DESIGN AND IMPLEMENTATION OF AN ORDERED MESH NETWORK INTERCONNECT

A Project Report

submitted by

ANJANA A J

in partial fulfilment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **Design and Implementation of an Ordered Mesh Network Interconnect**, submitted by **Anjana A J**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master Of Technology**, is a bona fide record of the research work done by her under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof
Dr V. Kamakoti
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 8th May 2017

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my guide, Dr. V.Kamakoti for giving me the freedom to explore and experiment and also for all the valuable advice, encouragement and motivation.

I would like to thank my co-guide Dr.Nitin Chandrachoodan and faculty advisor Dr.Shreepad Karmalkar who have patiently guided us throughout the program.

Further, I extend my sincere regard and appreciation to MS scholars Rahul Bodduna and Arjun Menon of RISE lab who have readily extended their learnings of Bluespec System Verilog and logic synthesis.

I also thank my fellow lab mates Syed M Md Zaid, Debpratim Adak, Sasidhar N and Gaurav Sehgal for all the help and support. Last but not the least, I thank my parents and all of my friends for their constant encouragement and support.

ABSTRACT

KEYWORDS: Chip Multi-processors, Network Topologies, Mesh Interconnect, Flow Control, Router Architecture, Flits, Virtual Channels, XY routing

In the last decade we have seen that as the Moore's law continues to hold true and more and more computing elements are being packed into the same area of silicon footprint, the performance per unit area has been increasing linearly but so is the requirement of heat dissipation. In order to deal with this problem the concept of multi core processors has been introduced wherein a single chip was used with multiple processing elements. Thus many core chips are rising as the design decision to give power effectiveness and enhance execution, while riding Moore's Law. The Shakthi processor project that is being undertaken at the RISE lab at IIT Madras aims to build 6 variants of processors based on RISC-V Instruction Set Architecture from UC Berkeley. The project aims to provide open source processor and SoC designs. As a part of that, this project aims to implement an ordered mesh network interconnect which can be used to implement snoopy cache coherence for mesh network on chip.

With multiple independent cores we have multiple shared caches in a shared memory architecture. The coherency traffic and other user data are exchanged between the multiple applications running on the multi core processors. Hence the overall performance will depend on the on-chip latency experienced by the packets in traversing through the interconnection network that connects the multiple cores. In this project an interconnection network has been implemented for mesh topology which has been proved to provide significant advantages in terms of power, performance and timing as compared to other popular network topologies like bus or ring. Cache requests will be broad casted throughout the entire mesh network and whenever a hit occurs at any of the nodes, then a response will be send back from that particular node to the requesting node. The requests from different sources are ordered by making use of a notification network so that all the nodes process the requests in the same order.

The interconnection network has been designed and implemented in an HDL named Bluespec System Verilog (BSV). In order to show the validity of the proposed design in real hardware resources, it has been synthesized onto a Xilinx synthesis tool VIVADO for Ultra-scale board.

Contents

List of Figures

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
RISE	Reconfigurable and Intelligent Systems Engineering
BSV	Bluespec System Verilog
HDL	Hardware Description Language
RISC	Reduced Instruction Set Computing
FIFO	First In First Out
CMP	Chip-Multiprocessors
VLSI	Very Large Scale Integration
CMOS	Complementary Metal Oxide Semiconductor
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
NoC	Network on Chip
VC	Virtual Channel

Chapter 1

INTRODUCTION

In 1965, Gordon Moore, the co-founder of Intel has observed that the number of transistors per square inch of an integrated circuit will double roughly in every eighteen to twenty-four months. This observation, generally known as the Moore's law presented the VLSI world with an opportunity to drastically increase the computing performance per unit area of silicon every few years since highly complicated circuits could be packed into a reduced semiconductor area. Also, with the advent of the faster and smaller CMOS technology, it was possible to drive these chips at higher clock frequencies which was another factor for increasing performance.

According to the Dennard scaling theorem, or the MOSFET scaling theorem, as the transistors become smaller in size, their power densities still remain constant. This implies that the power consumed by a circuit is proportional to its area. As a result, a functionally equivalent circuit would consume lesser power for the same frequency, and in order to improve the performance, the frequency could be increased without having to provide

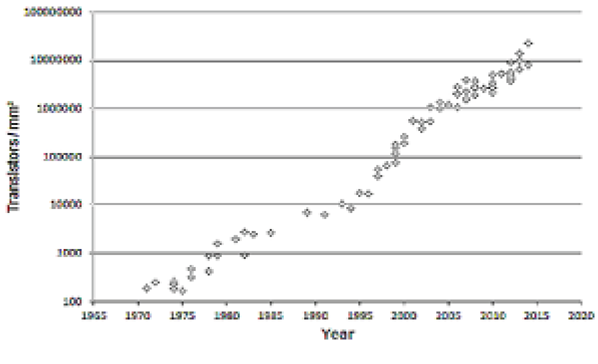


Figure 1.1: Moore's Law till date

higher power. As a result, for almost two decades after 1985, the processor manufacturers were able to improve processor performance by more than 50 % annually.

But in the recent years after 2005, Dennard scaling seems to have stopped and although Moore's Law continues to hold true, the proportional improvement in performance could not be obtained by merely increasing the clock frequency. As sizes decreased, the leakage currents increased and started consuming a huge part of the total power supply. Also, any attempt to increase frequency in the absence of scaled down voltages led to heating up of the chips and in some extreme cases even thermal runaway, thus further increasing the energy costs.

To overcome the above mentioned problems, the semiconductor industry has moved to the new found area of chip-multiprocessors (CMPs). The aim was to pack identical cores or processing elements on a common substrate. The cores being identical were easily replicated. All such cores ran on the same highest clock frequency which could be used without being concerned with elaborate cooling requirements. The idea was to run those applications that lend themselves well to parallel processing on these CMPs. That is, the problems being solved by the applications could be broken down into smaller and independent problems that would be managed by threads of the application and each thread would run on a dedicated core assigned to it. Thus, as of now, multi-core approach seems to be the most viable option for realizing performance that is keeping up with the Moore's law.

To derive the maximum benefits of scaling down of transistor sizes it is desirable to pack in as many cores per chip as possible resulting in increased performance per unit area. Also, it becomes essentially important that a scalable and high-bandwidth communication fabric needs to be developed in order to connect them together. An interconnect is required for inter-core and off-chip transport of user packets or flits and also for the coherency of data. A smaller number of cores may be connected using the conventional bus interconnect. However such an interconnect is not found to be scalable beyond a few number of cores. The reason for this is that the bus, although ordered and excellent for cache coherence traffic using snoopy protocol, does not support higher bandwidths as core count increases. The throughput declines significantly as more and more cores are added to the network. The power requirement for driving the bus also increases drastically as the bus expands. Another undesirable feature of the bus is the additional logic required for arbitration policy that adds to network latency.

Another simple interconnect design is that of a crossbar. The crossbar does not suffer from the low throughput as in the case of bus but as the number of cores increases, the area requirement of crossbars blow up and so does its power requirement. Many ideas to overcome these difficulties of buses and crossbars have been proposed like the segmented buses and hierarchical crossbars. However the Network on Chips (NoCs) have emerged as the preferred media due to their ability to scale, consume less power and occupy lesser chip area.

