FORTH-ICS/TR-391 July 2007

Approaching Ideal NoC Latency with Pre-Configured Routes

George Michelogiannakis

July 2007

Abstract

In multi-core ASICs, processors and other compute engines need to communicate with memory blocks and other cores with latency as close as possible to the ideal of a direct buffered wire. However, current state of the art networks-on-chip (NoCs) suffer, at best, latency of one clock cycle per hop.

We investigate the design of a NoC that offers close to the ideal latency in some preferred, run-time configurable paths. Processors and other compute engines may perform network reconfiguration to guarantee low latency over different sets of paths as needed. Flits in non-preferred paths are given lower priority than flits in preferred paths to enable the latter to provide low latency.

To achieve our goal, we extend the "mad-postman" technique [1]: every incoming flit is eagerly (*i.e.* speculatively) forwarded to the input's preferred output, if any. This is accomplished with the mere delay of a single pre-enabled tri-state driver. We later check if that decision was correct, and if not, we forward the flit to the proper output. Incorrectly forwarded flits are classified as dead, and are eliminated in later hops.

We use a 2D mesh topology tailored for processor-memory communication, and a modified version of XY routing that remains deadlock-free. We also propose an extension which enables a switching node to switch to adaptive routing when its benefits are required.

Our evaluation shows that, for the preferred paths, our approach offers typical latency around 500 ps versus 1500 ps for a full clock cycle at 667 MHz or up to 135 ps for an 1 mm ideal direct connect, in a 130 nm technology; non-preferred paths suffer a one clock cycle delay per hop when there is no contention, similar to that of other approaches. Performance gains are significant and can prove quite useful in other application domains as well.

Approaching Ideal NoC Latency with Pre-Configured Routes

George Michelogiannakis

Computer Architecture & VLSI Systems (CARV) Laboratory – member of HiPEAC
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Science and Technology Park of Crete
P.O. Box 1385, Heraklion, Crete, GR-711-10 Greece
Tel.: +30-2810-391660 Fax: +30-2810-391661
email: mihelog@ics.forth.gr

Technical Report FORTH-ICS/TR-391 – July 2007

Work performed as a M.Sc. Thesis at the Department of Computer Science, University of Crete, under the supervision of Prof. Manolis Katevenis

Abstract

In multi-core ASICs, processors and other compute engines need to communicate with memory blocks and other cores with latency as close as possible to the ideal of a direct buffered wire. However, current state of the art networks-on-chip (NoCs) suffer, at best, latency of one clock cycle per hop.

We investigate the design of a NoC that offers close to the ideal latency in some preferred, run-time configurable paths. Processors and other compute engines may perform network reconfiguration to guarantee low latency over different sets of paths as needed. Flits in non-preferred paths are given lower priority than flits in preferred paths to enable the latter to provide low latency.

To achieve our goal, we extend the "mad-postman" technique [1]: every incoming flit is eagerly (*i.e.* speculatively) forwarded to the input's preferred output, if any. This is accomplished with the mere delay of a single pre-enabled tri-state driver. We later check if that decision was correct, and if not, we forward the flit to the proper output. Incorrectly forwarded flits are classified as dead, and are eliminated in later hops.

We use a 2D mesh topology tailored for processor-memory communication, and a modified version of XY routing that remains deadlock-free. We also propose an extension which enables a switching node to switch to adaptive routing when its benefits are required.

Our evaluation shows that, for the preferred paths, our approach offers typical latency around 500 ps versus 1500 ps for a full clock cycle at 667 MHz or up to 135 ps for an 1 mm ideal direct connect, in a 130 nm technology; non-preferred paths suffer a one clock cycle delay per hop when there is no contention, similar to that of other approaches. Performance gains are significant and can prove quite useful in other application domains as well.

Acknowledgments

This work was conducted as part of a FORTH-ICS graduate student fellowship, supported by the European Commission in the context of the SARC (Scalable Computer Architecture) integrated project #27648 (FP6), and the HiPEAC network of excellence.

I am grateful for my family's continuing support, which has allowed me to achieve this work.

I would like to recognize the contribution of my supervisor, Prof. Manolis Katevenis, and Prof. Dionisios Pnevmatikatos for the completion of this work. I thank each of them for the guidance, support, constructive remarks, devoted time, as well as the opportunites and challenges they presented me.

I wish to thank my colleagues and the people who helped me, both locally and throughout HiPEAC and SARC, for their assistance and support in all the good and bad times: Christos Sotiriou, Spyros Lyberis, Pavlos Mattheakis, Stamatis Kavvadias, Nikolaos Andrikos, Vasilis Papaefstathiou, Michalis Papamichail, Kees Goossens, Giuseppe Desoli, Krisztian Flautner, Chris Jesshope, Jose Duato, and Georgi Gaydadjiev.

It’s hard to work fast without sweating.

Contents

1 Introduction **1**
1.1 Related Work 4

2 Preferred Paths & Routing **6**
2.1 Mad-Postman 6
2.2 Preferred Paths 7
2.3 Routing 8
2.3.1 Packet Format 8
2.3.2 Routing Logic 9
2.3.3 Duplicate Flits 9
2.3.4 Adaptive Routing Extension 10
2.4 Deadlock-Freedom 12
2.5 Preferred Path Reconfiguration 13
2.6 Backpressure 15

3 Switch Architecture & Topology **16**
3.1 Switch Architecture 16
3.1.1 Routing Logic Implementation 18
3.1.2 Virtual Channels 21
3.2 Network Topology 21
3.2.1 Network Interfaces 26

4 Layout Results **29**
4.1 Preliminary Results 29
4.2 RAM Blocks 30
4.3 Switch P&R Results 32

5 Conclusions **35**
5.1 Future Work 35
5.2 Conclusion 36

References **37**

List of Figures

1.1 Substituting inverter cells to create a network. 1
1.2 Preliminary simulation circuits. 2
1.3 Example of a reconfiguration to an uneven distribution of RAM blocks to processors' low-latency regions, to cover run-time demands. 3

2.1 Original mad-postman network routing. 7
2.2 Flit format. 39 Bits total - 7 control, 32 payload 8
2.3 Correct eager forwarding scenario that does not comply with strict XY routing. . 9
2.4 Duplicate flit scenario. 10
2.5 Circle formed by four preferred paths with exatly one turn. 12
2.6 Out-of-order delivery scenario. 14

3.1 Switch architecture. 17
3.2 A macroscopic illustration of a 4x4 switch input queueing approach, without preferred paths and VCs. 17
3.3 5-1 multiplexer with 3-1 cells. 18
3.4 Preferred path bus. 19
3.5 The simple 2D mesh topology. 22
3.6 2D mesh topology with two subnetworks. 23
3.7 2D mesh topology with two subnetworks - one for each axis. 24
3.8 RAM blocks rotated to place switches every two X axes. 24
3.9 Rectangular-shaped floorplan. 25
3.10 2x1 switching logic. 26
3.11 Cross-shaped floorplan. 27

4.1 RAM block pin placement. 31

List of Tables

4.1 Multiplexer simulation results. 30
4.2 Single-port RAM attributes. 31
4.3 Two-port RAM attributes. 32
4.4 Dual-port RAM attributes. 32
4.5 Switch p&r results (typical). 33
4.6 Switch area results for the bar and cross floorplans. ”a”, ”b” refer to Figures 3.9 and 3.11. 33

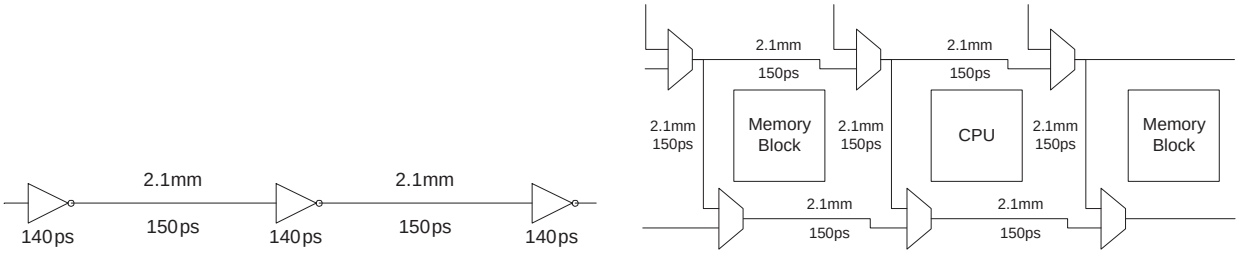
Chapter 1

Introduction

System-on-Chip (SoC) technology is the ability to place multiple function ”systems” -Intellectual property (IP) blocks- in a single silicon chip [2]. As SoCs grow in area, complexity and functionality, so do their communication requirements in terms of performance (latency and throughput) and number of interconnected components. These communication demands as well as the heterogeneity of IP block network interface logic necessitate the development of a structured on-chip communication infrastructure. Networks-on-Chip (NoCs) [3] have by now proven themselves to be key components of the emerging SoCs. Specialized NoCs are used in other applications as well, such as large processor chips or chip multiprocessors (CMPs) [4, 5].

NoCs significantly affect performance, latency and area of the chip. Reducing NoC latency is crucial for SoC performance since it is introduced to every communication pair within the SoC. Latency may become vital in the case of SoCs with critical timing demands (real-time SoCs). It may also play an especially important role in the case of processor units communicating with other processor units, local memory, shared memory or cache blocks.

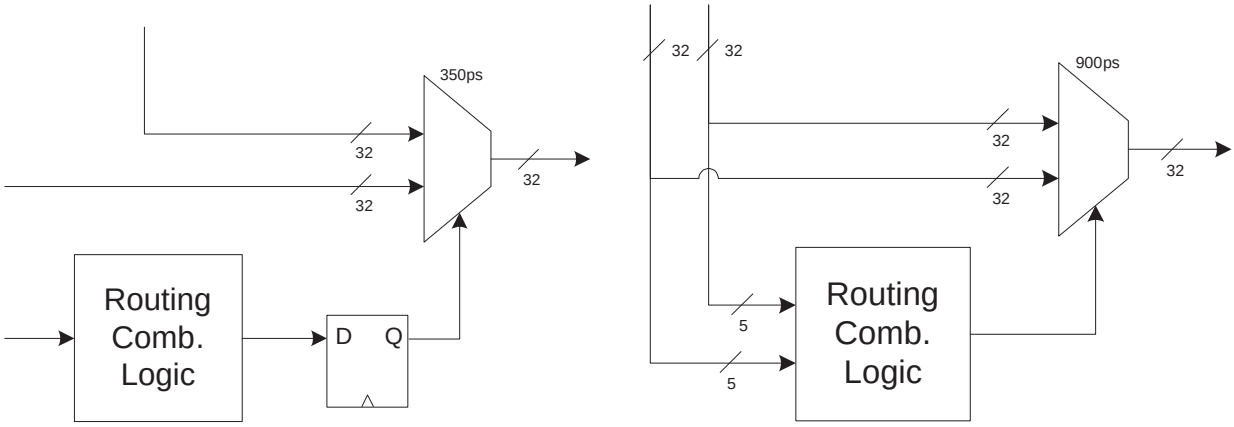
In this master's thesis we propose a NoC with latency close to the ideal, *i.e.* that of long buffered wires which are the simplest and fastest interconnection method. Our approach is fundamentally based on the simple observation that buffer or inverter cells in those long wires can be replaced with tri-state drivers or multiplexer cells to provide a basic network infrastructure, as shown in Figure 1.1. We would like to design a network that replaces long wire inverter cells and achieves latency close to that of those inverter cells in some optimistic scenarios (*i.e.* when speculation succeeds). Indeed, we are able to achieve latency of approximately 400 ps per hop



(a) Long wire with inverter cells.

(b) Primitive multiplexer cell network infrastructure.

Figure 1.1: Substituting inverter cells to create a network.



(a) Pre-configured multiplexer cell. (b) Multiplexer cell with no control input register.

Figure 1.2: Preliminary simulation circuits.

in those good scenarios and a single clock cycle -similar to that of other approaches as explained in section 1.1- otherwise.

To be able to achieve the above, we extend speculation to routing decisions. We predict each flit's output and forward it via only a single pre-enabled tri-state driver. Pre-enabling is an important factor of our achieved performance, since switch outputs have a substantial fanout of 11 in our implementation, and also because examining a flit by routing logic is time consuming. Preliminary simulation results to research the validity of this approach were conducted in our 130 nm implementation library under worst case conditions. They showed that a pre-configured multiplexer cell as shown in Figure 1.2(a), imposed a 350 ps delay. Inverter cells were placed approximately every 2.1 mm, wires between them imposed up to 150 ps latency and the cells themselves imposed an 140 ps latency. Finally, a multiplexer cell which computes it's control input according to it's current inputs with a very simple combinational logic, as shown in Figure 1.2(b), imposes a delay of 900 ps. We therefore confirm that pre-enabling control signals is crucial to approach the latency of an inverter cell.

To apply this concept in a fully-featured NoC, we resurrect the "mad-postman" [1, 6] technique proposed two decades ago for inter-chip communication networks. Mad-postman networks were the first to introduce the concept of eager (*i.e.* speculative) flit forwarding. We extend this technique to define run-time reconfigurable preferred paths in our network. They are formed by pre-driving tri-state select signals within a switch to form a connection between input/output pairs. Therefore, flits will be eagerly forwarded to their input's preferred outputs. Preferred path delay per hop is solely that of a pre-enabled tri-state driver. Pre-enabling is even more beneficial in a fully-featured NoC because these control signals fan out to many bits, thus driving them incurs considerable delay. We examine our proposal in a chip consisting of many processor units and RAM blocks. However, our ideas are general and can be easily adapted to other NoC styles.

Packets in our NoC may consist of a single or multiple flits [7]. Flits that are eagerly forwarded to a wrong switch output are terminated later in the network as "dead". They are forwarded by the switch they were misrouted at to their correct switch output through a non-preferred path at a lower priority than flits which originate from the input having that output as

preferred (if any), and suffer a latency of one clock cycle when there is no contention.

