NETWORK-ON-CHIP DESIGN FOR A CHIPLET-BASED WAFERSCALE PROCESSOR

BY

NICHOLAS CEBRY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Associate Professor Rakesh Kumar

ABSTRACT

Motivated by the failing of Moore's law and Dennard scaling, as well as increasingly large parallel tasks like machine learning and big data analysis, processors continue to increase in area and incorporate more computational cores. This growth requires innovation in manufacturing processes to build larger systems, and architectural changes to enable performance to scale acceptably. One significant architectural change is the shift from bus and crossbar based processor interconnections to networks-on-chip (NoCs). This thesis details the design of an NoC to enable a shared memory architecture in a chiplet-based waferscale processor with architectural support for up to 14,336 cores.

For my wife who loves and supports me and my parents who raised me.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my adviser, Rakesh Kumar, for his invaluable support of my research. I would particularly like to thank Professor Kumar for directing me while pursuing this research. His guidance and knowledge motivated and enabled my accomplishment of this thesis. I would also like to thank the Department of Electrical and Computer Engineering in the University of Illinois Grainger College of Engineering for providing me with a position as a teaching assistant.

This research would not have been possible without the collaboration of colleagues at the University of California Los Angeles, particularly Saptadeep Pal and Puneet Gupta. Saptadeep was integral in the development of the project as well as this thesis. I would like to sincerely thank Saptadeep for his input and guidance.

I would also like to thank the members of my research group at the University of Illinois, including Jingyang Liu and Matthew Tomei for their partnership in this project, and the entire Passat research group for their camaraderie the past two years.

My colleagues and mentors from both my academic and professional career so far have been vital to the development of my knowledge and interest in computer architecture. Firstly I would like to thank professor Taskin Padir and Velin Dimitrov with whom I worked in the Robotics and Intelligent Vehicles Research Lab at WPI. I would also like to acknowledge Bryan McLaughlin of Micro-Leads Inc., who welcomed me in an internship role and encouraged me to pursue grad school. I would also like to express my sincere thanks to Arthur Kinzinger, who was a mentor to me throughout my time at Draper.

Additionally, I would like to recognize my wife for her continued love and support. My research at the University of Illinois would not be possible without her commitment and understanding. I appreciate my family, especially my parents, for their lifelong support of my educational aspirations. Finally, I would like to thank my communities of faith for their encouragement, including my friends from University Lutheran Church in Champaign and the Christian Bible Fellowship at WPI.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AHB	Advanced High-Performance Bus
AMBA	Advanced Microcontroller Bus Architecture
AMD	Advanced Micro Devices
APU	Accelerated Processing Unit
BFS	Breadth First Search
CAS	Compare and Swap
CoWoS	Chip on Wafer on Substrate
CPU	Central Processing Unit
DMA	Direct Memory Access
DOR	Dimension Order Routing
EHP	Exascale Heterogeneous Processor
GPU	Graphics Processing Unit
IBM	International Business Machines
JTAG	Joint Test Action Group
MIC	Many Integrated Core
NoC	Network on Chip
PCB	Printed Circuit Board
RTL	Register Transfer Level
SCC	Single-chip Cloud Computer
SoC	System on Chip
Si-IF	Silicon Interconnection Fabric
TSMC	Taiwan Semiconductor Manufacturing Company

CHAPTER 1

INTRODUCTION

In pursuit of continued performance improvements despite the end of Moore's law and Dennard scaling, computer architects have turned to increasing the parallel computation ability of processors by increasing the number of cores. For the last several years, the number of cores in processors has grown steadily. Coupled with the increasing popularity of workloads like machine learning with large amounts of parallel computation, processors are likely to continue to increase in core count and chip size.

A natural direction of this trend is the production of a waferscale processor — that is, a processor with so many cores that it occupies the entirety of the silicon wafer used in the manufacturing process. Typically, many separate processors are manufactured on a silicon wafer, and the wafer is cut up afterwards into the separate processors. The goal of a waferscale processor is to use all the space on a wafer for a single processor with many cores. To date, the commercial production of waferscale processors has not been viable due to imperfections in the manufacturing process leading to yield issues. When a wafer is divided into many processors, this results in some of the processors being nonfunctional. But if the entire wafer is a single processor, the imperfections make the entire wafer nonfunctional. Creating a waferscale processor requires new manufacturing methods to address yield issues, and new architectures to support the large number of cores in the processor.

Several chiplet-based manufacturing techniques have arisen to address the manufacturing issues with large chips. Theses manufacturing techniques break large chips up into a number of smaller chiplets and then connect them together into a large chip. The chiplets are made through the traditional process of manufacturing many devices on a single wafer, and then dicing the wafer. Individual chiplets can be tested for functionality, and functional chiplets can be stitched together with a high-yield interconnection method to create a single large chip.

The rising importance of parallel workloads like neural networks, graph processing, and cloud computing is continuing to push increases in processor core counts and chip sizes. The traditional methods for interconnecting the processors on a chip — buses and crossbars — scale poorly. Contention issues, where nodes have to wait to access the interconnect, lead to limited bandwidth. Additionally, as the number of nodes in these interconnects increases and their area grows, their maximum clock speed shrinks due to the length of the wires increasing.

As a result, architects are shifting away from buses and crossbars as interconnection methods and toward routed networks-on-chip (NoCs) in processors with a large number of cores. NoCs provide better scaling characteristics by allowing more nodes to inject traffic at once and limiting the length of the wires between nodes in the system. The use of NoCs in processors adds new design parameters that must be considered when building large systems. This thesis studies this design process by examining the design of an NoC for enabling shared memory computation in a waferscale processor with architectural support for up to 14,336 cores.

In this thesis, we explore the trend of increasing core counts and processor sizes, discuss the design of NoCs as an enabling technology for large scale manycore processors, and present an implementation of an NoC for a waferscale processor.

In Chapter 2, we discuss the motivation behind the trend toward higher core counts and larger systems. The end of Moore's law and Dennard scaling has caused architects to look for improved performance through increased parallelism. This parallelism also lends itself well to prominent workloads like machine learning, big data, and cloud computing. We also describe some of the manufacturing techniques being developed to support increasing chip size.

In Chapter 3, we present the network-on-chip as an interconnection technology for enabling larger processors, and discuss some of its design parameters. An architect designing a network-on-chip for a processor must consider the topology, routing algorithm, and data transmission mechanisms. Common examples and techniques of each are described.

In Chapter 4, the author's design of a network-on-chip for a shared memory waferscale processor prototype is detailed as an example of what the next generation of large scale processors may look like. The prototype is composed of tiles consisting of 14 ARM Cortex cores connected by a bus. An open source NoC router is used to connect the tiles together. The author designed custom logic for interfacing the bus on the tiles with the routers, enabling globally shared memory among all the cores in the processor.

The work described in this thesis was part of a collaborative project undertaken by the author, students in the NanoCAD Lab at UCLA, and other students in the Passat Research Group at UIUC. The author was the primary developer of the RTL for interfacing between the on-tile bus and the NoC routers, and was also responsible for creation of the basic test programs used to assess functionality of the network. Other students assisted with debugging of the RTL, developed the JTAG interface for programming and debugging the prototype, designed power and clocking infrastructure for the device, and wrote higher level programs that tested the processor's ability to perform useful work like the parallel breadth first search algorithm.

We propose waferscale processing as the next step in building large highly parallel processors. The architecture proposed here, combined with the manufacturing process being employed, demonstrates a promising direction for practical waferscale integration, a decades old ambition in computer architecture.

CHAPTER 2

MANYCORE ARCHITECTURES AND LARGE SCALE SYSTEMS

2.1 Multicore Processor Architectures

Computer architects are constantly searching for ways to improve performance. For many years, compute performance has doubled roughly every 18 months. This rate of improvement was largely explained by Moore's law and Dennard scaling. Moore's law is a prediction that was made in 1965 that the number of transistors in an integrated circuit would double approximately every two years as manufacturing technology improved and transistor sizing and spacing shrank [1]. Dennard scaling is a trend identified in 1974 that the power density of transistors is roughly constant. This means that as transistors reduce in size, the number of transistors that can be used at the same power budget increases. Additionally, smaller transistors reduce propagation delays, allowing for higher frequency of operation. When combined with the observation by Moore about the rate at which transistors shrink, Dennard Scaling predicts that performance per watt doubles approximately every 18 months [2].

While the trends of Moore's law and Dennard scaling held, architects used the additional transistors to add extra features and make deeper pipelines, while running their chips at higher clock speeds. However, as transistors have continued to become smaller, leakage current has overcome switching power as the performance limit. Reducing transistor size does not reduce energy consumption (and heat generation) as much. This has caused the breakdown of Dennard scaling, and slowed the rate of performance improvement [3].