The basic requirement of an interconnect fabric for multi-core processors is to provide a scalable medium of communication between the computing cores. So the interconnect network is one of the topmost contenders for optimizations in the architecture. It is also known to consume a significant chunk of the total power. Also it should meet the facilities required for supporting cache coherency. Meeting such demands come at a cost. As the interconnect fabric becomes more complicated, more hardware is required to meet such demands and this translates to additional power consumption.

The researchers today have identified the interconnect as a critical optimization area and is trying to efficiently design it using various forms of optimization. One aim is to improve performance and decrease the data exchange costs between the cores through effective circuit design. The choice of architecture and logic selected for the design of the interconnect fabric also has a vital role to play in achieving the required functionality and the performance figures of the network. Another important aspect that affects the interconnect performance is the type of routing algorithm used. These optimizations have a direct relation to the power consumed by the interconnect as they determine the amount of transitions that occur on the network which is closely related to the power consumed. Hence it is desirable to implement a network that reduces the overall power consumed by the processors. Areas of network intelligence may also be explored in the course of building better and more efficient NoCs, for example, we may use a type of router that purposely drops flits based on the congestion in the network in order to reduce congestion.

Chapter 2

BACKGROUND AND LITERATURE REVIEW

2.1 MEMORY CONSISTENCY AND CACHE COHERENCE

In a multi-core processor the various cores working simultaneously share the same hard disk, memory and other memory devices. This provides several advantages as follows:

1. The cost is reduced as multiple processors share the same resources like mother board, power supply etc.
2. There is increased reliability as the failure of one processor does not affect the other processors though it will slow down the machine a little.
3. There is increased throughput as an increase in the number of processes completes the work in less time.

Cache memory is a computer memory with very short access time which is used for the storage of frequently used instructions or data. It is an extremely fast memory that is built into a computer's central processing unit (CPU), or located next to it on a separate chip. The CPU uses cache memory to store instructions that are repeatedly required to run programs. When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is available in the cache . Cache has various advantages in all shared memory multiprocessor system :

1. To reduce the average data access time: The CPU does not have to use the motherboard's system bus for data transfer thereby improving the overall system speed.

2. Reduced bandwidth demands are placed on shared interconnects.

3. It is a volatile memory, so it is reusable and it does not occupy much space.

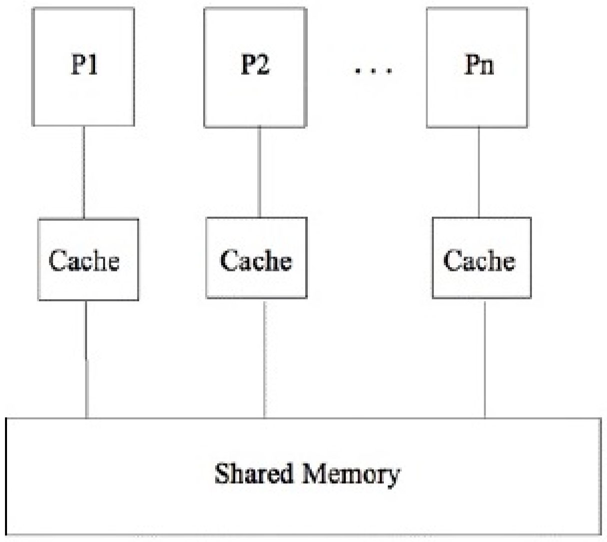


Figure 2.1: Several processor cores connected to cache

But certain problems arise while using cache memory for multi-processor systems. Since all the processors share the same address space, it is possible for more than one processor to cache an address at the same time. This is referred to as the coherence issue. Also, if one processor updates a data item without informing the other processor, inconsistency may result and cause incorrect execution. This is referred to as the consistency issue.

2.2 COHERENCY MECHANISMS AND PROTOCOLS

Cache coherence protocols attach permissions to each block stored in a processor's cache. On every load or store, the access is sent to the local cache and an appropriate action is taken after looking at the permission for that block. These permissions are referred to as the state of the cache line. The commonly used stable states in a cache are as described below:

1. Invalid (I): The block in the cache is not valid. The cache either does not contain the block, or it is a potentially stale copy that it cannot read or write.
2. Shared (S): The block in the cache is valid, but the cache may only read this block. This state typically implies that other processor caches in the system may be sharing this block with read permission.
3. Modified (M): The processor cache holds the block, and has read and write access to the block. This also implies that this cache holds the only valid copy of the block in the system, and must therefore respond to requests for this block.

In addition to the above states, two additional states may be added to improve performance – the Exclusive (E) state assigned to a block on a read request if there are no copies cached at other caches, allows for implicit write access, thereby saving time, and the Owned (O) state assigned to a block indicates that this cache will source a remote coherence request, which prevents slow DRAM access. Most cache coherence protocols allow cache blocks to transition to a different state through a set of transactions.

Throughout literature, there are two major approaches to cache coherence protocols, as follows:

1. **Broadcast-based snoopy protocols:** All coherence controllers observe (snoop) all coherence requests in the same order, and collectively take the correct action to maintain coherence. By requiring that all requests to a block arrive in order, a snooping system enables the distributed coherence controllers to correctly update the finite state machines that collectively represent a cache block's state. Snoopy protocols rely on ordered interconnects, like bus or tree networks, to ensure total ordering of transactions. A total ordering of transactions subsumes the per-block orders thus maintaining coherence. Additionally, the total ordering makes it easier to implement memory consistency models that require a total ordering of memory transaction, such as sequential consistency.

2. **Directory-based protocols:** The key principle in directory-based protocols is to establish a directory that maintains a global view of the coherence state of each block.

The directory tracks which caches hold each block and in what states. A cache controller that wants to issue a transaction sends the same to the directory, and the directory looks up the state of the block and determines the appropriate action. A sharer list for each block enables the directory to avoid broadcasting messages to all nodes, instead sending targeted messages to the relevant nodes. Like snoopy protocols, directory protocols also need to define when and how coherence transactions are ordered with respect to other transactions. In most directory protocols, the transactions are ordered at the directory. However, additional mechanisms are required to enforce sequential consistency.

Directory protocols have the advantage of requiring low communication bandwidth and are thus scalable to large number of cores. However the scalability comes at the cost of higher latencies caused by directory indirection. Further, as the number of cores increases, maintaining a directory incurs significant area and power overhead. In addition, directory protocols are harder to implement and require significant verification effort to enforce memory consistency guarantees.

Snoopy protocols have traditionally dominated the multiprocessor market. They enable efficient direct cache-to-cache transfers that are common in commercial workloads, have low overheads in comparison to directory-based protocols and are simple to design. In addition, the total ordering of requests in snoopy coherence, enables easy implementation of desirable consistency guarantees, such as sequential consistency. However, there are two primary impediments in scaling snoopy coherence to many-core chips, which are, broadcast overhead and reliance on ordered interconnects.

