Petabit Switch Fabric Design

Yue Cao
Jen-Hung Lo
Jingxue Zhou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-88
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-88.html

May 13, 2016

Copyright © 2016, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Special thanks to our advisors Elad Alon and Vladimir Stojanovic and our faculty consult committee John Wawrzynek. Also many thanks to the graduate students at BWRC who helped us tremendously with the tools setup for our project: Christopher Yarp, Angie Wang, Taewhan Kim, Paul Rigge, Ranko Sredojevic. Also, we would also like to thank to LBL staffs Farzad Fatollahi-Fard and David Donofrio for letting us use their OpenSoC as our project baseline and giving us a presentation about their research

University of California, Berkeley College of Engineering

MASTER OF ENGINEERING - SPRING 2016

Electrical Engineering and Computer Science

Physical Electronics and Integrated Circuits

Petabit Switch Fabric Design

Yue Cao

This **Masters Project Paper** fulfills the Master of Engineering degree requirement.

Approved by:

1. Capstone Project Advisor:

Signature: ____________________________ Date _____________

Print Name/Department: Vladimir Stojanovic/EECS

2. Faculty Committee Member #2:

Signature: ____________________________ Date _____________

Print Name/Department: John Wawrzynek/EECS

Table of Content

Chapter 1 Technical Contributions

1. Introduction 3
2. Knowledge domains 5
3. Methods and materials 9
4. Results and discussions 13
5. Conclusions 23

Chapter 2 Engineering Leadership

1. Introduction 25
2. Industrial analysis 26
3. Technical strategy 28
4. Marketing 29

Reference 32

Appendix 34

Chapter 1 Technical Contributions

1. Introduction

A router is a networking device that connects endpoint devices. The goal of our project is to design a high performance router. The network performance is related to the network delay, and the network delay is related to the number of router hops. The number of router hops is defined as the number of routers that data needs to go through from the packet source to the packet destination. The number of router hops can be reduced by increasing the router radix, which is the number of ports on the router. Therefore, for this project we designed a high radix router to achieve the high-performance goal.

For the starting point of our project, we were provided with the OpenSoC Fabric code from the Lawrence Berkeley Lab (LBL). The OpenSoC Fabric is an open source project that implements a router network generator in Chisel. For the purpose of this project, we are only interested in testing a single router. Hence our goal is to extract a router from LBL's router network, test the design with different parameters, evaluate the design with metrics such as area, power and performance, and finally conclude an optimal construction of a high radix switch.

Before moving to the details of our project, it is necessary to first introduce our project breakdown. The breakdown flow diagram is shown in Figure 1.1. Because the goal of this project is to optimize a router design implemented by Chisel, the first stage of our project is learning the concept of a router and the usage of Chisel. We got the concept of router from Becker's thesis *Efficient microarchitecture for network-on-chip routers* (2012) and Dally's book *Principles and Practices of Interconnection Networks* (2004). As mentioned previously, part of our goal is to extract and use a single router from the OpenSoC Fabric code. Therefore, the second stage of the project is reading and understanding the source code.

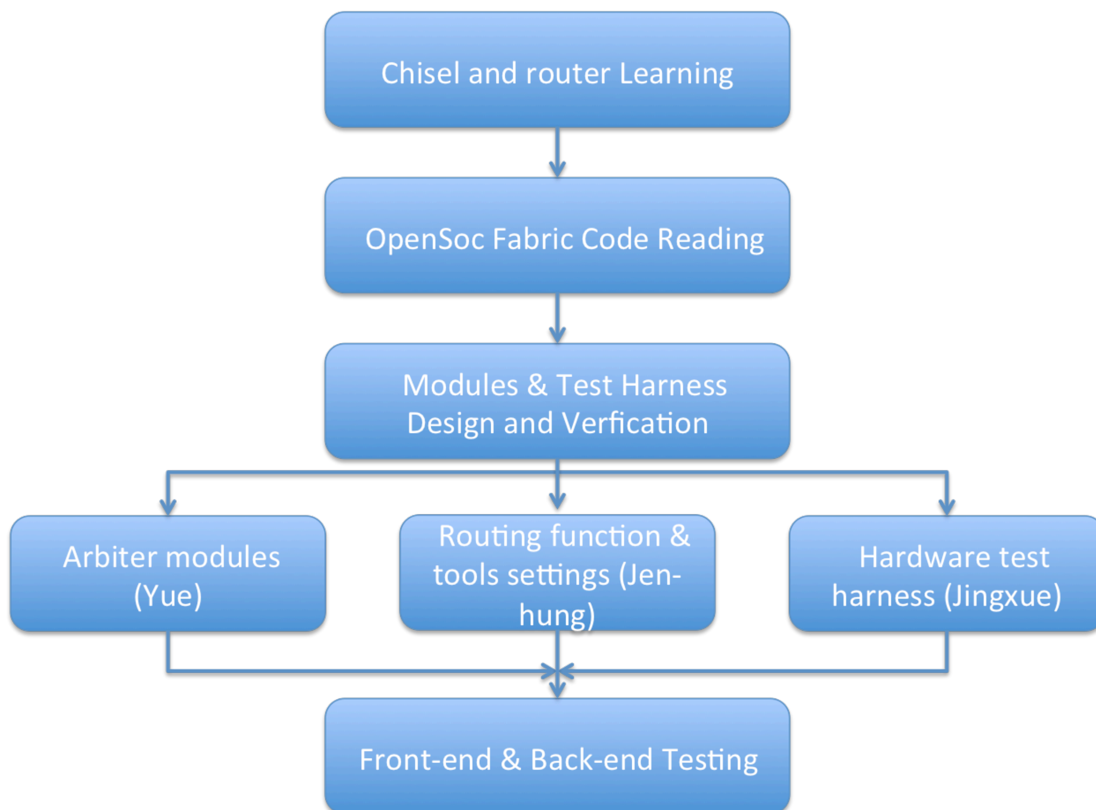

Figure 1.1 Work breakdown diagram

After finishing the required learnings, the next step is to complete the modules and the test design. This step has been divided into many small tasks, which were completed by us separately. The router in the OpenSoC Fabric code has limited number of ports, so in order to change this router to the high-radix design that we want, the modification on the routing function is necessary. This task was completed by Jen-hung Lo (Lo, 2016). He also worked on tool settings. Meanwhile, the hardware test harness for this design was written by Jingxue Zhou (Zhou, 2016). Finally, my tasks were to design two more types of the arbiter, which is one of the modules inside a router. Details of my work will be introduced in the following sections.

After the router's design and test have completed, we moved on to the testing stage. This part was completed by all the teammates. After testing the function correctness of the register-transfer level (RTL) design, we pushed the design through synthesis, evaluated the impact of the design parameters, and filtered out the constructions that exhibited lower

performance. Finally, we ran place and route for the few router designs that we have selected and decided on the optimal construction based on the performance metrics we have defined.

2. Knowledge Domains

2.1 Router Blocks Overview

A router is a networking device that connects the endpoint devices. In general, the function of a router is to route the incoming data packets from its input ports to its outputs based on the information contained within the packets. Figure 2.1 shows the block diagram of a virtual-channel (VC) router. The explanation of each functional stage is adapted from Becker's thesis *Efficient microarchitecture* (2012) and Dally's Book *Network Interconnection* (2004).

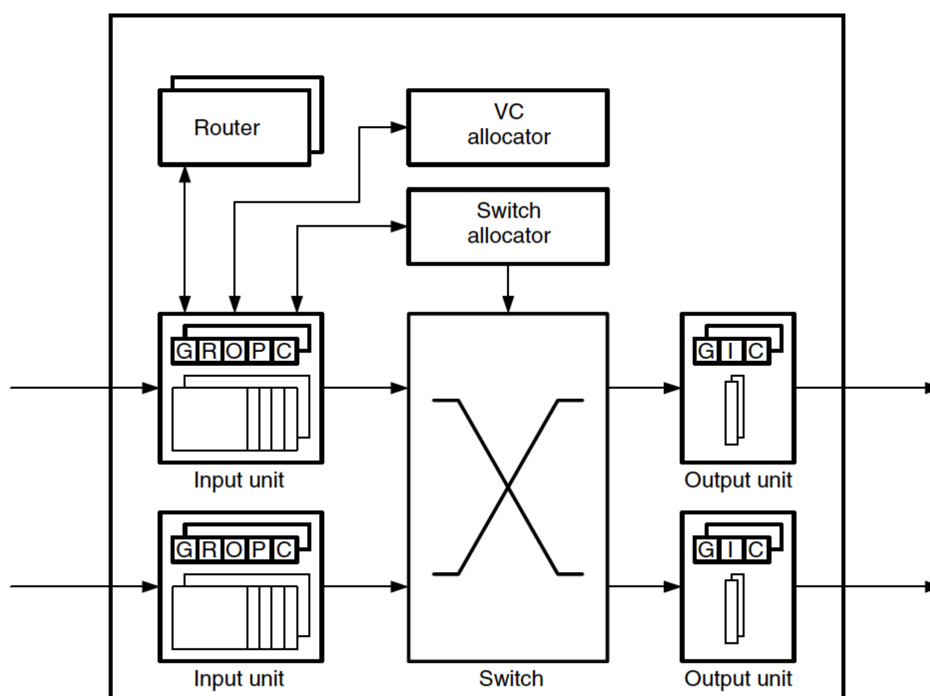

Figure 2.1 Virtual-channel router block diagram (Dally, 2004)

Input buffering: The data packets are sent in the form of sets of flits. When a flit arrives at an input port of the router, it will be first stored in the input buffer. After that, the routing computation and allocation will be conducted. The data flits are held in the input buffer until

they can be forwarded. An input buffer also keeps the state of the stored data packets. The main functioning module for this stage is the input unit in Figure 2.1.

Route computation: The first flit of a data packet, or the head flit, contains the routing information for the whole packet. The function of this block is to select the desired output port based on the information stored within the head flit. Usually, the route computation has a direct relationship to the topology of the router network. However, due to the scope of our project, we are only interested in a single router and do not want to be restricted by a specific topology. To that end, we decided to implement a programmable lookup table as the routing function as opposed to a topology-restrictive routing function such as dimension order routing.

VC allocation: Once the output port is selected, the router needs to allocate one of the output virtual channels to the input flits and the access to the output VC should be exclusive. The allocation is conducted only for the head flit of a data packet. The body flits can bypass this step and simply utilize the allocation that was assigned to the head flit.

Switch allocation: After completing the VC allocation, the switch allocation is conducted. The switch allocation's role is to provide a connection between the input port and the output port. This module needs to make sure that only one input port is linked with one output port. Moreover, it provides the time-slots that the connection needs to be maintained.

Switch traversal: Once the switch allocation is finished, the crossbar switch will receive the granted signal and build the link between the input and the output port. Next cycle, the head flit will traverse through the crossbar switch and be stored in the output buffer. The body flits will follow the head flit in the following cycles. After the tail flit finished the traversal, the connection will be released and the input port can move to the next loop. As for the output buffer, the data flits will be sent once the downflow device is ready. The main

functioning module for this stage is the switch in the diagram. After all of these stages, the data packet has been sent to the output unit of the specified output port.

2.2 Chisel Introduction

For this project, we were using Chisel (Constructing Hardware In a Scala Embedded Language) to build this router. Chisel is a hardware description language embedded in Scala, a high-level programming language (Bachrach, 2015). Chisel is generated by including hardware construction primitives in an existing language, which is easier compared to building a hardware language from scratch. Also, since Scala is a modern programming language, Chisel inherits some features from it, such as object-oriented programming, type inference and support for functional programming. These are all features not presented in conventional hardware description language such as Verilog and VHDL. Moreover, Chisel implementation can be transformed into both C++ form and Verilog form. Since the software (C++) simulation is faster than the hardware (Verilog form) simulation, the verification and testing for Chisel-implemented designs should be faster.