In order to provide preferred paths with flexibility and to be able to distinguish incorrect eager forwarding, we utilize a modified version of XY routing which remains deadlock-free. According to it, a flit is considered to have been correctly eagerly forwarded if it moves closer to its destination in any of the two axes. A flit is considered dead if the distance between it and the destination increased in any of the two axes with its last hop. This way, we can easily distinguish an incorrect eager flit forwarding as well as a dead flit in the network. We also propose a routing logic design which can dynamically switch to adaptive routing. Thus, our NoC is able to be assisted from adaptive routing's benefits when they are required.

Network reconfiguration is possible at any time by any processing element (PE) or other user block in the network. It serves the purpose of enabling PEs to dynamically setup low-latency paths to cover run-time demands. As the example illustrated in Figure 1.3 shows, processors may dynamically allocate RAM blocks to their low-latency regions to cover program needs. Reconfiguration is accomplished by sending specially formatted single-flit packets to the switching nodes that need to be reconfigured. Reconfiguration can be requested at any time, but is carefully applied to the switching node to prevent out-of-order delivery of flits belonging to the same packet. Dealing with out-of-order flit delivery complicates NoC - PE interface logic and is rarely allowed in NoCs.

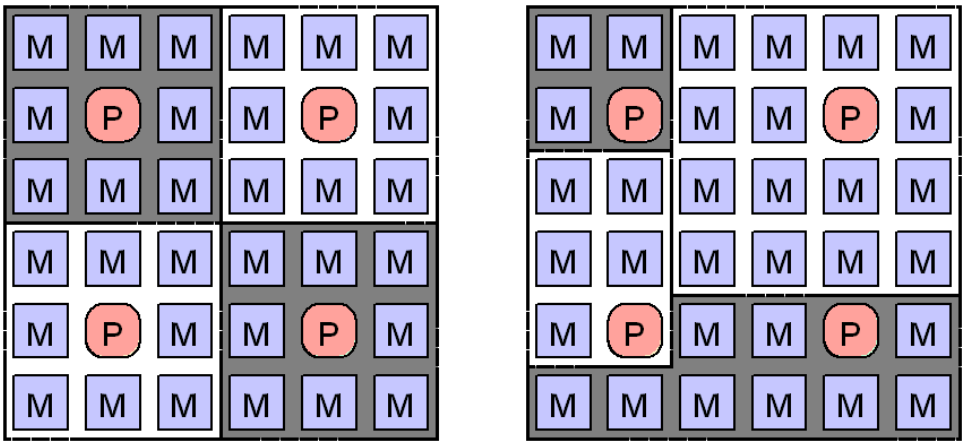


Figure 1.3: Example of a reconfiguration to an uneven distribution of RAM blocks to processors' low-latency regions, to cover run-time demands.

To fully exploit the mad-postman technique and ensure its proper operation, we take a slightly different approach for switching node architecture than most past research. Our switch resembles a buffered crossbar [8], having one FIFO at each crosspoint and schedulers at each output. The scheduler monitors the FIFOs and the preferred path, and determines which FIFO it can serve next, if any. At each input a combinational routing logic determines if the incoming flit needs to be forwarded to a non-preferred output. If so, it enqueues the flit in the appropriate crosspoint FIFO.

We evaluate our proposed approach on a 2D mesh topology [9] tailored for our target application, *i.e.* processors communicating with RAM blocks. We attempt to minimize the number of switching nodes by placing one switch per 4 RAM blocks. This provides significant savings in area, power, and latency by reducing the number of hops between two endpoints. The RAM

blocks are placed without any free space between them, essentially forming a bigger block. We also investigate floorplan options for our switching nodes by evaluating two different shapes (rectangular and cross-shaped), and outline some modifications to our switch to further reduce occupied area. Topology and floorplan choices, however, do not affect our low-latency contribution and are made according to application and optimization needs.

Simulation results show that, in a 130 nm technology, our design functions at 667 MHz under typical case conditions. It offers preferred path latency of approximately 360 ps per hop that increases to approximately 500 ps per hop when taking into account an 1 mm long wire at each output. This is compared to up to 135 ps latency for straight wires of a similar length that offer no configuration or routing capability. Non preferred path latency is one clock cycle when there is no contention. Our base switching node design, for 39-bit wide datapaths, occupies an area of 637 μm $\times$ 310 μm in a rectangular floorplan. We believe that our proposed NoC concept is the means to approach the ideal latency as closely as possible. It may also be combined with orthogonal past NoC research to further improve performance as well as other aspects.

The rest of this master's thesis is organized as follows: Section 1.1 provides a summary of past NoC research. Chapter 2 explains the mechanism for pre-configured low latency paths and our network's routing logic. Chapter 3 presents our proposed switch architecture and describes our NoC's topology. Chapter 4 presents our placement and routing results. Finally, chapter 5 provides our conclusions and identifies room for future work.

1.1 Related Work

Most of past research assumes switch-based, packet-switched architectures. Regular topologies such as 2D meshes [9] are utilized and are becoming standard practice in general-purpose NoCs. They provide the basis for NoC development for the vast majority of applications and environments. Run-time reconfigurable interconnects have been developed to meet specific application demands or provide an optimized network for specific application environments. FLUX network [10] and CoNoChi [11] assume an FPGA infrastructure, while DYNOC [12] is designed for ASIC flow. Additional topologies [13, 14, 15] have been proposed to reduce hop count and therefore network latency. They implement extra links between nodes for this purpose, resulting in various network topologies such as torus or even 3D mesh. Research in this area can be applied to our proposed NoC according to the desired aspect for optimization or application domain.

Research has also examined performance-enhancement techniques [16, 17, 18]. These approaches are based on pre-computing routing, virtual channel (VC) allocation, and arbitration decisions, as well as speculative pipelines to minimize deterministic routing latency. Implementations of these approaches with VCs and various datapath widths are able to function with a clock frequency of around 500 MHz in technologies ranging from 0.07 um to 0.13 um. While these approaches can yield per hop latency of one clock cycle, this latency is not guaranteed. These designs suffer high penalties from contention and blocking delays, that significantly increase latency. Moreover, one clock cycle per hop is their minimum possible latency, while our proposed NoC provides constant minimum per-hop latency, independent of the clock period. Latest research in this area achieves the same network performance with half the required buffer size or a 25% performance increase and lesser performance degradation as traffic increases with the same buffer size, by dynamically allocating VC buffers to input/output ports [19]. Minimum latency remains, however, 1 clock cycle per node. Asynchronous approaches achieve 2 ns per

hop [20] for highest-priority flits.

Power-efficient and thermal-aware systems [21, 22], fault-tolerant mechanisms [23, 24], and area-constrained designs [25, 26] have also been examined. They can be applied to our NoC depending on each application's needs. Some of these concepts may introduce significant changes to our NoC design or switching node architecture. However, such changes will not affect our low-latency contribution.

Another important issue in NoCs is routing for which many algorithms [27] have been proposed. Many recent NoCs utilize adaptive routing algorithms [28, 29, 30, 31] to route around congested or other problematic areas according to some criteria. As explained in section 2.3, our NoC utilizes deterministic routing algorithm under normal operation conditions but can switch to adaptive routing when it's benefits are required. Flexibility in preferred paths is provided at all times.

Chapter 2

Preferred Paths & Routing

In this chapter we present our design for low-latency preferred paths and our network's routing logic. In section 2.1 we begin by explaining mad-postman [1, 6] and it's operation as originally designed. We move on to present preferred paths in our network in section 2.2. In section 2.3 we present our packet's format and our network's routing logic. We then move on to discuss our NoC's deadlock-freedom in section 2.4. After that we present our run-time reconfiguration mechanism in section 2.5. Finally, in section 2.6 we outline our backpressure mechanism.

2.1 Mad-Postman

Mad-postman [1, 6] was introduced in inter-chip packet-switched communication networks. It offered minimal per-hop latency by eagerly forwarding an incoming flit to the same direction in the same axis that it entered the switch from. For instance, if a flit was moving along the X axis from right to left, it would enter the switch from the rightmost input and be eagerly forwarded to the leftmost output. There was no logic or delay during this forwarding more than that of simple multiplexor or tri-state cell. Incoming flits were also stored in the switch for checking that they were correctly eagerly forwarded. The network strictly followed XY routing algorithm. According to it, a flit must complete its traversal in the X axis before switching to the Y. A flit was regarded as correctly eagerly forwarded if it followed XY routing. Incorrectly forwarded flits remained in storage in the switch and were later sent to the appropriate output. We find that this concept can be applied to NoCs.

The original mad-postman strictly followed XY routing. Therefore, a flit would suffer a routing logic and buffering penalty once at its final hop (in order to be ejected to the local PE output), and possibly once more when it changed axes when traversing the network. Consider the example of Figure 2.1. Source S transmits a packet to destination D. The packet follows XY routing, illustrated with solid lines. The packet is correctly eagerly forwarded in all hops of this path, apart from switches A and D. The packet must change axes in switch A, while in switch D the packet must be forwarded to the switch's local PE output. At those points the packet suffers a routing logic and buffering penalty since it must be stored in the switch, wait for it's proper output be determined and content for that output. However, since eager forwards are completed before the switch knows the packet's proper output, switches A and D will incorrectly eagerly forward the flit. This dead flit will continue to traverse the network through incorrect eager forwards (illustrated with dashed lines) until it reaches a switch (in our example switch B) which

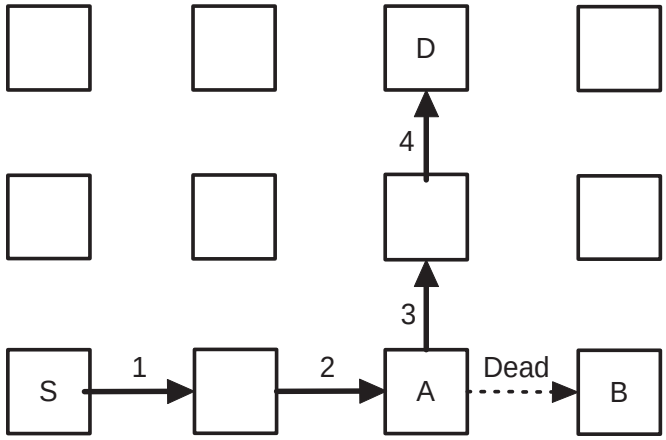


Figure 2.1: Original mad-postman network routing.

cannot eagerly forward the flit in the same manner since it is located at a network's edge.

We would like our NoC to be able to provide complete paths with the minimum per-hop latency. Moreover, we would like to provide the flexibility to change those paths at run-time to meet various application demands, such as a processor in a CMP allocating more RAM blocks to it's low-latency region, as illustrated in Figure 1.3. To meet these goals we introduce preferred paths.

2.2 Preferred Paths

In our switches each input is directly connected to a tri-state buffer at each other port's output. We do not connect an output to its own port's input, as we consider that flits will not desire to leave the switch from where they entered it. Each output has at most one preferred input. That input's tri-state driver is pre-enabled. Therefore, an incoming flit to that input would be eagerly forwarded to each output having this input as preferred. This is achieved solely with the delay of a pre-enabled tri-state driver. Thus, preferred paths are formed. Note that an input may have multiple preferred outputs. Therefore, preferred paths can split and simulate a broadcast network if so desired at run-time. However, preferred paths may not converge as only one tri-state may safely drive a wire at any time.

Each input also features a combinational routing logic which examines each incoming flit and determines whether it must be forwarded to an output other than the preferred. If so, it enqueues it in the appropriate crosspoint FIFO to be later forwarded by that output's arbitration logic. Switch architecture is discussed in section 3.1. If fair arbitration is desired without demands for very low latency at some part of the network, that part can be reconfigured to remove any preferred outputs from switch inputs. A flit needs to be forwarded to an output if the eager forwarding was incorrect. Later hops regard that flit as dead.

Dead flits are not forwarded to any output by routing logic. They propagate through the network in preferred paths until they reach an input with no preferred outputs. Then, they are either terminated or forwarded to an output by routing logic and possibly enter a circle, as discussed in section 2.4. Dead flits occupy preferred paths and therefore may be a nuisance. However, mad-postman networks indicate that this effect does not reduce the performance of the network

beyond that of virtual cut-through or wormhole networks [1]. Since flits in mad-postman networks are misrouted almost twice by average, we therefore expect the dead flit effect in our network to be similar if flits have the same mis-routing rate. Moreover, in our network we have the flexibility to configure preferred paths such as dead flits will occupy parts of the network that we know do not carry any, or at least any critical, traffic. This may make dead flits have a minimal effect. However, dead flits still consume power with every hop. As of the time of writing of this master's thesis, a cycle-accurate network simulator is still under development and therefore exact experimental confirmation of this issue is left as future work.

2.3 Routing

2.3.1 Packet Format

Packets may consist of a single or multiple flits, in the manner described in [7]. Single-flit packets are used for reconfiguration and read requests. Multi-flit packets are used for transferring multiple words of data to write to a RAM, or from a RAM as a reply to a read request. Flits feature 6 packet ID and 1 flit type control bits marking the initial flit of a packet as request (single-flit packet) or address, and thereafter data flits with the same packet ID as body or tail. The 32 payload bits contain data in the case of data flits and destination address, byte enables and packet type in the case of address or request flits. This translates into an 18% overhead since each flit has 7 control and 32 payload bits. Data bits may be increased if less overhead is desired with an increased switch area and power demand. This will also reduce multi-flit packet latency since fewer flits will be needed, at the cost of higher fanout of the MUX control signals. In our implementation, 32 payload bits were chosen to match RAM block data width.

Each switch is identified by unique X,Y coordinates. The flit's final destination is determined by two extra bits specifying the user block (PE) among the 4 the switch is connected to. Flit format is illustrated in Figure 2.2.

Packet ID (6)	Flit Type (1)	Packet Type (4)	Byte Enables (4)	Destination Address (22)	PE Output (2)

(a) Address flit.

Packet ID (6)	Flit Type (1)	Flit Data (32)

(b) Data flit.