Chapter 3

INTERCONNECTION NETWORKS

Interconnection networks are composed of switching elements. An interconnection network in a parallel machine transfers information from any source node to any desired destination node. This task should be completed with as small latency as possible and it should also allow a large number of such transfers to take place concurrently. Packet-switched networks-on-chip (NoC) provide scalable bandwidth for multi-core processors and they also have relatively low latency. An NoC usually comprises of a collection of routers and links that connect various processor nodes. Routers take care of the communication between cores by multiplexing the traffic flowing in from different input links onto the output links. A message transmission in a NoC occurs by breaking the message into packets before injection into the network. A packet comprises one or more flow-control-units or flits which are the smallest units of the message on which flow-control is performed. A flit, in turn, is composed of one or more physical-digits or phits which is the number of bits that can be transmitted over a physical link in a single cycle. The primary features that characterize a NoC are:

1. Topology

2. Routing algorithm

3. Flow control mechanism

4. Router micro-architecture

3.1 TOPOLOGY

It refers to the physical layout and connection of the nodes and links in the network. The performance and cost of the network is affected by the topology that is being used. It determines the number of hops that a message takes to reach its destination, as well as the wire length of each hop. Thus topology significantly influences the latency of the network. The power consumption also depends on the number of hops since it involves storing and forwarding of the packets. The topology also affects the path diversity available in the network, which refers to the number of unique paths that exist between a pair of source and destination nodes. A higher path diversity leads to greater robustness from failures, while also providing greater opportunities to balance traffic across multiple paths. The three major interconnect topologies in use are ring, mesh and ring-mesh hybrid interconnect or torus.

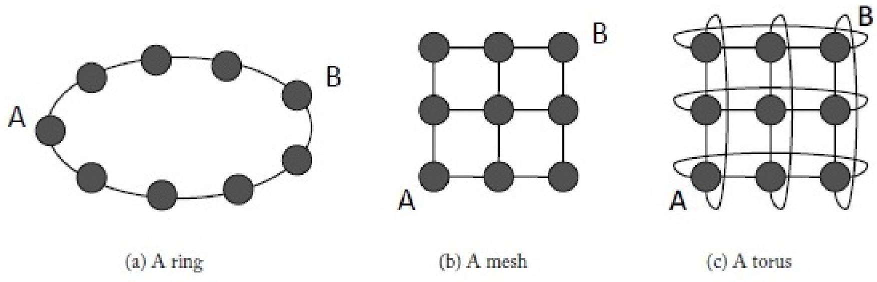


Figure 3.1: Popular network topologies

Figure 3.1 shows these three main interconnect topologies. The degree of a node refers to the number of links connected at the node. The greater the number of links the lower will be the latency but at the cost of increased silicon area and wire lengths. Also, it can be seen that for equal number of nodes, hop count will be maximum for ring topology and minimum for torus.

3.1.1 OTHER RELEVANT TOPOLOGIES

Apart from the basic topologies discussed above, there are other relevant topologies in use which are derived from the basic topologies. One among them is the butterfly interconnect as shown in figure 3.2. There are a number of switches in between the the source and destination nodes which performs the routing. Unlike mesh or torus the hop count here between any two pairs of nodes is exactly the same as the packets pass through the same number of stages. The hop count seems to have reduced but there is the disadvantage of an increased number of wires used in the design.

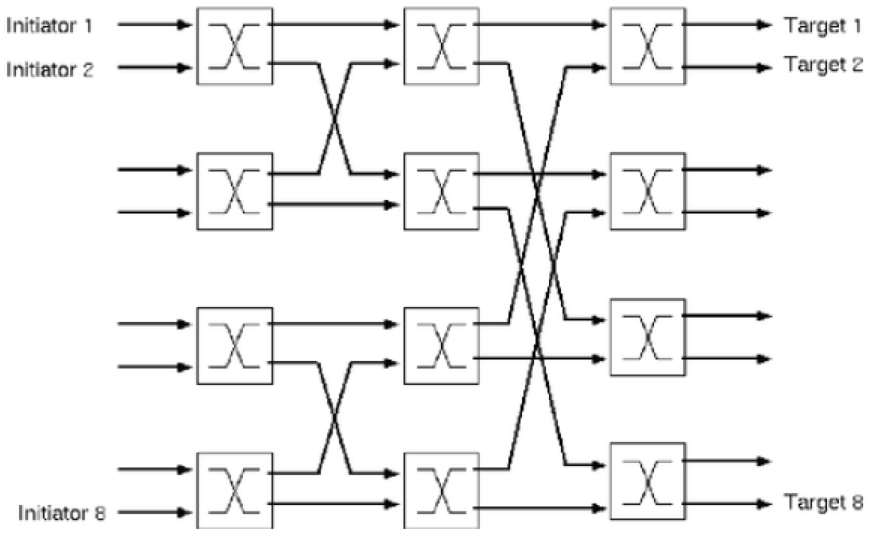


Figure 3.2: The butterfly interconnect

Another popular topology is the crossbar. It makes use of an arbitration unit which performs the control logic required for routing the packets. All the source and destination nodes are directly connected to the crossbar.

Then there is the fat tree topology which is in the form of a binary tree. Here the source and destination nodes are the leaf nodes and the internal nodes are the switches.

In this project the implementation has been done on the mesh network. Mesh network provides scalable bandwidth. It has several advantages over the other popular network topologies. The problem with ring is delay, with crossbars it is area and with bus it is