2.3 OpenSoC Fabric Overview

The starting point of our project is the OpenSoC Fabric, so it is important to explain an overview of this design. The following descriptions are adapted from the user and reference manual of OpenSoC Fabric (Fatollahi-Fard, 2015).

OpenSoC Fabric is a highly parameterizable and hierarchical on-chip network generator, which is implemented in Chisel. Using the object-oriented constructs, OpenSoC consists of a hierarchy of modules, and each module has an abstract definition. These abstract modules contain the input and output ports as well as the common functionalities. The child module extended from the abstract modules defines its specific attributes, A set of parameters can be set on the top level, and each child module can inherit these parameters and set them as values. With inheritance, a user can extend or redefine the modules by following the object-

oriented programming principles. The tree of the hierarchical classes is shown in Figure 2.2. This picture gives the overall structure of the OpenSoC Fabric design.

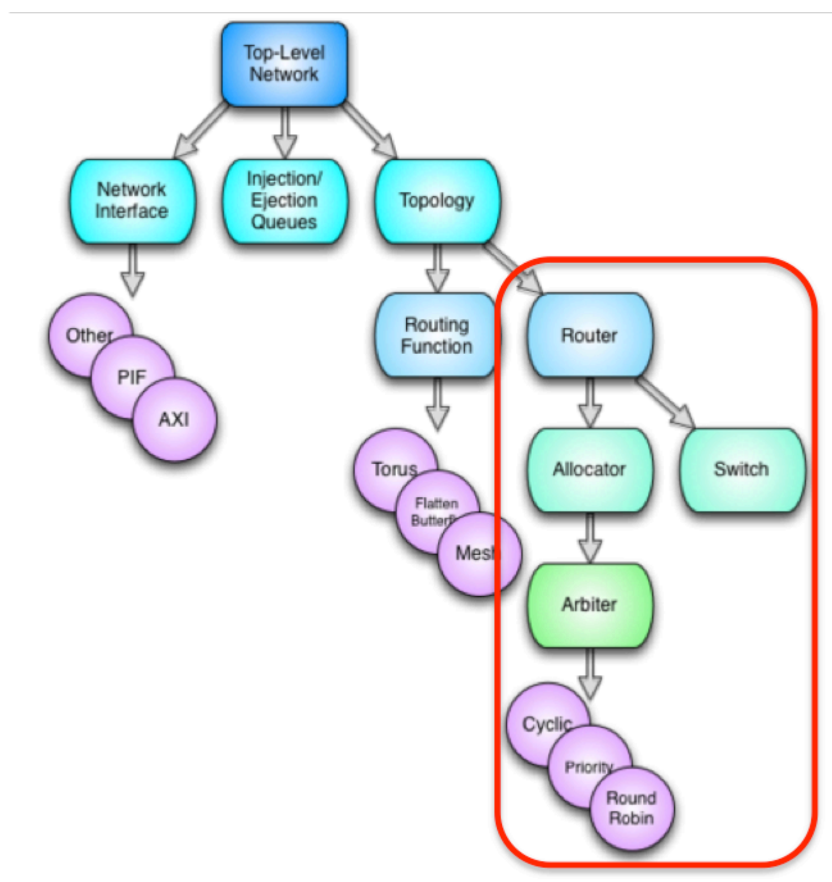

Figure 2.2 Hierarchy tree of the classes in on-chip network

Our project is to design a high radix router. Therefore we put our focus on the router class as well as all the subclasses instantiated within the router class. The classes in concern are circled in Figure 2.2. This implementation of the simple VC router is the baseline of our project. As mentioned before, modifications to the router implementation are still necessary. First, the default route computation for this router is dimension order routing; we need to change it to the lookup table as mentioned previously. Second, the current arbiter type is round-robin. We aim to implement two more types of arbiter. Finally, the original test harness is built for the router network. Hence a new test harness needs to be created for a single router.

3. Methods and Materials

3.1 Arbiters

Monitoring the access of a shared resource among several agents is called arbitration, and a module that does arbitration is called an arbiter. An arbiter is one of the key components for VC allocator and switch allocator, which makes it valuable to analyze. There are different types of arbiters, and the original arbiter in OpenSoC is round-robin. As mentioned in the section above, we decided to implement two more arbiters, carry-lookahead arbiter and matrix arbiter. The algorithms of these two arbiters are extracted from *Network Interconnection* (2004).

Carry-lookahead arbiter: A carry-lookahead arbiter is a fixed-priority arbiter, which means that the priority of the input ports will not change. The most intuitive way of implementing arbitration is looking at the highest priority port, if there is a request, grant this request; if not, move on to the second highest priority port…continue until one request has been granted or all the input ports have been checked. When the priority is fixed, it is possible to generate the grant signals based on the requests on the input ports, instead of the granted signals of the higher priority ports. In this way, the delay on the lowest priority port can be reduced, and the critical path delay can be reduced. By implementing such an arbiter, the clock period of the router can be reduced. However, such an unfair arbitration algorithm may cause the data congestion at the input ports.

Matrix arbiter: A matrix arbiter applies the least recently served priority scheme by keeping a matrix of state bits w_{ij}. The bit w_{ij} in row *i* and column *j* equals to 1 indicating that request *i* has higher priority comparing to request *j*. All the diagonal elements are not needed and $w_{ij} = \neg w_{ji}$. By ANDed with the state bits on its corresponding matrix column, a request can disable all the lower priority requests. After that, the outputs of the AND gates in a column will be ORed together to generate a disable signal for the corresponding request.

When the disable signal is 0, the request will propagate to the grant signal. Once the grant signal is asserted, all the bits on the row will be set to 0 and all the bits on the column will be set to 1. This means that this input port is assigned with the lowest priority. The circuit of a matrix arbiter with 4 input ports is shown as Figure 3.1. The boxes in the circuit implement the matrix elements mentioned above, and the bolded boxes mean that only the upper triangle of the matrix contains independent information, while the lower triangle just keeps the complement values of the upper half elements. Compared to the carry-lookhead arbiter, the matrix arbiter needs a more complicated structure to support the complex algorithm, and the complicated structure induces a longer delay. However, the least-recently served algorithm should give a fairer data distribution.

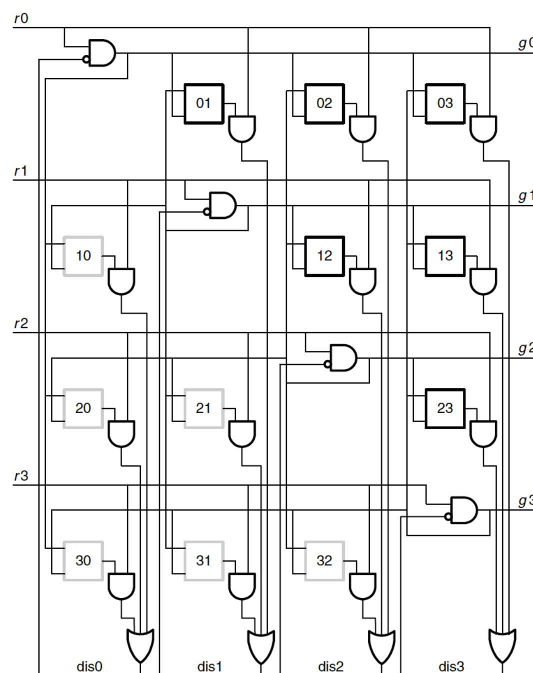

Figure 3.1 Circuit of a matrix arbiter (Dally, 2004)

To summarize, the reason for implementing these two arbiters is to strengthen the arbiter in two different directions. The carry-lookahead arbiter is priority fixed, but it has shorter delay. The matrix arbiter needs a complex implementation, but it uses a fair arbitration algorithm. While the original round-robin arbiter takes the middle of these two directions, by

testing the performance over these three arbiters, we should be able to find a reasonable choice for router optimization.

3.2 Chisel Implementation

As mentioned previously, we used Chisel to implement our design and I mainly designed the two arbiters. After the two arbiters were implemented, by changing the arbiter type inside the *swAllocator* and the *vcAllocator* module in router, we tested the router with different types of the arbiter. The detailed Chisel code is included in Appendix.

The first one is the carry-lookahead arbiter. For simplicity, the port 0 is set to be the highest priority port. Based on the algorithm mentioned above, the grant signals can be calculated directly using request signals. However, since the design needs to be parameterized, the direct logic gate implementation is difficult. After looking at the implementation of the round-robin arbiter from OpenSoC, I used a similar design method, and the snippet of key selection code is shown as below.

```
passSelectL1 := ~requestsBits + Bool(true).toUInt
winner := passSelectL1 & requestsBits
nextGrant := Mux(orR(winner), winner, nextGrant)
```

By adding one to the inverted request signals, inside *passSelectL1*, the bit that corresponds to the lowest port with valid request will be set to 1. The lower bits that correspond to the ports without requests will be 0, and the higher bits should be the inverted of the request bits. By ANDing with the request bits again, only the lowest port with valid port would be granted. In this way, the requests from 0 to *numRadix-1* are assigned with the highest to the lowest priority.

The second one is the matrix arbiter. Similarly, for simplicity, the initial state of the matrix sets the request 0 to the highest priority. The initial priority of a port decreases as port index increases and each port has a different priority. In this way, the arbiter can make sure that at most one request will be granted. The implementation follows the algorithm of matrix arbiter described above. In general, this arbiter assigns the lowest priority to the recently

granted request and should give a fairer data distribution. However, the number of registers used to implement this arbiter grows quadratically. Since the number of registers in the other two arbiters grows linearly, it is expected that the area of this arbiter will be much larger than the area of other arbiters for high radix.

The data packets used in OpenSoC code include the priority information and the round-robin arbiters with priority selection are used in the switch allocator. In order to follow the similar pattern, the priority carry-lookahead and the priority matrix arbiter are also designed. The priority selection algorithms for the two arbiters are the same. When the arbitration is requested, a vector of registers called *PArraySorted* takes in the current request priorities and finds the current highest priority *pmax*. Then, only the requests with the *pmax* priority will go to the normal arbitration process. By adding the priority arbiters into the router design, the router can also utilize the different priorities of the individual data packets.

Finally, some testbenches have been written to verify the correctness of the two arbiters. The testbenches are written for the arbiters of radix 8. The arbiters were checked to see if their arbitrations followed their expected algorithm and the request lock on their input ports worked correctly. Also, if an arbiter is free and no valid request exists at a cycle, the arbiter should report that the granted signal is invalid. To run the testbenches, the modules and harnesses have been added to the main function. The detailed code has been included in Appendix.

3.3 Application Specification Integrated Circuit (ASIC) Design flow

This router design is an Application Specification Integrated Circuit (ASIC) design. Our design follows the usual ASIC design flow of three stages: RTL design and verification, logical synthesis, and place-and-route.

RTL design and verification: The first step of ASIC design is to generate the high-level implementation of the design, that is, generate the Register Transfer Language (RTL)

design. Being specific to our project, this step is to write the behavioral modules using Chisel. Next step in this stage is to verify the behavior of the code. We need to verify the functionality at Chisel level. It can be achieved by building the design and the test harness in sbt, which is an open source build tool for Scala. For this stage, to save time, we discarded some configurations that give large latency.