Figure 2.2: Flit format. 39 Bits total - 7 control, 32 payload

The initial flit of a packet is an address or request flit. Data flits in the same packet have the same ID and will be treated by each switch as the corresponding address flit was. Since flits are eagerly forwarded without being able to process their headers, all flits in a packet will be incorrectly eagerly forwarded in the same way throughout the network. The same applies to the duplicate flit complication, explained in subsection 2.3.3. Attempting to do otherwise would require combinational logic in preferred path hops and thus would dramatically increase per-hop latency.

2.3.2 Routing Logic

Based on our need to accurately classify flits as dead, we choose to implement a deterministic routing algorithm for normal network operation. Non-deterministic (adaptive) routing algorithms introduce uncertainty in dead flit classification. This uncertainty is due to the fact that conditions, and therefore adaptive routing decisions, are subject to change at any time. Therefore, the switch currently examining a flit is unsure if the flit's previous hop regarded this switch as the best next hop at the time, or if the flit was incorrectly eagerly forwarded. Since making switches aware of neighbouring network configuration is too costly, we adopt a deterministic routing algorithm. However, in subsection 2.3.4 we propose a design extension that switches to adaptive routing algorithm when its benefits are needed.

As a result we chose a slightly modified version of XY routing. XY routing instructs a flit to first complete its movement in the X axis, and then switch to the Y axis to reach its destination. Our NoC follows this routing algorithm, but is more flexible in allowing eager forwards that do not adhere to strict XY routing. Specifically, a flit is considered to have been correctly eagerly forwarded, and therefore is not forwarded to another output by the switch, simply if it is approaching its destination in any of the two axes. This may result in a flit reaching its destination via a route that does not comply with strict XY routing.

In the example Figure 2.3 illustrates, the flit arrives from source S to destination D solely through preferred paths (solid lines). Switch A sees that the flit approached destination D in the Y axis, and therefore regards this eager forwarding as correct. XY routing would have the flit pass through non-preferred paths (dashed lines) and switch dimensions at node B, after having fully completed its traversal in the X axis. Because we would like to provide preferred paths with full flexibility, and also because disallowing these paths by forwarding flits again in non-preferred paths introduces an unnecessary overhead, we chose to modify our XY routing algorithm accordingly. Similarly, a flit is considered dead simply if it moves away from its destination in any of the two axes

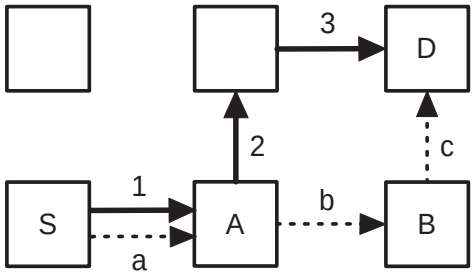


Figure 2.3: Correct eager forwarding scenario that does not comply with strict XY routing.

2.3.3 Duplicate Flits

Due to the above mechanisms, our NoC faces the complication of multiple copies of the same flit reaching their destination via different routes. An example of such an occurrence is illustrated in Figure 2.4. In that example, the flit leaving source S will be eagerly forwarded via the preferred path (solid lines) until it reaches destination D. However, switch A will regard this eager forwarding as mistaken since it has no preferred path knowledge for its neighbours and the flit's distance

from destination D increases in the Y axis. Supplying neighbouring preferred path knowledge to switches would lessen this problem, but is too costly. Therefore, it will forward another copy of the flit to destination D via non-preferred paths (dashed lines). Duplicate flits must be handled at the network interface logic. Network interface issues are addressed in subsection 3.2.1.

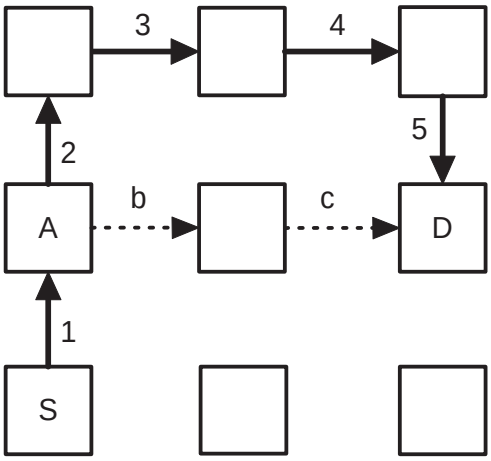


Figure 2.4: Duplicate flit scenario.

2.3.4 Adaptive Routing Extension

Adaptive routing is the ability to dynamically decide the best switch output for flits according to some criteria [27]. For instance, a switch might desire to avoid congested areas in the network and therefore route flits around those areas until congestion is resolved. Similarly, a switch might want to avoid heated areas in thermal-aware applications [21, 22].

Implementation of an adaptive routing algorithm in our NoC introduces uncertainty in dead flit classification. Since adaptive routing decisions depend on conditions (*i.e.* factors external from the switch), they are subject to change at any time. Therefore, a switch examining a flit to determine if it's dead is unsure whether that flit's previous hop regarded this switch as the best choice at the time, or if the flit was incorrectly eagerly forwarded. Neighbouring preferred path knowledge would enable a switch to distinguish between those two cases, but is not implemented due to the high cost it would impose.

On the other hand, adaptive routing has benefits to offer. Not only can it route flits around congestion points thus allowing them to resolve quicker and avoid flit delays, but it can also assist with any design priority such as the whole chip maintaining low temperature. Moreover, it can route around faulty NoC areas which become unusable due to some silicon defect - a factor which is becoming more important as technologies become narrower. For this reason we propose an extension in this section that allows parts of our NoC that require assistance from adaptive routing to operate under it. This operation mode change could either trigger from heavy congestion nearby a switch, silicon fault or any other priority the target application domain dictates. Implementation details of adaptive routing (routing factors, conditions it takes into account and priorities, decision thresholds) depend on application demands. This operation mode lasts only for as long as required according to some threshold set at design time. Our NoC operates under

normal conditions with the deterministic modified XY routing algorithm explained in subsection 2.3.2.

The decision to change to adaptive routing operation mode is made by each output individually since we would like to be able to apply adaptive routing exactly where needed, and also because each output features independent logic as explained in section 3.1. When this decision is made, combinational routing logic blocks located in inputs are alerted. After that time and until that output alerts the routing logic blocks that it has left adaptive routing operation mode, if an input would forward the initial flit of a packet to that output according to the modified XY routing described in subsection 2.3.2, the flit is examined by adaptive routing. It is then forwarded to a switch output according to adaptive routing instead.

As outlined in subsection 2.3.1, a packet is formed by possibly many flits. The initial flit of a packet contains the destination address. Routing logic calculates the switch output this packet should be forwarded to and stores that output and the packet's ID in an array. Since subsequent data flits do not contain the destination address, they can only be forwarded according to that array entry. That array entry is removed when the tail flit is encountered.

While this prevents out-of-order delivery of flits belonging to the same packet from occurring due to operation mode changes, it introduces the complication that if an output's operation mode changes, later flits of a packet, including flits in the output's FIFOs at the time the operation mode changes, will have to be forwarded according to previous-mode routing decisions. In case of adaptive routing operation mode being triggered due to congestion or thermal awareness, forwarding a few more flits to those areas is not an issue and can be compensated by lowering the decision threshold to change to adaptive routing. In case of an area of the NoC turning faulty, the first flits of the packet which have already been forwarded there would have already been lost if the error affected the packet's path. Therefore, forwarding and possibly loosing the subsequent flits makes no difference. There are cases where the first flits of a packet were forwarded safely before the fault which will affect subsequent flits occurs. However, the rarity of silicon defects combined with the low chance of these cases occurring does not justify modifying our implementation to cover these scenarios.

When a switch output decides to change to adaptive routing it disables (breaks) it's current preferred path by disabling the appropriate tri-state driver. Leaving any preferred paths enabled would mean that a substantial number of traffic may pass through to NoC areas we would like to protect through adaptive routing. Flits travelling in that preferred path will be handled as flits in all other paths. Later flits of a packet that had its first flits forwarded by a preferred path will have its later flits forwarded according to that preferred path as well, for the reasons explained in the previous paragraph and to prevent out-of-order delivery. Out-of-order delivery is discussed in section 2.5.

In case of congestion, preferred path performance may already be lost as described in section 2.4. Therefore, breaking preferred paths while operating in adaptive routing mode due to congestion does not introduce any performance degradation for flits travelling in preferred paths that would not already occur. In case of other factors, such as faulty areas, we might not want to forward flits there under any circumstances. In any case, if an application does not mind leaving preferred paths intact for some conditions (*e.g.* temperature), such an implementation is feasible.

Finally, since adaptive routing may forward flits to outputs that would make next hops consider them as dead, we need to label them accordingly. Therefore, if a a flit would be forwarded to an output due to adaptive routing, a bit in the packet type field, as illustrated in Figure 2.2(a),

is asserted. Switches examining a flit with this bit asserted never regard it as dead. They also clear it before forwarding it to the next hop. If a flit labelled this way enters a preferred path, this bit will only be cleared when the preferred path ends and in any other intermediate switches that decide to forward it again in between. This may result in duplicate flits reaching their destination, but this issue is already accounted for and handled as explained in subsection 2.3.3.

Design and implementation of the exact adaptive routing algorithm depends on application needs and priorities. The techniques presented in the subsection provide the mechanisms to allow operation of parts of the NoC under adaptive routing.

2.4 Deadlock-Freedom

XY routing is deadlock-free [32]. Therefore, network deadlock hazards in our NoC are introduced by adaptive routing and preferred paths since they do not necessarily follow XY routing and flits propagate in them without any control. End-to-end deadlocks are discussed in subsection 3.2.1. To guarantee that a flit will never wait indefinitely to be served, a switch needs to be able to serve FIFOs, and therefore resolve contention, if the preferred path has been continuously active for an unreasonably long period of time with a FIFO non-empty. The adaptive routing extension presented in subsection 2.3.4 also provides the option of utilizing adaptive routing in this or nearby switches. FIFOs are then served until all are empty. During this time, flits arriving in the broken preferred path will be enqueued in the appropriate FIFO behind previous flits following the same path. However, we need to investigate the possibility of a flit traversing the network indefinitely. We combat this issue in two ways.

First, we provide constraints which, if followed, guarantee that no flit will indefinitely travel through the network under normal conditions (without adaptive routing). If all preferred paths in the NoC are straight lines, flit propagation follows strict XY routing. Therefore, every turn is handled by routing logic and flits cannot enter a circle. This is the case with original mad-postman networks [1, 6].

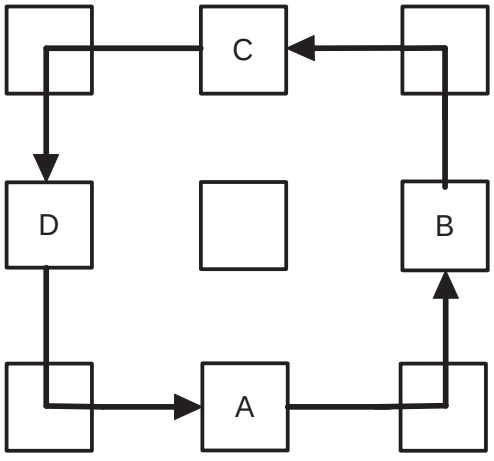


Figure 2.5: Circle formed by four preferred paths with exatly one turn.

Flits cannot enter a circle also in the case of preferred paths having exactly one turn. In this case, circles are formed by four different preferred paths as illustrated in Figure 2.5. Therefore, a flit would be examined by routing logic four times before completing a single loop. At these

times, the flit will either be considered dead and therefore be terminated, or it will be propagated according to XY routing. In the second case, in two out of the four checkpoints the flit will be forwarded according to the circle. However, in at least one of the other two it will be forwarded in the other axis and leave the circle.

Preferred paths with two turns may form a circle if the two routing logic checkpoints forward the flit according to the circle. Therefore, if preferred paths in our network contain up to one turn, no flits will indefinitely propagate in circle. This restriction does not take into account turns with switch data ports as they cannot be part of a circle.

Adaptive routing may construct scenarios that have a flit propagating in circles, depending on the implemented algorithm. For instance, even under the restriction that preferred paths cannot feature more than one turn, adaptive routing may forward the flit according to the circle in all four points of the circle that the flit is examined by routing logic and would otherwise leave the circle by XY routing. Also, some adaptive routing algorithms face deadlock issues regardless of preferred paths. Thus, the guarantee, as well as the extra required restrictions if any, that such a problem will not occur depends on the implemented adaptive routing algorithm.

Second, we investigate the consequences of a formed circle. As already described in subsection 2.3.2, each switch in the circle will examine if the preferred path forwarding was correct and forward a copy as necessary. This guarantees that a copy of the flit in the circle will be delivered to its destination. The flit will continue to propagate inside the circle. Other flits contenting for occupied resources will be forced to wait. If the flit in the circle propagates such as the preferred path is not idle at any clock cycle, contenting flits will face an increased queueing delay. However, as already described in the beginning of this subsection, contenting flits will eventually be served. This poses a performance issue, but no deadlock will occur. When FIFOs are served, the flit in the circle will be examined by routing logic and therefore may be terminated as dead.

2.5 Preferred Path Reconfiguration

Reconfiguration of our network's preferred paths consists of changing outputs' preferred inputs in the appropriate switches. Any PE or other user logic block can request reconfiguration by sending properly formatted single-flit configuration packets. These packets contain the destination node, the output to be reconfigured and the new preferred input. Configuration flits are enqueued in the appropriate crosspoint FIFO of their destination even if they follow a preferred path. When a configuration flit is selected by the output's arbiter, it is stored in the output's configuration register which stores the active configuration, instead of being forwarded to the next hop.

If we were to immediately alter the tri-state enable signals, we would risk out-of-order delivery of flits belonging to the same packet. Consider the example of Figure 2.6. Switch S transmits a packet to destination D. The initial flit (flit I) is constantly in non-preferred paths (dashed lines) and therefore is forwarded at every hop by XY routing logic. If a user block in the network was to reconfigure switch A to select input 1 as preferred for output port 2, later flits (flit II) would now reach destination node D via a preferred path (solid lines). Therefore, if flit I is in transit and switch A is reconfigured before the last flits of that packet reach it, those last flits could reach destination D before the first flits.