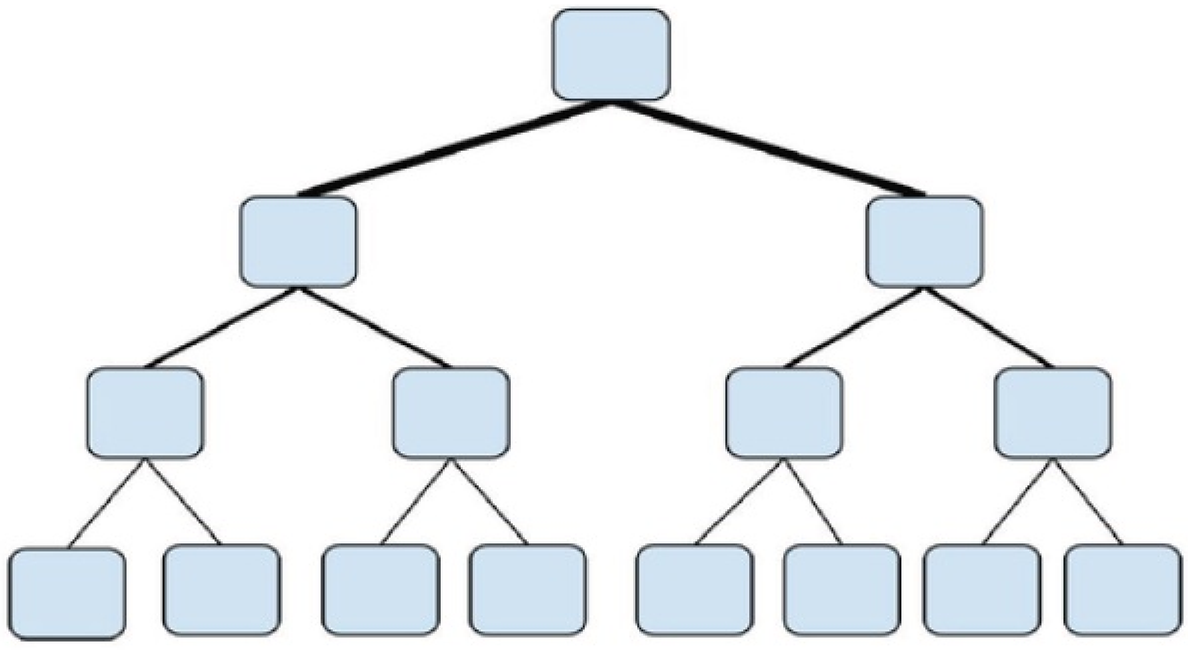


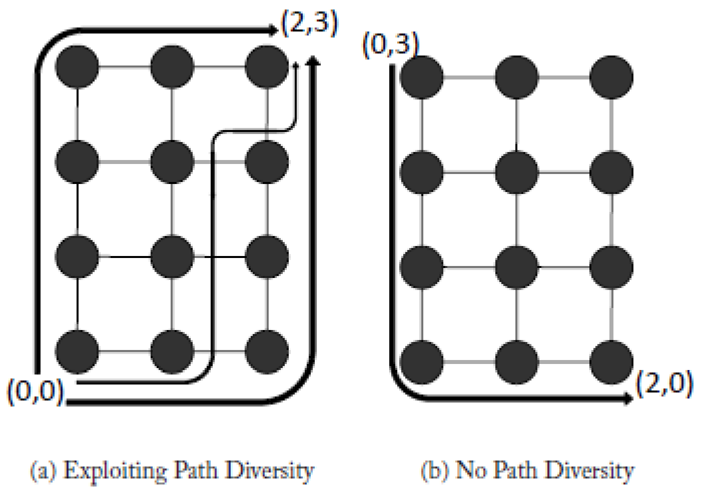
Figure 3.3: The fat tree topology

bandwidth. Also mesh can handle high traffic and it is robust. Mesh interconnect is suitable for higher dimension NoCs and is extensively used in the research of low power and low latency routers.

3.2 ROUTING ALGORITHM

For a given network topology routing algorithm determines the path taken by the packet from the source node to the destination node through the network. A good routing algorithm should aim to distribute the network traffic evenly, thereby avoiding network congestion and it should also provide good latency.

Routing algorithms are basically classified into three, which are, deterministic, oblivious and adaptive routing. In deterministic routing, the packets always follow the same path for a given pair of source and destination nodes. In oblivious and adaptive routing, for the given source and destination pairs, the packet may follow different paths at different times. This is because, in deterministic algorithms the path is chosen without taking into consideration the state of the network but in the other algorithms network congestion and other factors are considered while choosing the path. Routing algorithms are also classified into minimal and non-minimal. In minimal routing algorithms there is always the fewest number of hops between a source and destination. In non-minimal algorithms the packet may be routed away from the destination temporarily.



A routing algorithm is chosen by considering its effect on the network latency, throughput, complexity, cost, delay etc. Oblivious and adaptive routing algorithms suffer from the problems of deadlock, live lock and starvation. A deadlock usually occurs when two or more packets are waiting for each other to be routed forward. These packets may have reserved some resources and are waiting for each other to release the resources. Live lock occurs when a packet keeps spinning around its destination without ever reaching it. It is a common problem that exists in non-minimal type of routing algorithms. Complex networking algorithm and different set priorities can often lead to circumstances where lower priority packets never reach their destinations. This is referred to as starvation. This mainly happens when the packets with higher priorities reserve the resources all the time.

3.3 FLOW CONTROL

Network resources usually include buffers, links, channels etc. Flow control determines how these resources are allocated to the packets. A good flow control mechanism should allocate resources evenly so that congestion is reduced,bandwidth is increased and latency is also reduced.

The most basic flow control technique is circuit switching. Here the message as a whole is transmitted from the source to destination in one go without waiting at any of the intermediate nodes. In order to enable this, a probe packet is send first in order to reserve the resources required for the message to pass. The actual transmission of message is initiated only after an acknowledgement signal is received after resource allocation. This provides power and latency savings. But the links remain idle until acknowledgement of resource allocation is received and it cannot be used by any other packet. Therefore this results in reduced throughput.

Another method that overcomes several of the disadvantages of circuit switching is packet switching. Here the messages are broken down into packets and are buffered. Switching decision is made at every intermediate node. Packets from several sources can share links and therefore there is better link utilization in this method. There are two variants of packet switching. In store and forward method of packet switching, a packet is transmitted further only if it has been received in its entirety at the current node. This requires large buffer spaces and can also introduce delays in the intermediate nodes. The other one is the virtual cut through method where a part of the packet is transmitted further even though some portion of it has not yet been received at the current node. This reduces latency but still requires buffer storage.

A flit based flow control technique has been found to be suitable for NoCs. Due to the granularity of the flits buffer space is reduced which provides area and power savings. Also flits could be transmitted forward independently as in virtual cut through technique. Therefore there is improvement in latency.

3.4 ROUTER MICROARCHITECTURE

The basic architecture of the router should be such that latency and throughput are improved. The flits while traveling through the network spend most of their time at the routers waiting for the switching decision. Therefore delay of the router affects the delay between hops and thus the overall delay suffered by the packets from source to destination. Thus it also affects the maximum frequency at which the system can operate. Typical routers are usually pipelined. Area and power considerations should also be taken into account while designing the router as it has a direct impact on those factors.

Chapter 4

BASIC ROUTER ARCHITECTURE AND IMPLEMENTATION

Figure 4.1 shows the basic architecture of the router. Since we are using mesh topology, the router has five input and output ports corresponding to the four neighbouring directions North, South, East and West and a Local port.

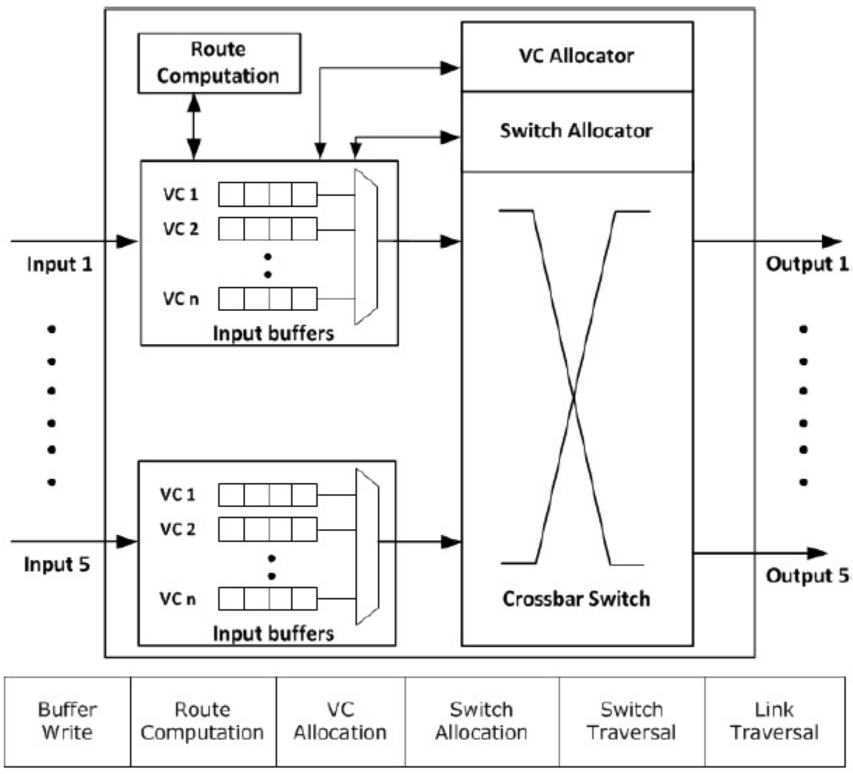


Figure 4.1: Basic router microarchitecture. The various pipeline stages are shown below the figure

The messages are broken down into packets which are further broken down into flits. This is schematically shown in figure 4.2. In this implementation we have a head flit, four body flits and a tail flit which indicates the end of the packet. Head flit contains the virtual-channel number, address required for matching and the source and destination information. Body flits and tail flits contain data. We use virtual channel (VC) flow control which allocates resources at flit granularity. Multiple VCs share the same physical link. Because of this even if one channel is full, the incoming flits in the same link can use the other free channels. Thus the use of VCs avoid head of the line blocking and deadlock. Head of the line blocking occurs when a packet at the head of the queue is blocked as its output port is held by another packet, while a packet further behind in the queue whose output port is available cannot make forward progress as it has to wait for the head of the queue to clear.