Logical Synthesis: The second stage is logical synthesis, which transforms the behavioral Verilog code into the gate-level circuit implementation. Logical synthesis uses a standard library that includes the logic gates like INV, AND, and XOR etc. to realize the behavior described in the given Verilog code. This stage also estimates the power, area, and clock period in a relatively inaccurate way since the routing information is missing. This stage is completed using the Design Compiler. After this stage, we did some evaluations based on the results.

Place-and Route: The final stage is place-and route, which takes in the gate-level netlist and generates the final layout for the design. To achieve better performance, placement optimization, clock tree synthesis, and routing optimization are necessary. After this stage, we have got the final design and the test results for this design will be reliable. The stage is completed using the tool IC compiler.

4. Results and Discussions

As mentioned previously, this project is to find the optimal construction for a high-radix router. Due to time and hardware limit, we just did the software simulations and the logical synthesis for the router switches with 2 to 64 ports. Based on these simulation results, we did the extrapolation for the information of the 128-port router and made the parameter decision. The goal of this project is to achieve better performance, and the performance can be evaluated in terms of the throughput or the average packet latency. Since both the throughput and the average latency should increase as radix grows and higher throughput and lower

average latency are preferred, we used throughput/latency as our key evaluation factor of the performance. Besides performance, area and power consumption were also considered. The design parameters that we have decided were the number of radix and the arbiter type. For this result section, we set the number of virtual channel per port to be 2, the flit width to be 55 and the injection rate to be 10%. To get a general result for this project, we used the random generated input packets.

Throughput: throughput is the number of data bits per second that a router can transfer from the input to output ports. Throughput can be calculated using the following equation. We included the channel utilization to calculate the real throughput, instead of the maximum throughput.

$$Throughput \ = Number\ of\ ports \cdot Flit\ size \cdot Clock\ frequency \cdot Channel\ Untilization$$

Average Latency: the average latency mentioned here means the average time cost for a data packet to pass the router switch. To get this value, we did the software simulation that injected 64 packets into each input port. The program recorded the latencies in cycles for each packet and we used the average value for our performance metrics.

4.1 Arbiter decision

First, we decided the arbiter type for our router among the round-robin (RR) arbiter, the matrix arbiter, and the carry-lookahead (CL) arbiter. To make a decision for the arbiter type, it is important to analyze the data distribution pattern of each arbiter. The following plots show the minimum latency, the maximum latency and the average latency for the router with these three types of arbiters.

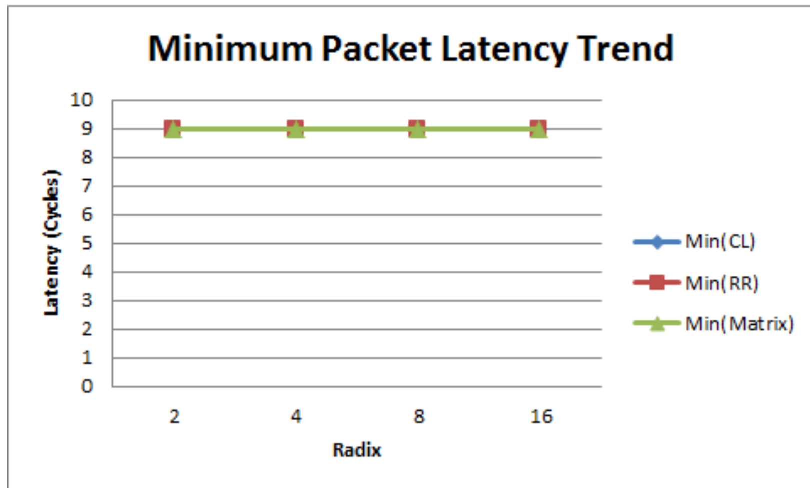

Figure 4.1 Minimum packet latencies for three arbiters

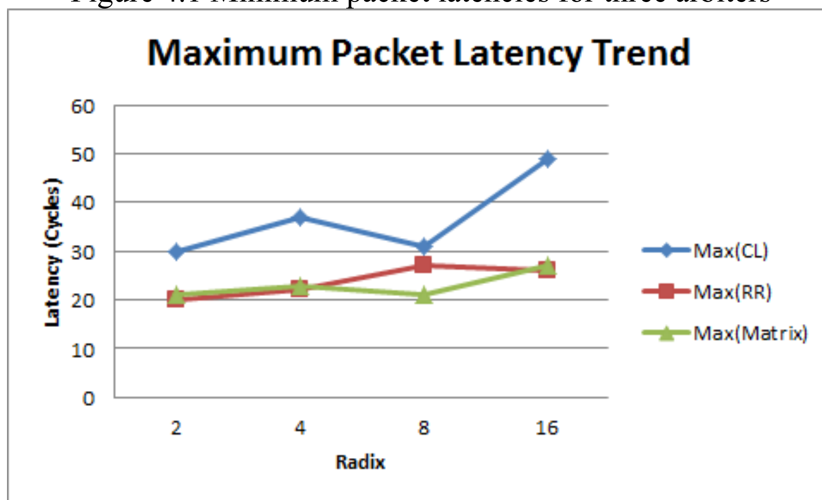

Figure 4.2 Maximum packet latencies for three arbiters

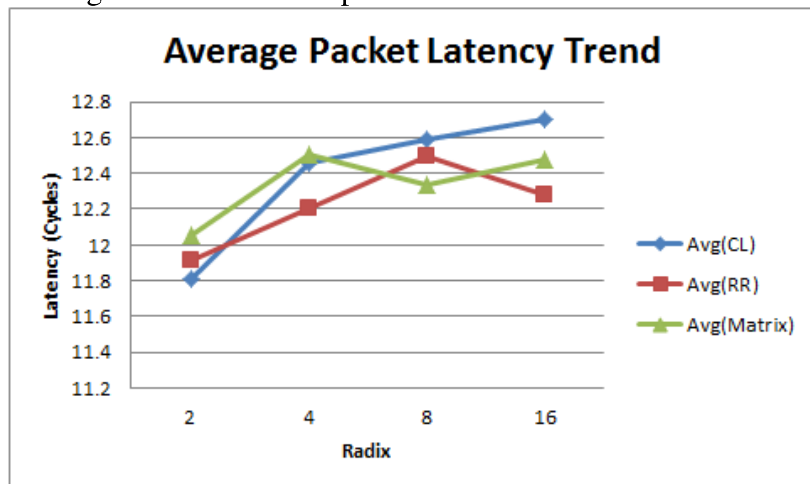

Figure 4.3 Average packet latencies for three arbiters

It can be seen on the plots that the minimum latency of each arbiter is the same, and the maximum latency of the CL arbiter is larger than the maximum latency of the other two arbiters. Also, the average latency of the CL arbiter grows faster than the other two arbiters.

Figure 4.4 shows the latency histograms for the three arbiters with 16 ports. We found that the RR and the matrix arbiter generated similar data transfer pattern, which gave a fairer data distribution, while the CL arbiter sent a few data packets with long delays. In general, it should be reasonable to conclude that the RR and the matrix arbiter transfer data packets in similar patterns but the CL arbiter transfers data in a less fair way.

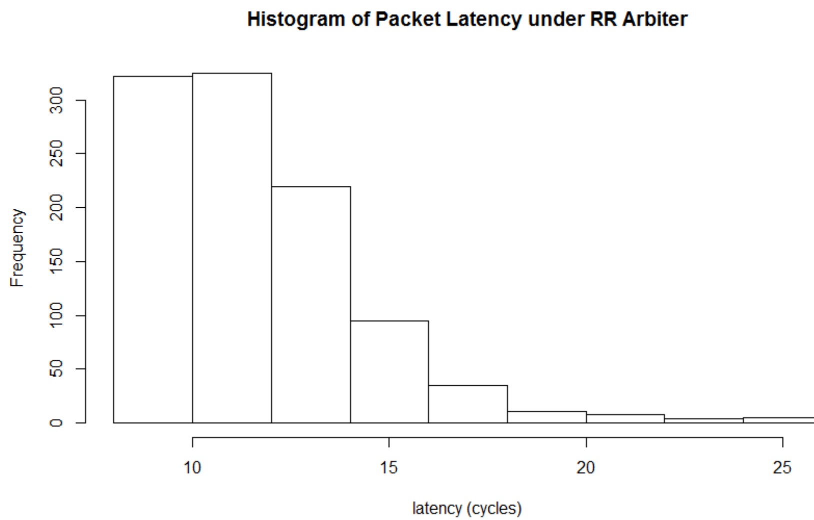

Figure 4.4a Packet latency histogram for RR arbiter

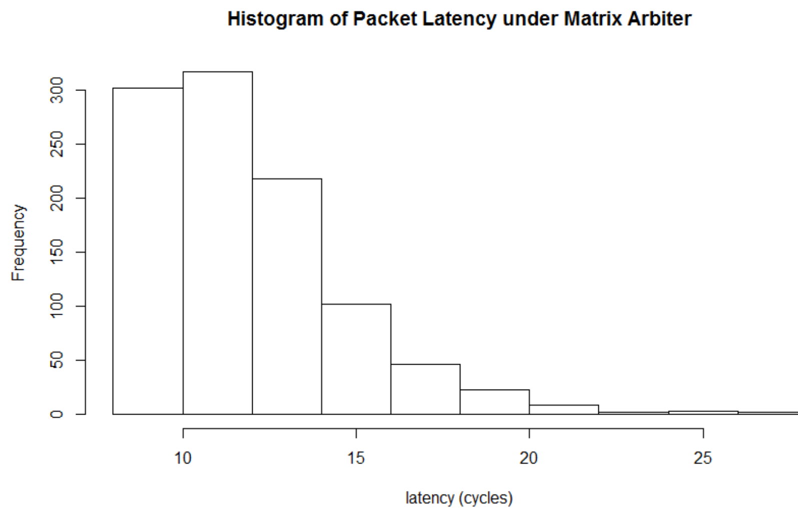

Figure 4.4b Packet latency histogram for matrix arbiter

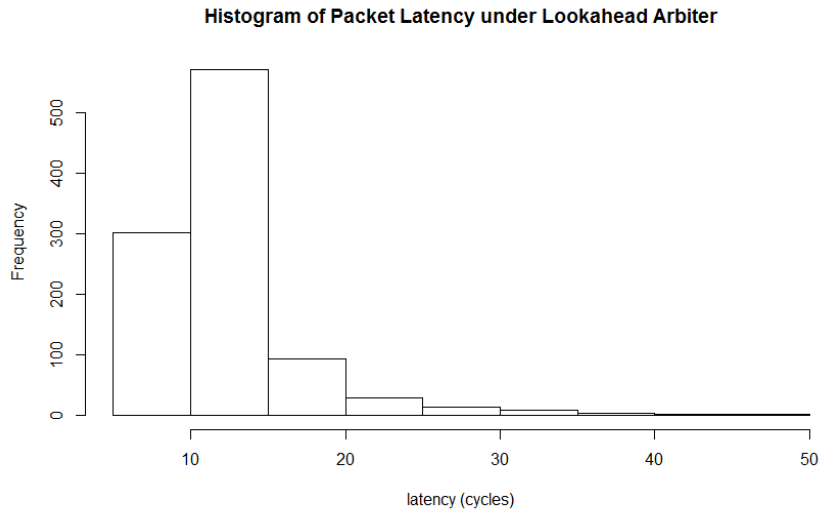

Figure 4.4c Packet latency histogram for CL arbiter

Besides the data transfer pattern, we also evaluated the arbiters over performance, area, and power. The following three tables show the throughput/latency, the total area and the total power of the routers using RR arbiters, matrix arbiters and carry-lookahead (CL) arbiters. Due to time limitation, we just compared these values for radix from 2 to 8. The bolded values are the best results for each condition.

