Petabit Switch Fabric Design

Jen-Hung Lo
Yue Cao
Jingxue Zhou

Acknowledgement

Special thanks to our advisors Vladimir Stojanovic, Elad Alon and our faculty committee member John Wawrzynek. Also many thanks to the graduate students at BWRC who helped us tremendously with our project: Christopher Yarp, Angie Wang, Paul Rigge, Ranko Sredojevic, and Taehwan Kim. Lastly, we would also like to thank Lawrence Berkeley National Lab staffs Farzad Fatollahi-Fard and David Donofrio for allowing us use their OpenSoCFabric as our project baseline and for giving us a presentation on the source code

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

Jen-Hung Lo

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 Contents

Chapter 1: Technical Contribution

I. INTRODUCTION

1.1 Motivation

Interconnection networks are used virtually in almost all digital systems that contain at least two components to connect be it CPU core, memory, or I/O device (Dally, 2004). As digital systems today continue to scale up, the interconnection network becomes one of the major bottleneck that limit the performance of the systems (Dally, 2004). Thus throughput and latency of the network are becoming more relevant than ever before when benchmarking a system’s performance. Thanks to the increase in pin bandwidth due to scaling of pin and wire density, the throughput of the network has been keeping pace with the rate at which systems are scaling. Furthermore, with high pin bandwidth a paradigm shift from low-radix wide-channel switch to high-radix narrow-channel switch can be seen (Abts, 2011). However, increasing the radix of a switch brings about various challenges as internal components such as crossbar switch and allocator scale at a faster rate than the radix (Kim, 2005). Our project, Petabit Switch Fabric Design, aims to sweep the design space of a high radix switch and draw a conclusion as to what the most optimal switch architecture is based on its performance metrics.

1.2 OpenSoC Fabric

The core of our project centers around the open source switch fabric code, named OpenSoC Fabric, implemented by Lawrence Berkeley National Laboratory (LBNL). OpenSoC Fabric was developed with the purpose of providing users a “complete and power on-chip network generator for evaluating future large-scape chip multiprocessors and system on chip” (Fatollahi-Fard at el, 2015). OpenSoC Fabric is implemented entirely in Chisel – which is a hardware construction language developed at UC Berkeley – and Scala. The primary use case for Chisel is describing diverse and highly parameterized hardware generators that traditional hardware description language such as Verilog lack (Izraelevitz, 2015). Moreover, Chisel can generate a software cycle-accurate simulator in C++ for verification as well as Verilog file that can be pushed through the ASIC or the FPGA flow (Bachrach at el, 2012). Leveraging the parameterization feature that Chisel offer, LBNL created a network generator that can be instantiated with user-defined parameters. Our project goal is to extract a single switch from LBNL's generator, implement other variants of the same blocks found within a the switch, adjust the parameters found within the design space and examine how these modifications may help or hinder the performance under the context of high-radix switch.

II. Switch in Literature

Our primary source of literature review is William Dally's book *Principles and Practices of Interconnection Networks*. The book describes the various sub-components within a switch. The main blocks of interest – namely virtual channels, routing computation, VC allocator, switch allocator and crossbar switch – are illustrated in high-level block diagram shown in Figure 1. In the next couple of sections, we introduce some essential switch terminologies – such as message, packet, flit and virtual channel – before dedicating the rest of the sections to each of the block's functional descriptions and architecture.

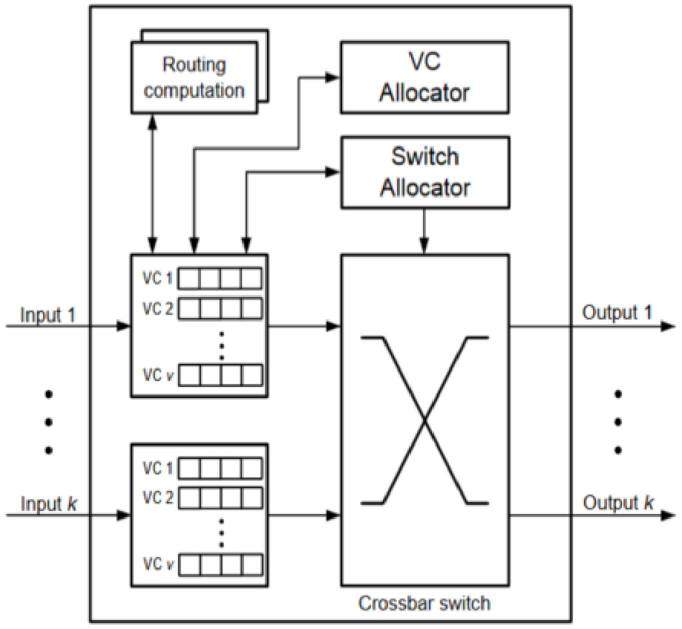


Figure 1. Router microarchitecture (Kim, 2005)

2.1 *Message, Packet and Flit*

Message, packet and flit are units of data transfer in which the network resources are allocated for. In Figure 2, message is the largest unit of transfer and it is usually sent from a source terminal such as processors or memories to the network. Since message can be arbitrarily large, message is broken down into several fix-sized packets. However, a packet can still be rather large compared to a port width of a narrow-channel high-radix switch, so a packet is further divided into flits. There are two kinds of flits within a packet: a head flit and a body flit. The head flit, as its name suggested, is the first flit of the packet and it contains information such as the packet destination, packet type, and packet ID. Body flits, on the other hand, are flits that follow the head flit and they contain the same packet type and packet ID as the head flit and additionally they also carry the packet payload. In our switch design, the resources are allocated for flits.

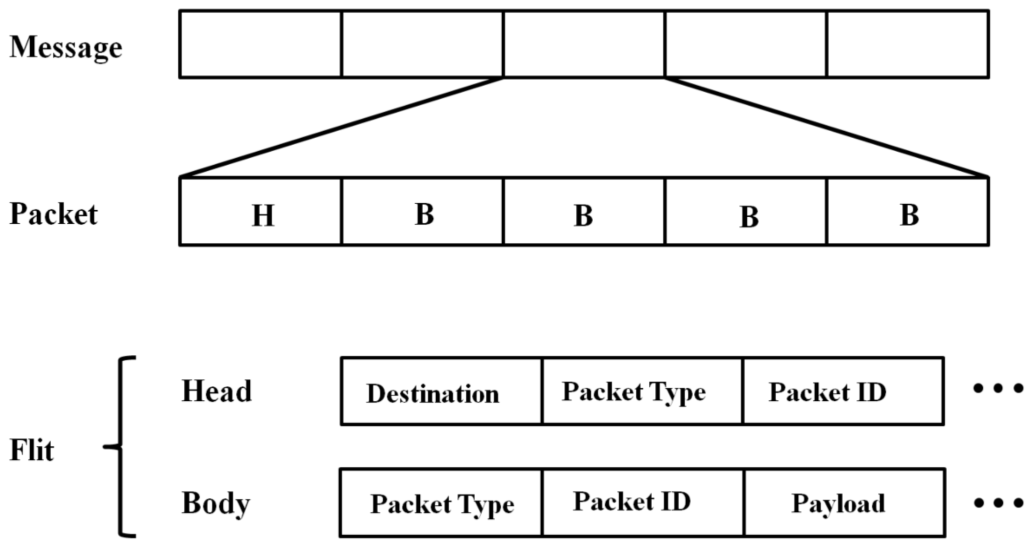


Figure 2. Message, packet and flit structures.

2.2 *Virtual Channel*

In Figure 1 the switch is shown to have *k* input ports on the left side and *k* output ports on the right side. Associated with these ports are *v* numbers of virtual channels, or VC. The idea of virtual channel is that when there are multiple different flits coming into Input 1, for example, these flits can be buffered in different virtual channels while they are waiting to be serviced. On the flip side, if there is only one virtual channel, then a single flit can potentially idle on the sole channel while other flits also wanting to enter the same port will be stuck in the buffer behind the idling flit. Hence, the benefit of having multiple virtual channels is to prevent the active flits from being obstructed by an idle flit as much as possible. Although it is not shown in Figure 1, virtual channels can also reside on the output ports. Physically, virtual channels are fixed-sized buffers and it is up to the designer to partition them as shown in Figure 3. Since the partitioning is user-defined, the number of virtual channels makes a good parameter choice in the design space.

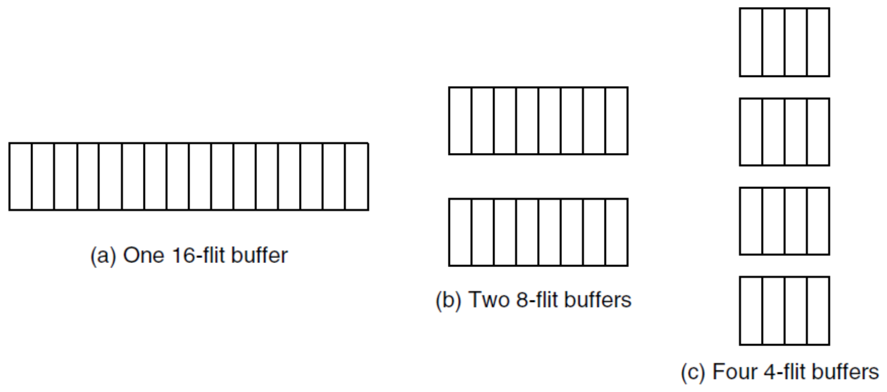


Figure 3. Buffer storage for 1, 2, and 4 virtual channels with fixed buffer size of 16 flits (Dally, 2004).

