D-Mesh 拓扑结构及其路由器编号例子），定义 PE 结点(nodes)和路由器(routers)间的网络连接，即解决在结点和路由器、路由器和路由器之间，谁输出到谁，谁从谁输入的问题。在网络(Network)模块中，每个结点都拥有自己的路由器。每个结点只能与自己的路由器传输数据，而每个路由器则能和自己的上下左右四个方向的路由器传输数据，所以，每个路由器可定义 5 个端口(ports)。例如，在图 3.2 中，datain[i][0]/dataout[i][0]表示 router[i]从 node[i]接收的数据/router[i]输出到 node[i]的数据，datain[i][1]/dataout[i][1]表示 router[i]从其上方 router 接收的数据/router[i]输出到其上方 router 的数据，以此类推。其中，i 表示路由器编号。

| | | | |
|---|---|---|---|
| 12 | 13 | 14 | 15 |
| 08 | 09 | 10 | 11 |
| 04 | 05 | 06 | 07 |
| 00 | 01 | 02 | 03 |

图 3.3　4x4 2D-Mesh 拓扑结构

## 3．3．1　路由器总体架构

系统的顶层模块是 Network 模块，Network 模块包含若干个 Router 模块，每个 Router 模块中均包含 Ant Agent 模块用于计算路由。

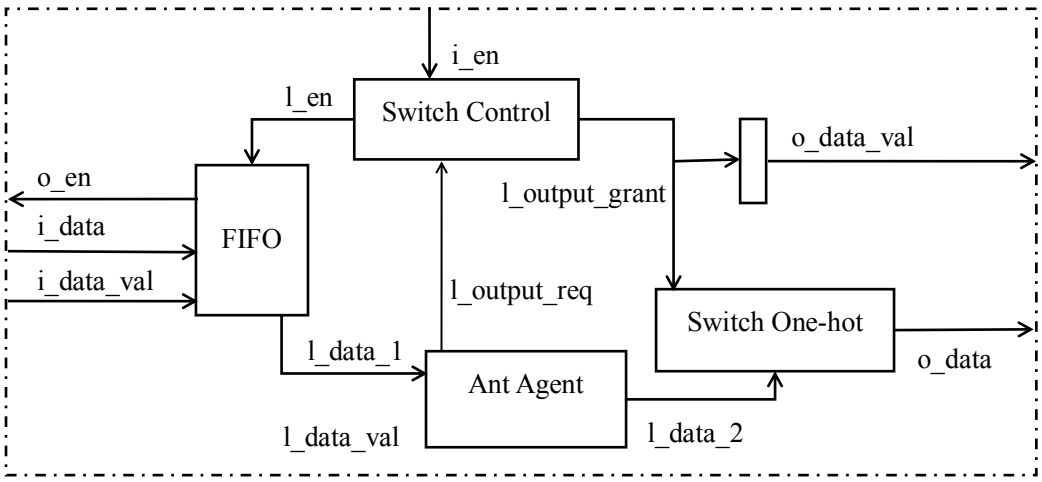



图 3.4　基于 SystemVerilog 的片上网络路由器内部结构图

在图 3.4 中，路由器（Router）模块包含 3 个输入端口（i_en,i_data,i_data_val）和 3 个输出端口（o_en,o_data,o_data_val）。输入的数据先发送给 FIFO 模块，FIFO 模块能在有数据输入而 l_en 为 0（请求输出方向不允许输出）时，将输入数据进行缓冲，并在自身缓冲空间满时，向周围结点或路由器发送输出使能信号(o_en)。路由代理(Ant

Agent)模块包含路由算法和选择算法两个子模块，能基于输入数据中的目的地等信息，根据路由算法计算出数据的输出方向，并将请求发送给开关控制模块。开关控制(Switch Control)模块通过其它路由器发送来的输入使能信号（i_en）、数据请求的输出方向以及可输出端口的轮换情况，给 FIFO 模块下达是否可输出数据的信号(l_en)，并给 Switch One-hot 模块发送输出请求应答信息。Switch One-hot 模块根据应答信息，将 FIFO 发送来的数据发往相应的输出方向。

## 3．3．2　各个模块功能和端口设计

### 3．3．2．1　网络模块功能和端口设计

每个路由结点都有 5 个输入输出端口，其中一个端口链接 PE 结点，其余的端口链接周围的路由器。通过[结点号 i][输入输出端口号 j]结构表示第 i 个结点第 j 个端口的输入输出，其中 j=0 表示与 PE 结点链接的端口、j=1 表示与北路由结点链接的端口、j=2 表示与东路由结点链接的端口、j=3 表示与南路由结点链接的端口、j=4 表示与西路由结点链接的端口。

表 3.1 中列出网络（Network）模块的各个端口设计。

表 3.1 Network 模块端口设计

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic | clk | 时钟信号 |
| logic | reset_n | 复位信号 |
| packet_t [0:`NODES-1] | i_data | 从 PE 结点接收数据，传入本地路由器 |
| logic [0:`NODES-1] | i_data_val | 指出输入端口（i_data）是否有数据输入 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| packet_t [0:`NODES-1] | o_data | 将从路由器接收到的数据输出到本地 PE 结点 |
| logic [0:`NODES-1] | o_data_val | 指出输出端口（o_data）是否有数据输出 |
| logic [0:`NODES-1] | o_en | 输出输入端口（i_data）的使能信号 |

### 3．3．2．2　路由器模块功能和端口设计

路由器（Router）模块拥有 5 个输入端口[local, north, east, south, west]、5

个输出端口[local, north, east, south, west]，分别用于与PE结点、北路由结点、东路由结点、南路由结点、西路由结点链接。模块对应单个路由结点的完整架构，其中包括输入输出缓存（FIFO）模块、消息包处理（agent）模块、路由算法(routing)模块、选择算法(selection)模块、开关控制(switch control)模块、交换开关(switch one-hot packet）模块。

表3.2中列出了路由器（Router）模块的各个端口设计。

表3.2 Router模块功能和端口设计

| 输入参数类型 | 输入参数名称 | 输入参数描述 |
|---|---|---|
| integer | X_LOC | 当前结点的X坐标 |
| integer | Y_LOC | 当前结点的Y坐标 |

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic | clk | 时钟信号 |
| logic | reset_n | 复位信号 |
| packet_t [0:`N-1] | i_data | 五个输入数据的端口 |
| logic [0:`N-1] | i_data_val | 指出五个输入端口（i_data）是否有数据输入 |
| logic [0:`N-1] | i_en | 输入五个数据输出端口（o_data）使能控制信号 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| packet_t [0:`M-1] | o_data | 五个输出数据的端口 |
| logic [0:`M-1] | o_data_val | 指出五个输出端口是否有数据需要输出 |
| logic [0:`M-1] | o_en | 输出五个数据输入端口（i_data）的使能信号 |