Throughput/latency(Gb/(s*cycle))	2	4	8
RR Arbiter	1.526	2.300	3.485
Matrix Arbiter	1.400	2.240	3.948
CL Arbiter	**1.633**	**2.641**	**4.282**

Table 4.1 Throughput/latency for the router of radix 2, 4, and 8

Area(um^2)	2	4	8
RR Arbiter	**104491.3082**	223193.841	486472.7992
Matrix Arbiter	105031.8725	229982.0274	548598.3014
CL Arbiter	104736.5572	**221551.0542**	**476628.531**

Table 4.2 Total area for the router of radix 2, 4, and 8

Power(uW)	2	4	8
RR Arbiter	**24800**	**48400**	**95600**
Matrix Arbiter	**24800**	50700	123000
CL Arbiter	26500	50700	101000

Table 4.3 Total power consumption for the router of radix 2, 4, and 8

Based on the results above, we got some features of the three arbiters. The RR arbiter gives a fair data distribution and consumes the lowest power. However, the performance of the router with RR arbiters becomes the worst for higher radix. The matrix arbiter also gives a better data distribution pattern among the three arbiters, and the performance at higher radix of the router with this type of arbiters is better than the performance of the router with RR arbiters. But the area and the power consumption of the matrix arbiter is the highest. For the CL arbiter, the router with this type of arbiters has the highest performance and the lowest area at higher radix, but the data distribution of this arbiter is less fair and the average packet latency for such a router grows faster with higher radix. Therefore, all of the three arbiters have their advantages and disadvantages, and can become the best arbiter for the router design under different conditions. For this project, the main goal is to improve the general router performance, so we decided to use the CL arbiter for our router design. The simulation results for the following section are based on the router with CL arbiters.

4.2 Number of radix decision

Since we have decided to use carry-lookahead arbiter in the previous section, we analyzed the effect of the radix number on the router design with carry-lookahead arbiter. Due to time limitation, we just did the software simulations and the hardware simulations for radix from 2 to 64. We extrapolated the simulation results to evaluate the design with radix 128.

To save time, we used the clock period from the post-synthesis design. Figure 4.5 shows the clock periods of the router designs from radix 2 to 128. Since the clock period grows

quite linearly, we modeled the clock period using a linear equation and extrapolated the clock period for the 128-port design. Similarly, we extrapolated the average packet latency for the 128-port design and Figure 4.6 shows the average packet latencies for routers with different radix numbers. Since the result for 2-port router differs a lot from the results for the other port numbers, we discarded the value for 2 ports for extrapolation. One of the possibilities of this difference is that the sample size of the 2-port router software simulation is quite small, which is 128, so the random test might cause a large variation among different simulations.

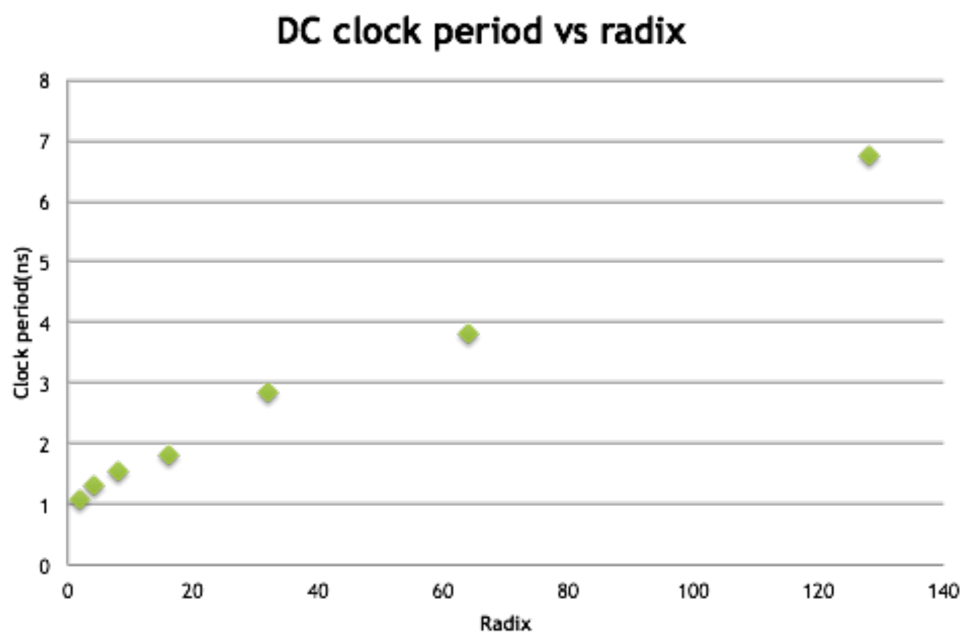

Figure 4.5 Post-synthesis clock period vs radix

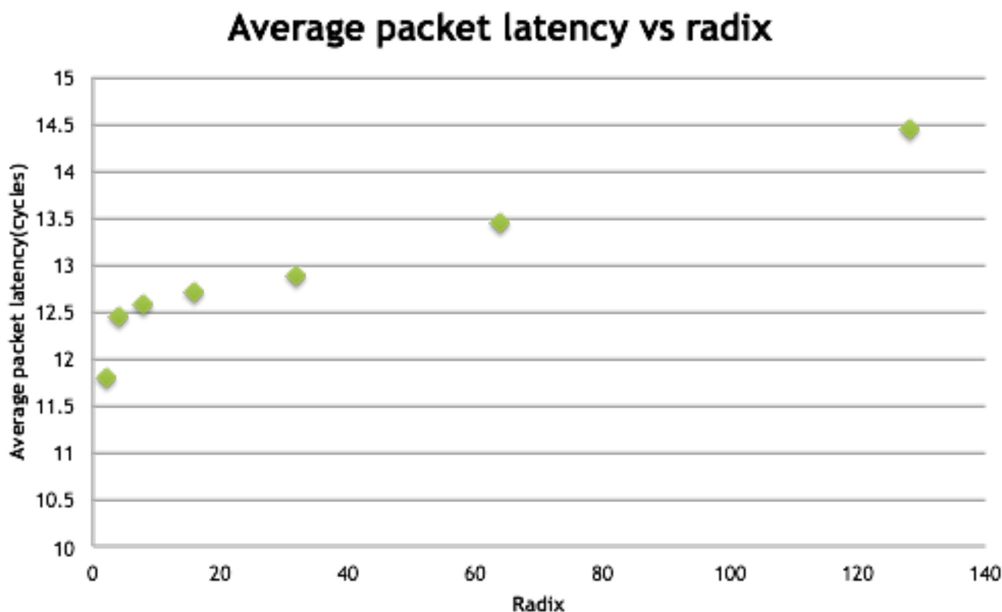

Figure 4.6 Average packet latency vs radix

To calculate the throughput, we extrapolated the channel utilization for radix 128. Since we found that the channel utilization rate is relatively random over different radix numbers, we used the average channel utilization for the 128-port design. Figure 4.7 gives the throughput of the routers from radix 2 to 128.

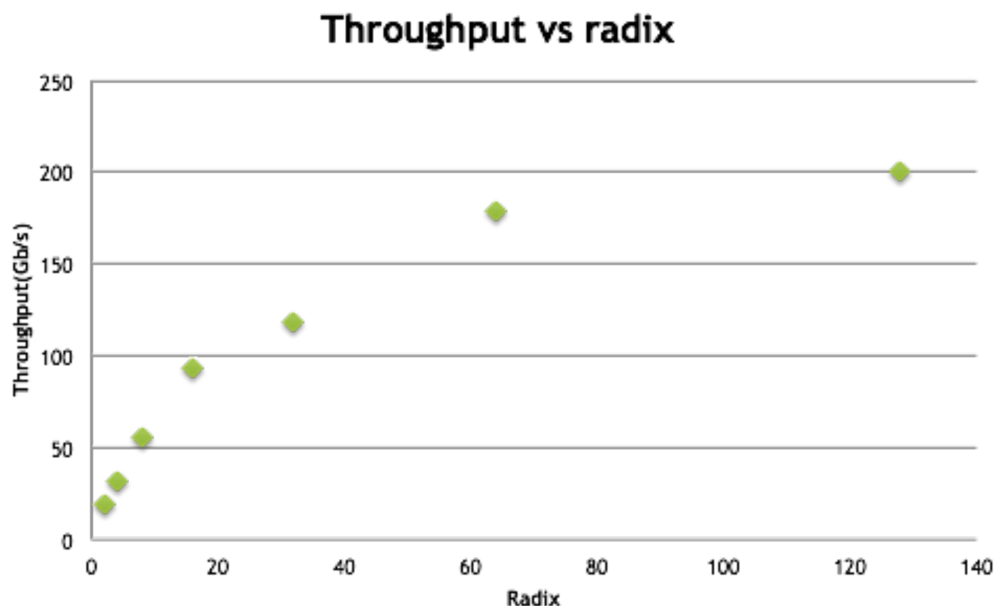

Figure 4.7 Throughput vs radix

Finally, we used the previous results to calculate the throughput/latency. Figure 4.8 shows the throughput/latency for the router designs with different radix numbers. We found that the highest throughput/latency is achieved at radix 64, which is 13.38Gb/(s*cycles). Therefore, we chose radix 64 for our design.

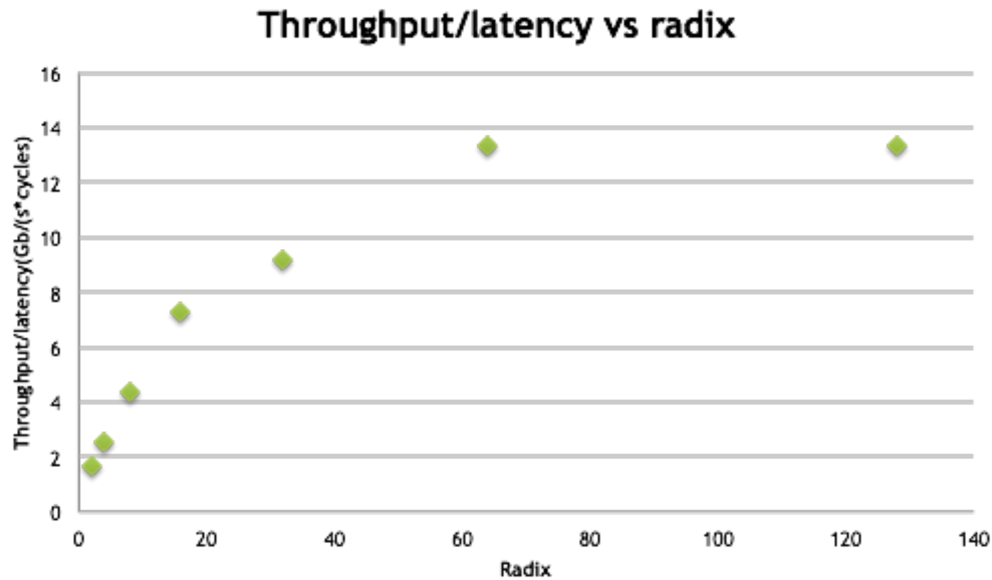

Figure 4.8 Throughput/latency vs radix

4.2 Chosen design exploration

Based on pervious evaluations, we chose radix 64 and CL arbiter for our router design. Therefore, we explored the information of the chosen design besides performance, including data transfer pattern, area and power consumption. Figure 4.9 shows the latency histogram of the router design. The plot shows that the data distribution is quite unfair, but most of the data packets have been transferred in a short delay. For a higher injection rate, the data congestion condition might get worse.