In an attempt to maintain regular performance gains in spite of the breakdown of Dennard scaling, architects turned their focus from increasing single core performance to increasing parallel performance [4]. Although clock speeds have stalled, increasing the number of compute units allows performance to keep improving, particularly in parallelizable workloads. Instead of using additional transistors to improve the speed at which a single core executes instructions, additional transistors are now being dedicated to additional cores, which programmers can take advantage of by writing parallel programs [5], [6].

The Piranha system was a prototype that explored the idea of trading compute core complexity for quantity, citing the abundance of thread-level parallelism in commercial workloads as a motivator [7]. Published in 2000, it was one of the first systems to be designed with scalability of core count and number of chips in mind. IBM released the first commercially available multicore processor, the Power4, about a year after the Pirnaha paper was published [8]. Consumer manufacturers Intel and AMD followed suit with their first multicore processors in 2005 [9], [10]. Today, processors in all domains from low power microcontrollers, to consumer electronics like phones, tablets, laptops, and desktops, and all the way up to server processors and supercomputers use multicore processors.

2.2 Manycore Processor Architectures

The shift from uniprocessors to multicore processors provided clear performance benefits, increasing computational throughput despite the relative stagnation of clock speeds. As a result, architects have continued to work in this direction, moving from multicore architectures with a few very powerful cores and specialized interconnections to manycore architectures with a vast number of cores connected in a regular and scalable fashion. Processor designs continue to grow in parallelism and number of cores [11]. Manycore systems have proved especially valuable for the trends of machine learning and big data processing. These operations are easily decomposed into many threads that can be run simultaneously by a large number of cores [12].

Architects are continuing to push the number of cores in a system, particularly for datacenter and supercomputing applications. Intel's single-chip cloud computer (SCC) and many integrated core architecture (MIC) are examples of the trend to increase the number of cores on a chip [13], [14]. Intel's SCC aimed to shrink data centers and improve their performance by condensing the resources of multiple servers onto a single chip. It took inspiration from data center clusters, connecting a series of compute tiles with a network like nodes in a server farm. The MIC architecture took a slightly different approach, aiming to increase core counts while focusing on maintaining compatibility with programming paradigms used in multicore chips. It implemented a shared memory model between a large number of cores on an accelerator card aimed to replace a GPU in scientific or compute applications.

The trend of growing chip sizes and core counts can also be seen in GPUs and high-performance computing. The exascale APU from AMD and MCM-GPU from NVIDIA are both examples of projects aimed at growing core counts [15], [16]. The exascale APU proposes the tight integration of a CPU and GPU into a single compute module for improved performance and power efficiency. The MCM-GPU is a method for coupling multiple GPU modules into a single larger device to keep up with the demand for increased performance scaling. Taking the idea one step farther is the proposed waferscale GPU, which uses the same GPU module idea as the MCM-GPU. However, instead of combining GPU modules by 3D integration in a package, many GPU modules are integrated on a single silicon wafer [17].

Especially in high performance computing, designs are moving from more connected computers or servers to much larger processors. Larger processors benefit parallelism by providing lower communication energy and increased compute per volume. Because the distance between cores is reduced when there are more cores present on a single chip (as compared to cores on separate chips or even in separate computers), less energy must be spent transmitting information between the cores. Larger chips also reduce the computer volume required to achieve the same number of cores in a system. The area cost of increasing the chip size to support more cores is much lower than the cost of adding more chips.

This trend is headed toward waferscale processors, which are processors the size of the entire silicon wafer used in the manufacturing process. Waferscale integration was first pursued in 1980 by Gene Amdahl's Trilogy Systems [18]. Within the next four years, at least two other companies, Mosaic Systems Inc. and Waferscale Integration Inc., had also started work on waferscale processors [19]. However, none of these companies ever released a waferscale processor product, as yield issues in the manufacturing process prevented them from creating one in a commercially viable manner. However, recent innovations in manufacturing techniques have shown that scaling chips up to the size of a wafer may now be viable.

To support this ever-increasing number of cores and size of chips, architectural changes must be made. Traditionally, the interface between cores and memory has been implemented by a bus, where a single node broadcasts messages to all other nodes. As core count and chip size grow, the bus becomes increasingly impractical. As the size of the bus grows, the energy necessary to drive it increases, and the maximum speed at which it can be run decreases. Additionally, as core count increases, the rate at which data must be retrieved from memory in order to keep the cores from stalling must increase. The bus becomes a bottleneck limiting memory bandwidth. As a result, networks-on-chip are replacing buses as the mechanism for interconnecting the cores in a processor. Networks allow multiple nodes to send messages simultaneously, reduce the maximum distance information must travel in a single hop, and allow for increased memory bandwidth by splitting the memory into multiple nodes.

2.3 Building Large Scale Processors

In addition to new architectural techniques, large scale processors also require new manufacturing techniques, as the physical size of the silicon chip grows with the number of cores. Interest in constructing processors the size of a silicon wafer has existed for decades. However, manufacturing issues have prevented successful commercialization thus far due to yield limitations [20]. New manufacturing techniques aim to solve these issues through chiplet-based designs with interposers and/or redundancy.

Chip-on-wafer-on-substrate (CoWoS) is an interposer-based approach to multi-chip modules which enables the construction of larger chips using micro-bumps and through-silicon vias to connect multiple smaller chips to the larger interposer [21]. The embedded multi-die interconnect bridge (EMIB) is also an interposer-based technology for the creation of large processors. The EMIB aims to increase interconnect wire density through the creation of small fine-pitch bridges that can be used to connect chips [22]. Researchers have also developed techniques for working with faulty silicon, such as routing the network connecting individual processing units around faulty cores

[23]. The Cerebras processor uses redundant cores to account for manufacturing errors at large scale, allowing a percentage of the cores on the chip to be nonfunctional without causing the whole chip to fail [24].

Another emerging technology is the silicon interconnection fabric, or Si-IF. Si-IF is an interposer technology like TSMC's CoWoS, but instead of microbumps it uses solderless thermal compression bonding of copper pillars to attach the smaller dies to the wafer [25]. This provides more mechanical rigidity and easier scaling to larger sizes than other interposer approaches [26]. The Si-IF approach can improve performance up to 20x compared to dies in separate packages on a PCB, and can approach the performance of single die interconnections [25].

CHAPTER 3

NETWORKS-ON-CHIP FOR MANYCORE PROCESSORS

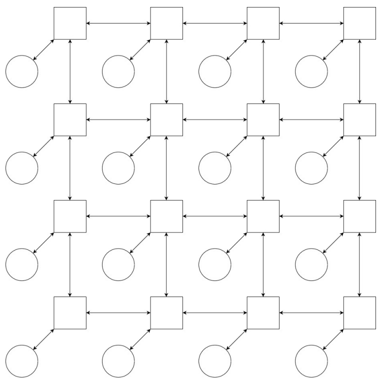
3.1 Network-on-Chip Design

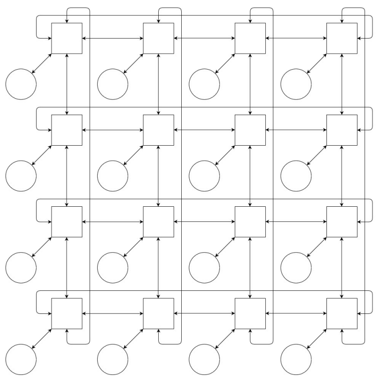
As the number of cores on a chip increases, the effects of the interconnection method on system design and performance increase [27]. When the number of cores in a system exceeds a certain point, it becomes infeasible to use a bus or crossbar to communicate with the memory subsystem. Cores spend too much time waiting to be granted control of the bus, and the larger the bus gets the more slowly it must be run due to propagation delays and increased capacitance. Networks-on-chip, or NoCs, are one way to replace buses in manycore architectures [28]. The use of NoCs introduces a whole set of design parameters and tradeoffs that are not present when systems utilize buses. Designers must now make decisions about network topology, routing algorithms, and data transmission mechanisms.