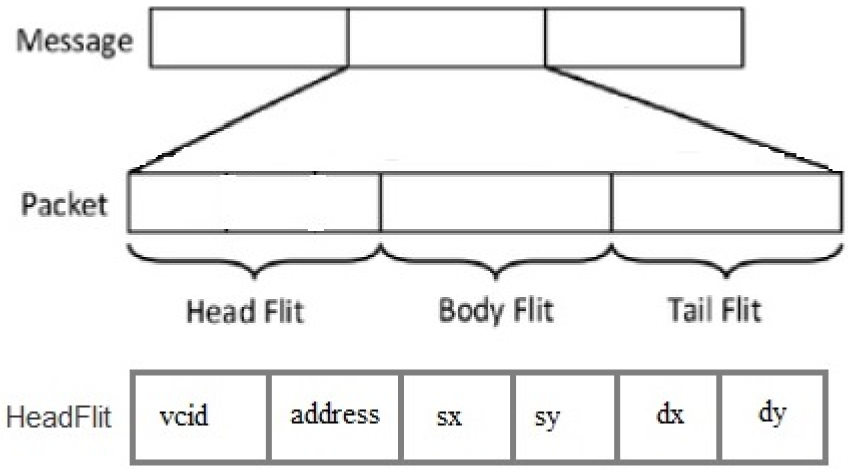


Figure 4.2: Breaking down the message into flits

The various stages that the flits pass through are shown at the bottom of the figure 4.1 and they are as follows:

Buffer Write:

The incoming flits are first written into the input buffers.

Route Computation:

Based on the source and destination information available on the head flits, route computation is performed in order to determine the output port for the flit. We have used the classic XY routing algorithm which is a distributed deterministic routing algorithm. It is a minimal algorithm suitable for mesh and torus topologies. XY routing algorithm routes packets first in x-direction (or horizontal direction) and then in y- direction (or vertical direction) to the destination. The addresses of the routers are their xy- coordinates. It has the advantage of never running into a deadlock or live lock.

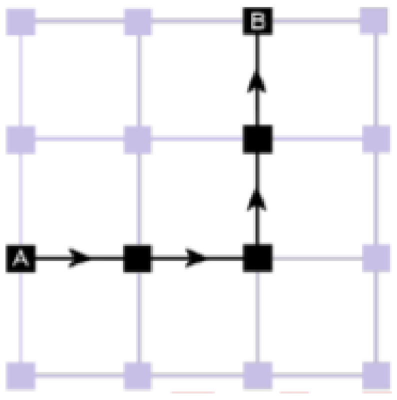


Figure 4.3: XY routing of packets from node A to node B in a mesh NoC

VC allocation:

In this stage the flit arbitrates for a virtual-channel corresponding to its output port. In order to perform VC allocation we make use of VC credits that indicate whether a VC is free or empty. If the credit is set to 1 it shows that the VC corresponding to it is free and if it is set to 0 then it indicates that the VC is filled. As flits travel downstream, credits flow upstream to grant access to the buffers that are vacated.

Switch Allocation:

In this stages the flit arbitrates for access to the crossbar switch. Here we make use of a direction arbiter in order to allocate the resources.

Switch Traversal:

On successfully winning the required output port, the flit traverses the crossbar.

Link Traversal:

The flit finally traverses the link to the next node.

Each flit arrives at the input of the router. The input unit contains the buffers in which the flits are held. The head flit includes a field called vcid which is the virtual channel number. Depending on the vcid the flit is stored in the corresponding VC.

The status of each of the virtual channels are maintained using two state fields which are Global (G) and Route (R). Global field can take on any of the values Idle(I), Routing(R), waiting for next VC (V) or Active (A). Route indicates the output port for the flit as obtained from route computation and can take any of the values North (N), South (S), East (E) or West (W).

Only the head flit passes through route computation and VC allocation. Body flits and tail flit follow the route setup by the head flit.

Chapter 5

THE ROUTER PIPELINE

Each head flit should go through route computation, virtual-channel allocation, switch allocation, and switch traversal. Each of these steps takes one clock cycle.

Suppose the head flit of a packet arrives at the router during cycle 0. The packet is directed to a particular virtual channel of the input port, as indicated by the vcid field of the head flit. At this point, the global state (G) of the target virtual channel is in the idle state(G = I). The arrival of the head flit causes the virtual channel to transition to the routing state (G = R) at the start of cycle 1.

During cycle 1, information from the head flit is used in order to perform route computation and to select one of the output port. The result of this computation updates the route field (R) of the virtual channel state vector and advances the virtual channel to the waiting for output virtual channel state(G = V) , both at the start of cycle 2. In parallel with this, the first body flit arrives at the router.

During cycle 2, the head flit enters the VC allocation stage and also the second body flit arrives at the router. During this stage, the result of route computation, held in the R field of the virtual channel state vector, is the input to the virtual channel allocator. If successful, the allocator assigns a virtual channel on the next router specified by the R field. The result of VC allocation updates the virtual channel id (vcid) of the flit and transitions the virtual channel to the active state(G = A) .

At the start of cycle 3 switch allocation begins. The head flit starts this process, but is handled no differently than any other flit. In this stage, any active virtual channel (G = A) that contains buffered flits and has downstream buffers available (C> 0) bids for a single flit time slot through the switch from its input virtual channel to the output unit. Depending on the configuration of the switch, this allocation may involve competition not only for the output port and the switch, but also competition with other virtual channels in the same input unit for the input port of the switch.

If successful switch allocation occurs during cycle 3 then the head flit traverses the switch during cycle 4. The head flit can start traversing the channel to the next router in cycle 5 without competition. While the head flit is being allocated a virtual channel, the first body flit is in the route computation stage, and while the head flit is in the switch allocation stage, the first body flit is in the VC allocation stage and the second body flit is in the route computation stage. However, since routing and virtual -channel allocation are per -packet functions, there is nothing for a body flit to do in these stages. Body flits cannot bypass these stages and advance directly to the switch allocation stage because they must remain in order and behind the head flit. Thus, body flits are simply stored in the virtual channel's input buffer from the time they arrive to the time they reach the SA stage.

As each body flit reaches the switch allocation stage, it bids for the switch and output channel in the same manner as the head flit. The processing here is per flit and the body flits are treated no differently than the head flit. As a switch time slot is allocated to each flit, the credits are updated to reflect its departure.

Finally, during cycle 6, the tail flit reaches the switch allocation stage. Allocation for the tail flit is performed in the same manner as that for head and body flits. When the tail flit is successfully scheduled in cycle 6, the packet releases the virtual channel at the start of cycle 7 by setting the virtual channel state for the output virtual channel to idle (G=I) and the input virtual channel state to idle(G=I) if the input is empty. If the input buffer is not empty, the head flit for the next packet is already waiting in the buffer. In this case, the state transitions directly to routing (G=R).

Each virtual-channel contains a buffer for head flit alone and a separate buffer for body flits and tail flit.