Because out-of-order delivery of flits belonging to the same packet can be a nuisance for destinations and also because address flits must always precede the corresponding data flits, it

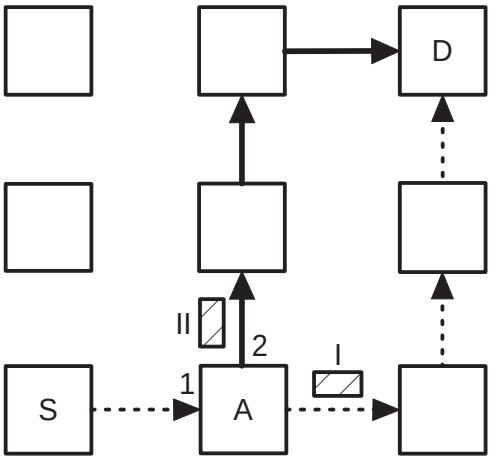


Figure 2.6: Out-of-order delivery scenario.

must be prevented by our NoC. This can be accomplished by delaying the application of the new configuration for each output until it is safe. Specifically, once a new configuration is received at an output, it is only applied when the old preferred path has been idle for 1 clock cycle, the new preferred path's FIFO is empty and no more flits from a packet are expected in those paths. As explained in subsection 2.3.1, flits forming a packet are labelled. Therefore, after receiving the packet's address flit and until receiving the tail flit, the arbitration logic knows that more flits are expected in this specific path and therefore delays the application of the new configuration. Blocks requesting reconfiguration are unsure exactly when it is applied, unless application demands dictate the implementation of a reconfiguration acknowledgment or polling mechanism. Due to this technique, no preferred path will change at each used switch from the time it receives the first flit of a packet until it forwards the last. This ensures that all flits of the same packet follow the same path and that a switch will not wait indefinitely for tail flits. Thus, all flits belonging to the same packet will be delivered in-order. As explained in subsection 2.3.4, adaptive routing does not challenge this guarantee since all flits of a packet are forwarded according to the decision made for the address flit, under the active routing algorithm at the time.

Different packets from the same source to the same destination may be delivered out-of-order. Since switches have no information regarding more expected packets in the same path, they may apply their new configuration if the preferred path becomes idle for one clock cycle. Even if the source transmits back-to-back, the preferred path may become idle for one clock cycle due to broken preferred paths to avoid starvation effects caused by contention, as explained in section 2.4. Therefore, this issue must be handled by the network interface logic as explained in subsection 3.2.1. Alternatively, the application may restrict network reconfiguration to be performed and applied only when the whole NoC or the part of the NoC to be reconfigured is idle. Implementation of this restriction may require software synchronization primitives.

Arbitration logic serves flits stored in FIFOs until all are empty, as described in section 3.1. If any flits arrive through the preferred path during this time they are enqueued in their preferred output's FIFO. Arbitration logic can then serve them in priority according to the implemented algorithm. This imposes a single clock cycle delay regardless of contention from other inputs,

in this infrequent scenario. However, attempting to re-enable the preferred path in the same clock cycle that flits arrive in would introduce combinational logic, extra preferred path latency and timing hazards. These flits will still be forwarded from FIFOs according to the preferred path. This serves the purpose of avoiding out-of-order delivery scenarios and taking advantage of preset multi-hop preferred paths.

2.6 Backpressure

Our NoC needs to provide a mechanism for not dropping flits due to full FIFOs. This mechanism must inform each output in a switch whether it can safely transmit a flit to the next hop. Likewise, the previous switch would also be informed if it can safely transmit to the current. Therefore, long packets reaching a congestion point will be stored in many, possibly consecutive, switches. Flits of the same packet may not be stored in consecutive switches in case of other flits stored in the FIFOs in an interleaved manner. Since flits of the same packet are guaranteed to follow the same path and arrive in-order, no recombination care must be taken.

Depending on area constraints and traffic patterns, we can adopt two different approaches. According to the first, if any of the next hop's FIFOs that have the output port in question as their input is almost full (to cover for backpressure signal propagation delay), the output's arbiter is alerted to not transmit any more flits until the signal is de-asserted and is therefore safe again. This approach requires only one wire from each output's next hop and the simplest logic.

According to the second approach, one wire from each of the next hop's FIFOs that have the output in question as their input alerts the switch exactly which crosspoint FIFO is almost full. This way, the arbiter needs to process packet IDs of flits in FIFO heads to determine if it can safely transmit any of them. With this approach, FIFOs that are not full are able to receive flits, and thus communication and FIFO utilization is more efficient. However, 6 wires are required at each output and also the arbiter must be able to process flit packet IDs in each FIFO head.

Extra care must be taken for flits forwarded in preferred paths. In our NoC we cannot know the final destination of flits travelling in preferred paths before they have been eagerly forwarded, nor can we control their transmission. Therefore, if a backpressure signal to an output port is asserted, the preferred path leading to this output is broken. Thus, all subsequent flits in that path are enqueued in the appropriate crosspoint FIFO and later forwarded according to the preferred path. Alternatively, the preferred path could remain intact, but flits in that path are still enqueued as above. Therefore, as long as a flit travels in a preferred path, it is not affected by contention or congestion. However, since preferred path flits take precedence over flits in FIFOs, congestion is slower to resolve.

Chapter 3

Switch Architecture & Topology

This chapter is divided into two sections. The first, section 3.1, explains our switching node's internal architecture and discusses virtual channels in our NoC. The second, section 3.2, presents our network's topology and outlines the role PE network interface logic plays.

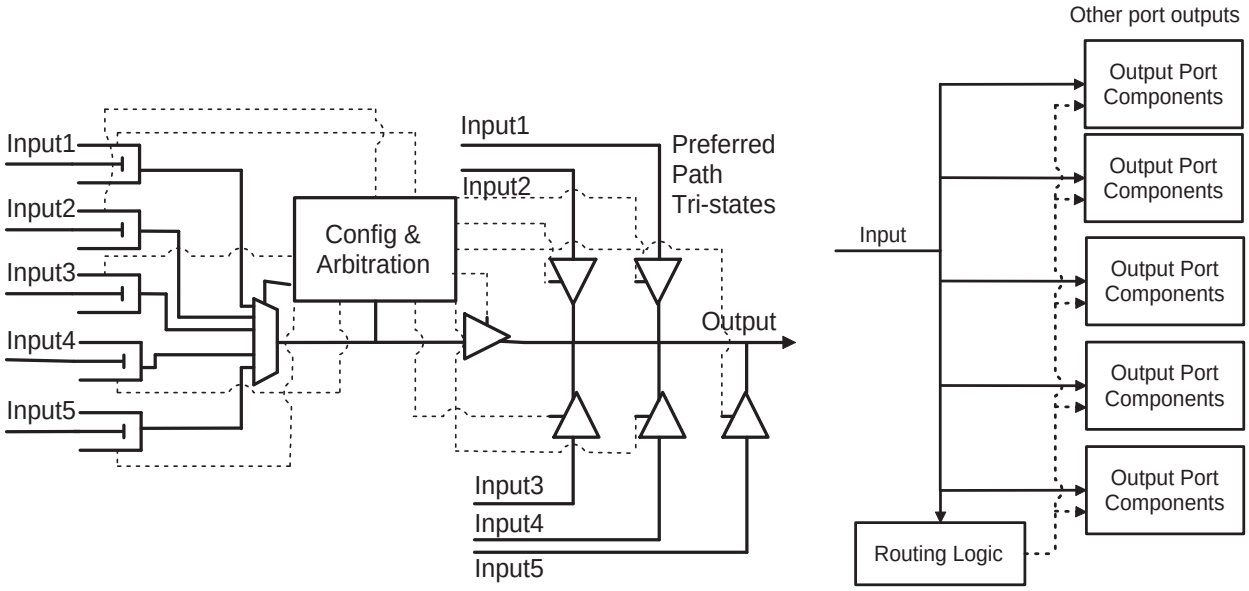
3.1 Switch Architecture

Switch input port components and connections are shown in Figure 3.1(b). Output port components are shown in Figure 3.1(a). Data wires are illustrated as solid lines and control wires as dashed lines. Our switch is a composition of the above figures, featuring 6 input/output ports. Each input is connected to each other port's output. We do not connect an output to it's own port's input as we consider that flits will not desire to leave the switch from where they entered. Our switch resembles a buffered crossbar [8] in that it features one FIFO at each crosspoint and independent configuration and arbitration logic at each output. Preferred paths resemble cut-through paths in buffered crossbars. A combinational routing logic block at each input decides at which FIFO, if any, should the incoming flit be enqueued.

This choice of switch architecture takes into account mad-postman's operation, since incoming flits are examined by the routing combinational logic before being able to be enqueued into FIFOs. Therefore, dead flits do not occupy FIFO lines. Moreover, since our current NoC does not include virtual channels, as addressed in subsection 3.1.2, crosspoint queueing removes the nuisance of head-of-line-blocking. Finally, the use of one arbitration and configuration logic block per output results in simpler logic and therefore shorter critical paths.

An alternative switch architecture examined featured FIFOs at input ports (input queueing). However, with this choice dead flits would occupy precious FIFO space and also have a greater impact on non-preferred flits since the latter would have to wait for any dead flits to be served from the FIFOs at a rate of 1 clock cycle per node, at best. Moreover, enqueueing and serving dead flits from the FIFOs is more wasteful as far as power is concerned. To avoid these and the head-of-line blocking problems, VCs would need to be included in this implementation. However, as explained in subsection 3.1.2, VCs are otherwise not needed since we already perform classification of our flits. Therefore, our switch architecture does not follow input queueing. A macroscopic illustration of the input queueing approach in a 4x4 switch, without showing preferred paths and without VCs, is provided in Figure 3.2.

Output configuration logic is responsible for storing and updating preferred path configura-



(a) Output port components. (b) Input port components.

Figure 3.1: Switch architecture.

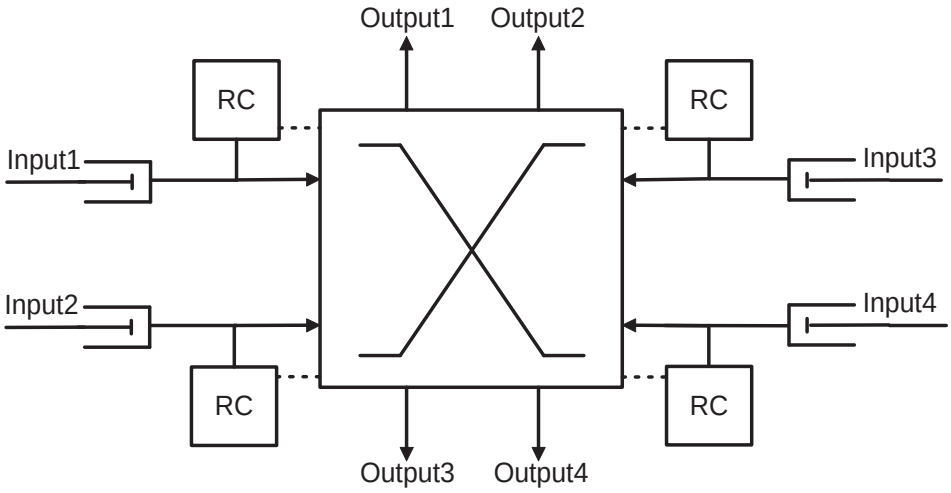


Figure 3.2: A macroscopic illustration of a 4x4 switch input queueing approach, without preferred paths and VCs.

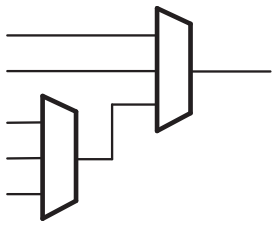


Figure 3.3: 5-1 multiplexer with 3-1 cells.

tion. Arbitration logic is responsible for serving the FIFOs. Non-empty FIFOs as well as FIFOs to which a flit is being enqueued in the current clock cycle are selectable. Arbitration logic selects a selectable FIFO to serve during the next clock cycle if the preferred path is idle during the current clock cycle. This serves the purpose of prioritizing preferred path flits without unreasonably preventing FIFOs from being served. It stops serving them when they are all empty. Arbitration takes place during the preferred path idle cycle for the next cycle. Therefore, our NoC achieves one clock cycle per-hop latency for non-preferred paths when there is no contention. Arbitration algorithm details may depend on exact NoC demands.

Each output is driven by tri-states directly connected via dedicated wires to each other port's input. Each output is also driven by a tri-state which connects the output wire with a multiplexer which forwards the FIFO flit being served by arbitration logic, if any. These tri-states need to drive one FIFO and one tri-state at each of the next hop's outputs other than the port that the current switch is connected to. They also need to drive the combinational routing logic located in the input. Therefore they have a fanout of 11.

Tri-state enable signals are driven by the output's configuration and arbitration logic. Preferred paths are thus formed by pre-enabling tri-states, therefore connecting an input with any number of preferred outputs. Tri-states were selected instead of multiplexers because our implementation library does not include a single 5-to-1 multiplexer (AND-OR or other equivalent) cell. Therefore, we would have to construct such a multiplexer cell with multiple library cells, imposing a greater latency than a single tri-state driver. We could implement this 5-1 multiplexer to have some inputs pass through a single multiplexer cell and others through multiple cells, but we do not want to prioritize paths at design-time, only at run-time according to the application's preference. An example of such an implementation is illustrated in Figure 3.3.

Depending on preferred path flexibility and area needs, an extra optimization may be necessary to further reduce switch area. Instead of directly connecting each input to each other output, a preferred path bus could be deployed, as in Figure 3.4. This vastly limits the number of preferred input-output pairs that can be configured to only one input with any number of outputs. However, intermediate designs can also be implemented. For instance, one such preferred bus in the X axis and one in the Y could be deployed, perhaps even connected to each other with tri-states. Therefore, depending on exact preferred path communication needs, NoC area overhead can be reduced.