For the post-synthesis design, the total area is $7736093.7304\mu m^2$ and the total power consumption is $1.32W$. The results are large, but should still be affordable under this radix number. However, if the design needs to be improved to a higher port number, it is possible that the area and the power will become the main limitation factors. Figure 4.9 and Figure 4.10 show the area and the power consumption distribution of the routers among the main

modules. The distributions for power and area are similar, and we found that the buffers took the largest portion. However, since the complexity of the switch and the switch allocator grows quadratically, it is reasonable to anticipate that they will become the largest modules for a higher radix router.

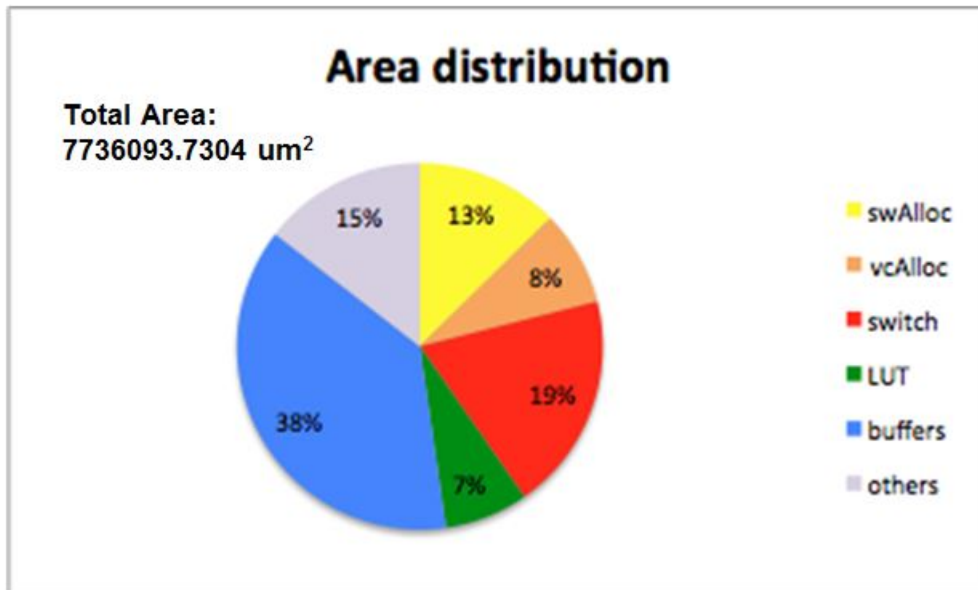

Figure 4.9 Area distribution for post-synthesis 64-port router switch

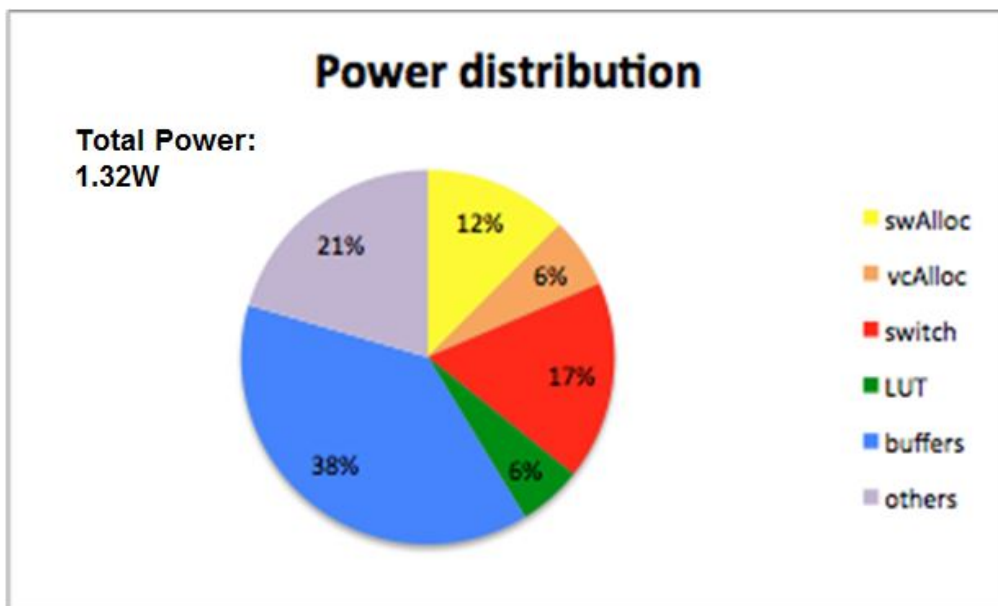

Figure 4.10 Power distribution for post-synthesis 64-port router switch

4.3 Post-route design information

We also pushed our router design through place-and-route to generate the layout for the chip design. Due to time and device limitation, the highest radix router that we pushed through ICC was 16-port design and the module-highlighted layout is shown in Figure 4.11. The yellow region corresponds to the switch allocator, the orange one corresponds to the VC allocator, the red one corresponds to the switch, the green one corresponds to the lookup table and the blue one corresponds to the buffers. We found that buffers took the largest portion, which was consistent with our post-synthesis condition. The clock period for this design is 2.91ns, the total area is 1441394.2407μm^2 and total power is 0.275W.

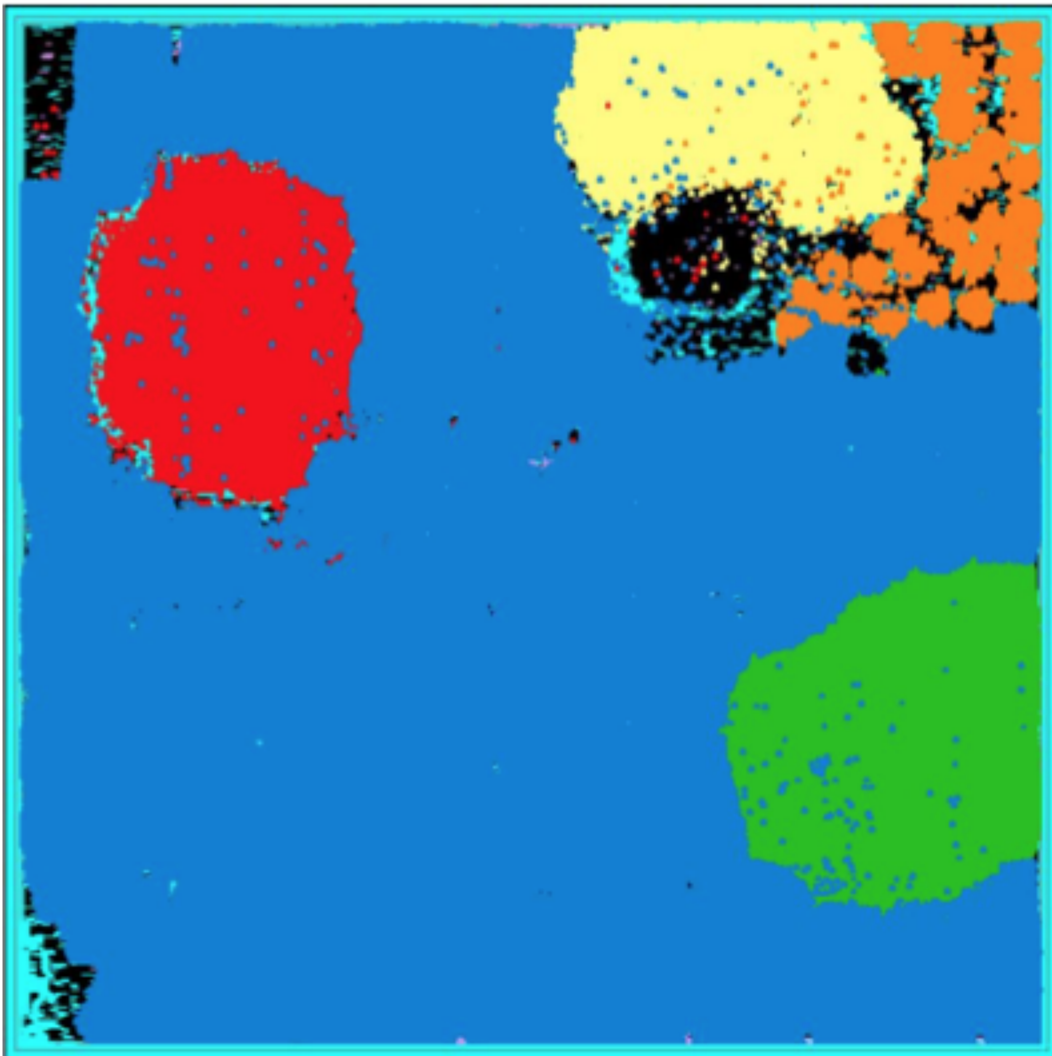

Figure 4.11 Highlighted layout for 16-port router design

5. Conclusions

In this project, we were trying to find an optimal construction for a high-radix router. To extend the design space, we decided to implement the carry-lookahead arbiter and the matrix arbiter. Besides, we determined to use a programmable lookup table for the route computation so that our design can fit in any network topologies. After setup the random test, we have done simulations for routers with different arbiters and different number of ports. Based on the evaluation of throughput/latency, we decided to use the carry-lookahead arbiter and the radix 64 for our router design. We did the software simulation and post-synthesis design analysis for the chosen design and pushed the 16-port router with CL arbiters through the place-and-route process to get a layout of the router chip.

Chapter 2 Engineering Leadership

1. Introduction

Driven by the growing demand for faster processing speed in recent years, chip companies such as Intel and AMD have turned to multi-core CPUs as the solution to scaling system performance (Wolfe, 2009). Unlike single-core processors, multi-core processors integrate hundreds or thousands of processing elements together on small chips. Given the physical proximity of myriads of processors on a single die, significant boost in performance can be achieved while maintaining minimal communication latency. As the number of architectural elements integrated on a single die continues to grow, the network-on-chip (NoC) implementation becomes the major bottleneck in how fast the multicore chip can operate (Becker, 2012). Network-on-chip is essentially the communication system integrated directly on the chip that ties all the processors, memories and external devices together. Figure 1 illustrates a multi-core NoC platform that features multiple cores, memories and other devices linked together by a central NoC switch fabric.

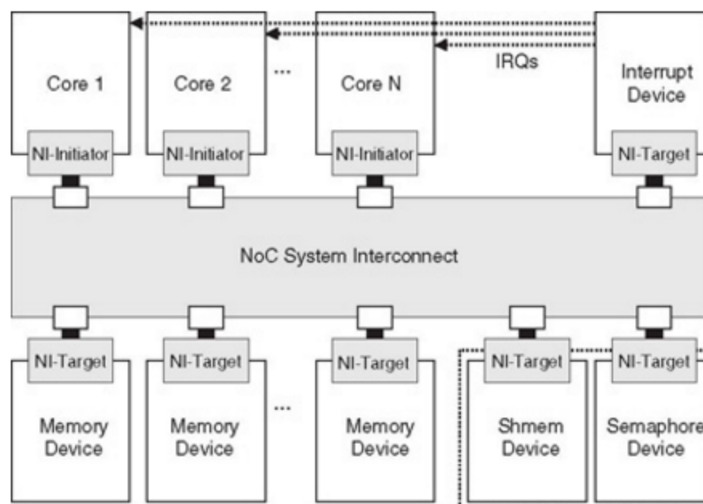

Figure 1. Multi-core Network-on-Chip Layout (Benini, 2007)

The switch fabric itself consists of several network nodes, or routers, that are interweaved together in certain geometrical topology to make up the entire NoC system. Hence, the times it takes to communicate between two network endpoints ultimately depends on the number of

router hops along the path of data traversal (Dally, 2004). The numbers of router hops are directly related to the number of ports –or radix – of a router, and by scaling up the radix of a router we can connect additional endpoint devices and communicate with fewer router hops, thus achieving the level of efficiency required by a multicore system. However, there exist design tradeoffs within router microarchitecture that limit the scope of radix's scalability, hence marking a point of diminishing return in network quality.