Vcstate represents the state of each VC and includes the G and R fields. To see if a packet is processed completely from the head flit to the tail flit a variable vacancy is used for virtual channels in each direction and it is updated with values ’Empty’, ’Filling’ or ’Filled’. When the VC is empty it holds the value ’Empty’. As head flit arrives the value is updated to ’Filling’ and it remains so as the body flits get filled in. Once the tail flit arrives the value is updated to ’Filled’ indicating that the entire packet has arrived.

Chapter 6

ORDERED MESH NETWORK INTERCONNECT

The design choices have been made by taking into account the various NoC components discussed earlier.

6.1 CHOICE OF LANGUAGE

C++, System C and System Verilog are some of the platforms that have been used for the design and synthesis of hardware for some years now. However, with the emergence of Bluespec System Verilog (BSV), more and more designers are opting it as their preferred platform for hardware design. The reasons for this are as follows:

1. BSV semantics like interfaces and atomic rules are at high level of abstraction for concurrency and hence it is much preferred to describe the parallelism found in hardware systems.

2. BSV's powerful types and static elaboration provides ease to describe the hardware architecture. The static elaboration is deterministic which helps to do a high level synthesis from BSV and arrive at a high quality of design much faster than the other platforms.

3. BSV provides strong parameterization due to which all modules and functions can be parameterized by other modules and functions. This gives a higher level of expressive power to designer.

4. Stronger type checking allows greater expansiveness with greater safety.

5. The code written using all the features is still synthesizable which really sets BSV apart from other languages.

The constructs and features as described above are either not available with other languages or are provided as add-ons and hence are not equally powerful. Hence the design has been implemented in BSV.

6.2 COHERENCE REQUESTS AND RESPONSES

We need to implement an interconnect fabric that can be used to employ snoopy coherence in a mesh network. Therefore in the network we deal with two types of messages which are request messages and response messages. Each is treated differently and separate algorithms are used for both the types of messages. Requests need to be broad casted throughout the network so that all the nodes can see it. But responses have to be routed correctly from the source node towards the destination node. As such, it is to be noted that in the case of request messages, VC allocation is performed directly after the buffer write stage thereby skipping the route computation stage in between.

Usually, in an interconnect network, global message ordering is realized by making use of a centralized ordering point. But such a centralized ordering point introduces two types of latency in the network:

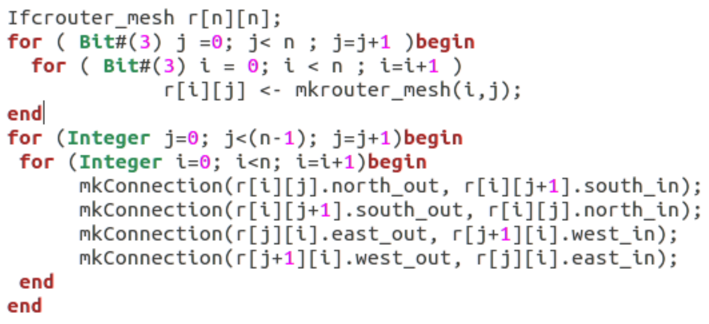
1. Indirection : This is the latency incurred by the message as it travels from the source node to the ordering point.

2. Serialization latency: This is the latency of the message that is waiting at the ordering point before it is correctly ordered and forwarded to the other nodes.

In order to eliminate this dependence on a centralized ordering point, we make use of two seperate networks in our implementation.

i. **The main network:** The main network is responsible for broadcasting coherence requests to all the nodes and delivering responses to the requesting node. The network is unordered and the requests from several source nodes may arrive at a particular node in any order. At every node in the main network we have 16 FIFOs which store the requests coming from different source nodes. For example, the requests coming from node 0 will always go to FIFO 0, requests coming from node 2 will be stored in FIFO 2 etc.

The routers are the basic elements of the main network. The router ports are connected together using mkConnection in order to create a mesh type network as shown. The main interface Ifcrouter_mesh has flit input and output methods and the interface Ifc_credit has the upstream and downstream availability of credits for all the directions.



```
Ifcrouter_mesh r[n][n];
for ( Bit#(3) j =0; j< n ; j=j+1 )begin
  for ( Bit#(3) i = 0; i < n ; i=i+1 )
             r[i][j] <- mkrouter_mesh(i,j);
end
for (Integer j=0; j<(n-1); j=j+1)begin
 for (Integer i=0; i<n; i=i+1)begin
      mkConnection(r[i][j].north_out, r[i][j+1].south_in);
      mkConnection(r[i][j+1].south_out, r[i][j].north_in);
      mkConnection(r[j][i].east_out, r[j+1][i].west_in);
      mkConnection(r[j+1][i].west_out, r[j][i].east_in);
 end
end
```

Figure 6.1: Connecting the routers together in order to create a mesh interconnect

Consider a 4 by 4 mesh network consisting of 16 routers. The routers are connected together as shown in figure 6.2. The routers are addressed by their x and y coordinates.

Every router has input ports namely north_in, south_in, east_in, west_in and local_in. Similarly the output ports are denoted by north_out, south_out, east_out, west_out and local_out. Whenever a request originates at any of the nodes, it is send to the local_out port from when it will be broadcasted to all the four directions. The broadcasting of requests is done throughout the network by following the given rules:

1. If a request is received at the local port, it will be send to all the four directions,

namely north, south, east and west ports.

2. If a request is received at the north port, it will be send to the south port.

3. If a request is received at the south port, it will be send to the north port.

4. If a request is received at the east port, it will be send to north, south and west ports.

5. If a request is received at the west port, it will be send to north, south and east ports.

Thus all the nodes are able to receive the requests. In figure, broadcasting is shown by the black arrows. Here, node (1,1) is the requesting node that sends the request.