### 3．3．2．3　路由器内部子模块功能和端口设计

（1）fifo_packet模块

该模块采用了同步时序逻辑，使用存储阵列缓存由网络仿真框架生成特定的数据包类型。内存中每个单元都关联读和写两种指针，以控制内存的读写。每个指针用一位的二进制数表示，同一时刻读指针和写指针都只有一个能为高位。通过读、写请求，读写指针的高位在内存单元间轮转，使得同一时刻只有一个单元可读和一个可写。并向上游路由结点发送缓存是否已满的信号。

表3.3中列出了fifo_packet模块的各个端口设计。

表 3.3 fifo_packet 模块端口设计

| 输入参数名称 | 输入参数描述 |
| --- | --- |
| DEPTH | 输入缓存队列长度 |

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
| --- | --- | --- |
| logic | clk | 时钟信号 |
| logic | ce | 使能信号 |
| logic | reset_n | 复位信号 |
| packet_t | i_data | 需存入 FIFO 的 packet_t 结构的数据输入端口 |
| logic | i_data_val | 指出输入端口（i_data）是否有数据输入 |
| logic | i_en | 输入数据输出端口（o_data） 的使能控制信号，控制数据的输出，当一个时钟上升沿来临时，若使能信号为高，则允许数据输出 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
| --- | --- | --- |
| packet_t | o_data | 需从 FIFO 发送的 packet_t 结构的数据输出端口 |
| logic | o_data_val | 指出是否有数据要从 o_data 端口输出，当 o_data_val=1 时，表示有数据输出 |
| logic | o_en | 输出数据输入端口（i_data） 的使能信号，表示缓存队列是否有空闲，若使能信号为高，则表示输入的数据可以被写入内存 |

（2）ant_agent 模块

该模块对需要输出的数据进行必要的处理和路由计算，并将处理后的数据发送给交换开关（switch one hot）模块，将路由计算后得到的输出请求发送给交换控制（switch control）模块。

表 3.4 中列出了 ant_agent 模块的各个端口设计。

表 3.4 ant_agent 模块端口设计

| 输入参数类型 | 输入参数名称 | 输入参数描述 |
| --- | --- | --- |
| integer | X_LOC | 当前结点的 X 坐标 |
| integer | Y_LOC | 当前结点的 Y 坐标 |

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic | clk | 时钟信号 |
| logic | reset_n | 复位信号 |
| packet_t [0:`N-1] | i_data | 输入的数据 |
| logic [0:`N-1] | i_data_val | 指出是否有数据输入 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| packet_t [0:`M-1] | o_data | 数据输出端口，输出给 switch one hot 模块 |
| logic [0:`N-1][0:`M-1] | o_output_req | 路由器 N 个输入端口的对输出端口的请求情况，输出给 switch control 模块 |

（3）switch_control 模块

交换控制。通过其它路由器发送来的输入使能信号（i_en）、数据请求的输出方向以及可输出端口的轮换情况，控制输入缓存通道的数据输出到各个输出端口。表 3.5 中列出了 switch_control 模块的各个端口设计。

表 3.5 switch_control 模块端口设计

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic | clk | 时钟信号 |
| logic | ce | 使能信号 |
| logic | reset_n | 复位信号 |
| logic [0:`M-1] | i_en | 从下游路由结点发来的信号，指出对应的下游路由结点是否可用（即是否有空闲） |
| logic [0:`N-1][0:`M-1] | i_output_req | N 个本地输入单元对 M 个输出端口的请求情况 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| logic [0:`M-1][0:`N-1] | o_output_grant | M 个输出端口分别对 N 个输入端口的请求作出的应答，输出给 switch onehot packet 模块 |
| logic [0:`N-1] | o_input_grant | 对 N 个输入缓存队列的应答信号，输出给 agent 模块 |

（4）switch_onehot_packet 模块

该模块实现了一个 N x M 交换开关，包括 N 个输入端口，M 个输出端口。每个输出端口分别对应一个 N 位的二进制数，N 位分别对应 N 个输入端口。输入端口的选择过程采用 one-hot 编码，即一位二进制数对应一个输入端口，且同一时刻最多只允许一位为 1。输出端口根据自己对应的 N 位 one-hot 编码选择一个输入端口。在本文中，5 位 one-hot 编码对应的请求输出端口分别对应[local, north, east, south, west]。

表 3.6 中列出了 switch_onehot_packet 模块的各个端口设计。

表 3.6 switch_onehot_packet 模块端口设计

| 输入端口类型 | 输入端口名称 | 输入端口描述 |
|---|---|---|
| logic [0:`M-1][0:`N-1] | i_sel | M 个 N 位 one-hot 编码 |
| packet_t [0:`N-1] | i_data | N 个数据输入端口 |

| 输出端口类型 | 输出端口名称 | 输出端口描述 |
|---|---|---|
| packet_t [0:`M-1] | o_data | M 个数据输出端口 |

# 4．FPGA 仿真实验与结果分析

## 4．1　测试平台

测试平台（Testbench）的编写是整个电路设计过程中不可缺少的一环。编写测试平台的主要目的是对用硬件描述语言（HDL）设计的电路进行仿真验证，测试设计电路的逻辑关系是否正确、验证电路功能和部分性能是否与预期相符。具体的说，测试平台向待测试模块输入已知的序列信号，然后观测待测模块的输出信号。输入信号的产生可以通过编程实现，也可以调用外部数据文件；输出信号的观测可以通过波形显示，或保存成文本形式等方法进行对比[33]。

编写测试平台进行测试的过程如下：
（1）产生模拟激励（波形），如时钟、复位等激励；
（2）将产生的激励加入到被测试模块并观察其响应的输出；
（3）将响应的输出与期望进行比较，从而判断设计的正确性。

本仿真测试中测试平台的基本结构如算法 4.1 所示。

```
`timescale 100ns/1ns   // 定义时间单位与时间精度
    module tb_network
    #(
        //定义周期时间、消息包注入率、测试包数量等参数
     )
        //测试信号定义声明
        //吞吐率、平均包时延等需测试的性能指标变量定义声明
        //例化设计模块
        //仿真模块需要的输入
        //使用 initial 或 always 语句来产生时钟、复位等激励（波形）
        //监控和记录模块输出响应
    endmodule
```