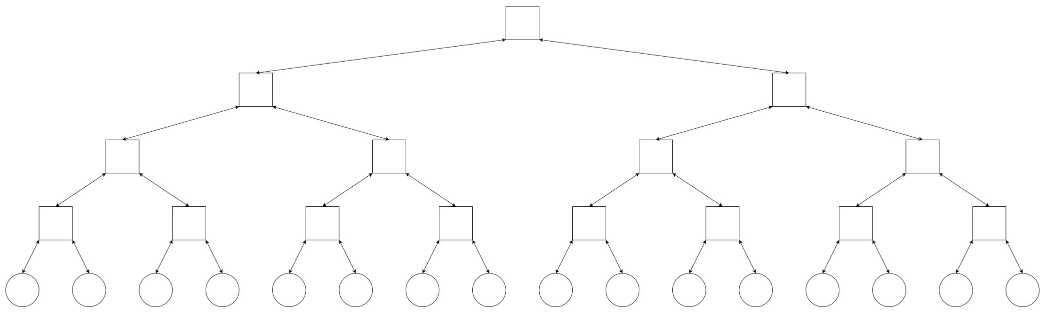
3.2 Network Topology

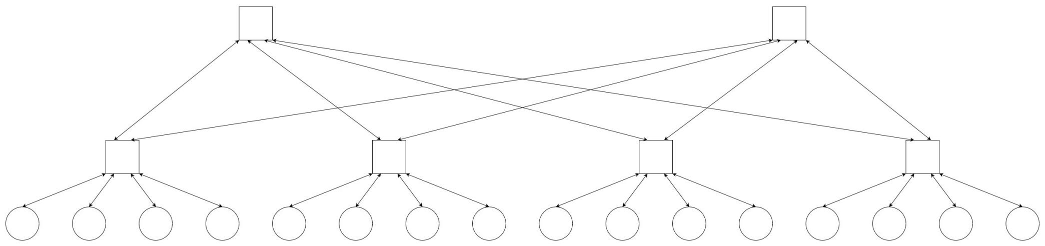
The first major design decision for a network-on-chip is the topology — that is, determining the connection pattern between different nodes in the network. Common topologies include mesh, torus, binary tree, and butterfly tree. Topologies differ in the number of routing units required per compute unit, the connection pattern between routing units, and the minimum, average, and maximum number of connections or hops between any two compute units. Diagrams of these network topologies appear in Figure 3.1.

In a mesh, the number of compute and routing units is equal, with each compute unit paired with a single routing unit. Routing units are designed with five ports: one for the connection to the processing unit and four for


(a) mesh topology


(b) torus topology


(c) binary tree topology


(d) butterfly tree topology

◯ = Compute Units □ = Routing Units

Figure 3.1: Common NoC Topologies

connections to other routing units, one in each cardinal direction. The compute and routing unit pairs are arranged in a grid, with each routing unit connecting to up to four neighboring routing units. Routing units on the edge of the grid have one or more unconnected ports. In this configuration, the minimum number of hops between two compute units is three, and the maximum number of hops is the width of the grid, plus the height of the grid, plus two.

A torus is also similar in design to a standard mesh, but with additional routes connecting the edges of the mesh to one another. Routers on the top edge are directly connected to routers on the bottom edge, and routers on the left edge are directly connected to routers on the right edge. This reduces the maximum number of hops that separate two nodes. In this configuration, the largest number of hops it takes to get between two compute nodes is one half of the width of the grid, plus one half of the height of the grid, plus two. However, the torus is much more difficult to manufacture, because the connections between the edges become long traces that are difficult to route. Additionally, the length of these traces limits the maximum clock speed.

Binary tree topologies consist of routers which have only three ports. Each router has a single parent and two children. The compute units form the leaves of the tree, and the routing units form all of the intermediate nodes in the tree structure. For N compute nodes, this topology requires $N - 1$ routing units. The minimum number of hops is two, between compute units connected to the same router. The maximum number of hops is $2\log_2(N)$. This topology could be beneficial in cases where traffic is known to be very regular, and high traffic paths can be split onto different branches of the tree where they will not cause congestion by sharing a router. Additionally, because each router has fewer ports, the required density of interconnect wires between routers is lower.

Butterfly tree topologies are similar to binary tree topologies, but each node has more children, and there are cross-connections between the branches. In the butterfly tree network, routing units have four ports for connecting to child nodes, and two ports for connecting to parent nodes. The compute nodes are the leaves, and are connected in groups of four to a single router. Each group of four routers on one level are connected by two different routers on the level above. For N compute nodes, this topology uses $\frac{N}{2} - 2$ routers. The tree will have $\log_2(N) - 2$ layers of routers. This means that the maximum number of hops between any two nodes is $2\left[\log_2(N) - 2\right]$. This topology therefore requires fewer routers than the binary tree, and has a lower maximum hop count, at the cost of more difficult routing algorithms. This also requires more complicated physical routing to connect all the routers on the chip.

3.3 Routing Algorithms

After the network topology has been selected, the routing algorithm to use must be determined. Applicable algorithms are governed, and in some cases completely determined, by the topology chosen. When a binary tree is used, there is only a single possible path between any two nodes, so network performance can only be affected by router arbitration scheme, as there is no variation in path selection. For mesh, torus, and butterfly tree topologies, there are multiple paths between points in the network, and routing algorithms must have some way to pick between the different paths available.

For mesh and torus networks, the simplest routing algorithm is dimension order routing (DOR). In dimension order routing, each router is assigned coordinates for its location in the mesh in a standard (X,Y) manner. Packets are routed between the source and destination with two straight lines, traveling to the appropriate dimension in first one coordinate, then the other. In XY DOR, the packet first travels to the appropriate column, and then the appropriate row. The opposite is true for YX DOR. Strict dimension order routing utilizes only a single path for traffic going between two nodes in a given direction.

Variations on dimension order routing can provide mechanisms for managing network congestion. Packets can be switched between XY and YX routing schemes midway while still traveling a path with a minimum number of hops. Additionally, if packets are allowed to take a non-minimum length route, even more paths between any two nodes can be considered. However, in this case algorithms must be carefully designed to ensure overall forward progress, as packets must be allowed to move farther away from their destination in order to travel on non-minimum length paths.

A basic binary tree algorithm assigns each compute node a single address value, with each layer in the tree corresponding to a different bit of the address. The top router has all compute nodes whose addresses begin with a zero on one branch, and all the compute nodes whose addresses have a one as the most significant bit on the other branch. The routers on the next layer down have a similar property, but for the second most significant bit. Packets leaving a compute node are routed up the tree until they reach a point where the most significant bits of their destination match the router, then they begin traveling back down. The path down the tree is effectively a binary search for the destination address.

Butterfly tree networks will use routing algorithms similar to those of binary tree networks, except that each hop between routers has two different potential paths, because each node is connected to two different nodes on the layer above it. Nodes must have some arbitration mechanism for selecting which path they will use to send their packets. Options include round-robin, priority, and congestion-based. In round-robin arbitration, nodes simply cycle through the different options each time they send a packet. The priority method entails nodes defaulting to one option every time, and only changing if they are unable to send a message due to congestion. Congestion-based methods require more advanced flow control schemes, in which the destination routers have some way of indicating how busy they are, allowing source routers to select the less busy destination option.

3.4 Data Transmission

The final design area for a network-on-chip is data transmission mechanisms. Each network packet will consist of some number of bits of information, and the designer must determine how the physical channels will transmit these bits. Important design choices are the physical width of the channel, and the number of virtual channels to support.

The physical width of the channel refers to how many bits of information move between two network nodes in parallel. If this number is less than the number of bits of information in a packet, the packet must be broken up into smaller transmission units known as flits. The use of flits can reduce the required number of wires between two nodes, but adds additional control overhead and latency. At a minimum, to achieve proper reassembly at the destination, each flit must indicate whether it is the start, middle, or end of a packet. Routing algorithms for NoCs generally restrict all flits in a packet to traveling the same path through the network, so the destination does not have to handle out-of-order flit arrival. Routing nodes must also be designed carefully to keep the flits of a packet together. If a router is transmitting flits from one packet to its neighbor, and a new packet arrives with the same destination, its arbitration scheme must be sure to send all the flits from the first packet before sending any flits from the new packet.

If the network uses multiple flits for at least some packets, the designer may also choose to implement virtual channels. Virtual channels allow flits from different packets to be interleaved on the same physical channel. All flits from a single packet travel between two nodes on the same virtual channel, in order relative to the other flits in their packet. But the physical link between two routers can change which virtual channel it is servicing mid-packet. Assigning different priorities to the virtual channels can allow more important packets to interrupt the transmission of less important packets and overtake them. Virtual channels increase router complexity by requiring the routers to keep track of which virtual channel buffered messages belong to, either by having separate buffers for each virtual channel or doing extra bookkeeping on a single set of buffers.

CHAPTER 4

NETWORK DESIGN FOR THE WAFERSCALE SYSTEM

4.1 Overview

To explore the impacts of a network-on-chip approach to the interconnection of cores in a processor and demonstrate the feasibility of an ultra-large-scale device, we designed the waferscale processor. Additionally, we wanted to answer the question of whether any aspects of network-on-chip design must be re-thought when increasing the size of the chip to the scale of a wafer. The processor we designed is intended to occupy an entire silicon wafer and has architectural support for 14,336 cores. It is made up of a series of small tiles which are bonded onto a full wafer interconnection device using copper pillar bonding. Each tile has an ARM AMBA AHB bus for intra-tile communication, and a network-on-chip approach is used to connect the different tiles together.