2.3 Routing Function

After the flit is buffered in one of the virtual channels, the output port through which the flit should go to needs to be determined. This is where routing function comes in. Routing function is an algorithm that determines the switch output that a flit should take based on the destination address embedded within the head flit. There are several approaches to implementing a routing function. Two implementations are considered in this paper.

One implementation is the dimension order routing (DOR). The idea is that the flit is first routed in a lower dimension to the correct coordinate before it is routed towards the next dimension (Yu, 2015). In a 3-dimensional mesh, we can define the DOR to first route in the X-dimension, then in the Y-dimension, and finally in the Z-dimension. Consider routing from (0,0,0) to (2,2,2) in a 3-D mesh indexed by Cartesian coordinates. The routing function in the switch indexed by (0,0,0) will choose an output port that leads to a neighboring switch indexed by (1,0,0). The routing function in the switch indexed by (1,0,0) will in turn choose the output port that leads to switch indexed by (2,0,0). Since the flit is at the correct location as far as X-dimension is concerned, the next switch is (2,1,0), follow by (2,2,0), (2,2,1), and finally (2,2,2). Figure 4 illustrates the path the flit takes from (0,0,0) to (2,2,2). Note that in a 3-dimensional mesh, the highest radix is limited to 6 (2 radix in each of the dimension). Hence, the dimension of the mesh has to scale up as the radix scales up.

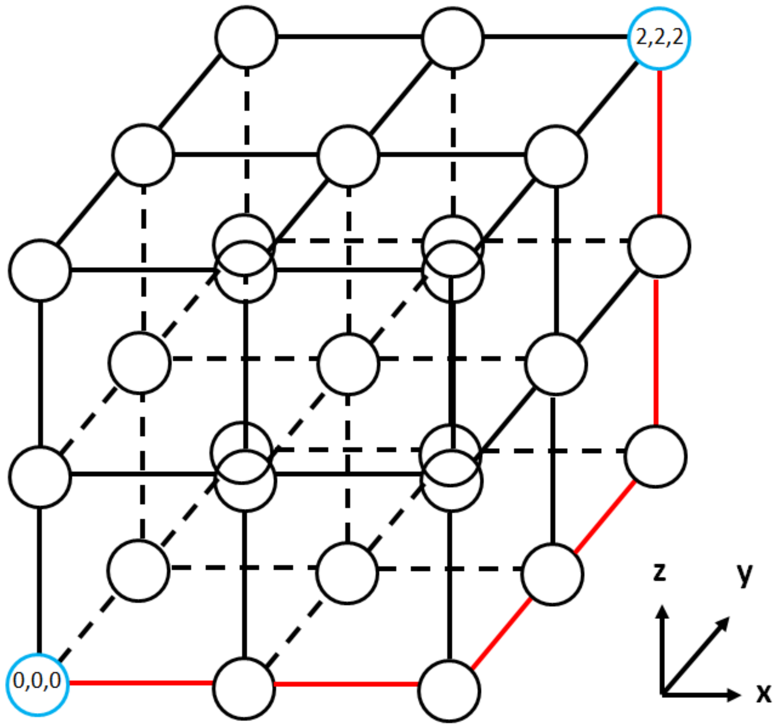


Figure 4. 3-Dimensional mesh with starting point switch indexed in (0,0,0) and ending point in (2,2,2).
The red path is chosen as path of traversal by the dimension order function.

The downside to this approach is that it restrains the network topology to a grid structure which limits the options of switch fabric infrastructure. Furthermore, the scaling of the dimension proportionally lengthens the time it needs to compute the destination port, thus incurring an undesirable critical path in the routing function block.

The second approach to routing function is lookup table and it alleviates the drawbacks experienced by DOR. A lookup table is a data table stored in a switch that lists the key-value pairs of all the addresses and their associated ports. The size of the address space depends on how many computing nodes there are in a network topology. A lookup table implemented in a 4-radix switch with 8 computing nodes in the network is shown in Figure 5. As we can see, address 0 and 6 maps to output port 3, address of 1 and 4 maps to output port 1, and so on. As opposed to the topology constraints imposed by dimension routing function, a lookup table has no such limitation as it is entirely user-defined. It also offers faster route computation than that of dimension order routing at the expense of area and power.

Address	**Output Port**
0	11
1	01
2	00
3	10
4	01
5	00
6	11
7	10

Figure 5. Lookup table for 8x8 switch

The lookup table is implemented simply as a register file that can be populated serially in the initialization process. Each virtual channel in a switch can read from the lookup table, hence there are as many read ports and multiplexors as there are numbers of virtual channels. Figure 6 shows a lookup table with *m* network nodes, *r* radix, and *n* read ports. Note that *n* usually equals to number of radix times number of VC. As radix scales up, the width of the lookup table scales up at the rate of log *r* and the number of multiplexers scales up at the rate of *r*. Therefore, the complexity of the lookup table as a whole scales at *r* log *r* relative to radix.

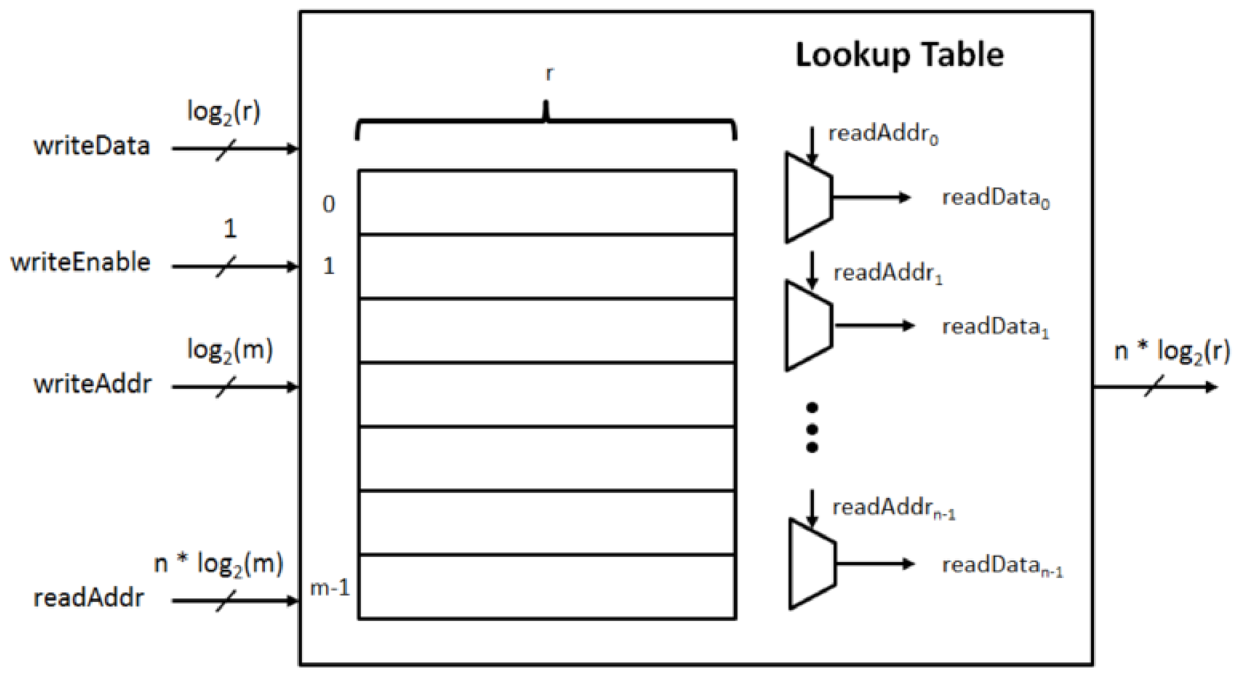


Figure 6. *m*-deep, *r*-wide lookup table with *n* read ports.

2.4 VC Allocator

Once the output port for the flit is determined using the routing function, the flit now needs to requests an output virtual channel from the virtual channel (or VC) allocator (Dally, 2004). There may be several flits from different input ports that are requesting the same output VC. To resolve the contention, it is the role of the VC allocator to allocate the output VC in question and ensure that only one request may obtain the output VC. In terms of implementation, an allocator is consists of several arbiters. An arbiter is a logic block that arbitrates among several requests competing for the same resource. An allocator leverages the functionality of several arbiters in order to achieve an arbitration scheme that arbitrates among several requests competing for a group of resources as opposed to just one resource. Figure 7 shows an allocator with *m* resources and *n*m* requests using *m* arbiters. In an allocator, *n* and *m* represent the number of input and output ports and vice versa. Therefore, scaling up the radix would increase the complexity of the allocator quadratically.