算法 4.1　测试平台的基本结构

## 4．2　实验配置参数和 NoC 路由算法性能测试指标

（1）实验配置参数

本文中所有实验均分为三个阶段，预热（warmup）阶段、详细测试（measure）阶段和排空（drain）阶段，只有在详细测试阶段进行性能测试。表 4.1 中给出了本实验中用到的配置参数。其中，消息包注入率（Packet Injection Rate，PIR），是指平均每个周期每个结点发送的消息包个数。

表 4.1 实验配置参数及描述

| 名称 | 取值 | | | | | | 描述 |
|---|---|---|---|---|---|---|---|
| num_nodes | 16 | | | | | | 网络中总共的结点数 |
| warmup_time | 1000 | | | | | | 用此周期数预热好网络状态 |
| measure_time | 10000 | | | | | | 详细测试阶段周期数 |
| drain_time | 3000 | | | | | | 维持当测试包发送完成后但未完全回收时网络状态 |
| max_cycles | 17000 | | | | | | 测试所允许的最大周期数，可根据测试过程调整 |
| router_input_queue_depth | 4 | | | | | | 路由器结点的各个端口的输入缓存队列长度 |
| PEnode_input_queue_depth | 20 | | | | | | PE 结点输入缓存队列长度 |
| create_ant_packet_period | 100 | | | | | | 生产蚂蚁包的间隔周期数 |
| pheromone_table_value_width | 8 | | | | | | 信息素表每一格值的位宽 |
| packet_injection_rate | 1 | 5 | 10 | 20 | 50 | 60 | 数据包的网络注入率 |
| aco_packet_injection_rate | 与 packet_injection_rate 取值相同 | | | | | | 蚂蚁包的网络注入率 |
| hotspot_packet_injection_rate | 10 | 20 | 30 | 50 | 100 | 100 | 热点数据包的网络注入率 |

（2）NoC 路由算法性能测试指标

对于路由算法的设计要求，不仅仅是简单的可运行和路由正确性，还要同时考虑网络性能、死锁、拥塞等各个方面。而在 NoC 路由算法的研究中，对于 NoC 的网络性能的衡量至关重要。本文将对以下两种性能测试指标进行具体分析。

吞吐率（Throughput），指每个周期内网络的每个节点平均可以发送或接收的数据量，其取决于路由、仲裁和网络拓扑[24]，其计算公式如下：

$$\text{吞吐率} = \frac{\text{测试周期内接收到的总的消息包数目}}{\text{总的测试周期数} \times \text{网络中总的结点数}}$$

平均消息包时延（Average Packet Delay），是数据包从源节点到目标节点所要消耗的平均时间代价，这不仅仅与网络路由结构、仲裁和网络拓扑，也与路由算法的优劣有关，其计算公式如下：

$$\text{平均消息包时延} = \frac{\sum \text{每个消息包的时延}}{\text{总的消息包数目}}$$

表 4.2 中列出了本文在测试平台中对实验结果进行统计的各个指标。

表 4.2 测试平台中实验结果统计指标

| 指标命名 | 指标描述 |
|---|---|
| total_cycles | 发送、接收所有阶段仿真包所用的周期数 |
| measure_cycles | 发送、接收详细测试阶段仿真包所用的周期数 |
| throughput | 记录详细测试阶段全部消息包的网络吞吐率（吐出率） |
| num_packets_transmitted | 记录详细测试阶段发送的全部消息包的数量 |
| num_packets_received | 记录详细测试阶段接收到的全部消息包的数量 |
| average_packet_delay | 记录详细测试阶段全部消息包的平均包时延 |
| max_packet_delay | 记录详细测试阶段全部消息包中的最大包时延 |
| packet.throughput | 记录详细测试阶段数据包的网络吞吐率（吐出率） |
| packet.num_packets_transmitted | 记录详细测试阶段发送的数据包的数量 |
| packet.num_packets_received | 记录详细测试阶段接收到的数据包的数量 |
| packet.average_packet_delay | 记录详细测试阶段数据包的平均包时延 |
| packet.max_packet_delay | 记录详细测试阶段数据包中的最大包时延 |
| acopacket.throughput | 记录详细测试阶段蚂蚁包的网络吞吐率（吐出率） |
| acopacket.num_packets_transmitted | 记录详细测试阶段发送的蚂蚁包的数量 |
| acopacket.num_packets_received | 记录详细测试阶段接收到的蚂蚁包的数量 |
| acopacket.average_packet_delay | 记录详细测试阶段蚂蚁包的平均包时延 |
| acopacket.max_packet_delay | 记录详细测试阶段蚂蚁包中的最大包时延 |

## 4．3　结果分析

在仿真测试中采用了三种输入模式（Traffic）来对算法进行测试，包括 Uniform、Transpose、Hotspot 三种。

Uniform 输入模式是指每个结点按正常的消息包注入率生成消息包，消息的目的地是随机选择的。

Transpose 输入模式是指每个结点按正常的消息包注入率生成消息包，消息的目的地与消息包的源地址关于某直线对称（在 2D-mesh 结构中一般采用关于对角线对称）。

Hotspot 输入模式是指非热点结点按正常的消息包注入率生成消息包，热点结点则按更高的消息包注入率生成消息包。同样地，消息的目的地是随机选择的。

本文中测试记录了消息包的网络吞吐率和平均包时延两种性能指标。基本实验结果见附表 1-3，其中，相较于 Odd-Even+Buffer Level 算法，Odd-Even+ACO 算法在 Uniform、Transpose、Hotspot 三种输入模式下能达到的吞吐率最大提升幅度为+23.81%（PIR=0.5）、+16.69%（PIR=0.5）和-0.96%（PIR=0.01），能达到的平均包时延最大降低幅度为+3.90%（PIR=0.2）、+9.73%（PIR=0.01）和+7.18%（PIR=0.6）。下面将分别对结果记录的这两种性能指标进行详细分析。

## 4．3．1　吞吐率

图 4.1-4.3 分别给出了 xy+random、odd even+random、odd even+buffer level、odd even+aco 四种组合的路由算法在 Uniform、Transpose、Hotspot 三种输入下的吞吐率（Throughput）。