The waferscale processor is composed of a two-dimensional array of tiles. The tiles are connected using a mesh topology, with each tile serving as a single routing node in the mesh. Physical channel width in the network is equal to packet size, so no flits are necessary and no virtual channels are used. Each of the tiles in the system has 14 ARM Cortex M3 cores, a series of memory banks, and routers for the on-chip network. Each core in a tile has a 64kB private instruction and data memory bank. Tiles also have five 128kB memory banks that are shared among all the cores. Four of these banks belong to the shared memory region accessible by every core on the wafer, making a total of 512kB of system shared memory per tile. The fifth bank is called the bookkeeping bank and is used for holding return values of memory requests made to other tiles in the system. This bank is only directly accessible by cores on the same tile. An AMBA AHB bus is used to connect all the components on the tile. The bus has 16 masters: the 14 ARM cores

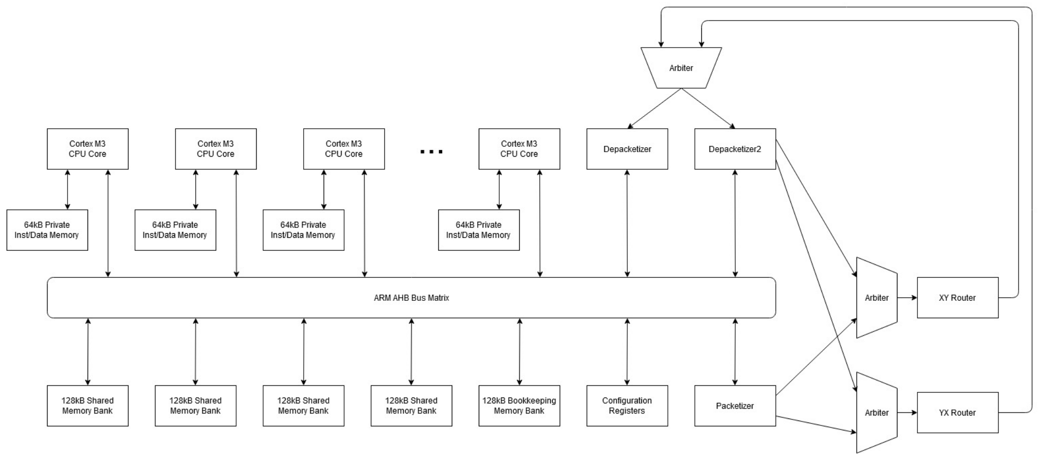


Figure 4.1: Block Diagram of a Single Tile

and two custom components for interfacing with the routers. There are seven slaves on the bus: the five memory banks, a set of registers for configuring various functions on the tile, and another custom component for interfacing with the routers. A block diagram of a single tile appears in Figure 4.1.

In the waferscale processor, we use two networks for tile interconnection. Both are mesh networks using dimension order routing. One of the networks performs X then Y dimension order routing, and the other network performs Y then X dimension order routing. A mesh network was chosen for ease of implementation and straightforward routing of wires on the interconnection wafer. Having two networks provides redundancy, reducing the effect of a tile being nonfunctional. Additionally, the traffic can be distributed between the two networks to reduce congestion. The redundancy motivation for using two networks to serve the same function is unique to particularly large chips like the waferscale processor. A smaller chip might achieve acceptable yields without having to implement a redundant network. And the fact that smaller chips do not use chiplet-based designs means that there is no worry about potential manufacturing issues in the mechanism connecting the chips together.

The system supports up to 128MB of shared memory that can be accessed by any core. Each tile holds 512kB, and the waferscale processor can contain up to 1024 tiles. The 128MB of memory occupy a 29-bit address space, with 10 bits used to indicate which tile the memory resides on, and 19 bits indicating the memory address on that tile. The 10 bits for tile location are further divided into two 5-bit fields, indicating the X and Y coordinate of

Table 4.1: Remote Write Address Ranges

	start address	end address
remote write over XY network	32'h80000000	32'h9FFFFFFF
remote write over YX network	32'hA0000000	32'hBFFFFFFF

Table 4.2: Memory Address Mapping for Remote Writes

bit	31..30	29	28..24	23..19	18..0
value	2'b10	network	dest x	dest y	memory address

the tile in the mesh network. Therefore, the maximum dimensions of the tile grid are 32x32.

4.2 Inter-Tile Communication

When a core needs to write to the shared memory, it performs a store operation on an appropriate address. The Cortex M3 has a 32-bit address space, and a contiguous 256MB section of this address space is reserved for performing writes to the shared memory. This 256MB section is divided in half, with half of the addresses used to perform writes over the XY network, and the other half used to perform writes over the YX network. The store operation is put onto the tile's AHB bus matrix, where it is routed to one of the pieces of custom logic for interfacing with the router, the packetizer. The packetizer takes the AHB bus message and reformats it as a network packet which it sends to either the XY or YX router on the tile, as indicated by the address used in the write. The packet travels through the network until it reaches the destination tile. The router at the destination then sends the message to another of the pieces of custom logic on the tile, the depacketizer. The depacketizer reformats the network message as an AHB write to the appropriate location in the memory on that tile. Tables 4.1 and 4.2 show the address ranges and memory mapping for writes to shared memory.

Reading from and performing CAS operations on shared memory requires a more complicated process. Because it would be impractical for a core to maintain ownership of its tile's bus matrix until it receives a response from memory on a tile that could be very far away, all reads and CAS operations are performed using a DMA-like method. To perform one of these operations,

Table 4.3: Memory Address Mapping for Bookkeeping Memory

bit	31..20	19..17	16..13	12..3	2	1..0
value	12'h200	3'b100	core	bucket	flag/data	2'b00

a core must execute a series of writes to a set of registers inside the packetizer. These registers indicate the address that the operation is to be performed on, a location in the bookkeeping memory to store the results of the operation, and the compare and swap values for CAS operations. This mechanism was particularly motivated by the scale of the processor. Because the maximum latency of a memory request increases with the size of the network/chip, it is unreasonable for a core in the waferscale processor to hold the bus for the entire duration of a request. In a smaller processor, however, it may be reasonable for a core to have exclusive access for the entire duration of a request, because the latency is smaller.

The 128kB of bookkeeping memory is divided up among the cores on the tile, so that requests from different cores will not interfere with one another. The bookkeeping memory has 17 bits of address space. Four bits are used as an indication of which core the section belongs to. Two of the bits are reserved for keeping memory accesses word aligned. The remaining memory is divided into logical buckets. A bucket consists of two memory words; one is for holding the return data from a request, and the other is a valid flag. One bit of the address is used for indicating whether a location is a data or flag address. The remaining 10 bits of address space indicate the index of the bucket. This allows each core to have up to 1024 requests in flight at once. Table 4.3 shows the address mapping for the bookkeeping memory.

We designed the packetizer so that each core has its own set of registers for configuring memory requests. This allows writes to the configuration registers from different cores to be interleaved while still producing correct operation. There are three registers for each core, with each register accessible from two different addresses. The three registers indicate the address to be accessed (with a similar mapping scheme to remote writes), the index of the bucket in the bookkeeping memory to put the response in, and the compare/swap values for CAS operations. There is a single 32-bit register for both the compare and swap values, and CAS operations are limited to 16-bit values. The two different addresses for each register are used to indicate

Table 4.4: Packetizer Register Address Mapping

bit	31..12	11..9	8	7..6	5..2	1..0
value	20'h60000	3'b000	send	register	core	2'b00

Table 4.5: Packetizer Register Address Ranges

	start address	end address
data registers, message not sent	32'h60000000	32'h6000003F
address registers, message not sent	32'h60000040	32'h6000007F
bucket registers, message not sent	32'h60000080	32'h600000BF
reserved	32'h600000C0	32'h600000FF
data registers, send message	32'h60000100	32'h6000013F
address registers, send message	32'h60000140	32'h6000017F
bucket registers, send message	32'h60000180	32'h600001BF
reserved	32'h600001C0	32'h600001FF

when the request configuration is complete and a network message should be generated. One address is used for modifying the register without generating a message, and the other is used for assigning the value and then generating a message on the network. This allows for more efficient operation if the core does not need to modify the values in every register between two requests. Tables 4.4, 4.5, and 4.6 indicate the memory address mapping scheme for the registers, the address ranges of the registers, and the mapping for the values written to the address configuration register.

After the core has set the register values, the packetizer generates a network message and sends it to the appropriate router on the tile. This message travels through the network to the destination tile, where it is received by the final piece of router interface logic, called the depacketizer2. The depacketizer2 will then use the bus matrix to perform the read or CAS operation, and generate a response message which it sends to the router on its tile. This response message will travel back through the network to the initial tile, where it is processed by the depacketizer, which writes the data into the indicated location in the bookkeeping memory then sets the bucket's valid flag.