3.1.1 Routing Logic Implementation

This subsection examines the input port's combinational routing logic without the adaptive routing extension presented in subsection 2.3.4. The routing logic's purpose is to determine which

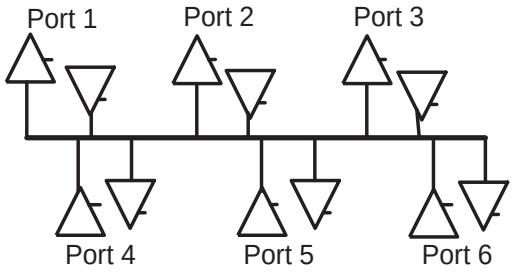


Figure 3.4: Preferred path bus.

output does each incoming flit need to be forwarded to. Incoming flits are examined by this logic as soon as they arrive, independently of preferred paths. This computation extends until FIFO enqueue enable inputs. The output arbiter regards as selectable, and includes in its arbitration, FIFOs to which a flit is currently being enqueued. All these computations and decisions constitute the critical path and are completed within a single clock cycle.

A flit should not be forwarded to any output if it is dead or if it was correctly forwarded by any of that input's preferred outputs. Otherwise, an output is chosen according to XY routing and the appropriate FIFO enqueue enable signal is asserted. The conditions for these decisions are presented in subsection 2.3.2. Input port components are illustrated in Figure 3.1(b).

To be able to detect flits that are moving away from their destinations, we implement comparators which inform us if the incoming flit's destination has a higher X coordinate than the switch examining the flit. This check is made only at X axis inputs since the flit did not traverse the Y axis with its last hop. The same check is made for the Y coordinate at Y axis inputs. No dead flit detection is performed at PE data input ports since these incoming flits were just generated by PEs and thus are never dead. We then evaluate the result depending on the input port that is examining the flit and therefore the direction the flit is travelling through the network. For example, if the leftmost input port of the X axis detects that the flit destination's X coordinate is smaller than that of the switch, it knows it is dead since the flit is travelling from left to right whereas the destination is to its left. The verilog code which performs dead flit detection is shown in listing 3.1.

Listing 3.1: Dead flit detection logic

```
1 wire larger_Y, equal_Y, lesser_Y;
2 wire larger_X, equal_X, lesser_X;
3
4 // Coordinate comparators. in[8:2] & in[15:9] are destination Y and X respectively.
5 comparator comp_Y (‘CURRENT_SWITCH_Y, in[8:2], larger_Y, equal_Y, lesser_Y);
6 comparator comp_X (‘CURRENT_SWITCH_X, in[15:9], larger_X, equal_X, lesser_X);
7
8 // Wires below detect flits moving away from their destination in any axis.
9 // There are two cases for moving away in the X axis, and two for the Y axis.
10 // ‘POSITIVE_X is the leftmost X input (flits travel in positive axis flow).
11 // Since the first part of the comparison is static (input port is also static),
12 // synthesis will remove it according to which input port we instantiated.
13 wire leaves_dest_X1 = (input_port == ‘POSITIVE_X & larger_X);
14 wire leaves_dest_X2 = (input_port == ‘NEGATIVE_X & lesser_X);
15 wire leaves_dest_Y1 = (input_port == ‘POSITIVE_Y & larger_Y);
```

```
16 wire leaves_dest_Y2 = (input_port == `NEGATIVE_Y & lesser_Y);
17
18 wire is_dead = leaves_dest_X1 | leaves_dest_X2 | leaves_dest_Y1 | leaves_dest_Y2;
```

The most common XY routing implementation has the source noting in the packet's header the number of hops that need to be made in each axis with one extra bit to mark the direction. Each hop checks if the X axis value is non-zero. If it is, it decrements that value by one and forwards it in the X axis appropriately. Otherwise, it checks the Y axis value in a similar fashion. If both are zero the packet is ejected to a local PE port. This implementation requires the simplest routing logic in switches. However, it is not an option for our NoC because switches cannot alter the headers of address flits travelling in preferred paths since forwarding is performed before the header is even processed.

Therefore, in our NoC address flits contain the destination's coordinates. Switches check if the destination's X coordinate is different than their own. If so, they forward the flit to the appropriate direction in the X axis. Y axis coordinates are handled in a similar way only if the destination's X coordinate matches that of the switch. The XY routing computation runs in parallel with the dead flit detection and uses the same comparators for the X and Y coordinates for efficiency. The results are then considered by a combinational logic which decides among X axis outputs first, then Y axis and finally local PE outputs.

Destination address information is only carried by address flits. Thereafter data flits need to be forwarded in the same manner using only their packet IDs. The routing and dead flit detection logics shown in this subsection make decisions for address flits only. These decisions are stored in a simple array holding packet IDs and switch outputs. Data flit packet IDs are looked up in that array. If an entry exists, the flit is forwarded to the stored output. If an entry does not exist, the flit is terminated as dead. The entry is removed when encountering the tail flit. Address flits destined to an adjacent PE are forwarded to a Y axis output whereas the appropriate switch data output for data flits is calculated and stored in the array instead. This array needs to hold only multi-flit packet IDs. In our implementation that array consisted of only four entries. This number may need to be increased depending on the expected traffic pattern and network size.

Since dead flit detection and XY routing run in parallel, merging of these two results is accomplished by few gates which receive both of their outputs. If the dead flit detection signal is asserted denoting that the flit is dead, the output port selection by the routing logic is ignored and no FIFO enqueue signals are asserted. The same occurs if the input is not receiving a valid flit either because no flit is being transmitted during the current clock cycle or because the flit has not fully arrived yet. This detection is performed by a simple logic which also runs in parallel. A flit is invalid if its type is 00 and its packet ID is not 11111, which denotes a configuration flit. The XY routing verilog code is shown in listing 3.2.

Listing 3.2: XY routing logic

```
1 // Flit needs to be sent to X axis, positive flow. Similarly for the other cases.
2 wire needs_X_pos = lesser_X;
3 wire needs_X_neg = larger_X;
4 // We will need to first check for X forwardings, then Y, then local PE.
5 wire needs_Y_pos = lesser_Y;
6 wire needs_Y_neg = larger_Y;
7
8 // Now we check for local PE forwardings. Flit delivered by this switch.
```

```
// Equal X is not needed due to output_port_select conditions order.
wire stop_here = equal_Y;
// This logic examines address flits only. Data flits will simply search the array.
// Below we decide the Y port for the address flit. in[1:0] select among the 4
// connected PEs. We store the appropriate data port into the array for data flits.
assign address_top = in[1] == 1'b0 & stop_here;
assign address_bottom = in[1] == 1'b1 & stop_here;

// Is the incoming flit valid.
wire is_valid = in[38:37] != 2'b00 | in[36:32] == 5'b11111;

// Logic deciding the output. The logic explained in the paragraph below is at
// output modules and thus not shown here. id_array[] is the data flit output
// array. Lookup logic is direct-mapped. It works sufficiently as a function of
// the packet ID, containing the source ID, if distant sources rarely send packets
// and all sources do not interleave packets. Generally, it needs to be changed.
assign output_port_select = (!is_valid) ? 3'b111 : (in[38] ?
id_array[in[36:35]][2:0] : (is_dead ? 3'b110 : (needs_X_pos ? 3'b000 :
(needs_X_neg ? 3'b001 : ((needs_Y_pos || address_top) ? 3'b100 :
((needs_Y_neg || address_bottom) ? 3'b101 : 3'b111))))));
```

If the flit is valid and non-dead, we need to check if it was forwarded by a preferred path to an output which makes the flit approach its destination in any axis. If XY routing decided to forward the flit to a X axis output, the acceptable outputs are that one plus the Y axis output that would be chosen, if any, had the flit reached the destination's X coordinate. If XY routing decided to forward the flit to a Y axis or a local PE output, there are no other acceptable outputs. Thus, each FIFO at a Y axis or PE output has an AND gate which ignores the incoming FIFO enqueue signal if the same path's preferred tri-state driver is enabled during the same clock cycle. If the flit is to be forwarded to a X axis output, the Y axis coordinate comparison result decides which of the two Y axis outputs we need to check if it has the input port receiving the flit as preferred. If that input port is indeed preferred, FIFO enqueue signals are de-asserted.

3.1.2 Virtual Channels

Virtual channels (VCs) are useful for defining multiple logical topologies within the network, adaptively routing around congested or faulty nodes and providing packet priority and thus guaranteed QoS classes [32]. However, in our NoC preferred paths already provide a means to prioritize packets compared to others as well as form different low-latency topologies. Moreover, our NoC's topology is already tailored to our specific application environment. Finally, we have already proposed a modification to implement adaptive routing without VCs explained in subsection 2.3.4, despite the increased challenges our NoC faces.

For these reasons, our current NoC does not include VCs. Introducing them would multiply FIFOs, which translates into a significant area overhead since our switch features one FIFO at every crosspoint. However, other NoC applications may have different design priorities and requirements which make VCs more attractive.

3.2 Network Topology

We tailor a 2D mesh topology to our target application, which is an array of processors and RAM blocks, aiming to minimize area overhead in addition to latency. We assume a flexible system

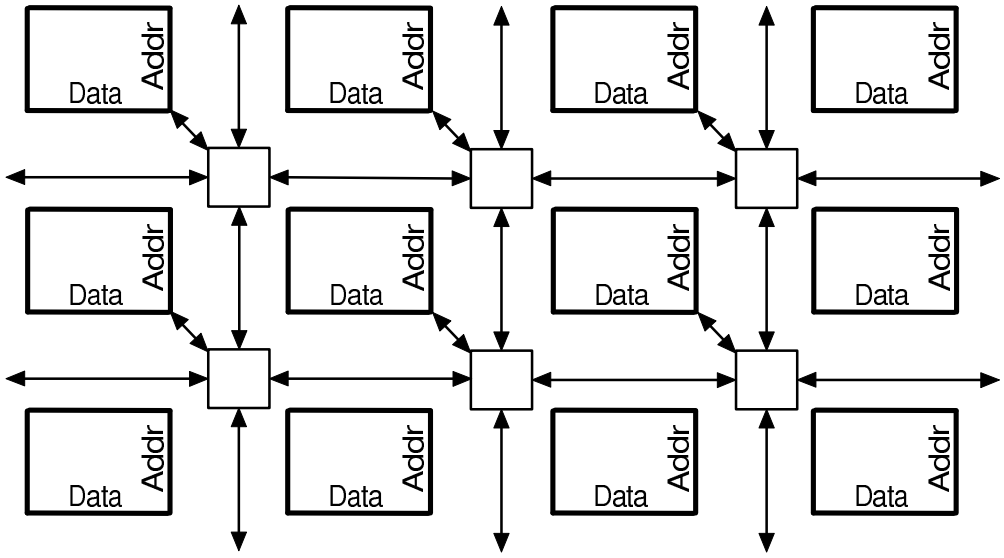


Figure 3.5: The simple 2D mesh topology.

that assigns memory blocks to processors' low-latency regions according to application needs, and therefore profits from the reconfiguration capabilities of our NoC. We used single-port RAM blocks as explained in section 4.2. In our 130 nm implementation library, RAM blocks feature data pins on one side of the X axis and address pins on one side of the Y axis. We are able to rotate and mirror RAM blocks to place their pins as needed by each topology option.

We considered several different network topologies:

- The simple 2D mesh topology shown in Figure 3.5. This topology was one of the first to be proposed [9]. It is regarded as the simplest choice and it is used by many general-purpose NoCs which do not have any a-priori application environment information to optimize their topology. The major drawback with this option is that switches, which are 5x5, are placed in the corners of PEs. Therefore, a significant amount of empty space has to be left between PEs. This problem could be lessened by floorplanning the switch as a cross or rectangle, but not completely solved. Another option to combat this problem is dividing the switch into multiple sub-switches and placing then in ascending order resembling a staircase, as proposed in [33]. Moreover, the switch, due to it's location, is equally far away from data and address pins, whereas we would like the switch to be right in front of both. However, the major reason we did not choose this topology is that it does not take any advantage of our application environment and the options we have, such as rotating RAM blocks and forming larger network blocks with several PEs.

- The second option we investigated, shown in Figure 3.6, is dividing the network into two sub-networks. The initial thought was to create address (request) and data sub-networks. This would enable us to perform optimizations in resources since the address sub-network would carry less traffic. Moreover, the address sub-network could be implemented as a broadcast tree to the RAM blocks nearest to the processor (in its local region). This would impose low latency and could prove handy to fully-associative cache implementations. Requests to distant RAM blocks would utilize the data sub-network. This option may very

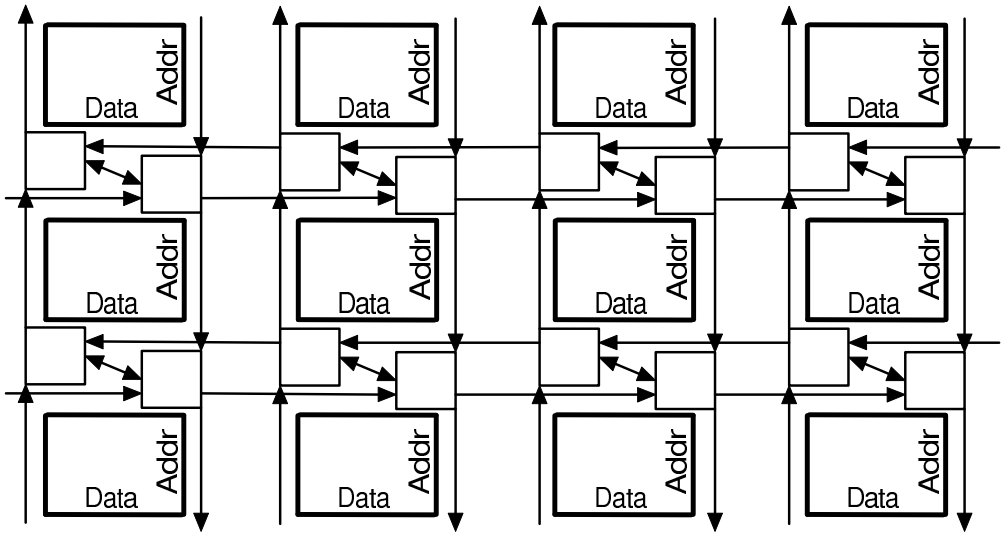


Figure 3.6: 2D mesh topology with two subnetworks.

well be the best choice for some application environments, but in the general case would prove area and power consuming. Moreover, RAM blocks need to be able to receive address flits from both sub-networks which complicates their network interface logic.