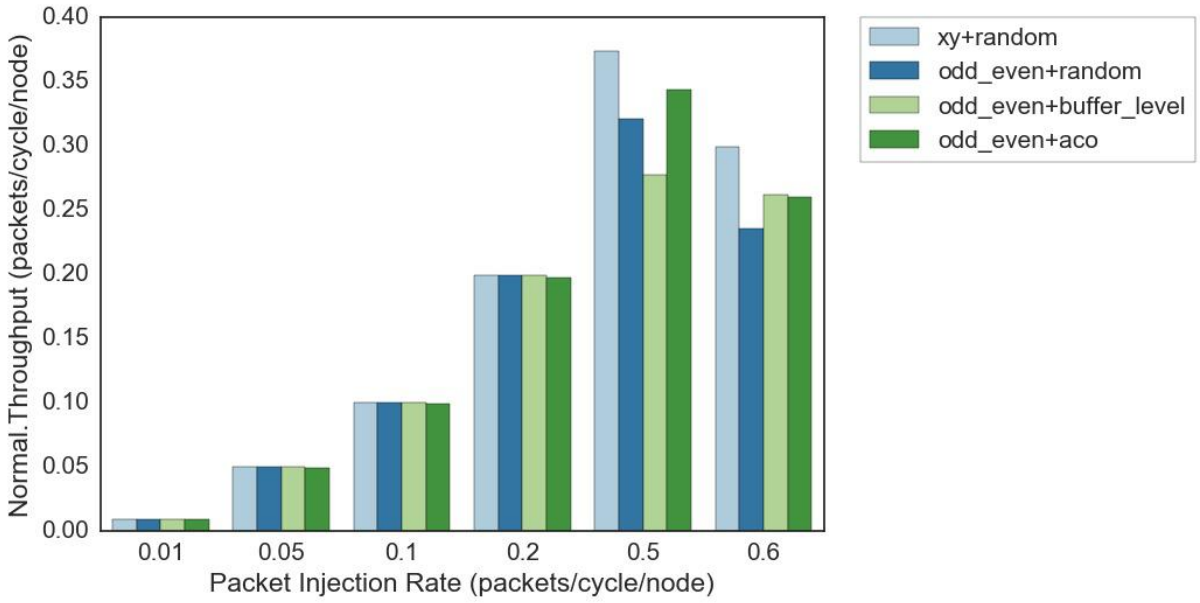



图 4.1 Uniform 输入下的吞吐率

图 4.1 是在 Uniform 输入模式下的仿真测试结果。可以看出，在消息包注入率较低（在未达到 0.5 时）的情况下，4 种组合的路由算法吞吐率在此网络中是平衡的。从 0.5 的注入率可以看出，随着注入率的增加，所有算法的吞吐率都受到了牵制，均未到达

50%。从 0.5 到 0.6 的注入率可以看出，4 种组合的路由算法的吞吐率都无法再上升。在注入率为 0.5 时，odd even+aco 组合的路由算法的吞吐率为 0.344，会优于 odd even+random、odd even+buffer level 组合的路由算法，但未超过 xy+random 组合的路由算法。而在注入率为 0.6 时，只有 odd even+buffer level 组合的路由算法的吞吐率未明显下降，已经略微超过了 odd even+aco 组合的路由算法的吞吐率。

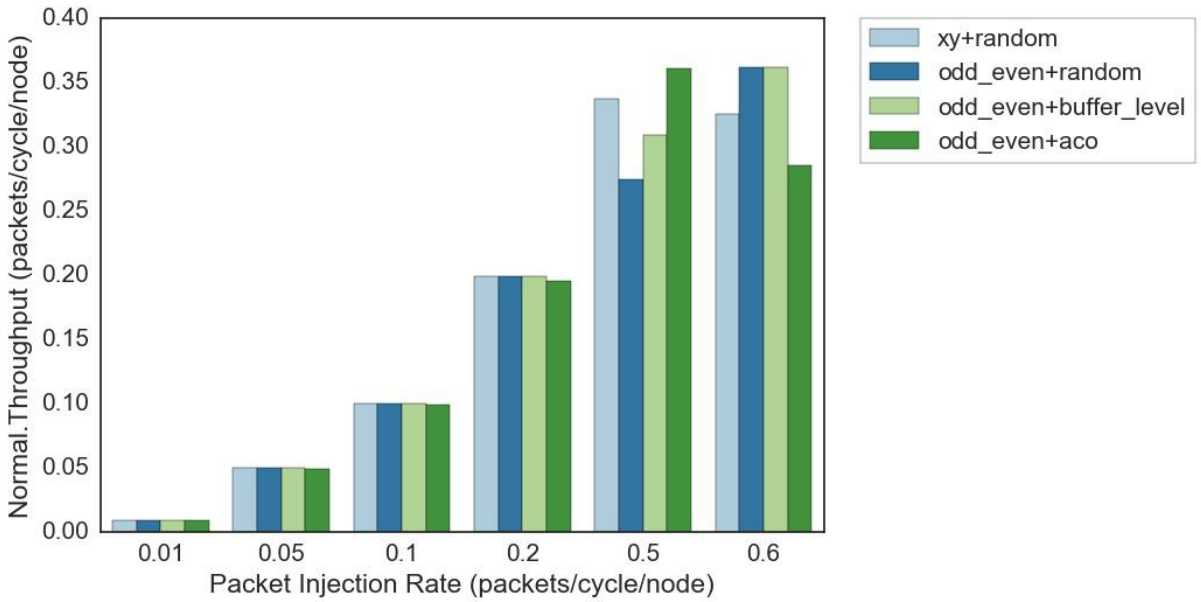



图 4.2 Transpose 输入下的吞吐率

图 4.2 是在 Transpose 输入模式下的仿真测试结果。可以看出，在消息包注入率较低（在未达到 0.5 时）的情况下，4 种组合的路由算法吞吐率在此网络中是平衡的。从 0.5 的注入率可以看出，随着注入率的增加，所有算法的吞吐率都受到了牵制，均未到达 0.5。从 0.5 到 0.6 的注入率可以看出，odd even+random、odd even+buffer level 组合的路由算法尽管受到牵制但还有上升的空间，而 xy+random、odd even+aco 组合的路由算法已经出现下降趋势。在注入率为 0.5 时，odd even+aco 组合的路由算法的吞吐率会优于 xy+random、odd even+random、odd even+buffer level 组合的路由算法，达到了 0.361 。而在注入率为 0.6 时，odd even+aco 组合的路由算法的吞吐率却成了最低的，为 0.286。