Table 4.6: Address Register Memory Map

bit	31	30	29	28..24	23..19	18..0
value	1'b0	CAS/read	network	dest x	dest y	memory address

Read and CAS operations use both networks to complete their tasks. The response messages are sent on the opposite network from the request message. This means that responses travel through the same set of tiles as their requests. This is important for having the redundant networks increase reliability of the system. If a tile is nonfunctional, it can break one of the two paths that exist between any pair of tiles. If the request and response were on the same network, it would require both paths to be functional for a read or CAS operation to complete.

4.2.1 Network Message Format

We designed the network to carry monolithic packets containing all the required data and address information in a single flit. The routers use a simple handshaking protocol with the transmitting router driving a valid signal, and the receiving router driving a ready signal. The entire packet is transmitted in parallel. The transmitting router puts the packet on the data lines and asserts the valid signal. It will hold the packet and valid signal until the receiving router asserts the ready signal. Packets in the system contain 99 bits of combined address and data information. The first three bits indicate the size of the data to be put on the AHB bus in bytes. The next 32 bits are only used for CAS messages and hold the two 16-bit values used in the CAS operation (compare value and swap value). The next 32 bits have a different purpose for each of the message types. For write messages, they contain the data to be written. For read and CAS messages, they contain the bucket index that the response is to be written into. For read response messages, they contain the data that was read out of memory. For CAS response messages, they indicate whether the CAS operation succeeded or failed. Next is a single bit indicating which network (XY or YX) the message will travel on. This is used by the depacketizer2 for generating response messages on the opposite network. After the network bit are two bits indicating packet type. The next 19 bits hold the address inside the destination tile's memory banks that the operation is accessing. The final 10 bits hold the X and Y coordinates of the destination tile. The upper five hold the Y coordinate, and the lower five hold the X coordinate. Table 4.7 shows a breakdown of the message format.

Table 4.7: Packet Format

bit	98..96	96..64	63..32	31	30..29	28..10	9..5	4..0
value	size	CAS data	pkt data	ntwk	type	mem addr	dest y	dest x

4.2.2 Packetizer Design

The packetizer is responsible for generating three of the four types of network messages the system uses: write messages, read request messages, and CAS request messages. It is a slave on the AHB bus matrix and sends messages to the routers to leave the tile. The bus matrix uses a two-phase approach to transmit messages, where the first phase is used to send the address and control signals for a message, and the second phase contains the message data. These phases are permitted to overlap, so the address phase of a message can occur concurrently with the data phase of the previous message. The routers use only a single phase to transmit messages, where all the information is transmitted at the same time. The packetizer control logic is governed by a state machine shown in Figure 4.2 with four states: an initial state, a state for generating write messages, a state for writing data to the request configuration registers, and a state for generating request messages.

State transitions in the packetizer are governed by messages coming from the bus matrix and by the availability of the router. Bus matrix messages cause the packetizer to enter either the write state or the register store state, depending on the destination address of the message. In the write state, the packetizer presents a message to the router, and waits for the router to indicate it has accepted the message. Once the router has accepted the message, the packetizer returns to the initial state if no AHB transactions are ready. If there is an AHB transaction waiting, the packetizer moves directly to the register store state, or back into the write state. In the register store state, the packetizer stores data from the bus matrix into one of the request configuration registers. If the address used to access the register was one of the designated transmission addresses, the packetizer moves into the request generation state. Otherwise, the packetizer can move back to the initial state, move to the write transmission state, or stay in the register store state, depending on the presence of an AHB transaction. In the request transmission state, the packetizer presents a message to the router, and waits for the router to indicate receipt of the message. When the router indicates it

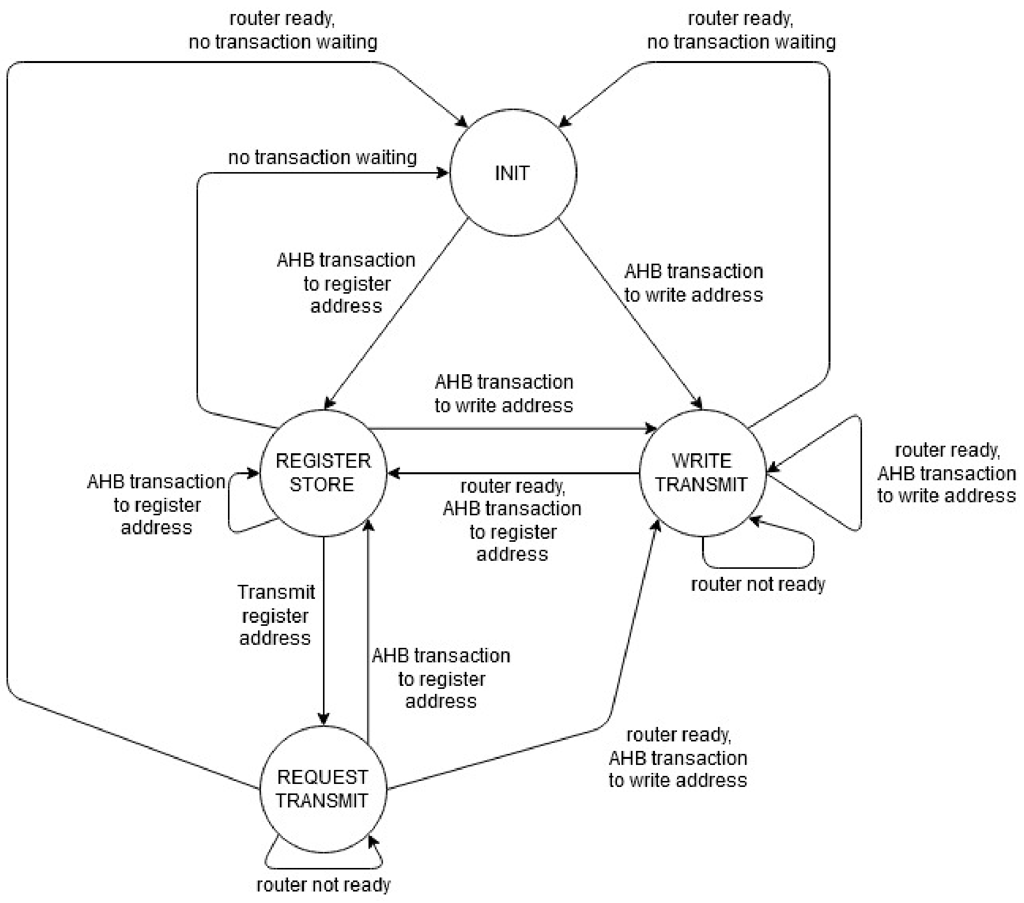


Figure 4.2: Packetizer State Diagram

has accepted the message, the state will transition to either the initial state, the register store state, or the write state, based on the presence of an AHB bus message.

4.2.3 Depacketizer Design

The depacketizer is responsible for receiving two types of network messages: writes and responses. It is a master on the AHB bus matrix and generates messages to write data into the memory banks. Write messages have their data written into one of the shared memory banks, according to the address of the write. Response messages have their data written into the bookkeeping memory bank, in the core and bucket location indicated when the request was generated. Writes to the bookkeeping memory are made up of two transactions, one to write the relevant data, and another to set the valid flag. The depacketizer control logic is based on a state machine shown in Figure 4.3 with six states: an initial state, a state for putting the address of a write on the bus matrix, a state for putting the data of a write on the bus matrix, a state for putting the address of a response bucket on the bus matrix, a state for putting the data of the response and address of the valid flag on the bus matrix, and a state for putting the data for the valid flag on the bus matrix.

State transitions in the depacketizer are governed by messages coming into the tile from the router and by the availability of the bus matrix. The write address state is entered when a write message comes in from the router. In this state, the depacketizer puts the address of the shared memory location it needs to write to on the bus and waits for the bus to give a ready signal. When the bus indicates ready, it puts the data from the write on the bus, and again waits for a ready signal. When the ready signal is received, it will either return to the initial state, go to the write address state, or go to the first response address state, depending on the status of the router. The first response address state is entered when a response message is received from the router. In this state, the address of the data bucket in the bookkeeping memory that is to be written to is placed on the bus, and the depacketizer waits for the bus to give a ready signal. When the ready signal is received, the depacketizer moves to the next state, where it puts the data of the response