Another option is deploying two sub-networks to serve different directions. In the example shown, one sub-network serves one direction in each axis. Therefore, traffic from a processor to a RAM block would probably utilize one network and the response generated from the RAM block the other. However, in many cases traffic would need to change sub-networks to reach it's destination. Thus, sub-networks need to be interconnected with each other, but not necessarily at every switch. This option, as shown, has only the advantage of dividing traffic (and therefore congestion) among sub-networks. However, interconnections between sub-networks may easily prove to be bottlenecks. Finally, this option's area overhead is significant.

- As a variation to the previous option we can assign a sub-network to each axis, shown in Figure 3.7. This option enables us to place a switch right in front of RAM pins, therefore saving in power and latency due to minimum length wires. Sub-networks still need to be interconnected as the previous option and those interconnects may prove to be bottlenecks depending on traffic patterns. This option still imposes a significant area overhead as the previous option. Moreover, this option's sole advantage does not outweigh it's disadvantages, especially considering that the independence between traffic in different axes is achievable even in the first option with the proper switch implementation.

- The option illustrated in Figure 3.8 takes into account RAM block pin placement. RAM blocks are rotated to place data pins every two X axes. This enables us to place switches only in the X axes with RAM data pins, therefore deploying half the amount of switches compared to the previous options. The switches are larger compared to those of the previous options, but deploying half the amount of switches provides us with considerable savings in power and area therefore making this option attractive. Wires in the Y axis

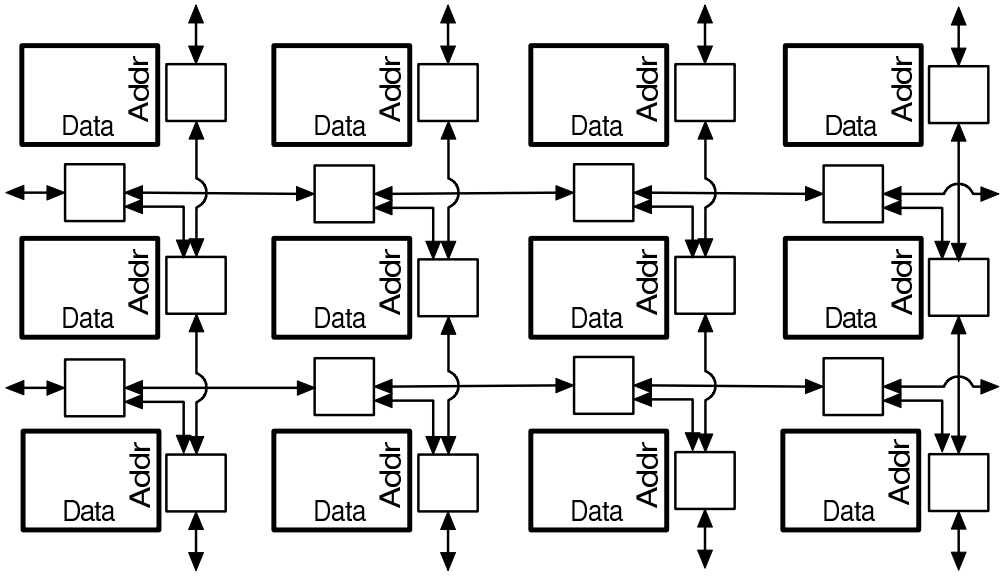


Figure 3.7: 2D mesh topology with two subnetworks - one for each axis.

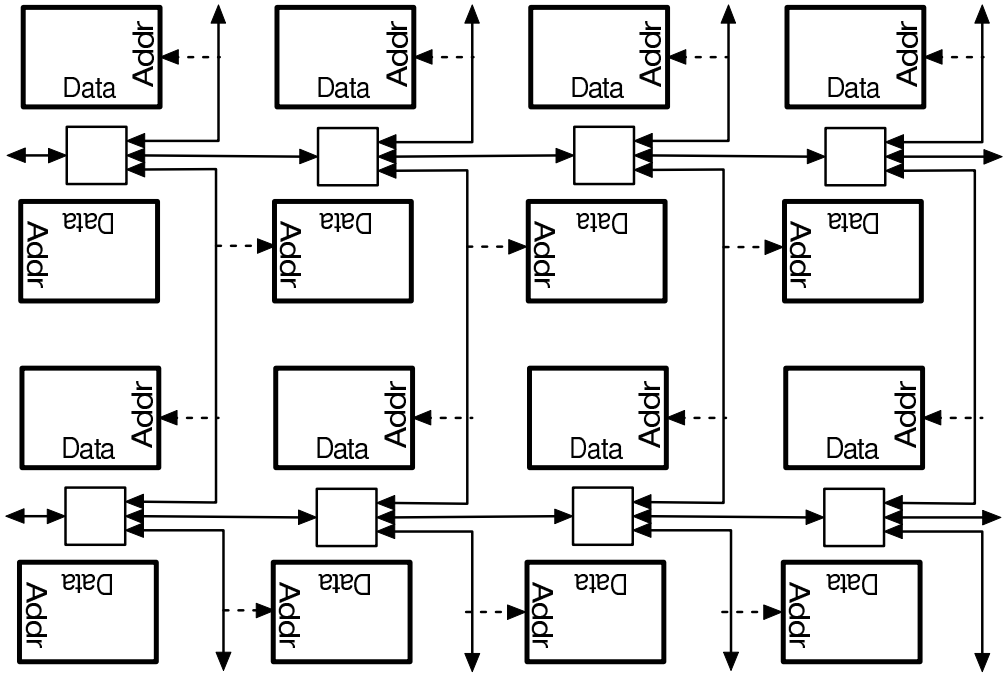


Figure 3.8: RAM blocks rotated to place switches every two X axes.

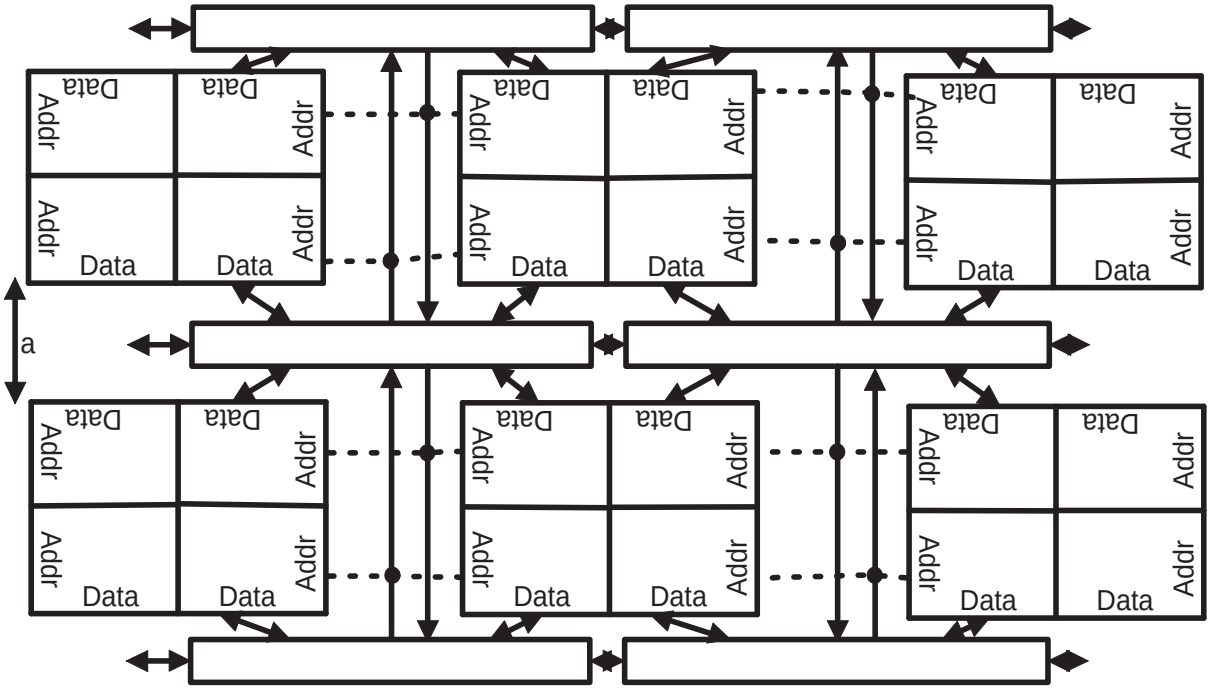


Figure 3.9: Rectangular-shaped floorplan.

have a length equal to twice a RAM block's height. We also investigated variations which featured wires only every two Y axes, as well as interconnecting switch (X,Y) with (X+1, Y+1) instead of (X, Y+1).

Our NoC's topology is a modification of the above last option. Since we would like to feature wires only every two Y axes and minimize empty space, we place four RAM blocks to form one larger network block. We rotate and mirror those RAM blocks to place all data pins on the X axis and all address pins on the Y axis. CPUs and other user blocks may be placed as part of such a block or as a whole network block themselves, depending on their size. We deploy one switch per network block (four PEs). This results in significant savings in area and power due to the limited amount of switches compared to topologies with one switch per PE. Moreover, this reduces the latency between two endpoints since flits need to traverse fewer switches. This topology, with the switch shaped as a rectangle in the X axis, is illustrated in Figure 3.9.

Each switch has 6 input/output ports. Each input is connected to each other port's output. Two of these input/output ports are used for inter-switch communication in the X axis, and other two in the Y axis. The rest two ports are used for communication with the data pins of the 4 adjacent RAM or other user logic blocks. Given the data pin placement, one port is used for each two RAM blocks facing each other in the X axis. One data output is wired to both RAM block data input interfaces. The two RAM blocks' data outputs are connected to a simple logic illustrated in Figure 3.10, resembling an one-way 2x1 switch.

That 2x1 switching logic has no routing logic and only features one FIFO which is used by the non-preferred input. From the moment a RAM block receives a read request from it's address interface until it is able to output data, it notifies this 2x1 switch to choose that RAM block's data output as preferred. This switch therefore imposes minimal latency impact. It may

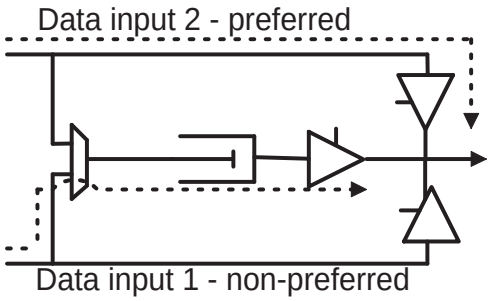


Figure 3.10: 2x1 switching logic.

also be reconfigured as other switches. If area permits, a request FIFO may be implemented to store the order of requests received by the RAM blocks. This will enable it to always anticipate the next generated flit, and thus avoid non-preferred path delays in case of multiple requests to both RAM blocks.

RAM block address interfaces are wired to the nearest Y axis output. These connections are illustrated with dashed lines. Thus, the RAM blocks immediately above or below a switch have their address interfaces wired to the Y output leading upwards or downwards respectively. This way, we avoid implementing extra output ports for address inputs. Therefore, a switch forwards address flits destined to a local RAM block to a Y output and data flits to a data output. This means that flits reach RAM blocks through different outputs according to their type, since we choose the switch port closest to the proper RAM pins. As data input interfaces, address interfaces monitor each incoming flit to determine if it is destined for that RAM block. The potential increased contention for outputs wired to address interfaces is outweighed by the significantly less area required by our switch. Our switch would have 8 inputs and 12 outputs without the described optimizations.

Finally, for switches that serve exclusively RAM blocks, we can further reduce the required switch area since RAM blocks will never need to communicate to each other directly. Therefore, each switch data input should not be connected to the other switch data output, therefore saving two internal switch connections and all the accompanying logic.

For switch placement, we examined two floorplan alternatives evaluated in chapter 4. In the first, switches are placed in the corners of larger network blocks as a cross, shown in Figure 3.11. This requires only a small distance between user blocks in each axis. Moreover, wire length, and therefore propagation delay, between each switch is minimal, even in the Y axis. The second placement, shown in Figure 3.9, has the switch solely in the X axis between two large blocks in a rectangular shape. User block distance in the Y axis is minimal, and is only used for memory address interface logic. Communication with switches in the Y axis is achieved by wires in higher metal layers routed above RAM blocks, or possibly in any metal layer routed above address interface logic. Y axis communication wire length is equal to twice the RAM block's height, approximating to 1 mm in our placement and routing.

3.2.1 Network Interfaces

PEs, RAM blocks and other user logic blocks need NoC interface logic. This logic is responsible for enabling communication between the NoC and the block. It is responsible for submitting

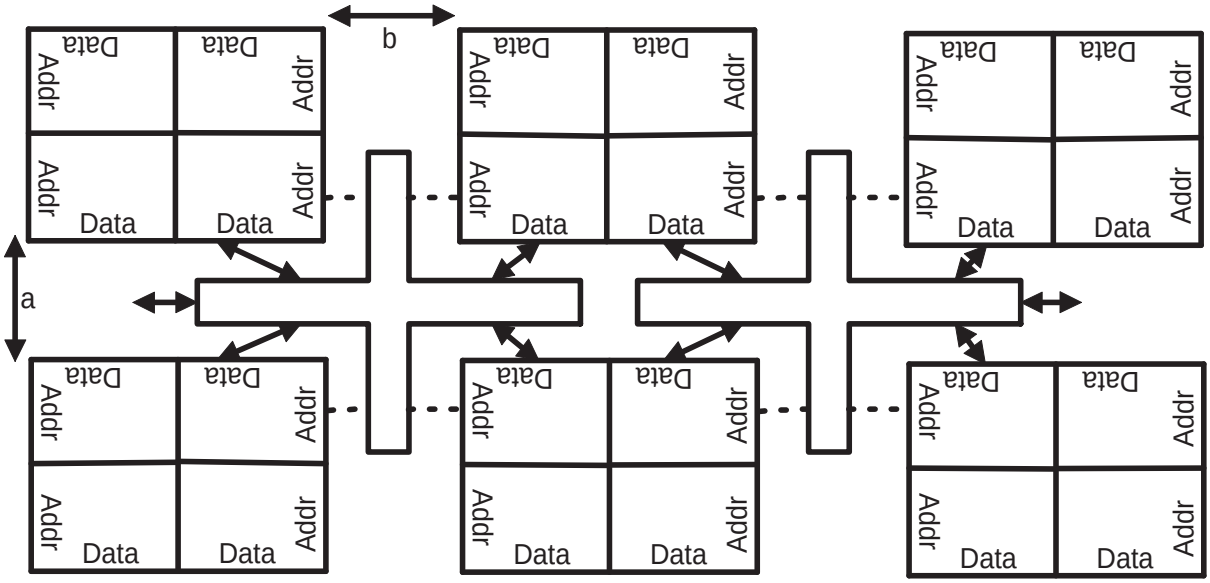


Figure 3.11: Cross-shaped floorplan.

properly formated packets divided in flits, as explained in subsection 2.3.1, as well as receiving flits destined to that user block. For data network interfaces, incoming flits must be briefly stored until the whole packet is complete and thus able to be submitted to the user block for processing. Address network interfaces only receive one flit per packet containing the address. In addition, network interface logic must be able to arbitrate between complete packets in a desired manner, *e.g.* submit read and write requests in the order they were transmitted by the source to satisfy sequential consistency.