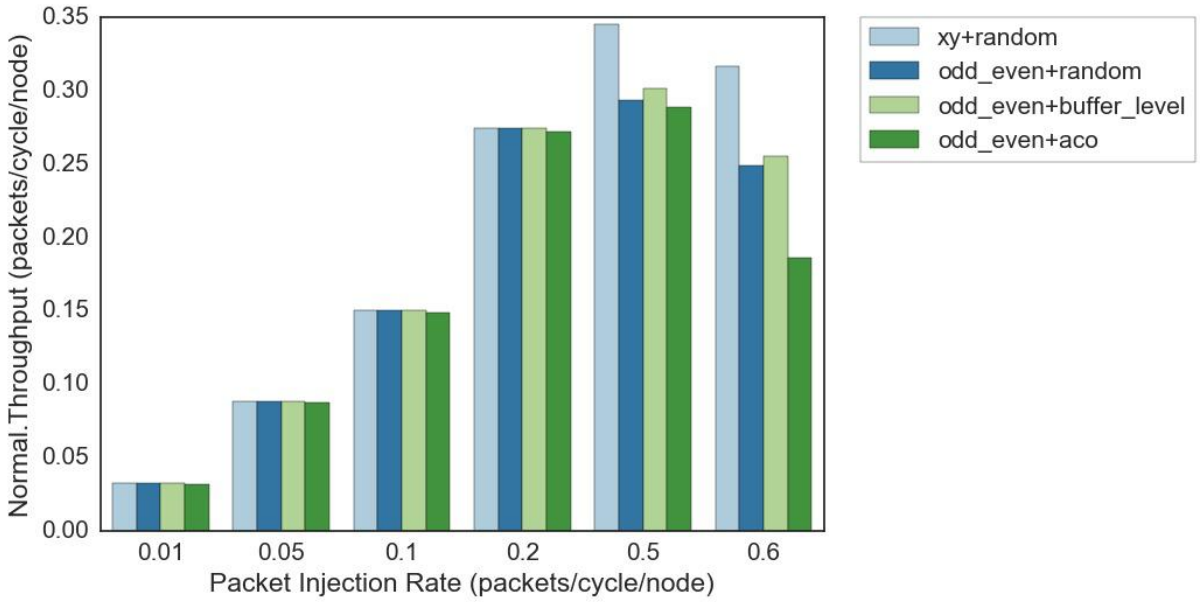



图 4.3 Hotspot 输入下的吞吐率

图 4.3 是在 hotspot 输入模式下的仿真测试结果。可以看出，在消息包注入率较低（在未达到 0.5 时）的情况下，4 种组合的路由算法吞吐率在此网络中是平衡的（会出现超过注入率的现象是因为 Hotspot 结点的注入率会比较大）。从 0.5 的注入率可以看出，随着注入率的增加，所有算法的吞吐率都受到了牵制，均未到达 0.5。从 0.5 到 0.6 的注入率可以看出，4 种组合的路由算法的吞吐率都无法再上升，且有明显下降。在注入率为 0.5 和 0.6 的情况下，xy+random 组合的路由算法的吞吐率都是最高的，而 odd even+aco 组合的路由算法的吞吐率都是最低的，在两种情况下分别相差 0.056 和 0.131。

综上所述，odd even+aco 组合的路由算法在提升饱和吞吐率方面没有明显的优势，但是没有太差的表现。

### 4．3．2　平均包延迟

下面三张图给出了 xy+random、odd even+random、odd even+buffer level、odd even+aco 四种组合的路由算法在 Uniform、Transpose、Hotspot 三种输入下的平均包时延（Average Packet Delay）。

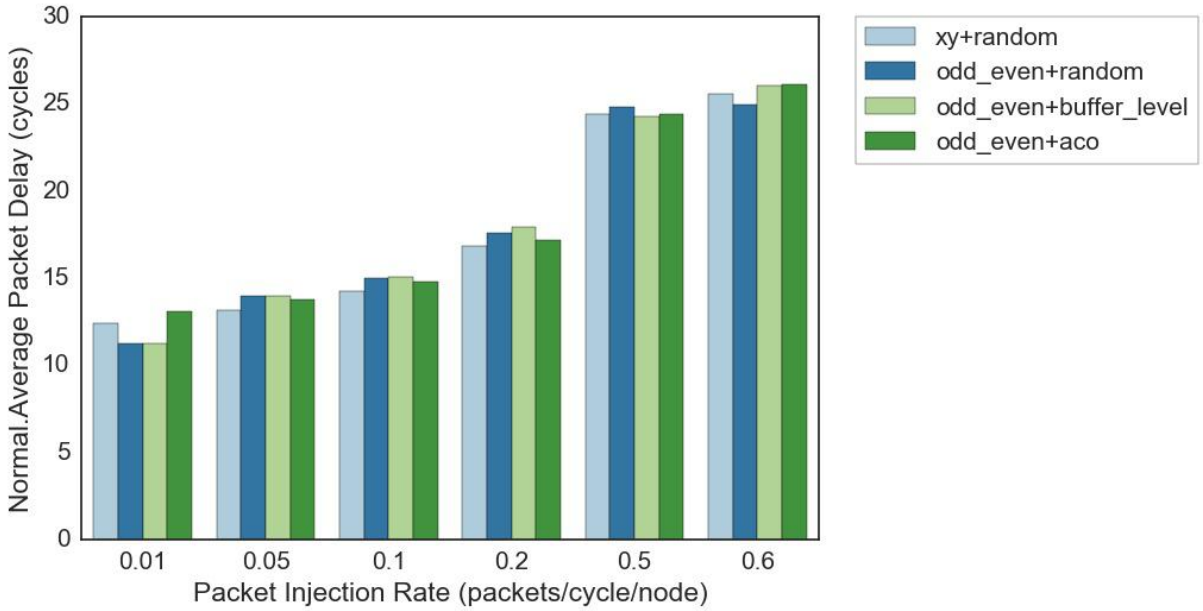



图 4.4 Uniform 输入下的平均包时延

图 4.4 是在 Uniform 输入模式下的仿真测试结果。从图中可以看出，4 种组合的路由算法的平均包时延都在平稳上升，且 4 种组合的路由算法在各个注入率下的平均包时延都较相近。