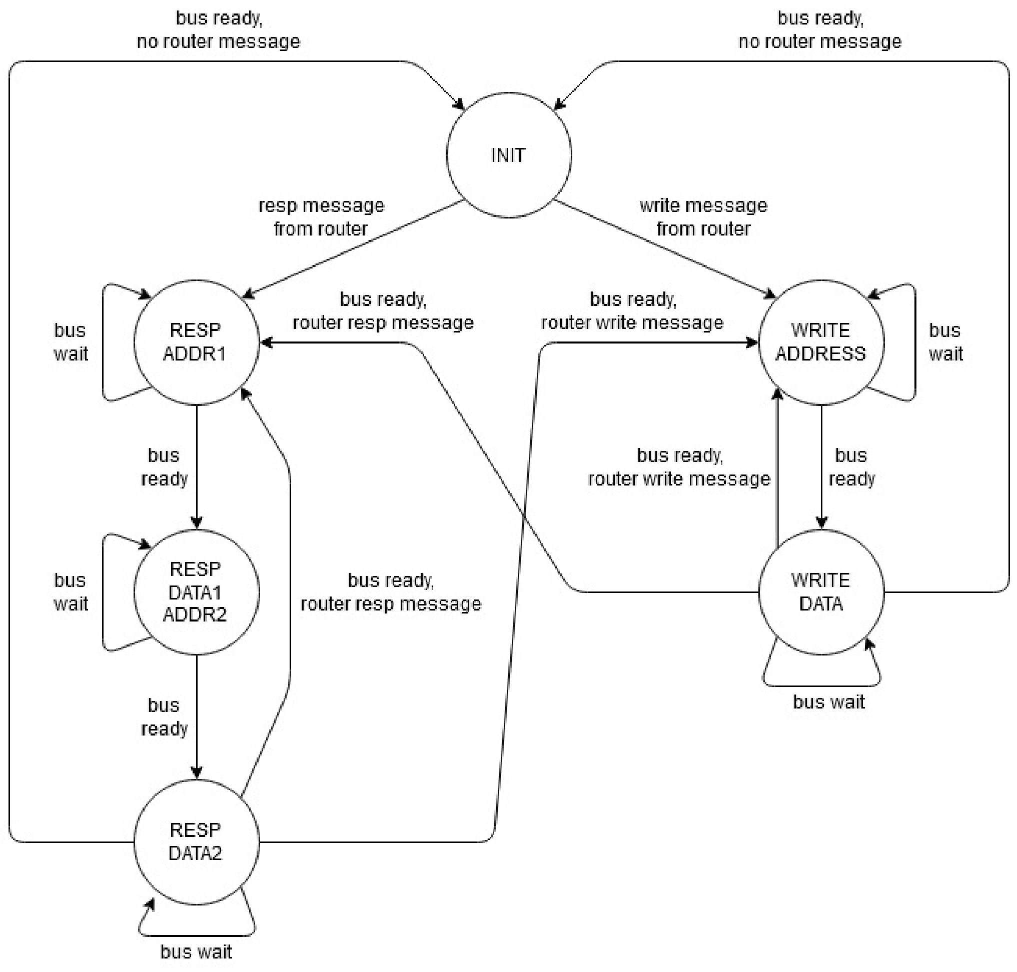


Figure 4.3: Depacketizer State Diagram

and the address of the valid flag for the bucket on the bus, completing the write of the data and beginning the write of the valid flag. Once the bus indicates it is ready, the depacketizer moves to the second response data state, where it puts the data for the valid flag on the bus, and again waits for the bus's ready signal. Once the bus indicates it is ready, the depacketizer will either return to the initial state, go to the write address state, or go to the first response address state, depending on the status of the router.

4.2.4 Depacketizer2 Design

The depacketizer2 is responsible for receiving read and CAS request messages from the router and generating response messages to send to the router. It is an AHB bus master so that it can read data out of the shared memory banks and write new data into them when performing CAS operations. The depacketizer2 uses the ARM bus matrix's master lock signal to prevent any other memory operations between its read and write transactions. The depacketizer2 logic is governed by a state machine shown in Figure 4.4 with eight states: an initial state, a state for putting the address of a CAS operation on the bus matrix, a state for performing the compare in a CAS operation, a state for writing the new data in a CAS operation, a state for sending the CAS return message to the router, a state for putting the address of a read operation on the bus matrix, a state for sending the data from a read operation to the router, and a state for waiting for the router to be ready to accept a response message.

State transitions in the depacketizer2 depend on messages coming into the tile from the router, availability of the bus matrix, and availability of the router to accept outgoing messages. An incoming read request from the router will put the depacketizer2 into the read address state, where it puts the address to be read onto the bus. It stays in this state until it receives an acknowledgement from the bus, at which point the depacketizer2 enters the read data state. It waits in this state for the bus to respond with the data requested. If the router is not ready to accept a new outgoing message when the bus responds with the data, then the router wait state is entered. The depacketizer2 remains in the router wait state until it receives a ready signal from the router, at which point it enters either the initial state, the

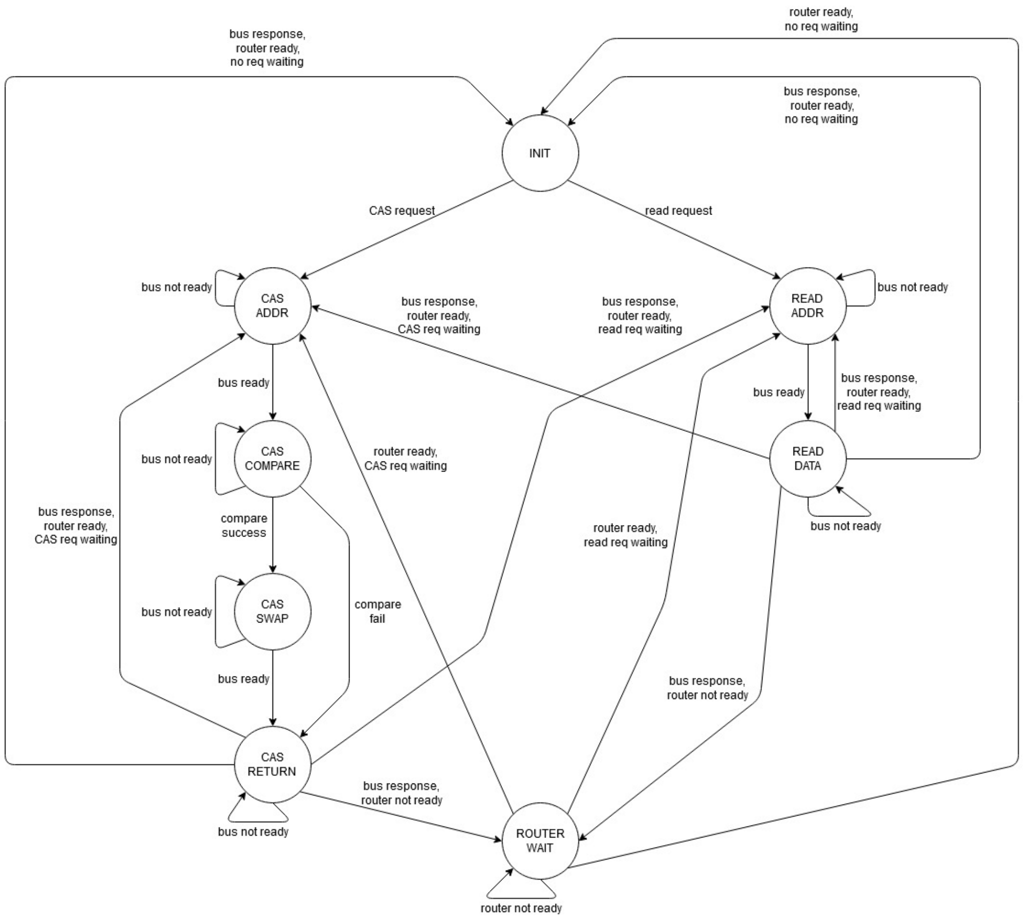


Figure 4.4: Depacketizer2 State Diagram

read address state, or the CAS address state, depending on the presence and type of a new incoming request. If the router is ready when the bus responds with the data, then the transition to the initial, read address, or CAS address state happens directly from the read data state. When the depacketizer2 gets a CAS request message, it enters the CAS address state, where it puts the address on the bus and waits for the bus to acknowledge it. Once the address has been acknowledged, the depacketizer2 waits in the CAS compare state until the bus responds with the data. When it receives the data, it transitions to either the CAS swap state or the CAS return state, depending on the result of the compare. In the CAS swap state, the depacketizer2 puts the address of the data to be modified on the bus and waits for the bus to acknowledge it. On acknowledgement, it moves to the CAS return state, where it puts the new data on the bus and again waits for acknowledgement from the bus. If the router is not ready to accept an outgoing message when the bus acknowledges the new data, the depacketizer2 enters the router wait state, where it stays until the router is ready to accept an outgoing message. If the router is ready to accept an outgoing message when the bus acknowledges the write, then the depacketizer2 moves from the CAS return state to either the initial, read address, or CAS address states, depending on presence and type of a new incoming request from the router.