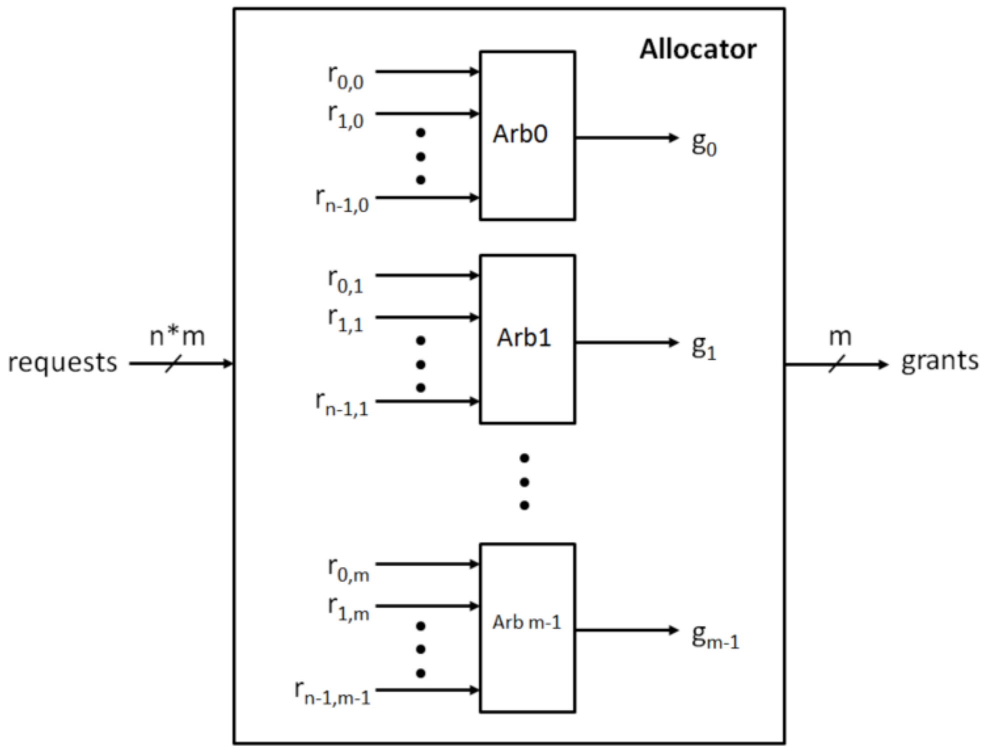


Figure 7. n-by-m allocator.

2.5 *Switch Allocator*

Once an output VC has been allocated by the VC allocator, a route between the input port and the output VC has been determined. The switch allocator's responsibility is to configure the select signals for the multiplexors located after the input VCs as well as the multiplexors within the crossbar switch in order to link together the input VC to output VC. The implementation for switch allocator is exactly the same as that of VC allocator and hence has the same complexity.

2.6 *Crossbar switch*

A crossbar switch connects multiple inputs to multiple outputs. The flit is able to physically traverse from input VC to output VC after the link is established between the I/O by configuring the select signals that control the multiplexors that reside within a crossbar. In Figure 8, it shows a schematic of the n-by-m MUX crossbar switch. This means there are *n* inputs and *m* outputs. Each of input and output ports are a flit wide as denoted by *fw*. Similar to an allocator, a switch grows quadratically as radix scales up. In fact, switch is the second largest block in the design as seen in the result section below. Hence, distributed MUX crossbar, another crossbar switch implementation, is considered. As shown in Figure 9, the idea of distributed MUX crossbar is to break a single MUX in Figure 8 into smaller MUXes in order to relieve some burden from the synthesis tool. As it turns out, distributed MUX crossbar not only offers very little area benefit compared to the regular MUX crossbar, but it also doubled the critical path in the switch. Hence, distributed MUX crossbar will not be further investigated in the rest of the paper. Chisel code for the distributed MUX crossbar can be found in the Appedix. We have also taken a look at Clos network as an alternative to MUX crossbar. Its implementation can also be found in the Appendix.

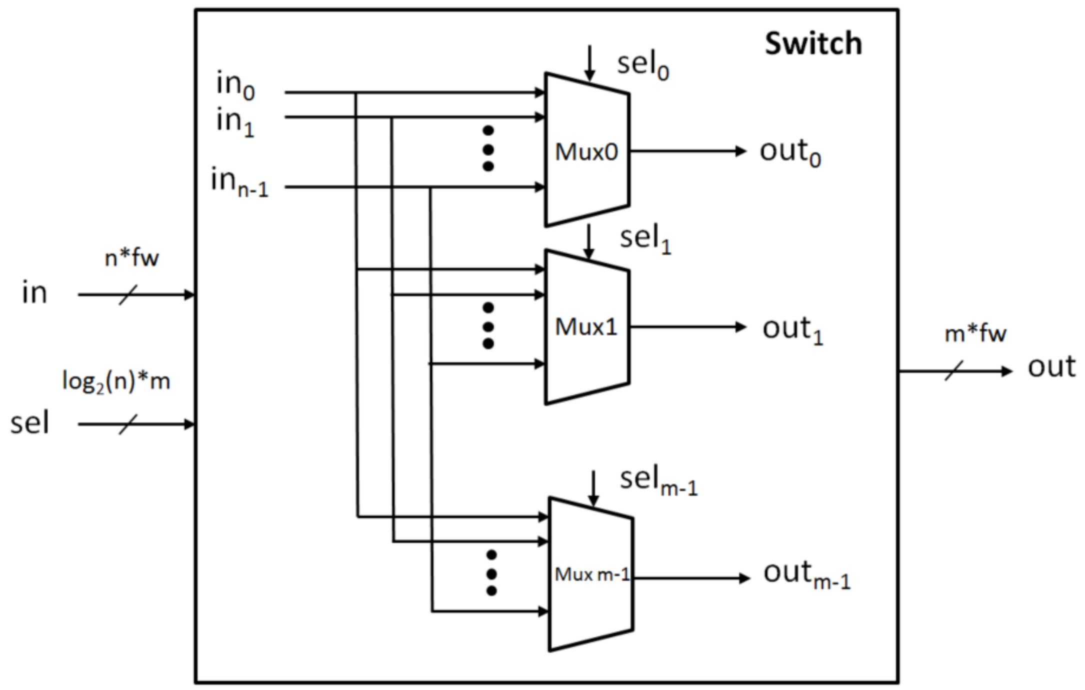


Figure 8. n-by-m switch. *fw* is the width of a flit.

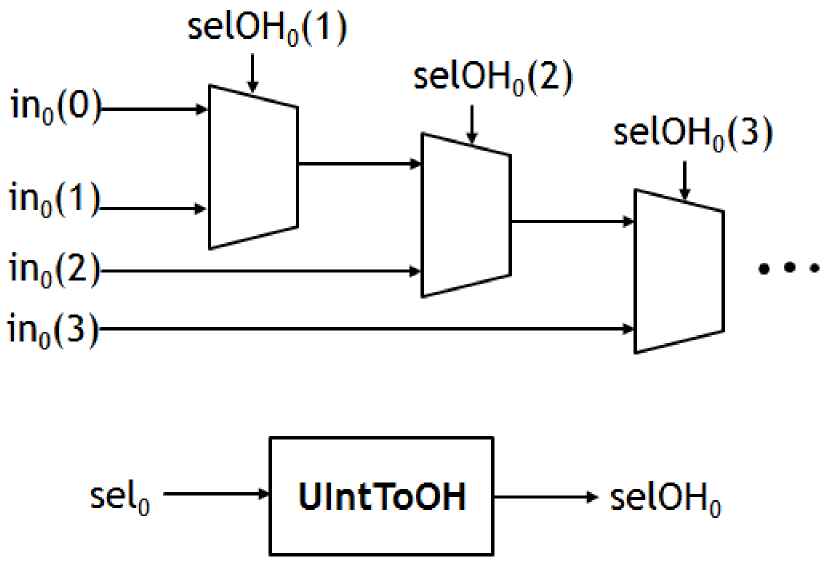


Figure 9. Small MUXes of distributed MUX crossbar that make up a larger MUX in a regular MUX crossbar.

III. Approach, Design and Validation

3.1 Work Breakdown

Figure 10 depicts the work breakdown structure that we have defined for our capstone project. Chisel and router learning is the first step that all members of the team have to undertake before moving on to comprehend OpenSoC Fabric code. These two task plans dominated most of the Fall Semester as they have steep learning curve. In the Spring Semester, we were able to achieve a decent understanding of the code base insomuch that we were able to distinctly split the work into three parts: (1) Implementation of arbiter modules; (2) implementation of routing function and back-end tool setup; and (3) extraction of a single switch and implementation of test harness. Lastly, the resulting design from our combined work will be pushed through the back-end ASIC flow which includes synthesis, floorplanning, place and route.

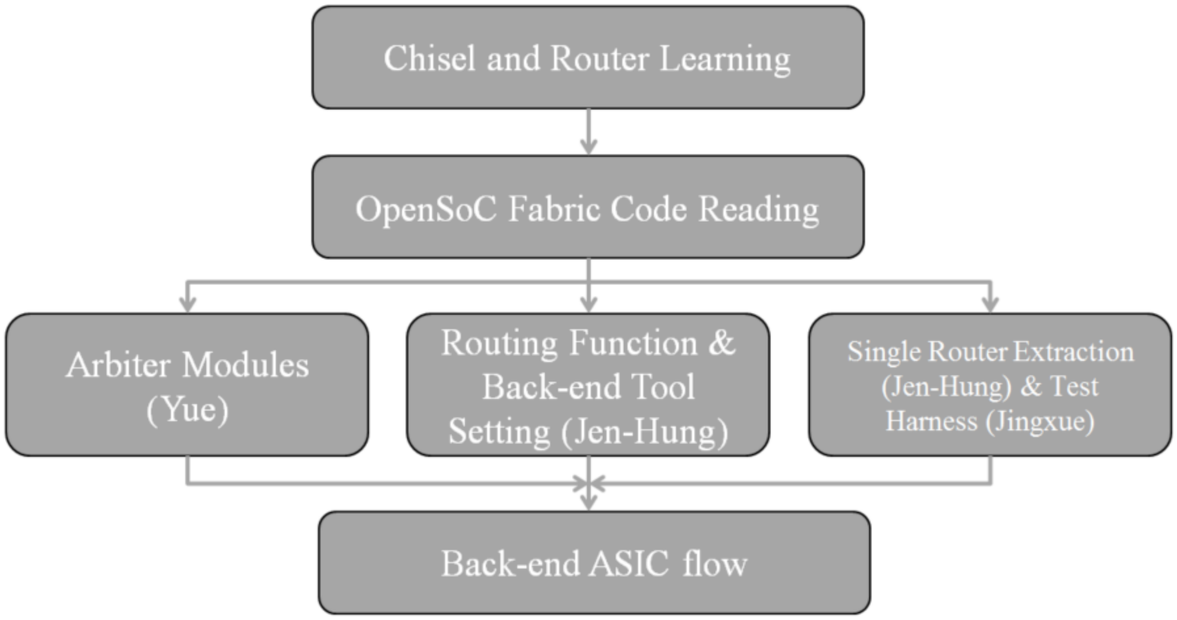

Chisel and Router Learning
OpenSoC Fabric Code Reading
Arbiter Modules (Yue)
Routing Function & Back-end Tool Setting (Jen-Hung)
Single Router Extraction (Jen-Hung) & Test Harness (Jingxue)
Back-end ASIC flow

Figure 10. Work breakdown structure.