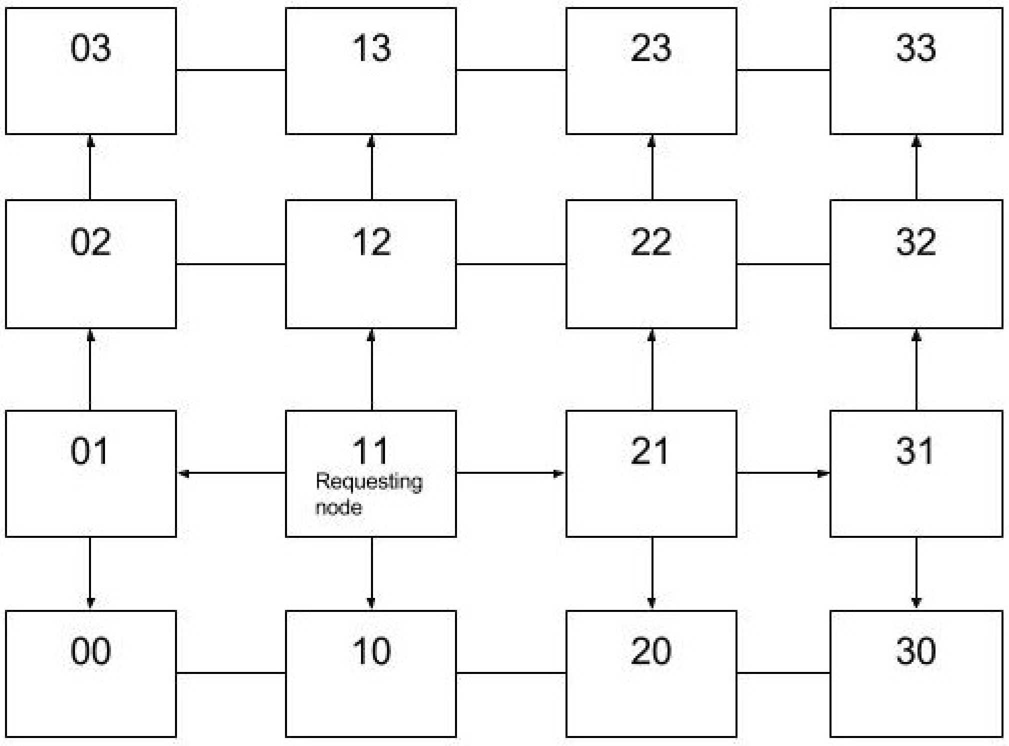


Figure 6.2: Broadcasting of request from the requesting node

ii.**The notification network:** Ordering of the requests from several source nodes is achieved by making use of the notification network. It is a fast, buffer less network. Whenever a request is send from the main network, a notification message encoding the source node's id is broadcasted in the notification network. The notification message is essentially a bit vector, where each bit corresponds to request from a particular source. Therefore requests from several sources are merged together by OR-ing the bit vectors in a contention less manner. The notification network consists of 5 N-bit bitwise OR gates where N is the number of cores.

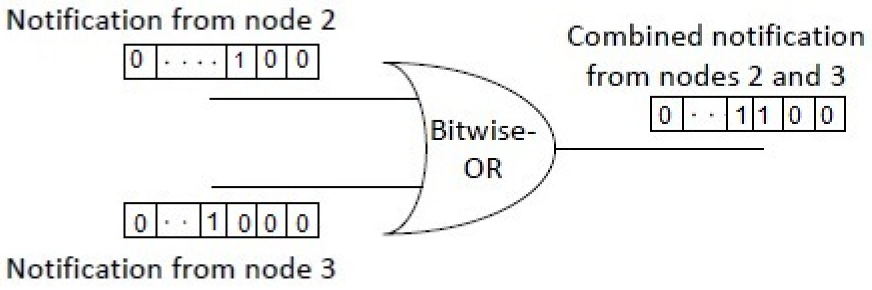


Figure 6.3: OR-ing of notification messages from different sources

From the notification message every node knows for which source's request it is waiting for. In an interconnect of 16 nodes, the highest priority is given to node 0 and the lowest priority to node 16.

When the request arrives at the node, the notification message is checked in order to determine the node whose request needs to be serviced first. Depending on this, the request message is dequed from the corresponding FIFO in the main network. Thus even though requests from different source nodes arrive at a particular node in any order, it will be serviced at all the nodes in a consistent global order. Thus, all the nodes locally determine the global order for servicing the requests.

If address matching is successful and hit occurs, then a response is send from that particular node to the requesting node. The response follows a path along the network as determined by the XY routing algorithm. The flit is first routed along the x direction and then along the y direction till it reaches the destination node.

In order to perform the action of address matching, a tag register was introduced at every node. In one of the nodes the tag value was kept equal to the value of address that we are passing in the request flit. Everywhere else the tag value is equal to zero. Therefore

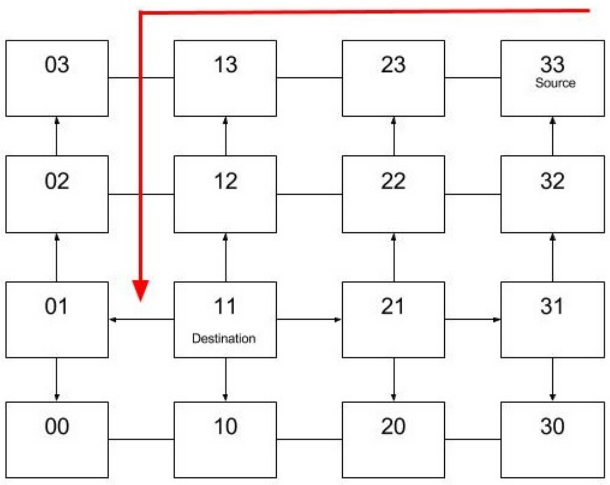


Figure 6.4: Routing the request back to the requesting node

when the request reaches that particular node with the corresponding tag value, hit occurs and response can be send back.

Chapter 7

SIMULATIONS RESULTS AND SYNTHESIS REPORT

7.1 HARDWARE DESIGN FLOW

Once the design has been coded in high level language, it needs to be realised into actual physical hardware. The VLSI design flow as depicted in Figure – outlines the major steps in converting the design towards physical realization.

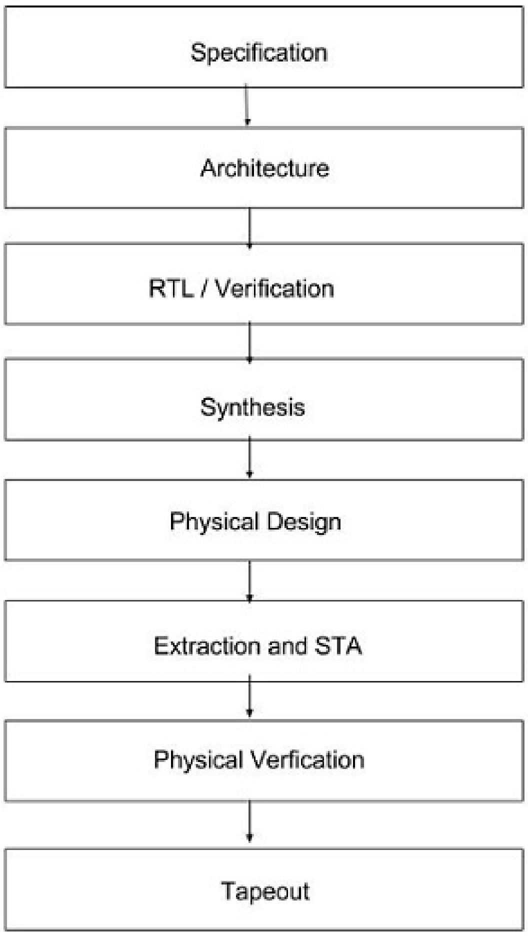


Figure 7.1: Flow of synthesis

The design of any product starts with an idea. The idea is born out of a client requirement. This idea is put down as a higher level behavioural model of the final product using the high level languages like BSV. The behavioural model is then compiled into a RTL using a suitable compiler. RTL is generally the description of the circuit at the module level where input output interfaces, clock and other signals are visible. Any design can be described in RTL using the Huffman's model.

Once the RTL is arrived at, the next step in the design flow is the logic synthesis. Using commercial EDA tools, the designer converts the RTL into a netlist which is nothing but a list of gates and wires whose inputs and outputs are specified. The EDA tools gives a lot of options like types of gates to be used, constraints for the design with respect to the power, area and timing, thus a highly optimized netlist is achieved after logic synthesis.