Our project, Petabit Switch Fabric Design, thus is to experiment and analyze the design tradeoffs in question and observe how they may help or hinder the performance of a router as it scales up its radix. Using the router design prototype based on the open source code developed by Lawrence Berkeley National Lab as a baseline, we will be investigating the ways in which different parameters may impact the performance of the router design. Ultimately, our end goal is to find the most efficient configuration for high radix router.

2. Industry Analysis

One of the biggest current technology trends is the shift towards cloud computing. Major companies like Dell, Microsoft, and Amazon have started to provide cloud computing services. For example, Dell announced the Dell Private Cloud Solution, which is powered by Intel architecture, and provides infrastructure that helps to reduce ownership cost by having superior automatic allocation of computing resources (2016). Instead of managing their own localized hardware, enterprises can rent data computing resources from these big companies to obtain more flexible resources and to reduce overall cost (Hassan, 2011).

Such trends lead to the collection of data computing resources towards the few big companies mentioned above. To provide the storage for such a large amount of resources, these companies need to construct data centers with warehouse-scale computers (WSC), that is, warehouses full of supercomputers interconnected together. In order for all the computers within such a warehouse to communicate to each other and to the outside world while

maintaining high performance, having powerful interconnection infrastructure is extremely critical. Hence, these data center giants become obvious target customers for our high-speed router.

To assess the profitability of this product, we will use the Porter's five forces model: new entrants, substitutes, buyers, suppliers, and existing rivals (2008). Firstly, consider the force of the new entrants. Routers are highly specialized pieces of hardware that are sold in the form of chips. The biggest part of the chip cost is the non-recurring engineering cost, which is the one-time cost for a chip design, so the overall cost of the chips will decrease drastically when increasing the sale amount. However, it is hard for new entrants to sell as many chips as the existing companies. Therefore, the new entrants have a critical cost disadvantage and thus the effect should be weak. Secondly, the substitute of a router chip is its software counterpart. Nowadays routers are a combination of software and hardware so as to fill in the shortcomings of each other. For example, Broadcom's Trident II ASIC switch is currently being used as top-of-rack switch configuration in Facebook's Wedge and FBOSS. Wedge is the physical hardware of the top-of-rack switch and FBOSS is the software agent that controls the ASIC (Simpkins, 2014). Therefore, the effect of the substitute software should be weak. Thirdly, the bargaining power of suppliers (the chip manufacturing companies) and the customers (warehouse-scale data centers) are quite strong since they don't come in high volume.

Finally, the rivals of our products are the products from existing network companies such as Cisco, Juniper and Broadcom. Since these companies are already firmly established in the networking landscape, the force of rivalry is strong. Fortunately, these companies are providing products with strong features instead of strong cost advantage, which may not have a great impact on the market price. For example, Broadcom announced the StrataXGS Tomahawk™ Series in September of 2014. This chip is used for Ethernet switch for cloud-

scale network and the promised bandwidth is 3.2 terabits per second (Broadcom, 2014). This product can support from 32 to 128 ports based on the speed of Ethernet, and the data transfer rate of the data center network can be largely improved while keeping the same cabling complexity and equipment footprint (Broadcom, 2014). This is a good example of competitors with powerful features.

After considering these five forces, we can see that except the rivalry force, we have two strong and two weak forces. As for rivalry, the strong force towards feature usually improves the profitability of the industry. However, our product will be a new entrant, which is determined as a disadvantage previously. In general, the profitability of our product should be on average level since the five forces are almost balanced. Based on the profit trend convention of rivalry above, we should focus on developing strong features to further improve the overall profit. Meanwhile, since it will be hard for us to compete with the existing strong rivals on all kinds of features, we should first concentrate on a niche market and design our product with few special features.

3. Technical Strategy

As mentioned previously, there is a clear indication in the current trend that enterprises and consumers alike are moving towards cloud services and solutions. A little more than a decade ago however, this trend was less obvious and most companies were still using localized servers with switches and routers they are managed individually (Morgan, 2015). Google, a pioneer in distributed computing and data processing, was the only company that foresaw the need of transformative networking technology required by the increasingly powerful computing infrastructure. Indeed, for the past decade or so, Google has been developing and deploying its own networking infrastructures to complement the computing power required from Google's large-scale cluster architecture starting from Google File System in 2002 to Spanner in 2012.

Armin Vahdat, the technical lead for networking at Google, succinctly described this mutual dependency between network and computing in his keynote in ONS 2015: "Networking is an inflection point and what computing means is going to be largely determined by our ability to build great networks over the coming years (2015)". By discovering before everybody else that traditional network was not able to scale up to meet the computing requirements in the near future and proactively improving and transforming their network infrastructure in response to the growing bandwidth demands from their servers, Google was able to become one of the biggest players in the computing industry today.

With the advancement of memory technology – for example, the 3D XPoint nonvolatile memory that offers up to 1,000 times the speed and up to 10 times the storage (Intel, 2015) – playing a major role in the future scene of datacenters, it is imperative for the networking technology to evolve even further than before. Vahdat has predicted in his keynote that a 5 Petabit per second network, in comparison to the Gigabit per second network commercially available today, may be needed in the near future (2015). Currently, Google's latest-generation network Jupiter employs high-radix switch with 128 ports and 40 Gigabytes per port, allowing it to deliver 1.3 Petabit per second (Singh et al, 2015). In light of the successful deployment of high-radix switch from Google and Vahdat's foresight on networking trend, our team pursues to find the optimal high-radix router architecture that enables data to be communicated at the Petabit level and beyond.

4. Marketing

Fast router technology has ample opportunities in the tech market because it addresses the need for fast and efficient network infrastructure. This section assesses the success of our router technology in the market by applying the 4P marketing analysis which considers four main aspects of go-to-market elements: price, product, promotion and place.

One can find routers being used in almost all digital systems where there are at least two endpoints that can communicate with each other. However, as a new entrant, it is important to find a specific niche market in which our product best fits. According to Andre Barroso, the manufacturing cost is directly proportional to the number of radix (Andre Barroso, 2013). The increased cost means that our product will be an enterprise, business-to-business product rather than a commodity sold directly to consumers. Moreover, companies such as Broadcom, Cisco and Juniper are already dominant in the networking world, thus making it a difficult process for us as new entrant to compete. As previously mentioned, our router technology is designed to enable fast and efficient communication between large collections of machines in computing centers. Therefore, it may be in our best interest to zoom in our market focus to companies such as Google and Facebook that house homegrown warehouse-scale datacenters. Moreover, in recent years many major players on par with Google and Facebook have starting to develop their own data servers, thus forming a growing pool of demand for robustness and efficiency in the underlying networking infrastructures.

Since our market segment is quite narrow and our product fits business-to-business commerce the most, our distribution channel should just be a team of professional salespeople that are highly familiar and experienced in this market. Therefore, the appropriate promotion strategy is definitely not huge-scale advertisement; rather, if our technology is exactly what Google or Facebook is looking for, their adoption of our product will publicize it to other potential customers. Another common way for new techs to raise awareness is by showcasing them at technology trade shows such as Consumer Electronic Show. Linksys and Netgear – companies that sells data networking hardware products – for example have seen huge success in CES with announcements of new generation of routers.

In general, as we determined our product as a business-to-business one, we will first focus our market on big companies such as Google and Facebook who need router

technology for their datacenters. As a new entrant, we will keep track on what our competitors are doing, and specialize in our feature – using high radix to achieve high speed. Once we succeed in our first target market, we plan to promote our product to a broader potential market to gain more recognition by publicizing the product through existing consumers and showcasing the product in Electronic Show.

Referenced

Andre Barroso, Luiz, Jimmy Calidaras, Urs Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.*" Morgan & Claypool Publishers, 2013.

Bachrach, Jonathan, Asanovi´c, Krste and Wawrzynek, John. *Chisel 2.2 Tutorial.* July 10, 2015. https://chisel.eecs.berkeley.edu/2.2.0/getting-started.html. Accessed March 20, 2016.

Becker, Daniel. *Efficient microarchitecture for network-on-chip routers*. Doctoral dissertation submitted to Stanford University. August 2012.

Benini, Luca and Giovanni De Micheli. "The Challenges of Next-gen Multicore Networks-on-Chip Systems." n.p. 26 Feb. 2007 <http://www.embedded.com/design/mcus-processors-and-socs/4006822/The-challenges-of-next-gen-multicore-networks-on-chip-systems-Part-4>

Broadcom. Broadcom Delivers Industry's First High-Density 25/100 Gigabit Ethernet Switch for Cloud-Scale Networks. Press Release. n.p., 24 Sept. 2014. Web. 1 Mar. 2015. <http://www.broadcom.com/press/release.php?id=s872349>.

Dally, William, and Towles, Brian. *Principles and Practices of Interconnection Networks.* San Francisco: Morgan Kaufmann Publishers, 2004.

Dell. "Dell Private Cloud Solutions*".* www.dell.com. n.d. Accessed 4 March 2016

Fatollahi-Fard, Farzad, Donofrio, David, Michelogiannakis, George, and Shalf, John. Lawrence Berkeley National Laboratory. *OpenSoCFabric: On-chip network generator*. Jun 22, 2015. Accessed March 20, 2016. <https://github.com/LBL-CoDEx/OpenSoCFabric/wiki>

Hassan, Qusay. *"Demystifying Cloud Computing". The Journal of Defense Software Engineering* (CrossTalk) (2011 Jan/Feb): 16–21. Retrieved 4 March 2016

Jiang, Nan, Becker, Daniel, Michelogiannakis, George, Balfour, James, and Towles, Brian[…]. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

Lo, Jen-hung. "Technical Contribution – Petabit Switch Fabric Design" Capstone Project Report. Berkeley, CA. 2016

Porter. *"The Five Competitive Forces That Shape Strategy".* hbr.org. January 2008. Accessed 4 March 2016

Simpkins, Adam. "Facebook Open Switching System ("FBOSS") and Wedge in the Open." n.p. 10 Mar. 2014. <https://code.facebook.com/posts/843620439027582/facebook-open-switching-system-fboss-and-wedge-in-the-open/>

Singh, Arjun, et al. “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network”. In SIGCOMM (2015)

Vahdat, Amin. “A Look Inside Google’s Data Center Network.” Open Networking Summits 2015. Santa Clara Convention Center, Santa Clara. 17 Jun. 2015. Keynote.