3.2 Arbiter Modules

Arbiter is the main component within an allocator and as mentioned previously, the complexity of an allocator increases quadratically with the number of radix. Hence it is important to explore various arbiter implementations to gauge the efficiency of each type in a high-radix switch. Yue has chosen two different implementations of arbiter to explore: matrix arbiter and lookahead arbiter. Since types of arbiter are part of the design space, they can be parametrized using Chisel's parametrization feature.

Matrix Arbiter implements the least recently served scheme, which means the most recently served port will be assigned the lowest priority in the next round of arbitration. Using the analogy of a street intersection, it is similar to giving the left turn light the lowest priority to be green after it was just green a moment ago. Matrix arbiter stores the priorities inside a matrix structure, hence the name of the arbiter. Lookahead arbiter is a variant of fixed priority round robin arbiter that is superior in the speed of arbitration. Figure 11(a) shows the implementation of a normal fixed-priority arbiter and Figure 11(b) shows the lookahead implementation of the arbiter. Since lookahead arbiter experiences less gate delay, it has relatively higher speed than a typical arbiter.

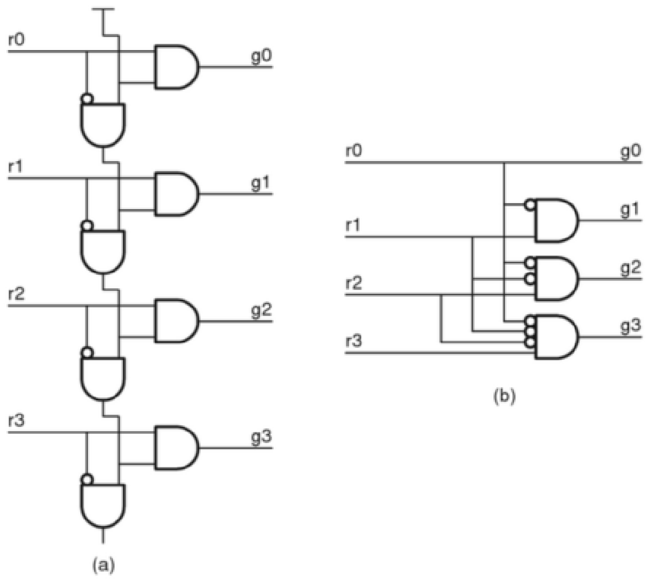
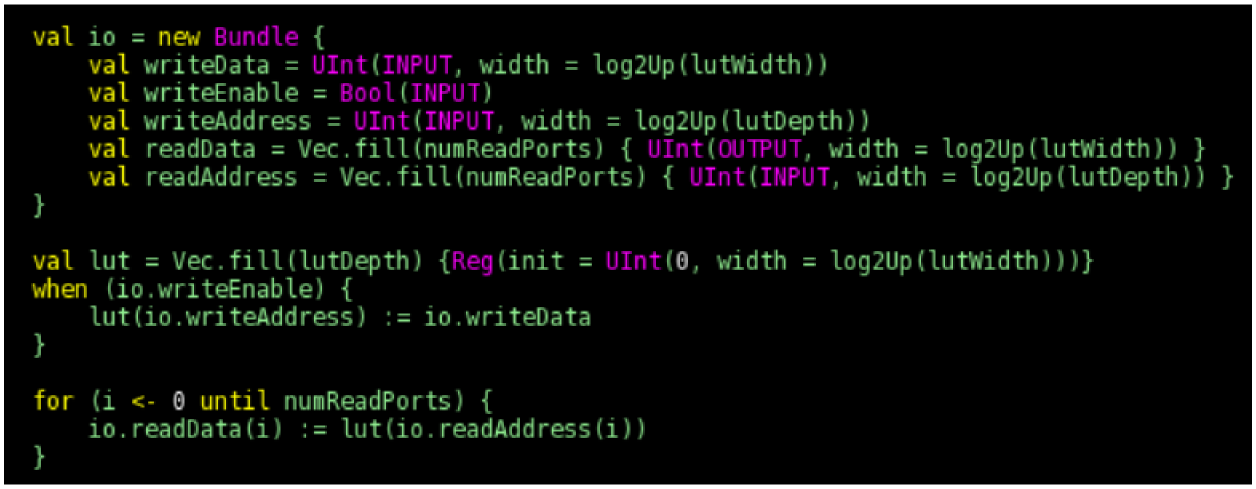


Figure 11. Two implementations of 4-bit arbiter: (a) Using iteration; (b) using lookahead (Dally, 2012).

3.3 Routing Function & Back-end Tool Setting

As mentioned in the previous section, lookup table was chosen as the routing function of the switch. A lookup table offers fast access and flexibility in topology infrastructure at the expense of area and power. Since I am responsible for the implementation, I have included the snippet of the lookup table Chisel implementation in Figure 12. For full implementation and test harness, see Appendix. The IOs are quite similar to a register file except that there can be more than 2 read ports. In fact, there are as many read ports as there number of radix times number of VCs. Also note that the only handshaking signal is writeEnable. This means that the lookup table can only be initialized/modified once before the functional traffic of the switch begins. Since write and read operations are never executed at the same time, there is no need to worry about race condition.



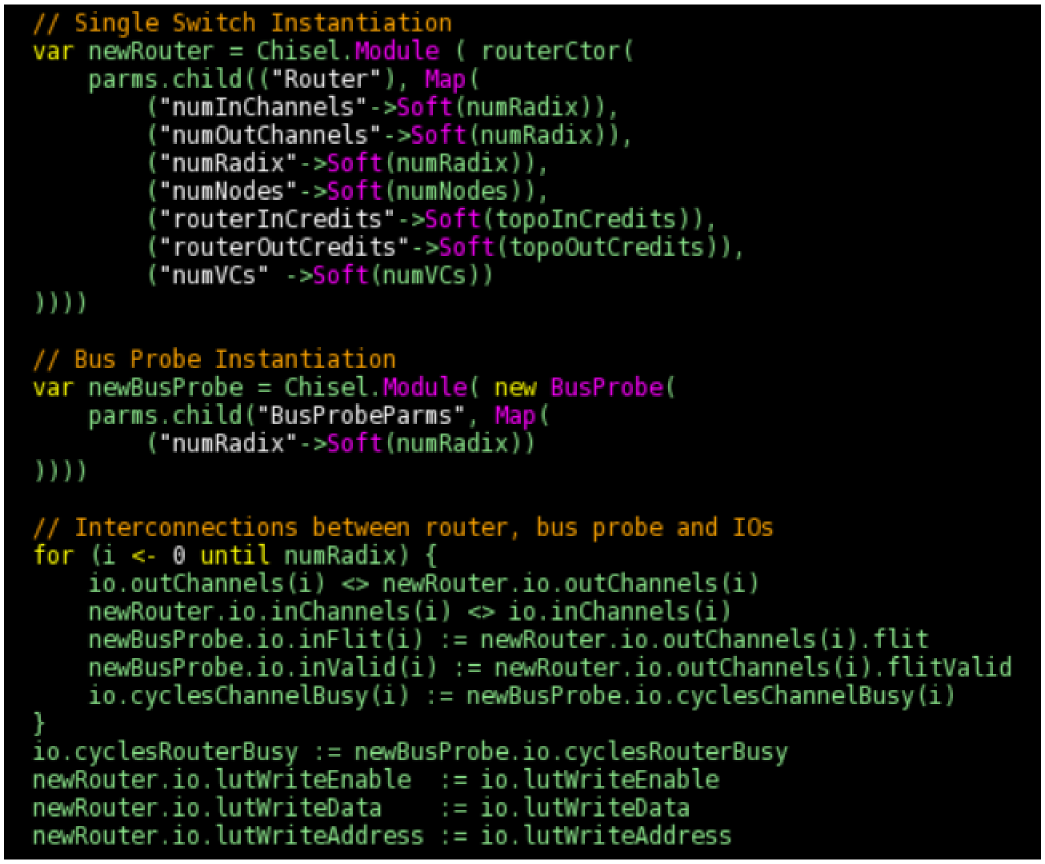
```
val io = new Bundle {
    val writeData = UInt(INPUT, width = log2Up(lutWidth))
    val writeEnable = Bool(INPUT)
    val writeAddress = UInt(INPUT, width = log2Up(lutDepth))
    val readData = Vec.fill(numReadPorts) { UInt(OUTPUT, width = log2Up(lutWidth)) }
    val readAddress = Vec.fill(numReadPorts) { UInt(INPUT, width = log2Up(lutDepth)) }
}

val lut = Vec.fill(lutDepth) {Reg(init = UInt(0, width = log2Up(lutWidth)))}
when (io.writeEnable) {
    lut(io.writeAddress) := io.writeData
}

for (i <- 0 until numReadPorts) {
    io.readData(i) := lut(io.readAddress(i))
}
```

Figure 12. Chisel code for lookup table

Apart from the lookup table implementation, I was also responsible for modifying all the modules and test harnesses that depended on dimension order routing – which was originally implemented by LBNL – to accommodate the integration of the lookup table. Lastly, I set up the environment for ASIC back-end flow which is described in detail in “3.5 Back-end ASIC Flow” section.