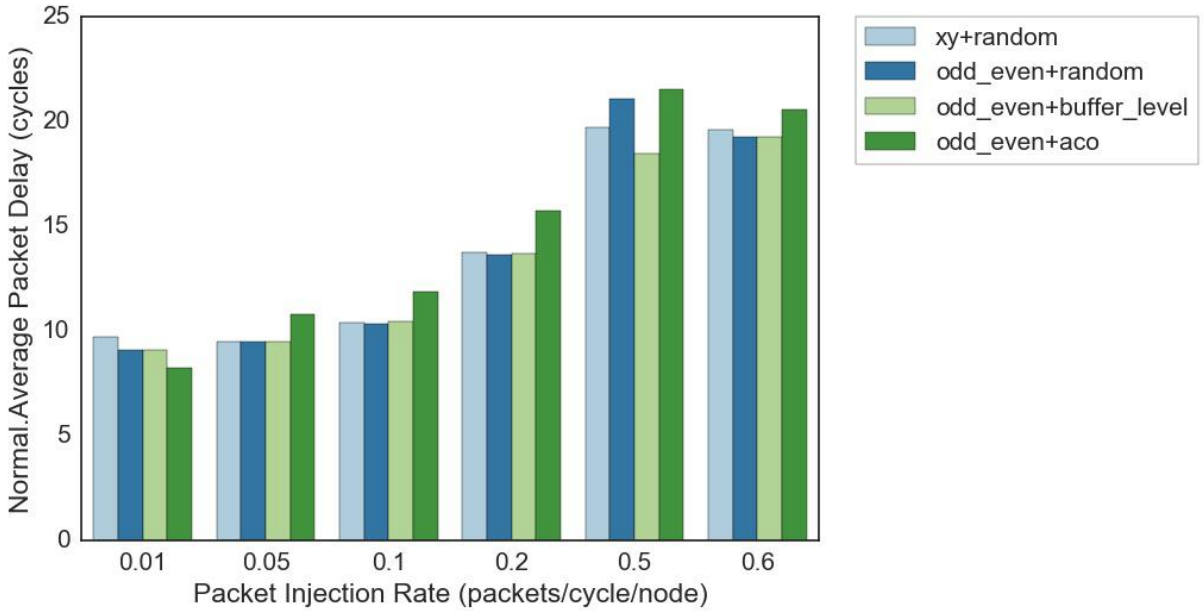



图 4.5 Transpose 输入下的平均包时延

图 4.5 是在 Transpose 输入下的仿真测试结果。从图中可以看出，在注入率为 0.01 时，odd even+aco 组合的路由算法的平均包时延会比其它三种低一点，而在其它注入率下，odd even+aco 组合的路由算法的平均包时延都比其它 4 种组合的路由算法高一些。

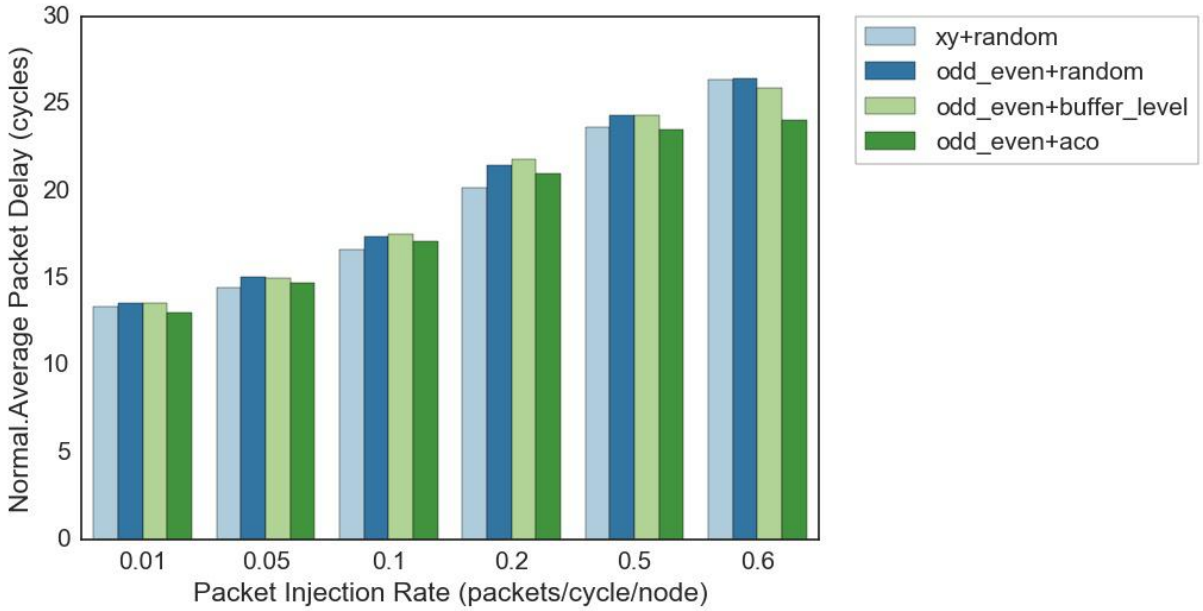



图 4.6 Hotspot 输入下的平均包时延

图 4.6 是在 Hotspot 输入下的仿真测试结果。从图中可以看出，4 种组合的路由算法的平均包时延都在平稳上升，在注入率较低时，odd even+aco 组合的路由算法的平均包时延与其它的路由算法相比，都不会太高。特别是在 0.6 的注入率时，它的平均包时延出现了稍微多些的优势。

综上所述，在 Uniform 输入下，odd even+aco 组合的路由算法表现出的平均包时延性能和其它组合的路由算法相接近。在 Transpose 输入下，其表现出的性能会比其它几种路由算法差些。而在 Totspot 输入下，在消息包注入率较高时，odd even+aco 组合的路由算法在降低平均包时延方面出现了些优势。

# 结　论

本文提出了一种适用于 NoC 的基于蚁群优化思想的自适应路由选择算法，将其与奇偶路由算法相结合并实现。通过在三种输入模式下的实验，将上述算法与另外三种主要路由算法对比，分析该算法在吞吐率和平均包时延上的性能表现。

FPGA 仿真实验结果表明，相较于 Buffer Level 选择算法，文中实现的 ACO 选择算法在 Uniform、Transpose、Hotspot 三种输入模式下能达到的吞吐率最大提升幅度为+23.81%（PIR=0.5）、+16.69%（PIR=0.5）和-0.96%（PIR=0.01），能达到的平均包时延最大降低幅度为+3.90%（PIR=0.2）、+9.73%（PIR=0.01）和+7.18%（PIR=0.6）。

从实验结果来看，本文中实现的选择算法表现出的性能不会太高也不会太差。其可能的原因有：

（1）测试硬件方面的问题，由于本文采用基于 Intel Pentium CPU B950c 处理器的笔记本电脑开发，因此路由选择算法的仿真测试实验中采用的是基于 4x4 2D-Mesh 片上网络拓扑架构，该结点数偏少，可能导致了几种路由算法差距不大的性能表现；

（2）由于时间关系，设计中实现的 NoC 架构不是很完善，考虑也不是非常周全，比如没有详细考虑实际系统中会出现的链路传输时延等，所以在系统设计上也可能影响了路由算法的测试结果；

（3）基于蚁群优化思想的路由选择算法还有许多需要权衡和改进的地方。本文中实现的算法不是根据信息素表浓度值进行概率选择，而是直接选择最大值对应的下一跳结点，以及在进行选择时也未多加入网络当前状态的考虑。另外，蚂蚁包与数据包的结合、蚂蚁包的注入率问题以及每一个参数的设置都会影响到实验结果；