Zhou, Jingxue. “Technical Contribution – Petabit Switch Fabric Design” Capstone Project Report. Berkeley, CA. 2016

Appendix Chisel code for arbiters and testbenches

```
package OpenSoC

import Chisel._
import scala.collection.mutable.HashMap
import scala.util.Random

/*This classes are from OpenSoC*/
class RequestIO(val parms:Parameters) extends Bundle {
   val numPriorityLevels = parms.get[Int]("numPriorityLevels")
   val releaseLock = Bool(OUTPUT)
   val grant = Bool(INPUT)
   val request = Bool(OUTPUT)
   val priorityLevel = UInt(OUTPUT,
width=log2Up(numPriorityLevels))
   override def clone = { new
RequestIO(parms).asInstanceOf[this.type] }
}

/*
class RequestIO extends Chisel.Bundle {
   val releaseLock = Bool(OUTPUT)
   val grant = Bool(INPUT)
   val request = Bool(OUTPUT)
   val prioritylevel = UInt(OUTPUT, width=3)
}
*/

class ResourceIO extends Chisel.Bundle {
   val ready = Bool(INPUT)
   val valid = Bool(OUTPUT)

}

abstract class Arbiter(parms: Parameters) extends Module(parms) {
   val numReqs = parms.get[Int]("numReqs")
   val io = new Bundle {
         val requests = Vec.fill(numReqs){ new
RequestIO(parms) }.flip
         val resource = new ResourceIO
         val chosen = UInt(OUTPUT, Chisel.log2Up(numReqs))
         }
}

/*Start of original implementation*/
// /////////////////Algorithm for Arbiter without priority
////////////////////////

class CLArbiter(parms: Parameters) extends Arbiter(parms) {
   val nextGrant = Chisel.Reg(init=UInt( (1 << (numReqs-1)) ,
width=numReqs))

   val requestsBits = Cat( (0 until
numReqs).map(io.requests(_).request.toUInt() ).reverse )
```

```
      val passSelectL1 = UInt(width = numReqs)
      val winner = UInt(width = numReqs)
    passSelectL1 := UInt(0)
    winner := UInt(0)

    val nextGrantUInt = UInt(width = log2Up(numReqs))
    nextGrantUInt := Chisel.OHToUInt(nextGrant)
    val lockRelease = Bool()
    lockRelease := io.requests(nextGrantUInt).releaseLock

    when ( nextGrant(nextGrantUInt) && requestsBits(nextGrantUInt)
    && ~lockRelease ) {
        /* If Locked (i.e. request granted but still requesting)
        make sure to keep granting to that port */
        winner := nextGrant
    } .otherwise {
        /* Otherwise, select next requesting port */
        passSelectL1 := ~requestsBits + Bool(true).toUInt
        winner := passSelectL1 & requestsBits
        nextGrant := Mux(orR(winner), winner, nextGrant)
    }

      (0 until numReqs).map(i => io.requests(i).grant :=
   winner(i).toBool() && io.resource.ready)
      io.chosen := Chisel.OHToUInt(winner)
      io.resource.valid := io.requests(io.chosen).grant &&
   io.resource.ready //(winner != UInt(0))
}

   /* Matrix Arbiter */
   class MatrixArbiter(parms: Parameters) extends Arbiter(parms) {

      val nextGrant = Chisel.Reg(init=UInt( (1 << (numReqs-1)) ,
   width=numReqs))

      val winGrant = UInt(width=numReqs)
      winGrant := UInt(1<<(numReqs-1))

      val requestsBits = Cat( (0 until
   numReqs).map(io.requests(_).request.toUInt() ).reverse )

      //pMatrix(i)(j) represents the element in column i, row j
     val  pMatrix  = Vec.fill(numReqs){Vec.fill(numReqs){Reg(Bool())}}
      val pMatrix_n = Vec.fill(numReqs){Vec.fill(numReqs){Bool()}}

      //initialization
      for (i <- 0 until numReqs) {
            for (j <- 0 until numReqs){
                  pMatrix_n(i)(j) := pMatrix(i)(j)
            }
      }

      //disable signal, each element for one request
      val disableSig = Vec.fill(numReqs){Vec.fill(numReqs){Bool()}}
      val disableSig_c = Vec.fill(numReqs){Bits(width = numReqs)}

      //connection for disableSig
```

```
    for (i <- 0 until numReqs) {
          for (j <- 0 until numReqs){
                disableSig(i)(j) := requestsBits(j) & pMatrix(j)(i)
          }
    }

    //final disable signal
    val dis = Vec.fill(numReqs){Bool()}

    for(i <- 0 until numReqs){
          disableSig_c(i) := disableSig(i).toBits
          dis(i) := orR(disableSig_c(i))
    }


    val winner    = UInt(width=numReqs)
    val winner_b      = winner.toBits

  winner := UInt(0)

  val nextGrantUInt = UInt(width = log2Up(numReqs))
  nextGrantUInt := Chisel.OHToUInt(nextGrant)
  val lockRelease = Bool()
  lockRelease := io.requests(nextGrantUInt).releaseLock

    when (orR(requestsBits)) {
          /* Locking logic encapsulating matrix logic */
          when (nextGrant(nextGrantUInt) &&
requestsBits(nextGrantUInt) && ~lockRelease ) {
                /* If Locked (i.e. request granted but still
requesting)
                      make sure to keep granting to that port */
                winGrant  := nextGrant
                nextGrant := winGrant
          }
          .otherwise {

                winner := (requestsBits & ~dis.toBits).toUInt
                //Matrix update
                when(orR(winner)){
                      for(i <- 0 until numReqs){
                            when(winner_b(i).toBool){
                                  for(j <- 0 until numReqs){
                                        when(UInt(i) != UInt(j)){
                                              pMatrix_n(i)(j) :=
Bool(false)
                                              pMatrix_n(j)(i) :=
Bool(true)
                                  }
                            }
                      }
                }
          }

                winGrant  := Mux(orR(winner), winner, nextGrant)
                nextGrant := winGrant
          }
```

```
     }.otherwise{
            winGrant  := requestsBits
            nextGrant := requestsBits
    }


      (0 until numReqs).map(i => io.requests(i).grant :=
   winGrant(i).toBool() && io.resource.ready)
      io.chosen := Chisel.OHToUInt(winGrant)
      io.resource.valid := io.requests(io.chosen).grant &&
   io.resource.ready //(winner != UInt(0))

      when(reset){
            for(i <- 0 until numReqs){ //start with 0 is highest
   priority
                  for(j <- i+1 until numReqs){
                        pMatrix(i)(j) := Bool(true)
                  }
                  for(j <- 0 until i+1){
                        pMatrix(i)(j) := Bool(false)
                  }
            }
    }.otherwise{
            when(io.resource.valid){
                  //printf("update\n")
                  pMatrix := pMatrix_n //update only if served

            }
    }
}

// //////////////////Algorithm for Arbiter with priority
////////////////////////
class CLArbiterPriority(parms: Parameters) extends Arbiter(parms) {

      val numPriorityLevels = parms.get[Int]("numPriorityLevels")
      val nextGrant = Chisel.Reg(init=UInt(1 , width=numReqs))
   // val nextGrant = UInt(width=numReqs)
   // nextGrant := UInt(1<<(numReqs-1))

      val winGrant = UInt(width=numReqs)
      winGrant := UInt(1)

      val requestsBits = Cat( (0 until
   numReqs).map(io.requests(_).request.toUInt() ).reverse )
      val PArraySorted =
   Vec.fill(numPriorityLevels){Vec.fill(numReqs){Reg(init=UInt(0,wid
   th=1))}}

      val passSelectL1 = UInt(width = numReqs + 1)

      val winner = UInt(width=numReqs)
      val pmax = UInt(width = log2Up(numReqs))

      passSelectL1 := UInt(0)
     winner := UInt(0)
      pmax := UInt(0)
```

```
 val nextGrantUInt = UInt(width = log2Up(numReqs))
 nextGrantUInt := Chisel.OHToUInt(nextGrant)
   val lockRelease = Bool()
 lockRelease := io.requests(nextGrantUInt).releaseLock

   when (orR(requestsBits)) {
         /* Locking logic encapsulating Round Robin logic */
         when ( nextGrant(nextGrantUInt) &&
requestsBits(nextGrantUInt) && ~lockRelease ) {
               /* If Locked (i.e. request granted but still
requesting)
               make sure to keep granting to that port */
         winGrant  := nextGrant
         nextGrant := winGrant
}
   .otherwise {
         //To refresh the Sorted Array
         for (i <- 0 until numPriorityLevels){
               PArraySorted(i) := UInt(0)
         }

         //To store the given priorities and the requesting ports
in a sorted array
         for (i <- 0 until numReqs) {
               for (j <- 0 until numPriorityLevels) {
                     when (io.requests(i).priorityLevel ===
UInt(numPriorityLevels-1-j)) {
                           when (requestsBits(i)){
                           PArraySorted(numPriorityLevels-1-j)(i) :=
UInt(1)
                           }
                     }
               }
         }

         //To find which is the max priority level available
         for (i <- 0 until numPriorityLevels) {
               for (j <- 0 until numReqs){
                     when(PArraySorted(i)(j) === UInt(1)) {
                           pmax := UInt(i)
                     }
               }
         }

         //To find the port number having this max priority
         passSelectL1 := (~PArraySorted(pmax).toBits) +
Bool(true).toUInt
         winner := passSelectL1 & (PArraySorted(pmax).toBits)

         winGrant  := Mux(orR(winner), winner, nextGrant)
         nextGrant := winGrant
   }
 }
   (0 until numReqs).map(i => io.requests(i).grant :=
winGrant(i).toBool() && io.resource.ready)
   io.chosen := Chisel.OHToUInt(winGrant)
```

```
   io.resource.valid := io.requests(io.chosen).grant &&
io.resource.ready //(winner != UInt(0))

}

class MatrixArbiterPriority(parms: Parameters) extends
Arbiter(parms) {

   val nextGrant = Chisel.Reg(init=UInt( 1 , width=numReqs))

   val winGrant = UInt(width=numReqs)
   winGrant := UInt(1<<(numReqs-1))

   val numPriorityLevels = parms.get[Int]("numPriorityLevels")

   val PArraySorted =
Vec.fill(numPriorityLevels){Vec.fill(numReqs){Reg(init=UInt(0,wid
th=1))}}

   val pmax         = UInt(width = log2Up(numReqs))

   pmax := UInt(0)

   val requestsBits = Cat( (0 until
numReqs).map(io.requests(_).request.toUInt() ).reverse )

   //pMatrix(i)(j) represents the element in column i, row j
  val pMatrix = Vec.fill(numReqs){Vec.fill(numReqs){Reg(Bool())}}
   val pMatrix_n = Vec.fill(numReqs){Vec.fill(numReqs){Bool()}}

   //initialization
   for (i <- 0 until numReqs) {
         for (j <- 0 until numReqs){
               pMatrix_n(i)(j) := pMatrix(i)(j)
         }
   }

   //disable signal, each element for one request
   val disableSig = Vec.fill(numReqs){Vec.fill(numReqs){Bool()}}

   //connection for disableSig
   for (i <- 0 until numReqs) {
         for (j <- 0 until numReqs){
               disableSig(i)(j) := Bool(true)
         }
   }

   //final disable signal
   val dis = Vec.fill(numReqs){Bool()}

   for(i <- 0 until numReqs){
         dis(i) := Bool(true)
   }



   val winner       = UInt(width=numReqs)
```

```
   winner := UInt(0)

   val nextGrantUInt = UInt(width = log2Up(numReqs))
   nextGrantUInt := Chisel.OHToUInt(nextGrant)
   val lockRelease = Bool()
   lockRelease := io.requests(nextGrantUInt).releaseLock

   when (orR(requestsBits)) {
         /* Locking logic encapsulating matrix logic */
         when (nextGrant(nextGrantUInt) &&
requestsBits(nextGrantUInt) && ~lockRelease ) {
               /* If Locked (i.e. request granted but still