Network interface logic must also identify which flits of a packet it has already received and discard duplicate copies. Duplicate flits were discussed in subsection 2.3.3. There are various implementations of this functionality according to design optimization priorities. To simplify this task, the flit control bits could be expanded, or the packet ID bits lessened, to include sequence number bits.

Out-of-order delivery of flits belonging to the same packet is impossible, as explained in section 2.5. However, flits belonging to different packets from the same source may be delivered in any order. Therefore, interface logic must be able to submit packets for processing once complete, regardless of other incomplete packets. To implement this functionality, we deploy multiple small FIFOs in data interfaces and registers in address interfaces, and enqueue flits accordingly. In case we would like packets to be submitted for processing in the order they were sent by the source, the relative packet order can be retrieved from the packet header.

Network interface logic must also handle multiple incoming packets from various sources. Flits from these packets may arrive in any order. If all buffer is used up, extra incoming flits remain in previous hops through backpressure. Therefore, no excessive buffer space is required.

Since one data interface FIFO is reserved per packet until it is complete and submitted for processing, and packets may arrive out-of-order, deadlocks may occur. Lets assume that packet A has partially arrived at the target RAM. Packet B from the same source arrives at the final hop before A's tail flit. However, all of the RAM's data interface FIFOs are reserved. Therefore,

packet B waits in the switches due to backpressure not allowing packet A's tail flit to arrive to complete the packet.

This scenario requires several packets arriving out-of-order through the same path with partially complete packets, since the data interface logic deploys several FIFOs. Assuming sources do not submit packets interleaved, out-of-order delivery from packets originating from the same source is only caused by reconfiguration. Therefore, limiting the number of active reconfigurations that can occur at any one time to less than the number of data interface FIFOs guarantees that at least one FIFO will be eventually freed and no deadlock will occur in this case. Implementation of this restriction may require software synchronization primitives, defining areas each CPU can reconfigure paths in, or a reconfiguration acknowledgment mechanism.

We can also investigate an implementation to detect and recover from end-to-end deadlocks that might rise if multiple sources are involved. For example, if in our application one PE could be a destination for more sources than the number of FIFOs in it's interface logic, these packets are multi-flit, follow the same path and get interleaved such as all address flits arrive before tail flits, a similar deadlock situation could occur. Some mechanisms for deadlock recovery we could implement, as well as a presentation of end-to-end deadlock issues, can be found in [34].

Chapter 4

Layout Results

This chapter provides results of our work. Section 4.1 begins with preliminary simulation results of some library cells. Section 4.2 presents some of our available RAM blocks and justifies our RAM block choice. Finally, section 4.3 presents results for our switching node after placement and routing. All experiments presented in this master's thesis used a 130 nm library. Synthesis was conducted with Synopsys Design Compiler version 2004.06-SP2, placement and routing with Cadence SOC-Encounter version 3.3 and simulation with Verilog-XL version 05.10.002-p.

4.1 Preliminary Results

As one of the first steps of our research, we conducted experiments to extract timing results from single library cells. They helped us understand what is the ideal latency of long wires and therefore how close our approach is compared to that. Moreover, through these results we confirmed that pre-enabling tri-state drivers or multiplexer cells is crucial for achieving low latency.

Long wires feature inverter cells and one buffer cell in case the total number of cells in the wire is not even. They are placed approximately every 2 mm of wire length. The drive strength of these cells varies and can even reach the library's maximum value of 32×, with the majority being 16× or 20×. Inverter cells of these drive strengths have an average latency of 140 ps. 2 mm long wires in between them imposed latency in the range of 65 - 150 ps depending on metal layer, driving strength of the cell driving them, and simulation conditions.

Wires between cells had a resistance of approximately 0.18 kΩ - 0.9 kΩ and a parasitic capacitance of approximately 45 fF - 250 fF, depending on their length and metal layer. Wires in higher metal layers are wider and therefore impose a smaller RC delay due to their reduced resistance. However, the connector (via) from a cell's connection point to a metal layer imposes an increased delay as metal height increases. Therefore, lower metal layers are preferable for short wires and higher metal layers for long wires. In our experiments, most wires were routed in metal layers 3 - 4 and 5 - 6 with many wires that connected neighbouring cells being routed in layer 1 - 2. Odd-numbered layers are used for horizontal wires (X axis) and even-numbered layers for vertical wires (Y axis). Our library features a minimum of 6 and a maximum of 8 metal layers. Wires in layers 7 - 8 can be routed above RAM blocks. Further information for wires and their attributes can be found in [35].

Preliminary simulations were conducted with 2-1 multiplexers. They were implemented with

Table 4.1: Multiplexer simulation results.

I/O Wires length	Typical case latency		Worst case latency	
	Pre-configured	Dynamic	Pre-configured	Dynamic
10 mm	254 ps	565 ps	260 ps	900 ps
8 mm	211 ps	496 ps	371 ps	836 ps
5 mm	240 ps	514 ps	368 ps	808 ps
2 mm	254 ps	590 ps	378 ps	893 ps
1 mm	196 ps	496 ps	395 ps	998 ps

single AND-OR cells. Implementations with NAND cells were also studied but they proved to be 60 ps - 270 ps slower. This is due to the fact that multiple NAND cells, which are faster than AND-OR cells, are required to implement a single multiplexer. Implementations with AND-NOR cells are only preferable if the output wire could have an odd number of inverter cells. If not, an otherwise unnecessary inverter cell needs to be added therefore making this implementation slower.

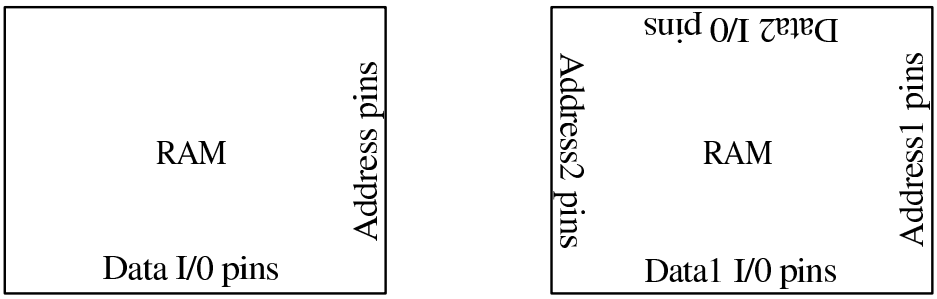
In order to determine the importance of pre-enabling multiplexers, we simulated a multiplexer which dynamically decides among it's inputs with a simple two-stage, five-input combinational logic of 6 logic gates which is driven by some of the multiplexer's current inputs. On the other hand, we simulated a multiplexer which stores the result of the same combinational logic in a register which drives it's select input. These circuits are shown in Figure 1.2. The length of the wires connected to it's inputs and outputs varied from 1 mm to 10 mm. Long wires featured inverter of buffer cells as already discussed.

Simulation results are given in Table 4.1. They present the latency from the time the multiplexer's inputs are stable until the multiplexer's output is stable. These results show that the dynamic multiplexer is 2 to 3 times slower than the pre-configured one. This confirms that pre-enabling select signals is important for reducing latency. As discussed in section 3.1, cells at the switch output driving the next switch have a fanout of 11. Thus, pre-configuration becomes even more important in our NoC due to this high fanout and therefore the increased latency to toggle the driven value. Finally, since our switch requires a 5-1 multiplexer at each output and our library does not include single AND-OR cells larger than 3-1, we deploy tri-state cells. However, the importance and benefits of pre-configuration remain the same.

4.2 RAM Blocks

We chose single-port RAM blocks of 4096 lines of 32 bits each (128 kbits in total), with a column multiplexer of 16. Without power rings, they are 715.07 μm long on the X axis, and 551.64 μm long on the Y axis. We deployed 25 μm wide power rings on each side adding 50 μm of length to each axis. These power rings allow our RAM blocks to operate at their maximum design frequency of 1 GHz. As shown in Figure 4.1(a), data input/output pins reside on one of the long sides of the RAM block (X axis), while address input pins reside on one of the short sides (Y axis). Control pins are on the same side with their respective port's data pins.

Latency and area attributes of several single-port RAM blocks are shown in Table 4.2. As



(a) Single-port RAM. (b) Dual-port and two-port RAMs.

Figure 4.1: RAM block pin placement.

Table 4.2: Single-port RAM attributes.

Lines	Bits per line	Column mux	Area	Clock-to-data latency	
				Typical case	Worst case
128	32	8	390 μm $\times$ 110 μm	1.1 ns	1.8 ns
256	32	8	380 μm $\times$ 140 μm	1.1 ns	1.9 ns
512	32	8	380 μm $\times$ 200 μm	1.2 ns	2.0 ns
512	64	8	720 μm $\times$ 200 μm	1.3 ns	2.2 ns
1024	32	8	380 μm $\times$ 330 μm	1.4 ns	2.4 ns
1024	32	16	720 μm $\times$ 190 μm	1.4 ns	2.3 ns
2048	32	8	380 μm $\times$ 560 μm	1.7 ns	3.0 ns
2048	64	8	720 μm $\times$ 560 μm	1.8 ns	3.2 ns
4096	32	8	380 μm $\times$ 1040 μm	2.2 ns	4.2 ns
4096	32	16	720 μm $\times$ 550 μm	1.9 ns	3.5 ns
8192	32	16	720 μm $\times$ 1020 μm	2.6 ns	4.9 ns

that table shows, the clock positive edge to valid output data latency increase between the 2048 $\times$ 32 and 4096 $\times$ 32 memories with a column multiplexer of 8 is 0.58 ns (under typical case conditions) which is disproportional compared to the 0.31 ns between the 1024 $\times$ 32 and 2048 $\times$ 32 memories. The same is true for the 0.68 ns between memories 4096 $\times$ 32 and 8192 $\times$ 32 with a column multiplexer of 16. Therefore, we chose the 4096 $\times$ 32 RAM block with a column multiplexer of 16. Choosing a larger RAM block would mean that our memories would be considerably slower than if we chose the previous-size blocks. On the other hand, choosing a smaller RAM block would mean that we would require more RAM blocks and therefore more wasted area on our chip due to network interface logic, power rings and other network overhead. Therefore, this RAM block balances area and latency efficiency which are the two areas we focus in our application environment.

64-bit wide RAM blocks are slightly larger than their equally-sized 32-bit wide RAM blocks. However, they are faster with the same column multiplexer. In our case, the 2048 $\times$ 64 RAM block with a column multiplexer of 8 is only slightly faster than the 4096 $\times$ 32 with a column multiplexer of 16. Our choice regarding RAM word length is also based on the fact that most

Table 4.3: Two-port RAM attributes.

Lines	Bits per line	Column mux	Area	Clock-to-data latency	
				Typical case	Worst case
512	64	4	630 μm $\times$ 370 μm	1.7 ns	3.1 ns
1024	32	8	630 μm $\times$ 360 μm	2.0 ns	3.8 ns

Table 4.4: Dual-port RAM attributes.

Lines	Bits per line	Column mux	Area	Clock-to-data latency	
				Typical case	Worst case
512	64	4	630 μm $\times$ 370 μm	1.6 ns	2.9 ns
1024	32	8	630 μm $\times$ 360 μm	1.6 ns	2.9 ns

open-source processors are 32-bits wide instead of 64. If that is not a concern for other application environments, 64-bit wide RAM blocks might be a more attractive choice depending on the other options at the desired size.

Two-port memories have one read and one write port. Dual-port memories have two independent read/write ports. Their pin layout is shown in Figure 4.1(b). Attributes for two RAM blocks of each type are shown in Tables 4.3 and 4.4. The number of possible RAM blocks is considerably smaller for these types since two-port and dual-port memories are restricted to a total size of 72 kbits and 64 kbits respectively. This is compared to 256 kbits for single-port RAMs. Moreover, two-port and dual-port memories are not nearly as area efficient as their respective single-port RAMs. Dual-port memories are almost twice as large compared to single-port memories of the same size, whereas two-port memories are approximately 75 % larger. Both are also considerably more power-consuming than single-port memories. Therefore, since our NoC or application environment has no specific demand for multiple-port memories, we chose single-port memories.

4.3 Switch P&R Results

We performed placement and routing of our design without the adaptive routing modification presented in subsection 2.3.4. Switch p&r results are shown in Table 4.5. Area results for the bar and cross floorplans discussed in section 3.2 are presented in Table 4.6. Results shown are under typical case conditions. Power consumption results are under heavy switching activity. Preferred path latency per switch ranged from 300 to 420 ps. If we also include a 1 mm long wire at the output, approximately twice a RAM block's height, latency increases to 450-550 ps, compared to 80 - 135 ps for straight wires of a similar length without any configuration or routing capability. When there is no contention, non-preferred path latency is one clock cycle. Contention without starvation effects increases non-preferred path latency depending on various factors, but does not affect preferred path latency. Our design functions at 667 MHz under our library's typical case conditions, and at 400 MHz under worst case conditions.