（4）该系统的实现测试不是一个完全随机的环境，比如每一次仿真时，随机产生的仿真数据都是一样的，这可以保证几个算法处理的输入是相同的，以提升公平性，但也丧失了仿真测试的普遍性。

因此，在以后的工作中，可以从以下方面对本文进行改进：构建更完善的 NoC 平台和 Testbench 平台，研究和实现更好的 ACO 路由选择算法以及优化电路的代码设计等。

# 致　谢

首先要感谢我的指导教师蔡旻老师。毕业设计是对大学四年的一次大总结，也是对自己的一次挑战。在大四上学期期末，刚拿到自己毕业设计题目的任务书时，对任务书内容中的许多名词都含有困惑甚至未曾了解过。寒假前期的知识储备阶段，面对我的诸多疑问，我的蔡旻老师总是耐心地给我解答。

在后来的实践阶段，因为自己以前没有特别练习过硬件描述语言的编程，以及在阅读英文文献能力方面较薄弱，在我使用软件、设计和编写程序的过程中也遇到许多自己无法解决的问题。蔡旻老师在自己繁忙工作的同时，也十分认真负责地同我一起解决问题。在毕业设计阶段中，蔡旻老师指导了我的设计思路，让我少走了许多弯路，培养我养成良好的编程习惯，还教会了我很多设计工具的使用和设计编程的方法，给了我很大的鼓励和帮助。在此，我真诚地向蔡旻老师表示感谢！

还要感谢我的舍友们。感谢在毕业设计阶段中，我的舍友们，在晚上熄灯后我还在敲键盘时对我的理解，在我遇到难题时对我的鼓励。虽然我们的课题不同，但我们的大目标相同，我们相互鼓励，相互提醒，相互督促，共同进步。

最后还要感谢我的家人们。在遇到烦恼无人倾诉时，家人们永远都在我身边。家人们对我无私关怀，让我不管遇到什么难题，都不会被压垮，都不会放弃。

感谢毕业设计阶段，每一个在我身边的你。

# 参考文献

[1] 蒋珊珊. 片上网络感知故障容错路由算法研究.电子科技大学,2015.
[2] 董少周. NoC 路由算法及仿真模型的设计与研究. 合肥工业大学, 2009.
[3] 岳耀强. 片上网路自适应路由算法的设计与性能分析.曲阜师范大学,2015.
[4] A. Hemani, A Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg and D. Lindqvist. Network on Chip: An architecture for billion transistor era. In Proceedings of the IEEE NorChip Conference.
[5] S. Kumar, et al. A Network on Chip Architecture and Design Methodology. In Proceedings of ISVLSI 2002, pp.117-124.
[6] Navaneeth Rameshan, V.Laxmi, M.Sgaur. Minmal path, Fault Tolerant. QoS aware Routing with Node and Link Failures in 2D Mesh No C. In Proceedings of the 2010 25th IEEE International Symposium on Defect and Fault Tolerance in VLSI System, 2010.
[7] 李丽,许居衍. 片上网络技术发展现状及趋势浅析. 电子产品世界. 2009,01:32-37.
[8] 张大坤,黄翠,宋国治. 三维片上网络研究综述. 软件学报. 2016,01:155-187.
[9] Singh, J.K. , Swain, A.K. , Reddy, T.N.K. , Mahapatra, K. Performance evalulation of different routing algorithms in Network on Chip. In Proceedings of the IEEE Asia Pacific Conference on Postgraduate Research on Microelectronics and Electronics (Prime Asia). 2013.
[10] 王峥,顾华玺,杨烨,乐天助. 片上网络交换机制的研究. 中国集成电路. 2007,12:22-27.
[11] 刘江,张金艺,周多,周文强. 一种结合路由器旁路通道的低延迟 NoC 容错机制. 微电子学与计算机. 2015,04:10-14.
[12] 黄翠,张大坤,宋国治. 三维片上网络映射算法研究综述. 小型微型计算机系统. 2016,02:193-201.
[13] Zhiliang Q ian, Paul Bogdan. A Traffic-aware Adaptive Routing Algorithm on Highly Reconfigurable Network on Chip Architecture. Computer Systems Organization. 2012: 161-170.
[14] Jiong Luo, N K. Jha. Power-profile driven variable voltage scaling for heterogeneous distributed real-time embedded systems. In Proceedings of the IEEE 16th International Conference on VLSI Design. 2003:369-375.
[15] Li M, Zeng Q A, Jone W B. Dy XY: A proximity congestion-aware deadlock-free dynamic routing method for network on chip. In Proceedings of the 43rd Annual Design Automation Conference. 2006: 849-852.
[16] Hu S, Lin X. A Symmetric Odd-Even Routing Model in Network-on-chip[C]. In Proceedings of the 11th International Conference on Computer and Information Science. Shanghai, China, 2012: 277-280.
[17] Hu J, Marculescu R. Communication and task scheduling of application-specific network on chip[C]. In Proceediongs of Computers and Digital Techniques. 2005: 643-651.
[18] Enright, N.; Peh, L. On-Chip Networks. In Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2009.

[19] L.G.Valiant and G.J.Brebner. Universal schemes for parallel communication. In Proceedings of the 13th Annual ACM Symposium on Theory of Computing. pp. 263 – 277, 1981.
[20] G.-M. Chiu. The Odd-Even Turn Model for Adaptive Routing. IEEE Trans. Parallel Distrib. Syst. vol. 11, no. 7, pp. 729-738, July 2000.
[21] H.-K. Hsin, E.-J. Chang, and A.-Y. Wu. Spatial-temporal enhancement of ACO-based selection schemes for adaptive routing in network-on-chip systems. IEEE Trans. Parallel Distrib. Syst. vol. 25, no. 6, pp. 1626 – 1637, Jun. 2014.
[22] G. Ascia, V. Catania, M. Palesi. Implementation and Analysis of a New Selection Strategy for Adaptive Routing in Networks-on-Chip. IEEE Transaction on Computers. v.57, I.6, pp. 809-820, 2008.
[23] 胡小兵. 蚁群优化原理、理论及其应用研究. 重庆大学,2004.
[24] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the Argentine ant. Natur wissen schaften. 76:579 – 581, 1989.
[25] 衡书会. 基于 FPGA 的片上网络验证测试平台设计与实现.西安电子科技大学,2013.
[26] Anandamoy Sen 2006. Swarm Intelligence based optimization Of MANET cluster formation.
[27] Schoonderwoerd, Ruud; Holland, Owen; Bruten, Janet; Rothkrantz, Leon 1996. Ant-based load balancing in telecommunications networks. Hewlelt-Packard Laboratories, Bristol-England, pp 162-207.
[28] J. Baras and H. Mehta. A Probabilistic Emergent Routing Algorithm for Mobile Ad hoc Networks (PERA). 2003.
[29] Gianni Di Caro, Frederick Ducatelle and Luca Maria Gambardella. AntHocNet: an adaptive nature-inspired algorithm for routing in mobile ad hoc networks. 2005.
[30] Di Caro G., Dorigo M. AntNet: Distributed stigmergetic control for communications networks. Journal of Artificial Intelligence Research. 9, pp. 317-365, 1998.
[31] M. Daneshtalab and A. Sobhani. NoC Hot Spot Minimization Using AntNet Dynamic Routing Algorithm. In Proc. IEEE Appl. Specific Syst., Architect. Processors Conf. 2006, pp. 33-38.
[32] G.-M. Chiu. The Odd-Even Turn Model for Adaptive Routing. IEEE Trans. Parallel Distrib. Syst. vol. 11, no. 7, pp. 729-738, July 2000.
[33] 苏阳,蒋银坪,邢培飞. 那些年,我们拿下了 FPGA. 北京：北京航空航天大学出版社, 2013.