3.4 Single Switch Extraction & Test Harness

OpenSoC Fabric is a generator for network mesh. However, within the realm of our project scope, we only need to explore and analyze a single router. Therefore, the first step is then to extract a single switch from the mesh provided in OpenSoC Fabric. The key to switch extraction is to recognize that the content of the module which contains network topology can be replaced by that of a single router without modifying the module’s I/Os. With this approach, there is no need to modify the top level modules as well as the child modules that interface with the topology module. Figure 13 shows the code snippet of the topology that instantiates one switch only. The bus probe that is instantiated alongside of the switch is used to monitor the flit traffic on the output ports.



```
// Single Switch Instantiation
var newRouter = Chisel.Module ( routerCtor(
    parms.child(("Router"), Map(
        ("numInChannels"->Soft(numRadix)),
        ("numOutChannels"->Soft(numRadix)),
        ("numRadix"->Soft(numRadix)),
        ("numNodes"->Soft(numNodes)),
        ("routerInCredits"->Soft(topoInCredits)),
        ("routerOutCredits"->Soft(topoOutCredits)),
        ("numVCs" ->Soft(numVCs))
))))

// Bus Probe Instantiation
var newBusProbe = Chisel.Module( new BusProbe(
    parms.child("BusProbeParms", Map(
        ("numRadix"->Soft(numRadix))
))))

// Interconnections between router, bus probe and IOs
for (i <- 0 until numRadix) {
    io.outChannels(i) <> newRouter.io.outChannels(i)
    newRouter.io.inChannels(i) <> io.inChannels(i)
    newBusProbe.io.inFlit(i) := newRouter.io.outChannels(i).flit
    newBusProbe.io.inValid(i) := newRouter.io.outChannels(i).flitValid
    io.cyclesChannelBusy(i) := newBusProbe.io.cyclesChannelBusy(i)
}
io.cyclesRouterBusy := newBusProbe.io.cyclesRouterBusy
newRouter.io.lutWriteEnable  := io.lutWriteEnable
newRouter.io.lutWriteData    := io.lutWriteData
newRouter.io.lutWriteAddress := io.lutWriteAddress
```

Figure 13. Chisel code for topology with only one switch instantiation

Test harness is a testing structure that wraps around the entire switch design. It injects packet traffic with various traffic patterns and measures the packet latency as well as the channel utilization. Due to the scope of the project, the only traffic pattern we considered is 10% injection rate without bursting. The methodology to test the switch functionality is simply injecting a packet into an input port and expecting it to exit the correct output port in a certain amount of cycles. After a simple packet traversal is validated, we then inject a total of 64 packets of varying lengths into each of the ports and monitor the arrival/departure of all the packets. Using these numbers, the latency and throughput are calculated like such:

$$Latency = \frac{1}{64 * radix} * \sum (individual\ packet\ latency)$$

$$Throughput = Radix * flit\ width * frequency * average\ channel\ utilization$$

Note that clock frequency in the throughput equation can be found in static timing report generated from the Design Compiler tool.

3.5 Back-end ASIC Flow

OpenSoC Fabric is designed entirely in Chisel. Chisel is capable of generating Verilog equivalent code that can be pass to an ASIC or FPGA flow. Once we have the Verilog code that is functionally the same as its Chisel counterpart, we can feed it to Design Compiler which is logic synthesis tool that synthesizes the hardware design described in HDL into a gate-level circuit representation. From this tool we can also get estimation on timing, area and power of the design. However, it is not accurate as no routing or placement has been done yet. We borrowed the setup scripts from the ASIC lab we took last semester and from the Petabit team previous year. The next step after Design Compiler is to push the design through IC Compiler which does floorplanning, placement and routing. Figure 14 highlights the steps in ASIC design flow.

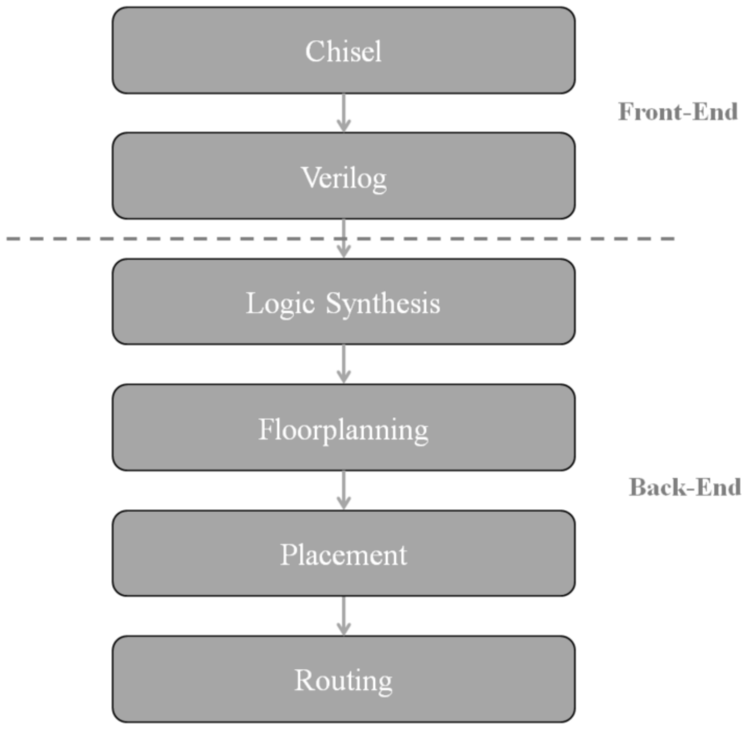


Figure 14. ASIC design flow

IV. RESULTS

4.1 Objective, Design Space, and Performance Metrics

As mentioned previously, our objective is to explore the design space of a switch and determine what configuration gives the most optimal performance. The parameters within the design space that we chose to vary are the types of arbiters and the number of radix. Figure 15 summarizes all the parameters. Note that we varied the radix from 2 up to 64 because the C++ simulator was not able to run for 128-radix or above configurations due to complexity and time. The primary performance metrics we decided to use to deduce the best configuration is the division of throughput by latency.

$$\frac{Throughput}{Latency} = \frac{Radix * flit\ width * chn\ util}{\frac{1}{64 * radix} * \sum(individual\ packet\ latency) * clock\ period} \quad \left({}^{bits}/_{s\ *\ cycle}\right)$$

Since we want to achieve the highest throughput and the lowest latency, the implementation that offers the highest division of throughput by latency is the most ideal. Area and power are also considered in choosing the optimal configuration.

Parameters	**Fixed or Vary**	**Value(s)**
Types of Arbiter	Vary	Round Robin
		Lookahead
		Matrix
Radix	Vary	2, 4, 8, 16, 32, 64
Num of VC	Fixed	2
Num of nodes	Fixed	256
Queue depth	Fixed	16
Flit size	Fixed	55 bits

Figure 15. Design parameters and their values

4.2 Choosing an Arbiter

There are three kinds of arbiters we implemented: Round robin, matrix and lookahead. Figure 16(a), 16(b) and 16(c) show how these arbiters fare under different metrics, namely power, area and throughput over latency. Note that the figures contain data up to radix 8 because we think the data is enough to show the general trend as radix scales up. Under 2-, 4- and 8-radix switch, lookahead arbiter proves to be the superior arbiter as it takes up the least areas, offers the most throughput over latency and generates similar power numbers as those of the other arbiters. It is also interesting to note that matrix arbiter sacrificed power and area for a slightly better performance than round robin.

Besides the metrics used above, it is also crucial to consider the fairness of each arbiter. Lookahead arbiter is a fixed priority arbiter and thus the channels that were assigned the lowest priorities will have starved packets. A side-by-side comparison of each arbiter's fairness is shown in Figure 17(a), 17(b) and 17(c). They show the histograms of the latencies of 1024 packets injected at 10% rate under each of the arbiter in a 16-radix switch. As seen in the lookahead arbiter's histogram, it has a relatively higher peak on the left follow by a long tail. This means that majority of the packets are serviced right away while the rest of the packets are idling waiting to be serviced. On the other hand, round robin and matrix arbiter rotate the priorities of the channel and hence have a relatively smoother gradient and shorter tail. For comparison, the longest latency in lookahead, round robin and matrix are 49, 26 and 27 cycles, respectively. Hence, at 10% injection rate, the longest packet latency in lookahead is twice as long as that of the round robin and matrix, which may or may not be desirable depending on the application.
Despite its inferior arbitration fairness, lookahead arbiter is still chosen as the best arbiter based on its performance and thus is used in the rest of the result analysis.