requesting)
                     make sure to keep granting to that port */
               winGrant  := nextGrant
               nextGrant := winGrant
         }
         .otherwise {

               //To refresh the Sorted Array
         for (i <- 0 until numPriorityLevels){
               PArraySorted(i) := UInt(0)
         }

         //To store the given priorities and the requesting ports
in a sorted array
                  for (i <- 0 until numReqs) {
               for (j <- 0 until numPriorityLevels) {
                     when (io.requests(i).priorityLevel ===
UInt(numPriorityLevels-1-j)) {
                           when (requestsBits(i)){
                           PArraySorted(numPriorityLevels-1-j)(i) :=
UInt(1)
                           }
                     }
               }
         }

         //To find which is the max priority level available
         for (i <- 0 until numPriorityLevels) {
               for (j <- 0 until numReqs){
                     when(PArraySorted(i)(j) === UInt(1)) {
                           pmax := UInt(i)
                     }
               }
         }

         for (i <- 0 until numReqs) {
               for (j <- 0 until numReqs){
                     disableSig(i)(j) := PArraySorted(pmax).toBits
& pMatrix(j)(i)
               }
         }

         for(i <- 0 until numReqs){
               dis(i) := orR(disableSig(i).toBits)
```

```
        }


        winner := (PArraySorted(pmax).toBits &
~dis.toBits).toUInt
        //printf("Not locked, winner:%b\n", winner_b)

        //Matrix update
        when(orR(winner)){
             for(i <- 0 until numReqs){
                  when(winner(i).toBool){
                       //printf("update" + i + "\n")
                       for(j <- 0 until numReqs){
                            when(UInt(i) != UInt(j)){
                                 pMatrix_n(i)(j) := Bool(false)
                                 pMatrix_n(j)(i) := Bool(true)
                            }
                       }
                  }
             }
        }

             winGrant  := Mux(orR(winner), winner, nextGrant)
             nextGrant := winGrant
        }
    }.otherwise{
             winGrant  := requestsBits
             nextGrant := requestsBits
    }


   (0 until numReqs).map(i => io.requests(i).grant :=
winGrant(i).toBool() && io.resource.ready)
   io.chosen := Chisel.OHToUInt(winGrant)
   io.resource.valid := io.requests(io.chosen).grant &&
io.resource.ready //(winner != UInt(0))

nextGrant: %d\n", winner, winGrant, io.chosen, nextGrant)
   when(reset){
        for(i <- 0 until numReqs){ //start with 0 is highest
priority
             for(j <- i+1 until numReqs){
                  pMatrix(i)(j) := Bool(true)
             }
             for(j <- 0 until i+1){
                  pMatrix(i)(j) := Bool(false)
             }
        }
    }.otherwise{
        when(io.resource.valid){
             pMatrix := pMatrix_n //update only if served

          }
    }

}
```

```
//tester for matrix arbiter
class MatrixArbiterTest(c: MatrixArbiter) extends Tester(c) {
   implicit def bool2BigInt(b:Boolean) : BigInt = if (b) 1 else 0

   def noting(i: Int) : Int = if (i == 1) 0 else 1

   val numPorts : Int = c.numReqs

   val inputArray = Array(
         Integer.parseInt("00000001", 2),
         Integer.parseInt("00000001", 2),
         Integer.parseInt("00000001", 2),
         Integer.parseInt("00000000", 2),
         Integer.parseInt("00100001", 2), //what's wrong for
00100001? Possible reason: matrix updates
         Integer.parseInt("00010100", 2),
         Integer.parseInt("10011110", 2),
         Integer.parseInt("11100000", 2),
         Integer.parseInt("00001001", 2),
         Integer.parseInt("00110010", 2),
         Integer.parseInt("00000111", 2),
         Integer.parseInt("00001010", 2),
         Integer.parseInt("00001000", 2),
         Integer.parseInt("00011100", 2),
         Integer.parseInt("01100110", 2),
         Integer.parseInt("01010001", 2),
         Integer.parseInt("10111000", 2),
         Integer.parseInt("00111100", 2)
         )
   val lockArray = Array(
         Integer.parseInt("00000000", 2),
         Integer.parseInt("11111110", 2),
         Integer.parseInt("11111110", 2),
         Integer.parseInt("11111111", 2),
         Integer.parseInt("00100000", 2),
         Integer.parseInt("11011111", 2),
         Integer.parseInt("00000100", 2),
         Integer.parseInt("11111011", 2),
         Integer.parseInt("10111111", 2),
         Integer.parseInt("11110111", 2),
         Integer.parseInt("11111101", 2),
         Integer.parseInt("11111110", 2),
         Integer.parseInt("11110111", 2),
         Integer.parseInt("11110111", 2),
         Integer.parseInt("11101111", 2),
         Integer.parseInt("11011111", 2),
         Integer.parseInt("10111111", 2),
         Integer.parseInt("01111111", 2),
         Integer.parseInt("11111011", 2),
         Integer.parseInt("00000000", 2)
   )
   val outputArray = Array(
         Integer.parseInt("00000000", 2),
         Integer.parseInt("00000001", 2),
         Integer.parseInt("00000000", 2),
         Integer.parseInt("00000000", 2),
```

```
        Integer.parseInt("00100000", 2),
        Integer.parseInt("00000100", 2),
        Integer.parseInt("00000100", 2),
        Integer.parseInt("01000000", 2),
        Integer.parseInt("00001000", 2),
        Integer.parseInt("00000010", 2),
        Integer.parseInt("00000001", 2),
        Integer.parseInt("00001000", 2),
        Integer.parseInt("00001000", 2),
        Integer.parseInt("00010000", 2),
        Integer.parseInt("00100000", 2),
        Integer.parseInt("01000000", 2),
        Integer.parseInt("10000000", 2),
        Integer.parseInt("00000000", 2)
        )
   val chosenPort = Array(0, 0, 0, 0, 5, 2, 2, 6, 3, 1, 0, 3, 3,
4, 5, 6, 7, 2)
   val resourceReady = Array(false, true, false, false, true,
true, true, true, true, true, true, true, true, true, true, true,
true, false)
   val cyclesToRun = 18

   step(1)

   for (cycle <- 0 until cyclesToRun) {
        poke(c.io.resource.ready, resourceReady(cycle))
        println("inputArray(" + cycle + "): " +
inputArray(cycle).toBinaryString)
        for ( i <- 0 until numPorts) {
              poke(c.io.requests(i).request, (inputArray(cycle) &
(1 << i)) >> i)
              poke(c.io.requests(i).releaseLock,
noting((lockArray(cycle) & (1 << i)) >> i))
        }
        step(1)
        expect(c.io.chosen, chosenPort(cycle))
        expect(c.io.resource.valid, resourceReady(cycle) &&
inputArray(cycle) != 0)
        for ( i <- 0 until numPorts) {
              expect(c.io.requests(i).grant, (outputArray(cycle)
& (1 << i)) >> i)
        }

   }
   step(1)
}

class CLArbiterTest(c: CLArbiter) extends Tester(c) { //for
highest priority is 7
   implicit def bool2BigInt(b:Boolean) : BigInt = if (b) 1 else 0

   def noting(i: Int) : Int = if (i == 1) 0 else 1

   val numPorts : Int = c.numReqs

   val inputArray = Array(
        Integer.parseInt("00000001", 2),
```

```
      Integer.parseInt("00000001", 2),
      Integer.parseInt("00000001", 2),
      Integer.parseInt("00000000", 2),
      Integer.parseInt("00100001", 2),
      Integer.parseInt("00000100", 2),
      Integer.parseInt("10001000", 2),
      Integer.parseInt("00000000", 2),
      Integer.parseInt("00000001", 2),
      Integer.parseInt("00000010", 2),
      Integer.parseInt("01100100", 2),
      Integer.parseInt("00001000", 2),
      Integer.parseInt("00001000", 2),
      Integer.parseInt("00010000", 2),
      Integer.parseInt("00100100", 2),
      Integer.parseInt("01000000", 2),
      Integer.parseInt("11100000", 2),
      Integer.parseInt("00000000", 2)
      )
val lockArray = Array(
      Integer.parseInt("00000000", 2),
      Integer.parseInt("11111110", 2),
      Integer.parseInt("11111111", 2),
      Integer.parseInt("11111111", 2),
      Integer.parseInt("11111110", 2),
      Integer.parseInt("11111110", 2),
      Integer.parseInt("11111011", 2),
      Integer.parseInt("11110111", 2),
      Integer.parseInt("11111111", 2),
      Integer.parseInt("11111110", 2),
      Integer.parseInt("11111101", 2),
      Integer.parseInt("11111011", 2),
      Integer.parseInt("11111111", 2),
      Integer.parseInt("11110111", 2),
      Integer.parseInt("11101111", 2),
      Integer.parseInt("11111011", 2),
      Integer.parseInt("10111111", 2),
      Integer.parseInt("11011111", 2),
      Integer.parseInt("11111111", 2),
      Integer.parseInt("11111111", 2)
)
val outputArray = Array(
      Integer.parseInt("00000000", 2),
      Integer.parseInt("00000001", 2),
      Integer.parseInt("00000000", 2),
      Integer.parseInt("00000000", 2),
      Integer.parseInt("00000001", 2),
      Integer.parseInt("00000100", 2),
      Integer.parseInt("00001000", 2),
      Integer.parseInt("00000000", 2),
      Integer.parseInt("00000001", 2),
      Integer.parseInt("00000010", 2),
      Integer.parseInt("00000100", 2),
      Integer.parseInt("00001000", 2),
      Integer.parseInt("00001000", 2),
      Integer.parseInt("00010000", 2),
      Integer.parseInt("00000100", 2),
      Integer.parseInt("01000000", 2),
```

```
          Integer.parseInt("00100000", 2),
          Integer.parseInt("00000000", 2)
          )
    val chosenPort = Array(0, 0, 0, 0, 0, 2, 3, 0, 0, 1, 2, 3, 3,
  4, 2, 6, 5, 0)
   val resourceReady = Array(false, true, false, true, true, true,
  true, true, true, true, true, true, true, true, true, true, true,
  false)
    val cyclesToRun = 18

    step(1)

    for (cycle <- 0 until cyclesToRun) {
          poke(c.io.resource.ready, resourceReady(cycle))
          println("inputArray(cycle): " +
  inputArray(cycle).toBinaryString)
          for ( i <- 0 until numPorts) {
                poke(c.io.requests(i).request, (inputArray(cycle) &
  (1 << i)) >> i)
                poke(c.io.requests(i).releaseLock,
  noting((lockArray(cycle) & (1 << i)) >> i))
          }
          step(1)
          expect(c.io.chosen, chosenPort(cycle))
          expect(c.io.resource.valid, resourceReady(cycle) &&
  inputArray(cycle) != 0)
          for ( i <- 0 until numPorts) {
                expect(c.io.requests(i).grant, (outputArray(cycle)
  & (1 << i)) >> i)
          }
    }
    step(1)
  }
```

Modications inside main.scala:

In moduleName:

```
case "MatrixArbiter" => (moduleToTest =()=> Module(new
MatrixArbiter(
      parms.child("MyMatrixArbiter", Map(
            ("numReqs"->Soft(8)),
      ("numPriorityLevels"->Hard(8))
            )
      ))
   ))
   case "CLArbiter" => (moduleToTest = () =>  Module(new CLArbiter(
      parms.child("MyCLArbiter", Map(
            ("numReqs"->Soft(8)),
            ("numPriorityLevels"->Hard(8))
            )
      ))
   ))
```

In harnessName:

```
case "MatrixArbiterTest"    => ( chiselMainTest(myargs, moduleToTest)
{ c => new MatrixArbiterTest(c.asInstanceOf[MatrixArbiter]) } )
```

```
case "CLArbiterTest"  => ( chiselMainTest(myargs, moduleToTest) { c
=> new CLArbiterTest(c.asInstanceOf[CLArbiter]) } )
```