4.3 Intra-Tile Message Routing

Because each tile has two routers on different networks, and three different entities that can source or sink messages, we designed additional modules for routing messages inside of the tiles. In order to improve performance, messages generated by the packetizer and depacketizer2 with a destination of the tile they are generated on are not actually sent to the routers, but instead looped back locally. The network is also susceptible to deadlock, due to the depacketizer2's interaction with both incoming and outgoing messages. It is possible for the depacketizer2 to be unable to send a message because the network is full, and for the network to be stuck in a full state because there are request messages waiting for the depacketizer2 to process them blocking progress. To reduce the likelihood of this deadlock occurring, we added queues to the tiles, allowing them to store incoming messages in the

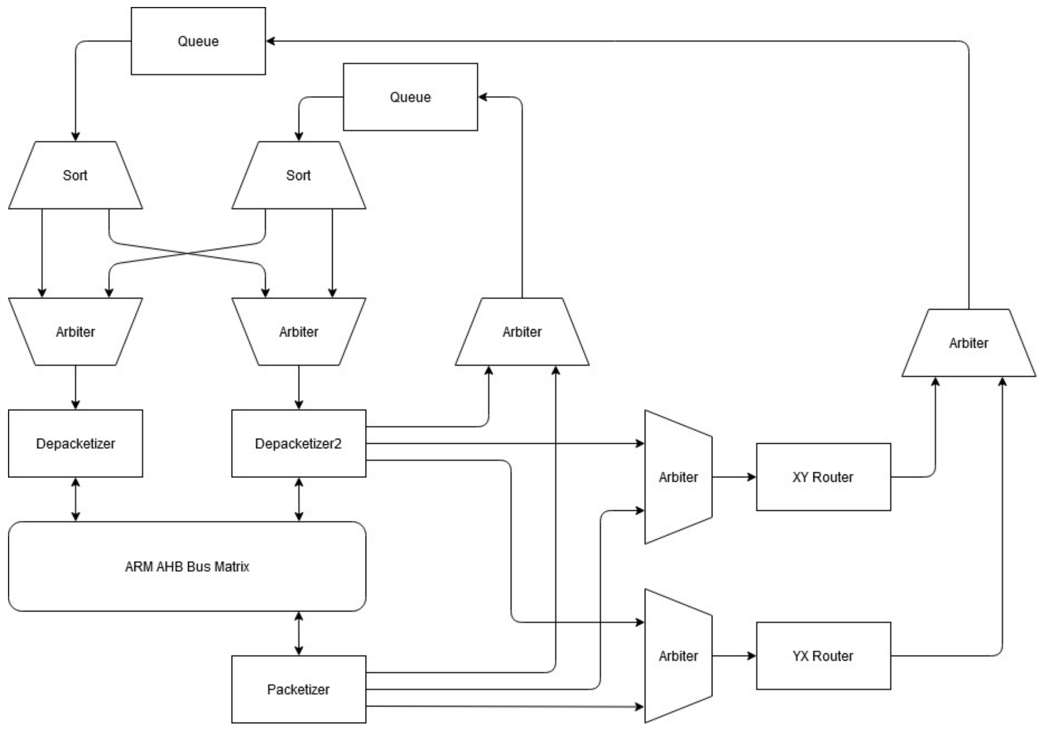


Figure 4.5: Network Interface Block Diagram

tile and free up the network. Figure 4.5 shows a block diagram with the details of the network interface logic on each tile.

Behavior of each arbiter in the system is configurable by writing to the tile's configuration registers. Default behavior of the arbiters is to alternate which input has priority every time there is a conflict. But the arbiters can also be configured in two different priority modes. In strict priority mode, one of the two inputs will always win the arbitration when there is a conflict. In relaxed priority mode, a count value is written into the configuration registers. A chosen input will win the arbitration the number of times indicated, then arbitration will prioritize the other input for a single conflict. This pattern will continue indefinitely. Priority arbitration can also help reduce the likelihood of deadlock. Response messages can be given higher priority to access the router than write and request messages, and the priority of messages coming over the network from other tiles can be adjusted relative to messages generated on the local tile.

4.4 Testing

We tested the network through functional RTL simulation using several programs written for the ARM cores. Programs were written in C, and compiled binaries were loaded into memory at the start of the simulation. We designed the first program to test the functionality of the remote write capability using circular queues. The circular queues were used to implement a mailbox. The transmitting tile was instructed to send a sequence of messages to the mailbox, and the receiving tile would verify that it received the exact same sequence of messages from the mailbox. A specific range of memory addresses in the shared memory on the receiving tile was designated for the data items in the circular queue. Additionally, an address in shared memory on the receiving tile was designated to hold the tail index of the queue, and an address in shared memory on the transmitting tile was designated to hold the head index of the queue. The transmitting tile writes data into the queue, and then adjusts the tail pointer accordingly. The receiving tile reads data from the queue, and then adjusts the head pointer to indicate it has processed the message. Both tiles can use the values of the head and tail indices to determine when the queue is full or empty. The transmitter performs an on-tile read to determine the value of the head index and knows what value it has written to the tail index. The receiver performs an on-tile read to determine the value of the tail index and knows what value it has written to the head index. Cores write to digital I/Os, connected to LEDs when the tests are run on an FPGA, to indicate whether the pattern matches the expected one or not.

We designed another program to test the functionality of remote reads and CAS operations. In this program, each tile initialized several addresses in its section of the shared memory with values based on tile coordinates and memory address. Then, tiles can perform read and CAS operations on various addresses and verify that the response is as expected. For CAS operations in this program, the swap value was always set the same as the compare value so that the value in memory did not actually change. This ensures that every core in the system should always know what value to expect in memory, and whether a CAS operation should return with success or failure. We wrote this program in a highly configurable manner using preprocessor directives. Changing some define statements at the top of the program before compiling could adjust the type of requests generated (read or CAS), the traffic pattern (random uniform or hot spot), the ratio of network use (XY vs YX), and the number of requests each core could have in-flight at a time. This test also uses writes to digital I/O by the cores to provide an easy indication of success or failure. Additionally, each core keeps track of the number of responses that meet and fail expectations in locations in its private memory bank. These values can be read via JTAG to determine whether the test was successful or not.

For a higher level system test, we also used a program that runs a parallel breadth first search (BFS) algorithm on a graph and verified proper behavior. Running the BFS algorithm demonstrates the processor's ability to perform useful work, and gave us the opportunity to begin developing a programming model for the system. Additionally, the BFS algorithm relies on CAS operations where the swap operation actually writes a new value, covering a hole in the earlier test program.

4.5 Analysis

For analyzing the theoretical performance of our network, we chose the bisection bandwidth metric. This number represents the maximum bandwidth between two halves of a bisected network. The network is bisected with the minimum number of inter-node links cut, producing the minimum bandwidth between the two halves.

The bisection of a mesh network with N nodes cuts through $\sqrt{N}$ links. Although our architecture supports a mesh size of 32x32, the physical area of the tile design and the size of the wafer being used to construct our prototype limit us to a mesh size of 25x25. Because our design uses two mesh networks, a bisection partitioning will actually cut twice as many links, because we are partitioning two networks at once. The links are bidirectional, and our anticipated clock speed is 275 MHz. We can therefore calculate our bisection bandwidth as follows:

$$\sqrt{25 \times 25} = 25 \text{ links cut} \tag{4.1}$$

$$\frac{25 \text{ links}}{\text{network}} \times 2 \text{ networks} = 50 \text{ links} \tag{4.2}$$

$$50 \text{ links} \times \frac{2 \text{ messages}}{\text{bi-directional link}} \times \frac{99 \text{ bits}}{\text{message}} = 9900 \text{ bits} \tag{4.3}$$

$$\frac{9900 \text{ bits}}{\text{cycle}} \times \frac{275 \times 10^9 \text{ cycles}}{\text{second}} = \frac{309 \text{ gigabytes}}{\text{second}} \tag{4.4}$$

This is the rate at which our network can transfer raw information, but many of the bits in our messages are control and address bits. The actual rate of data transmission is lower at only 32 bits per message:

$$50 \text{ links} \times \frac{2 \text{ messages}}{\text{bi-directional link}} \times \frac{32 \text{ bits}}{\text{message}} = 3200 \text{ bits} \tag{4.5}$$

$$\frac{3200 \text{ bits}}{\text{cycle}} \times \frac{275 \times 10^9 \text{ cycles}}{\text{second}} = \frac{100 \text{ gigabytes}}{\text{second}} \tag{4.6}$$

By also calculating the theoretical maximum performance of the processor independent of the bandwidth, we can develop a roofline model for the waferscale processor prototype. For our maximum performance model, we assume each core in the system is able to complete a floating point operation per cycle.

$$25 \times 25 \text{ tiles} = 625 \text{ tiles} \tag{4.7}$$

$$625 \text{ tiles} \times \frac{14 \text{ cores}}{\text{tile}} = 8750 \text{ cores} \tag{4.8}$$

$$8750 \text{ cores} \times \frac{1 \text{ flop}}{\text{cycle}} \times \frac{275 \times 10^9 \text{ cycles}}{\text{second}} = \frac{2187.5 \text{ gigaflop}}{\text{second}} \tag{4.9}$$