# 附录

附表 1 Uniform 输入模式下不同路由算法的吞吐率和平均包时延

| 包注入率<br>(包/时钟周期/结点) | 路由算法+选择算法 | 吞吐率<br>(包/时钟周期/结点) | 平均包时延<br>(时钟周期) |
|---|---|---|---|
| 0.01 | xy+random | 0.00965 | 12.4053 |
| | odd_even+random | 0.00965 | 11.2985 |
| | odd_even+buffer_level | 0.00965 | 11.2985 |
| | odd_even+aco | 0.0094875 | 13.0785 |
| 0.05 | xy+random | 0.0502 | 13.1636 |
| | odd_even+random | 0.0502 | 14.0191 |
| | odd_even+buffer_level | 0.0502 | 14.0191 |
| | odd_even+aco | 0.0496062 | 13.8105 |
| 0.1 | xy+random | 0.100062 | 14.2942 |
| | odd_even+random | 0.100069 | 15.0278 |
| | odd_even+buffer_level | 0.100062 | 15.072 |
| | odd_even+aco | 0.0990188 | 14.7892 |
| 0.2 | xy+random | 0.199469 | 16.8563 |
| | odd_even+random | 0.19945 | 17.6398 |
| | odd_even+buffer_level | 0.19945 | 17.9195 |
| | odd_even+aco | 0.197431 | 17.2206 |
| 0.5 | xy+random | 0.374256 | 24.4487 |
| | odd_even+random | 0.321188 | 24.8305 |
| | odd_even+buffer_level | 0.277919 | 24.2938 |
| | odd_even+aco | 0.344094 | 24.4299 |
| 0.6 | xy+random | 0.298969 | 25.5796 |
| | odd_even+random | 0.235538 | 24.985 |
| | odd_even+buffer_level | 0.262225 | 26.053 |
| | odd_even+aco | 0.260562 | 26.138 |

附表 2 Transpose 输入模式下不同路由算法的吞吐率和平均包时延表

| 包注入率<br>(包/时钟周期/结点) | 路由算法+选择算法 | 吞吐率<br>(包/时钟周期/结点) | 平均包时延<br>(时钟周期) |
|---|---|---|---|
| 0.01 | xy+random | 0.00965 | 9.71901 |
| | odd_even+random | 0.00965 | 9.13027 |
| | odd_even+buffer_level | 0.00965 | 9.13027 |
| | odd_even+aco | 0.009475 | 8.24206 |
| 0.05 | xy+random | 0.0502 | 9.48238 |
| | odd_even+random | 0.0502 | 9.52342 |
| | odd_even+buffer_level | 0.0502 | 9.4757 |
| | odd_even+aco | 0.0495937 | 10.8245 |
| 0.1 | xy+random | 0.100062 | 10.4082 |
| | odd_even+random | 0.100062 | 10.3389 |
| | odd_even+buffer_level | 0.100056 | 10.4598 |
| | odd_even+aco | 0.099 | 11.8799 |
| 0.2 | xy+random | 0.199431 | 13.7599 |
| | odd_even+random | 0.199419 | 13.6387 |
| | odd_even+buffer_level | 0.199431 | 13.6862 |
| | odd_even+aco | 0.195794 | 15.7702 |
| 0.5 | xy+random | 0.337056 | 19.7518 |
| | odd_even+random | 0.274612 | 21.0964 |
| | odd_even+buffer_level | 0.309513 | 18.5069 |
| | odd_even+aco | 0.361162 | 21.5565 |
| 0.6 | xy+random | 0.325362 | 19.5897 |
| | odd_even+random | 0.361694 | 19.2473 |
| | odd_even+buffer_level | 0.361694 | 19.2473 |
| | odd_even+aco | 0.285562 | 20.56 |

附表 3 Hotspot 输入模式下不同路由算法的吞吐率和平均包时延表

| 包注入率<br>(包/时钟周期/结点) | 路由算法+选择算法 | 吞吐率<br>(包/时钟周期/结点) | 平均包时延<br>(时钟周期) |
|---|---|---|---|
| 0.01 | xy+random | 0.0327125 | 13.3734 |
| | odd_even+random | 0.0327125 | 13.5607 |
| | odd_even+buffer_level | 0.0327125 | 13.5605 |
| | odd_even+aco | 0.0323312 | 13.0176 |
| 0.05 | xy+random | 0.0883063 | 14.4821 |
| | odd_even+random | 0.0883063 | 15.0661 |
| | odd_even+buffer_level | 0.0883063 | 15.0446 |
| | odd_even+aco | 0.0873875 | 14.7438 |
| 0.1 | xy+random | 0.150306 | 16.6223 |
| | odd_even+random | 0.150294 | 17.3981 |
| | odd_even+buffer_level | 0.150294 | 17.5238 |
| | odd_even+aco | 0.148844 | 17.1458 |
| 0.2 | xy+random | 0.274738 | 20.1789 |
| | odd_even+random | 0.274625 | 21.5066 |
| | odd_even+buffer_level | 0.274713 | 21.8148 |
| | odd_even+aco | 0.271925 | 21.0343 |
| 0.5 | xy+random | 0.345175 | 23.6816 |
| | odd_even+random | 0.293794 | 24.3535 |
| | odd_even+buffer_level | 0.301731 | 24.3802 |
| | odd_even+aco | 0.289069 | 23.5356 |
| 0.6 | xy+random | 0.316744 | 26.3923 |
| | odd_even+random | 0.249381 | 26.4823 |
| | odd_even+buffer_level | 0.255794 | 25.9167 |
| | odd_even+aco | 0.186138 | 24.0546 |