Power (uW)	Radix 2	Radix 4	Radix 8
RR Arbiter	**24800**	**48400**	**95600**
Matrix Arbiter	**24800**	50700	123000
CL Arbiter	26500	50700	10100

Figure 16(a). Power figures for different types of arbiters and radix

Area (um^2)	Radix 2	Radix 4	Radix 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**

Figure 16(b). Area figures for different types of arbiters and radix

Throughput/Latency (Gb/s*cycle)	Radix 2	Radix 4	Radix 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**

Figure 16(c). Throughput/Latency for different types of arbiters and radix

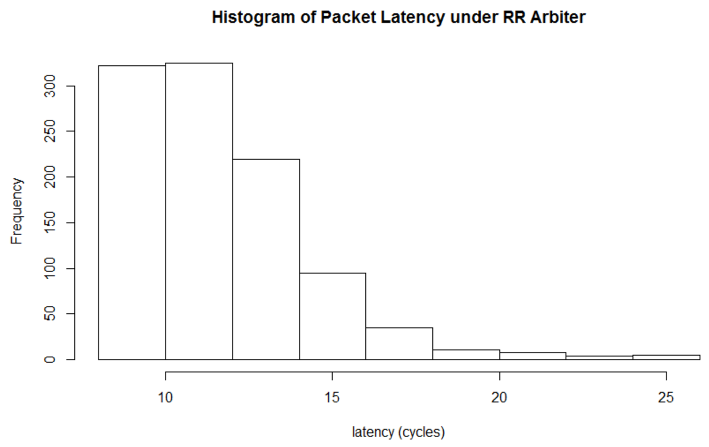


Figure 17(a). Histogram of packet latency under round robin arbiter

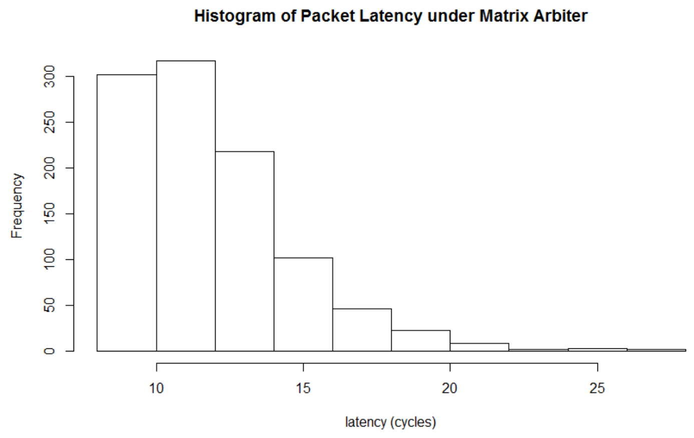


Figure 17(b). Histogram of packet latency under matrix arbiter

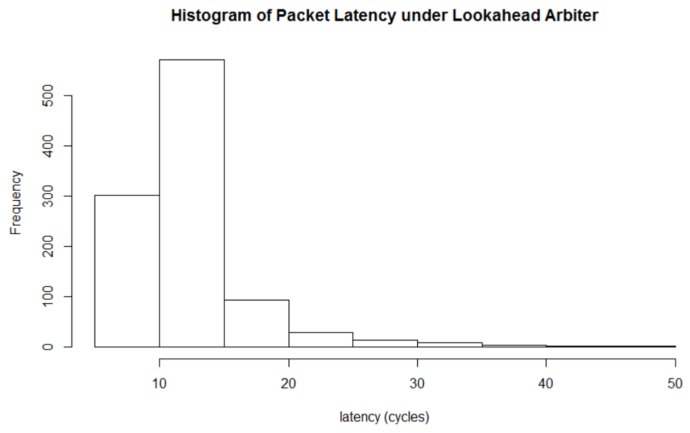


Figure 17(c). Histogram of packet latency under lookahead arbiter

4.3 Choosing a Radix

Similar to choosing the arbiter, we decided on the best radix based on the throughput over latency metric. Figure 18(a), 18(b) and 18(c) show the latency, throughput and throughput over latency versus radix, respectively. Note that the data points for radix-128 are extrapolated and the rest is obtained from software simulation and DC synthesis. In Figure 18(c), it shows that 64-radix switch yields the highest performance and droops a little bit at 128-radix. Before claiming 64-radix switch as the best configuration, we also have to consider its area and power figures to see if they are reasonable.