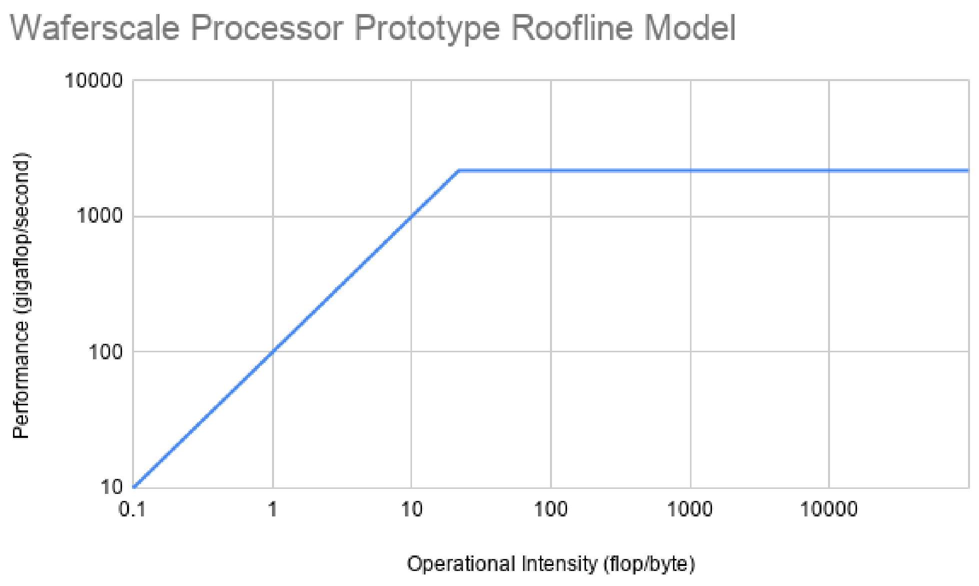


Figure 4.6: Roofline Model of the Waferscale Processor Prototype

We assume that each floating point operation requires two memory reads and one memory write. Given our ARM cores are 32 bit cores, this means each operation requires 96 bits or 12 bytes of data. Using this information, we generate the roofline plot in Figure 4.6.

CHAPTER 5

CONCLUSION

Core counts and processor sizes continue to increase. Moore's law and Dennard scaling no longer provide the straightforward improvement in processor performance they once did, so parallel performance is becoming increasingly important. Additionally, important tasks such as machine learning, cloud computation, and big data analysis are all highly parallelizable and benefit easily from increased parallelism in processors.

To support this trend, new manufacturing methods like the silicon interconnection fabric and new architectural elements like networks-on-chip are required. Current manufacturing standards do not support chips beyond a certain size at acceptable yields, limiting the number of cores that they can integrate on a chip. New techniques for composing chips out of smaller modules called chiplets look to address this limitation. The NoC is an important design element of large processors, as it is responsible for providing computational cores with the data they need, thereby governing the system's maximum throughput. To do meaningful work, processing cores must be able to communicate with memory and with one another efficiently. NoCs are the current state of the art for providing this communication in an efficient and scalable way.

In this thesis, we presented the design of an NoC developed for enabling shared memory processing on a waferscale processor. Through this design process, we noted that considerations about redundancy in the network and maximum latency of the network take on special importance when designing a large processor like the waferscale prototype we developed. Several of the design choices made for the NoC presented here were made with an eye toward simplicity of developing a working prototype device. Future work could examine the potential benefits of a torus topology, which would shorten the maximum number of hops between two nodes in the system but complicate the physical routing of the wafer and the network routing algorithm. Additionally, the use of smaller flits and virtual channels could be explored. This would allow variation in packet sizes, different priority for different packet types, and reduce the total number of physical wires necessary on the wafer. Both of these changes would also complicate the design of the router, which implements the routing algorithm and message buffering policies in hardware.

The prototype device developed here could also be used as a tool for exploring various parallel programming models. For example, transactional memory is a model for shared memory computation that eliminates the need for locks to protect data. It is generally proposed along with specific hardware to monitor memory interactions and buffer values before committing them to memory. The waferscale processor could implement a version of this scheme using low-level firmware and the private memory bank each core possesses.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.

[2] R. H. Dennard, F. H. Gaensslen, Hwa-Nien Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Proceedings of the IEEE*, vol. 87, no. 4, pp. 668–678, 1999.

[3] R. S. Williams, "What's next? [The end of Moore's law]," *Computing in Science Engineering*, vol. 19, no. 2, pp. 7–13, 2017.

[4] B. A. Nayfeh and K. Olukotun, "A single-chip multiprocessor," *Computer*, vol. 30, no. 9, pp. 79–85, 1997.

[5] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[6] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[7] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 282–293, 2000.

[8] "Power4 the first multi-core 1GHz processor," online, IBM, https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

[9] "Dual core era begins, PC makers start selling intel-based PCs," press release, Intel, https://www.intel.com/pressroom/archive/releases/2005/20050418comp.htm, 2005.

[10] "AMD ships first dual-core processors," online, NBC, http://www.nbcnews.com/id/7589302/ns/technology_and_science-tech_and_gadgets/t/amd-ships-first-dual-core-processors/, 2005.

[11] S. Sapatnekar, E. Haritan, K. Keutzer, A. Devgan, D. A. Kirkpatrick, S. Meier, D. Pryor, and T. Spyrou, “Reinventing EDA with manycore processors,” in *2008 45th ACM/IEEE Design Automation Conference*, 2008, pp. 126–127.

[12] R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, “Machine learning and manycore systems design: A serendipitous symbiosis,” *Computer*, vol. 51, no. 7, pp. 66–77, 2018.

[13] “Introducing the single-chip cloud computer,” White Paper, Intel, 2010.

[14] A. Duran and M. Klemm, “The Intel® many integrated core architecture,” in *2012 International Conference on High Performance Computing Simulation (HPCS)*, 2012, pp. 365–366.

[15] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, “Design and analysis of an APU for exascale computing,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 85–96.

[16] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C. Wu, and D. Nellans, “MCM-GPU: Multi-chip-module GPUs for continued performance scalability,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 320–332.

[17] S. Pal, D. Petrisko, M. Tomei, P. Gupta, S. S. Iyer, and R. Kumar, “Architecting waferscale processors - a GPU case study,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 250–263.

[18] E. N. Berg, “Can troubled trilogy fulfill its dream?” *The New York Times*, https://www.nytimes.com/1984/07/08/business/can-troubled-trilogy-fulfill-its-dream.html, July 8, 1984.

[19] J. Schefter, “Giant microcircuits for super-fast computers,” *Popular Science*, Jan. 1984.

[20] J. F. McDonald, E. H. Rogers, K. Rose, and A. J. Steckl, “The trials of wafer-scale integration: Although major technical problems have been overcome since WSI was first tried in the 1960s, commercial companies can’t yet make it fly,” *IEEE Spectrum*, vol. 21, no. 10, pp. 32–39, 1984.

[21] W. C. Chen, C. Hu, K. C. Ting, V. Wei, T. H. Yu, S. Y. Huang, V. C. Y. Chang, C. T. Wang, S. Y. Hou, C. H. Wu, and D. Yu, “Wafer level integration of an advanced logic-memory system through 2nd generation CoWoS® technology,” in *2017 Symposium on VLSI Technology*, 2017, pp. T54–T55.

[22] R. Mahajan, R. Sankman, N. Patel, D. Kim, K. Aygun, Z. Qian, Y. Mekonnen, I. Salama, S. Sharan, D. Iyengar, and D. Mallik, “Embedded multi-die interconnect bridge (EMIB) – a high density, high bandwidth packaging interconnect,” in *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, 2016, pp. 557–565.

[23] F. T. Leighton and C. E. Leiserson, “Wafer-scale integration of systolic arrays,” *IEEE Transactions on Computers*, vol. C-34, no. 5, pp. 448–461, 1985.

[24] S. K. Moore, “Huge chip smashes deep learning’s speed barrier,” *IEEE Spectrum*, vol. 57, no. 1, pp. 24–27, 2020.

[25] S. Jangam, S. Pal, A. Bajwa, S. Pamarti, P. Gupta, and S. S. Iyer, “Latency, bandwidth and power benefits of the superchips integration scheme,” in *2017 IEEE 67th Electronic Components and Technology Conference (ECTC)*, 2017, pp. 86–94.

[26] S. Pal, D. Petrisko, A. A. Bajwa, P. Gupta, S. S. Iyer, and R. Kumar, “A case for packageless processors,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 466–479.

[27] R. Kumar, V. Zyuban, and D. M. Tullsen, “Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling,” in *32nd International Symposium on Computer Architecture (ISCA’05)*, 2005, pp. 408–419.

[28] W. J. Dally and B. Towles, “Route packets, not wires: on-chip interconnection networks,” in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 684–689.