Table 4.5: Switch p&r results (typical).

Implementation library	130 nm		
Power supply	1.2 V		
Clock frequency	667 typical - 400 worst case (MHz)		
Input/Output ports	6		
FIFOs	30		
FIFO lines	2		
Flit width	39 bits		
Pref. path latency/hop	300-420 ps - 450-550 ps incl. 1 mm wire		
	Full switch	Preferred bus	Change
Gates	44874	38865	-13 %
Cells	15001	13369	-11 %
Cell area	195228 μm^2	183056 μm^2	-6 %
Internal nets	13595	12703	-6.5 %
Combinational area	84424 μm^2	72420 μm^2	-14 %
Non-combinational	110798 μm^2	110632 μm^2	-0.1 %
Leakage power	91 μW	85 μW	-7 %
Dynamic power	80 mW	77 mW	-3 %

Table 4.6: Switch area results for the bar and cross floorplans. ”a”, ”b” refer to Figures 3.9 and 3.11.

	Bar floorplan	Overhead	Cross floorplan	Overhead	Change
Full switch	a = 170 μm	13%	a = 130 μm b = 140 μm	18%	+38%
Preferred bus	a = 133 μm	10%	a = 118 μm b = 114 μm	16%	+60%
Change	a: -22%	-23%	a: -9% b: -18.5%	-11%	

FIFOs were implemented with registers due to their number and small size of 2 lines each. Reducing the number of lines from 3 to 2 reduced occupied switch area by approximately 20%. As explained in section 3.1, a FIFO is selectable for serving during the next clock cycle even if a flit is being enqueued in the current clock cycle. This increases the arbiter’s critical path to include the FIFO’s input. We investigated an implementation which regards a FIFO as selectable only when it’s non-empty. This means than flits in non-preferred paths need 2 clock cycles when there is no contention, since they would need to be enqueued in the first clock cycle, and only after that their FIFO can be served during the next (second) clock cycle. Therefore the arbiter’s critical path is reduced to begin only at the FIFO’s head, which is a group of registers. This design was able to function with a clock period of 1.2 ns (833 MHz) under typical conditions, instead of 1.5 ns (667 MHz). However, since non-preferred paths impose a two clock cycle delay, this translates into a 2.4 ns delay for the short critical path implementation compared to 1.5 ns for the long critical path. Therefore, the long critical path implementation imposes a shorter latency.

We deployed a 25 μm wide power ring for each RAM block. Therefore, their effective size is 740.07 μm $\times$ 576.64 μm. In an orthogonal shape, one switch occupies a minimum area of 637 μm $\times$ 310 μm. In the rectangular placement option as explained in section 3.2 and illustrated in Figure 3.9, switch height (a) is 170 μm at minimum. Since we require one switch every 4 RAM blocks (or user blocks of roughly the same size), NoC area overhead is 13%. In the cross placement option as depicted in Figure 3.11, switch height in the X axis (a) is 130 μm, while switch length in the Y axis (b) is 140 μm. In this case, NoC area overhead is 18%. This shows that area efficiency drops in the second case. However, cross-shaped switches have the least possible distance between each other even in the Y axis, therefore minimizing propagation delay between them.

P&r details of the reduced-area single-preferred path switch explained in section 3.1 are also shown in Tables 4.5 and 4.6. In the rectangular placement option, switch height (a) is 133 μm (22% decrease). In the cross placement option, switch height in the X axis (a) is 118 μm (9% decrease), while switch length in the Y axis (b) is 114 μm (18.5% decrease). This imposes a NoC area overhead of 10% in the first case and 16% in the second. These results show that the area gain is small, but in some applications it could outweigh the loss in preferred path flexibility.

Chapter 5

Conclusions

This chapter concludes this master's thesis. In section 5.1 we identify room for future work. Finally, in section 5.2 we summarize this work's key elements.

5.1 Future Work

A number of issues should be addressed in the future. Firstly, our NoC needs to be made fault-tolerant since faults may appear in a chip's lifetime, especially in technologies narrower than 130 nm. This will impose an unavoidably increased area overhead since it will require redundancy or at least error-detection circuits. However, technology trends indicate that designs must be fault-tolerant in some way in order to be trusted for future designs.

Secondly, our NoC needs to be evaluated in a complete system under various workloads and demands. This will enable us to accurately analyze our contribution's impact on a complete system's performance. It will also give us an understanding of how dead flits affect our NoC's performance, and under which configurations and traffic patterns is this impact lessened. A simple metric that can give us this understanding is how many times a flit was forced to wait in a FIFO due to a dead flit in a preferred path. We can also investigate what is the optimal strategy for choosing preferred paths based on the expected traffic pattern in the NoC, and what performance increase does the optimal choice bring. As of the time of writing of this master's thesis, a complete system cycle-accurate simulator is under development, which will enable us to run simulations to answer all the above questions.

Finally, our NoC can face synchronization issues which may result in design limitations. Since preferred paths are purely combinational, flits traversing them can arrive at their destinations and other switches at any point during the clock cycle. Thus, flits may violate flip-flop setup or hold time upon arrival. This may set the flip-flop to a metastable, or at least to an unknown, state and may therefore cause multiple problems.

There are several approaches to combat this issue. Firstly, we could impose a constraint on our preferred paths so that this problem will never occur. This will allow our NoC to function without altering our design, thus without having a decreased area or power efficiency. Depending on the constraint, latency and other performance aspects may be negatively affected. For example, we could limit the number of continuous preferred path hops such that flits will enter a non-preferred path before the end of the clock cycle that they entered their current preferred path in. In this case, assuming that flits are submitted from non-preferred paths in the very be-

ginning of the clock cycle, that number is $[\frac{ClockPeriod}{PrefPathHop+Wire\ Latency}]$. This constraint disallows us to reach very low latency numbers, which we would otherwise be able to.

Furthermore, we could deploy synchronizers at every switch and PE interface logic. While they will not affect preferred paths until flits exit them, their imposed latency in non-preferred paths and preferred path exit points is excessive. Finally, our switch components can easily be implemented asynchronously with known design methodologies. The problem in this approach lies in the necessary handshake between switches or PE network interface logic blocks to guarantee that no flit fragments will be routed through the network. This handshake may be implemented in preferred path entry and exit points instead of simply adjacent switches, but it will still impose a delay that will prevent us from offering our current low per-hop latency, both in preferred and non-preferred paths. Alternatively, we could implement encoding schemes such as dual-rail [36] which also provide this guarantee. However, such implementations will almost double our NoC's required area and also require significantly more power.

Our choice of which option to implement depends on where we are willing to take a penalty. We could either implement the handshake solution and increase our NoC's latency, implement an encoding scheme and increase our NoC's area and power, or we could impose restrictions in preferred paths and lessen our NoC's preferred path flexibility. This choice may differ in different application domains, therefore a different decision may be made for each one.

5.2 Conclusion

We presented a NoC design that offers low latency in pre-configured paths. This latency is close to that of long buffered wires. To achieve our goal, we have resurrected and tailored mad-postman, a technique proposed two decades ago. According to our implementation, an incoming flit is eagerly forwarded to the input's preferred outputs, if any. This is achieved solely by pre-enabled tri-state drivers, and therefore with the least delay possible. Flits are then checked by routing logic to determine if they were correctly eagerly forwarded. If not, flits are forwarded to their correct output. Incorrectly forwarded flits are terminated in later hops as dead. When there is no contention, non-preferred paths impose a single clock cycle per-hop delay.

For routing, we implement XY routing. However, we make the modification that a flit is considered to have been correctly forwarded if it approaches the destination in any of the two axes, even if it does not follow strict XY routing. This way, flits may take different paths and gain from increased preferred path flexibility. A flit is considered dead if its distance from the destination increases in any of the two axes. We have also proposed a modification to allow parts of our NoC to switch to adaptive routing when it's benefits are needed.

Path reconfiguration occurs at run-time. Any user block can transmit configuration packets to any switch in the NoC. Switch architecture resembles that of a buffered crossbar [8]. FIFOs are placed at crosspoints, and each output port has independent arbitration and configuration logic. We presented a 2D mesh topology tailored to our CMP environment to reduce area, power and latency by limiting the number and complexity of the switches.

P&r results show that preferred path latency varies from 300 ps to 550 ps, depending on placement and wire length. Our NoC imposes a 13% area overhead for the whole chip.

We believe that our work provides a different approach in some areas and can form the basis for future NoC implementations which focus on low latency. Our main contribution for low-latency preferred paths can apply in different NoC environments than the one presented, with the

optimizations and assumptions necessary for that environment. Our work provides the means to achieve latency less than one clock cycle per node, which appears to be the current lower limit. While there are open issues left for future work, a substantial number of past NoC research can be applied and therefore provide solutions. Depending on exact application needs, further latency, area or energy optimizations may be made.

Bibliography

[1] C. R. Jesshope, P. R. Miller, and J. T. Yantchev. High performance communications in processor networks. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 150–157, New York, NY, USA, 1989. ACM Press.

[2] Reinaldo A. Bergamaschi and William R. Lee. Designing systems-on-chip using cores. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 420–425, New York, NY, USA, 2000. ACM Press.

[3] B. Luca and D. Giovanni. Networks on chips: A new paradigm for system on chip design, 2002.

[4] Michael Bedford Taylor and Walter Lee. Scalar operand networks. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):145–162, 2005. Member-Saman P. Amarasinghe and Member-Anant Agarwal.

[5] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996.

[6] R. Bevide C. Izu and C Jesshope. Mad-postman: a look-ahead message propagation method for static bidimensional meshes. In *Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing*, pages 117–124. IEEE Computer Society Press, 1994.

[7] Li-Shiuan Peh and William J. Dally. Flit-reservation flow control. In *In Proc. of the 6th Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 73–84, January 2000.

[8] D. Simos M. Katevenis, G. Passas et al. Variable packet size buffered crossbar (cicq) switches. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1090–1096, Paris, France, June 2004.

[9] Wiliam J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 684–689, Las Vegas, NV, June 2001.

[10] Stamatis Vassiliadis and Ioannis Sourdis. Reconfigurable flux networks. In *IEEE International Conference on Field Programmable Technology (FPT)*, December 2006.

[11] Roman Koch Thilo Pionteck and Carsten Albrecht. Applying partial reconfiguration to networks-on-chip. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 155–160, Madrid, Spain, August 2006.

[12] Christophe Bobda, Ali Ahmadinia, Mateusz Majer, Juergen Teich, Sandor P. Fekete, and Jan van der Veen. Dynoc: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *In Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pages 153–158, Tampere, Finland, August 2005.

[13] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[14] William J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Trans. Comput.*, 40(9):1016–1023, 1991.

[15] Feihui Li, Chrysostomos Nicopoulos, Thomas Richardson, Yuan Xie, Vijaykrishnan Narayanan, and Mahmut Kandemir. Design and management of 3d chip multiprocessors using network-in-memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 130–141, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Jongman Kim, Chrysostomos Nicopoulos, and Dongkook Park. A gracefully degrading and energy-efficient modular router architecture for on-chip networks. *SIGARCH Comput. Archit. News*, 34(2):4–15, 2006.

[17] Jongman Kim, Dongkook Park, T. Theocharides, N. Vijaykrishnan, and Chita R. Das. A low latency router supporting adaptivity for on-chip interconnects. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 559–564, New York, NY, USA, 2005. ACM Press.

[18] Robert Mullins, Andrew West, and Simon Moore. Low-latency virtual-channel routers for on-chip networks. *SIGARCH Comput. Archit. News*, 32(2):188, 2004.

[19] Chrysostomos A. Nicopoulos, Dongkook Park, Jongman Kim, N. Vijaykrishnan, Mazin S. Yousif, and Chita R. Das. Vichar: A dynamic virtual channel regulator for network-on-chip routers. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 333–346, Washington, DC, USA, 2006. IEEE Computer Society.

[20] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous noc architecture providing low latency service and its multi-level design framework. In *ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 54–63, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Li Shang, Li-Shiuan Peh, Amit Kumar, and Niraj K. Jha. Thermal modeling, characterization and management of on-chip networks. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–78, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. A technology-aware and energy-oriented topology exploration for on-chip networks. In *DATE '05: Proceedings of the*

conference on Design, Automation and Test in Europe, pages 1238–1243, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Radu Marculescu. Networks-on-chip: The quest for on-chip fault-tolerant communication. In *ISVLSI '03: Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, page 8, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Sam Kerner Tudor Dumitra and Radu Marculescu. Towards on-chip fault-tolerant communication. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 225–232, New York, NY, USA, 2003. ACM Press.

[25] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419, Washington, DC, USA, 2005. IEEE Computer Society.

[26] Cristian Grecu and Michael Jones. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Trans. Comput.*, 54(8):1025–1040, 2005. Student Member-Partha Pratim Pande and Senior Member-Andre Ivanov and Senior Member-Resve Saleh.

[27] 11-Gu Lee, Jin Lee, and Sin-Chong Park. Adaptive routing scheme for noc communication architecture. In *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, pages 1180– 1184. IEEE Communications Society, 2005.

[28] Li Shang, Li-Shiuan Peh, Amit Kumar, and Niraj K. Jha. Thermal modeling, characterization and management of on-chip networks. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–78, Washington, DC, USA, 2004. IEEE Computer Society.

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

[30] Srinivasan Murali, David Atienz, Luca Benini, and Giovanni De Michel. A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 845–848, New York, NY, USA, 2006. ACM Press.

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

[32] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. pages 492–506, 2000.

[33] Manolis Katevenis and Miriam Blatt. Switch design for soft-configurable wsi systems. In *IFIP WG 10.5: Proceedings of the workshop on Wafer Scale Integration*, pages 255–270, Grenoble, France, 1986. Saucier, Trilhe, Eds, North Holland Co.

[34] Michael Taylor. The raw prototype design document, 2000.

[35] M. Horowitz, R. Ho, and K. Mai. The future of wires, 1999.

[36] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. Coping with the variability of combinational logic delays. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*, pages 505–508, Washington, DC, USA, 2004. IEEE Computer Society.