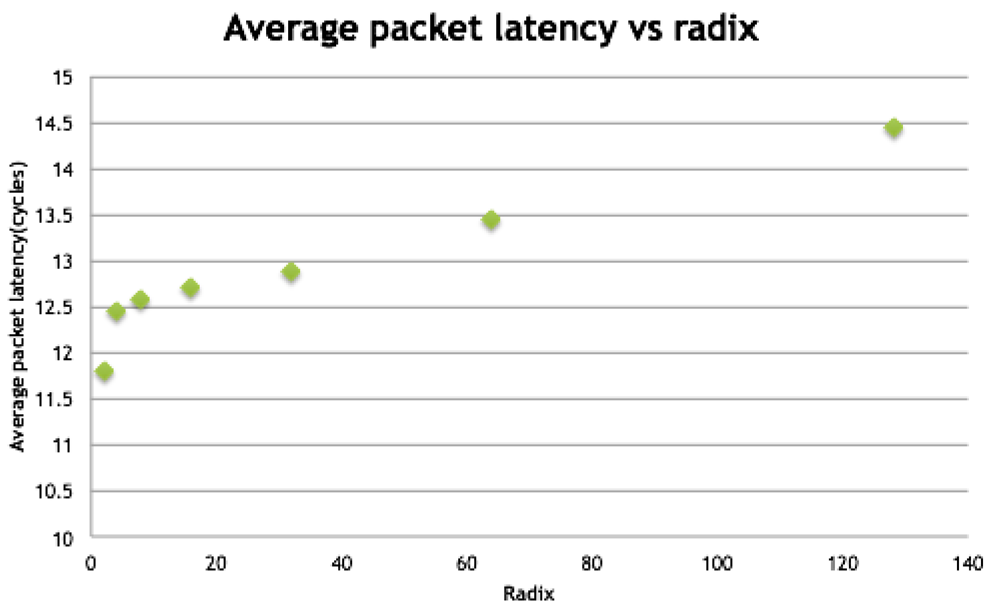


Figure 18(a). Average packet latency vs. radix

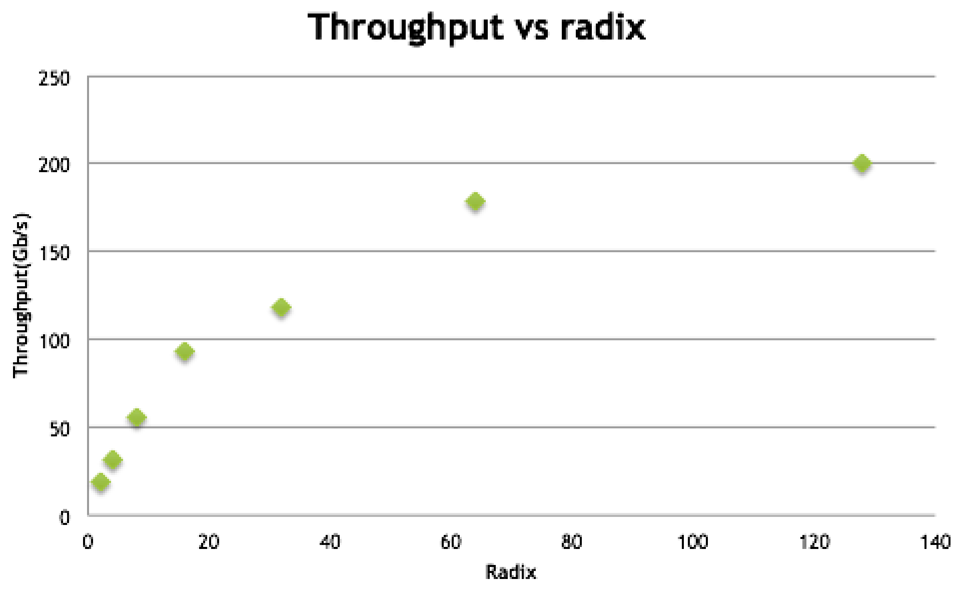


Figure 18(b). Throughput vs. radix

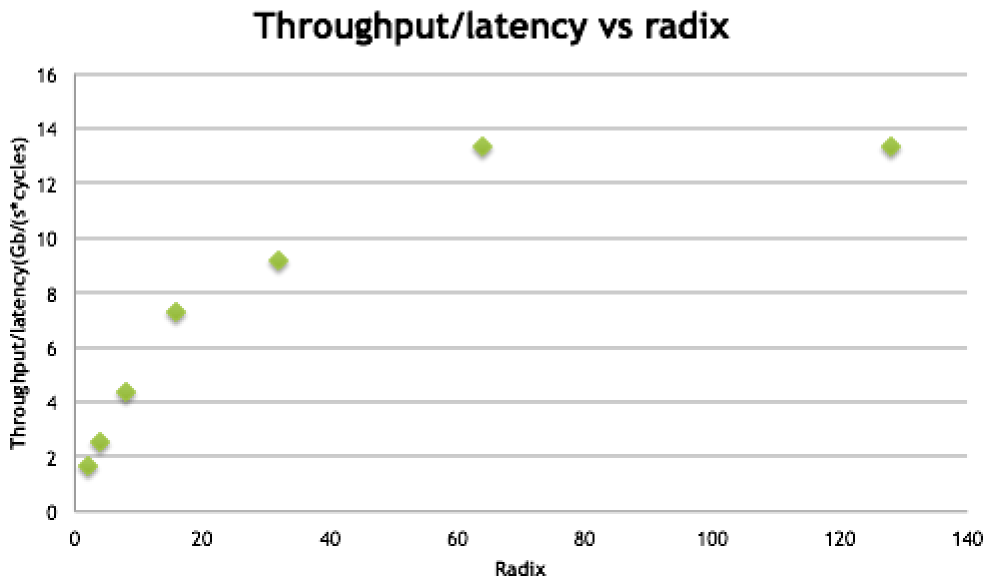


Figure 18(c). Throughput / latency vs. radix

Figure 19(a) and 19(b) illustrate the area and power distribution of 64-radix switch. We can observe that the buffers consume majority of the area and power. Buffers include injection/ejection queues and VC buffers that are 16 registers deep and are allocated for 55-bit flit. The next largest block is the crossbar switch. As mentioned previously, crossbar switch is innately large and it grows quadratically with respect to radix. Hence it is likely that the crossbar may overtake buffer – which grows linearly – as the largest component beyond 128 radix. This is a major concern, so pipelining and efficient crossbar design such as Clos network should be look into as future work. One more concerning thing to note is the rest of the area and power denoted as "others". This mainly includes the logics on the toplevel, with the biggest hitter being the scoreboard registers that track all the valid grants of the allocators. The scoreboard is a sequential element that also grows quadratically with respect to radix. This may be the reason why its power is higher than that of the switch, which is purely combinational. Overall, the total power is 1.32W and the total area is 7,736,094 um^2 or 7.7 mm^2 which comes down to a 2.8mm by 2.8mm chip if the aspect ratio is 1. A caveat is that these figures might be an underestimation as they are obtained from DC reports.

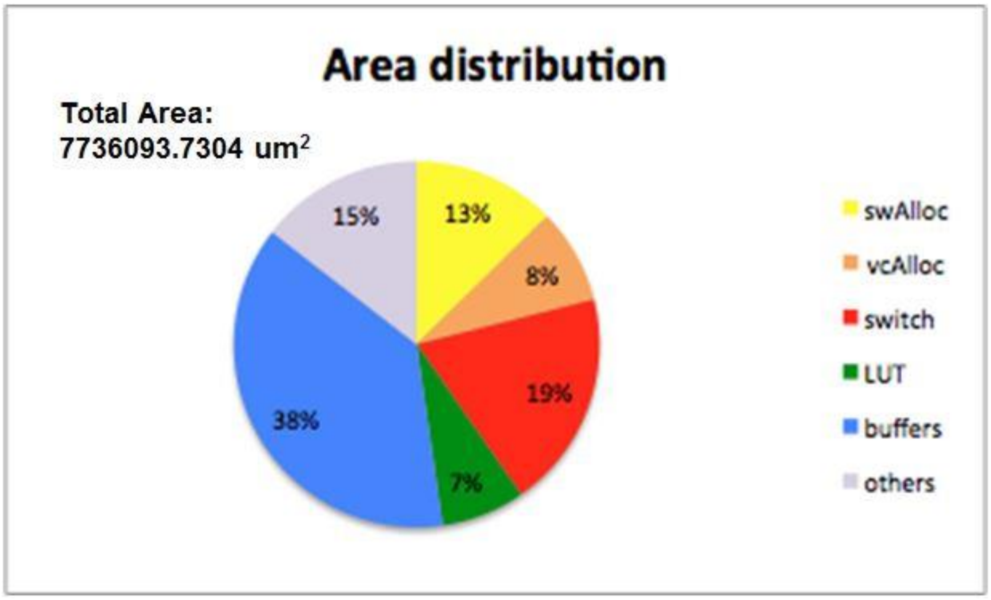


Figure 19(a). Area distribution for 64-radix switch

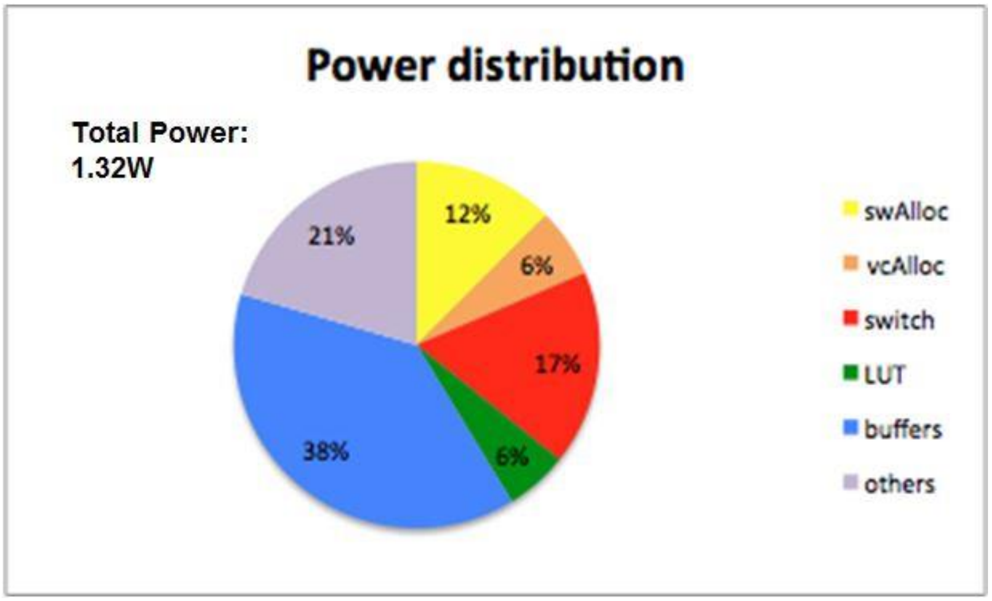


Figure 19(b). Power distribution for 64-radix switch

4.4 Concluding Remarks and Future Work

Since all the power and area figures are generated from Design Compiler, they do not account for wiring congestion and parasitics. However, these figures still show a general trend that shed lights on the growth rates of area and power with respect to radix. Using the three performance metrics – throughput over latency, power and area – we were able to find the most suitable arbiter, the lookahead arbiter. Even though it has an inferior arbitration scheme than the other two arbiters, it gives us the least amount of area while having the best throughput over latency. Using the same performance metrics, we were able to find the most ideal radix, 64, under the switch architecture implemented by LBNL.

For future work, we are hoping to push 64-radix switch through ICC. Currently, we were only able to push 16-radix switch through ICC, hence no significant ICC data was reported in this paper. Although not executed, we have been looking into the hierarchical synthesis flow as a potential solution. It is a methodology where each of the components are synthesized individually and then put together on the toplevel to be synthesized again. With this approach, we may be able to synthesize 32- or high radix switch in ICC.

Industry and Market Analysis

I. 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.

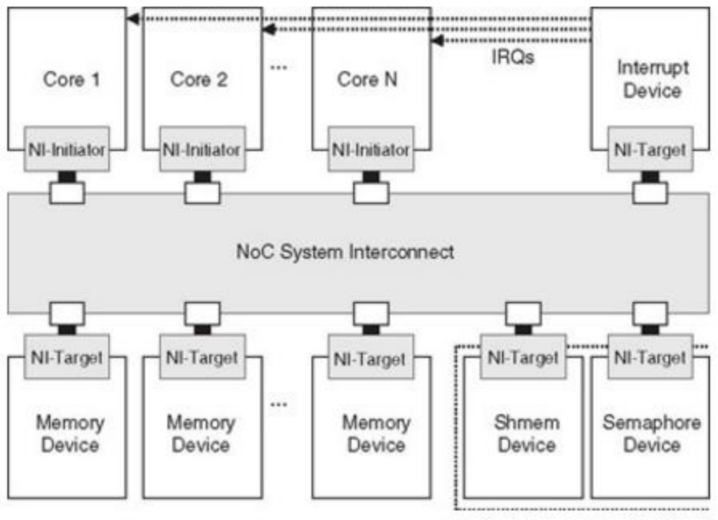


Figure 1. Multi-core Network-on-Chip Layout (Benini, 2007)

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.

II. 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.

III. TECH 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 that 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.

IV. MARKET ANALYSIS

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.

Work Cited

Abts, Dennis, and John Kim. *High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities*. Morgan & Claypool Publishers, 2011.

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, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, Krste Asanovic. “Chisel: Constructing Hardware in a Scala Embedded Language”. DAC 2012, June 3-7, 2012, San Francisco, CA. 2012.

Becker, Daniel. “Efficient microarchitecture for network-on-chip routers”. Doctoral dissertation submitted to Stanford University. August 2012. http://purl.stanford.edu/wr368td5072

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

Dally, Williams and R. Curtis Harting. *Digital Design: A Systems Approach*. Cambridge University Press, 2012

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

Fatollahi-Fard, Farzad, David Donofrio, George Michelogiannakis, John Shalf. *OpenSoC Fabric: User and Reference Manual*. 22 June 2015. <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

John Kim, William J. Dally, Brian Towles, Amit K. Gupta. “Microarchitecture of a High-Radix Router”. Proceedings of the 32nd annual international symposium on Computer Architecture, 2005: 420-431. Washington, DC. 2005

Izraelevitz, Adam, *Advanced Parametrerization Manual*. 22 May 2015. <https://chisel.eecs.berkeley.edu/2.2.0/chisel-parameters.pdf>

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.

Yu, Zhigang, Dong Xiang, Xinyu Wang. “Balancing Virtual Channel Utilization for Deadlock-free Routing in Torus Networks”. *The Journal of Supercomputing*, 17(8), 3094-3115. August 2015.

Appendix

lookUpTable.scala

```
package OpenSoC

import Chisel._
import scala.util.Random

class LookUpTable(parms: Parameters) extends Module(parms) {
    val lutWidth = parms.get[Int]("widthLookUpTable")
    val lutDepth = parms.get[Int]("depthLookUpTable")
    val numReadPorts = parms.get[Int]("numReadPorts")

    val io = new Bundle {
        val writeData = UInt(INPUT, width = log2Up(lutWidth))
        val writeEnable = Bool(INPUT)
        val writeAddress = UInt(INPUT, width = log2Up(lutDepth))
        val readData = Vec.fill(numReadPorts) { UInt(OUTPUT, width =
log2Up(lutWidth)) }
        val readAddress = Vec.fill(numReadPorts) { UInt(INPUT, width =
log2Up(lutDepth)) }

        // Debug Ports
        val readOutAddress = Vec.fill(numReadPorts) { UInt(OUTPUT, width =
log2Up(lutDepth)) }
    }

    val lut = Vec.fill(lutDepth) {Reg(init = UInt(0, width = log2Up(lutWidth)))}
    when (io.writeEnable) {
        lut(io.writeAddress) := io.writeData
    }

    for (i <- 0 until numReadPorts) {
        io.readData(i) := lut(io.readAddress(i))
    }

    // Debug Ports
    io.readOutAddress <> io.readAddress
}

class LookUpTableTest(c: LookUpTable) extends Tester(c) {
      implicit def bool2BigInt(b:Boolean) : BigInt = if (b) 1 else 0

    val lutWidth : Int = c.lutWidth
    val lutDepth : Int = c.lutDepth
    val numReadPorts : Int = c.numReadPorts

    val nums = (0 until lutDepth).map(x =>
BigInt(Random.nextInt(Math.pow(2,log2Up(lutWidth)).toInt)))

    for (i <- 0 until lutDepth) {
        poke(c.io.writeAddress, i)
        poke(c.io.writeEnable, true)
        poke(c.io.writeData, nums(i))
        step(1)
    }

    poke(c.io.writeEnable, false)

    for (i <- 0 until numReadPorts) {
        poke(c.io.readAddress(i), i % lutDepth)
        expect(c.io.readData(i), nums(i % lutDepth))
    }
    step(1)
}
```

topology.scala

```
package OpenSoC

import Chisel._

abstract class VCTopology_LUT(parms: Parameters) extends Module(parms) {
    val numVCs          = parms.get[Int]("numVCs")
    val numRadix        = parms.get[Int]("numRadix")
    val numNodes        = parms.get[Int]("numNodes")
    val topoInCredits   = parms.get[Int]("topoInCredits")
    val topoOutCredits  = parms.get[Int]("topoOutCredits")
    val routerCtor      = parms.get[Parameters=>VCRouter]("routerCtor")

    val counterMax = UInt(32768)

    val io = new Bundle {
        val inChannels  = Vec.fill(numRadix) { new ChannelVC(parms) }
        val outChannels = Vec.fill(numRadix) { new ChannelVC(parms).flip() }
        val lutWriteEnable      = Bool(INPUT)
        val lutWriteData        = UInt(INPUT, width = log2Up(numRadix))
        val lutWriteAddress     = UInt(INPUT, width = log2Up(numNodes))

        val cyclesRouterBusy    = UInt(OUTPUT, width=counterMax.getWidth)
        val cyclesChannelBusy   = Vec.fill(numRadix){UInt(OUTPUT,
width=counterMax.getWidth)}
    }
}

class SimpleVCRouterTopology(parms: Parameters) extends VCTopology_LUT(parms) {
    var newRouter = Chisel.Module ( routerCtor(
        parms.child(("Router"), Map(
            ("numInChannels"->Soft(numRadix)),
            ("numOutChannels"->Soft(numRadix)),
            ("numRadix"->Soft(numRadix)),
            ("numNodes"->Soft(numNodes)),
            ("routerInCredits"->Soft(topoInCredits)),
            ("routerOutCredits"->Soft(topoOutCredits)),
            ("numVCs" ->Soft(numVCs))
    ))))
    var newBusProbe = Chisel.Module( new BusProbe(
        parms.child("BusProbeParms", Map(
            ("numRadix"->Soft(numRadix))
    ))))

    for (i <- 0 until numRadix) {
        io.outChannels(i) <> newRouter.io.outChannels(i)
        newRouter.io.inChannels(i) <> io.inChannels(i)
        newBusProbe.io.inFlit(i) := newRouter.io.outChannels(i).flit
        newBusProbe.io.inValid(i) := newRouter.io.outChannels(i).flitValid
        io.cyclesChannelBusy(i) := newBusProbe.io.cyclesChannelBusy(i)
    }
    io.cyclesRouterBusy := newBusProbe.io.cyclesRouterBusy
    newRouter.io.lutWriteEnable  := io.lutWriteEnable
    newRouter.io.lutWriteData    := io.lutWriteData
    newRouter.io.lutWriteAddress := io.lutWriteAddress
}
```

switch_t.scala (distributed MUX crossbar)

```
package OpenSoC

import Chisel._

class Switch_t(parms: Parameters) extends Module(parms) {
        val numInPorts  = parms.get[Int]("numInPorts")
        val numOutPorts = parms.get[Int]("numOutPorts")
        val switchWidth = parms.get[Int]("switchWidth")
        val io = new Bundle {
        val inPorts = Vec.fill(numInPorts) { UInt(INPUT, width = switchWidth) }
        val outPorts = Vec.fill(numOutPorts) { UInt(OUTPUT, width = switchWidth) }
        val sel = Vec.fill(numOutPorts) {UInt(width = log2Up(numInPorts))}.asInput
    }
        val selOH = Vec.fill(numOutPorts) {UInt(width = numInPorts)}
        for( i <- 0 until numOutPorts ) {
            selOH(i) := UIntToOH(io.sel(i))
        }
        for (i <- 0 until numOutPorts) {
            val data = Vec.fill(numInPorts) {UInt(width = switchWidth)}
            data(0) := io.inPorts(0)
            for( j <- 1 until numInPorts) {
                data(j) := Mux(selOH(i)(j), io.inPorts(j), data(j-1))
            }
            io.outPorts(i) := data(numInPorts-1)
        }
}
```

clos.scala (incomplete clos network, missing the select scheme)

```
package OpenSoC

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

class Clos(parms: Parameters) extends Module(parms) {
    val numInPorts  = parms.get[Int]("numInPorts")
    val numOutPorts = parms.get[Int]("numOutPorts")
    val switchWidth = parms.get[Int]("switchWidth")
    val switchCtor  = parms.get[Parameters=>Switch_t]("switchCtor")

    var n : Int = 6
    var k : Int = 11
    var r : Int = 11

    if (numInPorts == 32) {
        n = 4
        k = 7
        r = 8
    } else if (numInPorts == 64) {
        n = 6
        k = 11
        r = 11
    } else {
        n = 6
        k = 11
        r = 11
    }

    val io = new Bundle {
        val inPorts = Vec.fill(numInPorts) { UInt(INPUT, width = switchWidth) }
        val outPorts = Vec.fill(numOutPorts) { UInt(OUTPUT, width = switchWidth) }
        val sel = Vec.fill(numOutPorts) {UInt(width = log2Up(numInPorts))}.asInput
    }

    def selectScheme(r1 : Int, r2 : Int) : Int = {
        //TODO
    }

    var firstStageMap = new HashMap[Int, Switch_t]()
    var secondStageMap = new HashMap[Int, Switch_t]()
    var thirdStageMap = new HashMap[Int, Switch_t]()

    for (i <- 0 until r) {
        var firstStageSwitch = Chisel.Module ( switchCtor(
            parms.child(("firstStageSwitch",i), Map(
                ("numInPorts"->Soft(n)),
                ("numOutPorts"->Soft(k)),
                ("switchWidth"->Soft(55))
            ))))
        var thirdStageSwitch = Chisel.Module (switchCtor(
            parms.child(("thirdStageSwitch",i), Map(
                ("numInPorts"->Soft(k)),
                ("numOutPorts"->Soft(n)),
                ("switchWidth"->Soft(55))
            ))))
        firstStageMap += i -> firstStageSwitch
        thirdStageMap += i -> thirdStageSwitch
    }
    for (i <- 0 until k) {
        var secondStageSwitch = Chisel.Module (switchCtor(
            parms.child(("secondStageSwitch",i), Map(
                ("numInPorts"->Soft(r)),
                ("numOutPorts"->Soft(r)),
                ("switchWidth"->Soft(55))
            ))))
        secondStageMap += i -> secondStageSwitch
    }

    for (i <- 0 until r) {
        for (j <- 0 until n) {
            if (i*n + j < numInPorts) {
                firstStageMap(i).io.inPorts(j) <> io.inPorts(i*n + j)
            }
            if (i*n + j < numOutPorts) {
                io.outPorts(i*n + j) <> thirdStageMap(i).io.outPorts(j)
            }
        }
        for (j <- 0 until k) {
            firstStageMap(i).io.outPorts(j) <> secondStageMap(j).io.inPorts(i)
            secondStageMap(j).io.outPorts(i) <> thirdStageMap(i).io.inPorts(j)
        }
    }
}
```