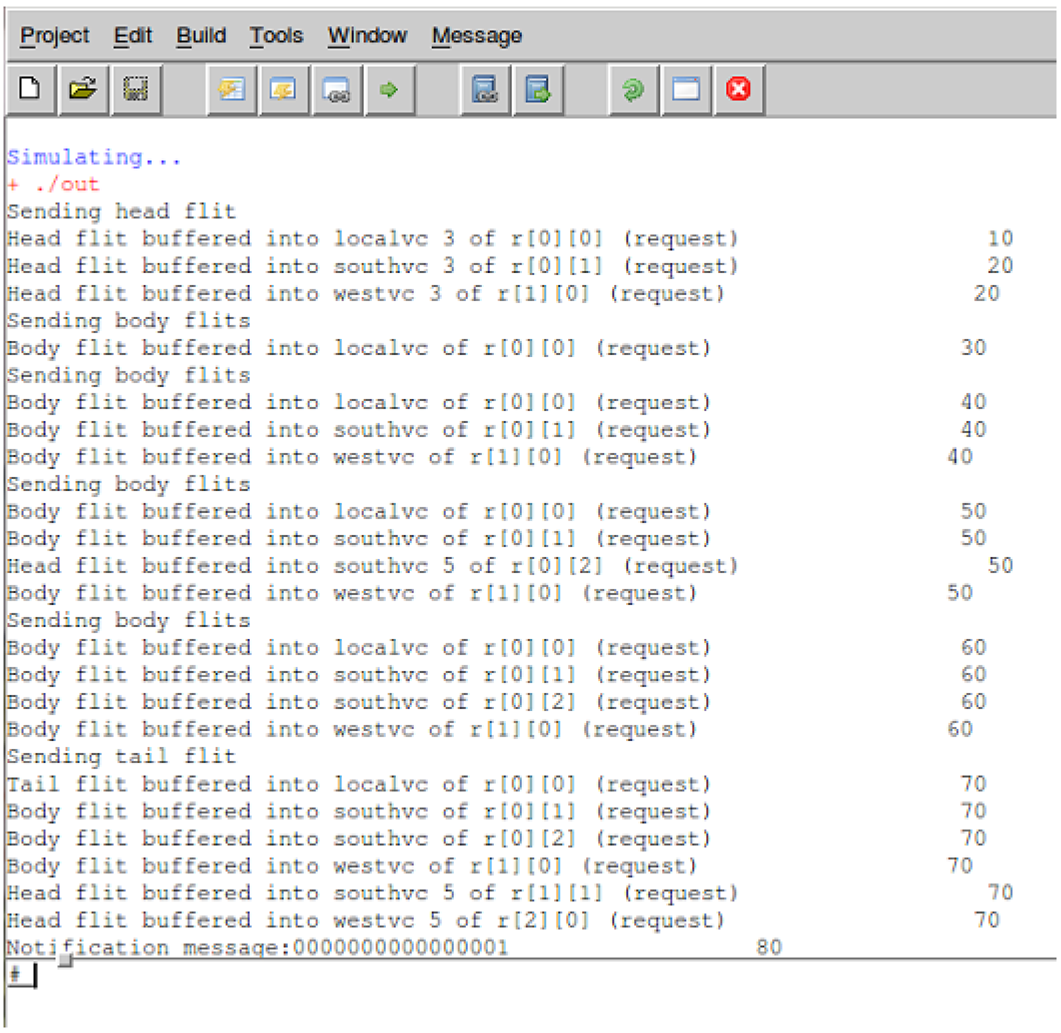
On getting the netlist, more EDA tools are used to do place and route of gates and wires or floor planning as it is popularly called. The result of place and route is the mask that could be handed over to the foundry for carrying out the fabrication of the chip. Two most important part of the design flow are the testing and verification. Testing is done to ensure final chip does not suffer from manufacturing defects and verification is done at each stage of the flow to ensure the design meets the requirements as were originally projected. In our case however, we limit the scope to design, implementation of the design in BSV, simulations and logic synthesis is done to verify performance.

7.2 SIMULATION

Once the BSV coding is done, the project is compiled with BSV compiler which gives options to compile for BSV simulator or to generate verilog files for further processing. We simulate the design using the Bluesim simulator. The results of the simulation are observed on the BSV GUI and recorded for analysis.The Simulation Results for some test cases are shown below.

7.2.1 TEST CASE: 1

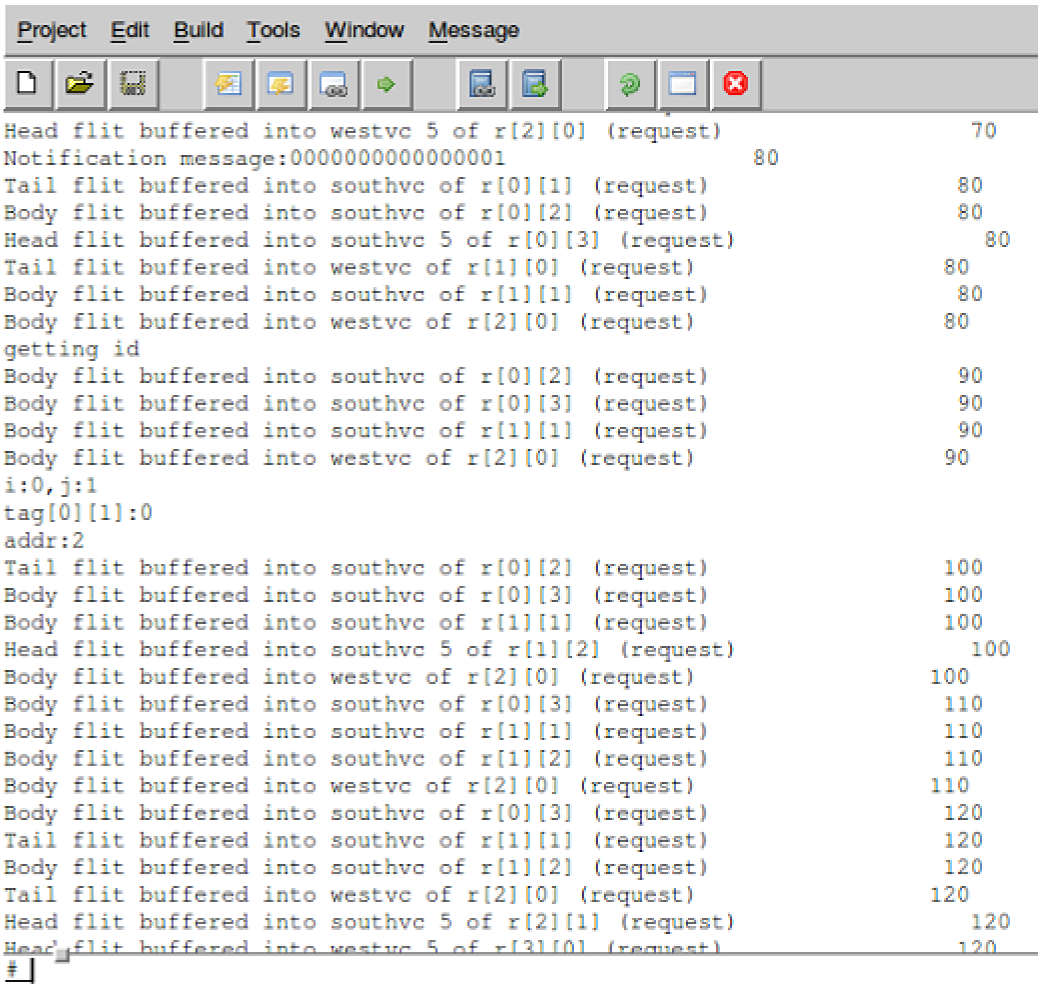
Let core 0 send out a request with address value equal to 2. The tag value in core 16, i.e., in core (3,3) is kept equal to 2 so that when address matching is done hit occurs here and a response signal is send back to core 0, i.e., core (0,0).



```
Simulating...
+ ./out
Sending head flit
Head flit buffered into localvc 3 of r[0][0] (request)          10
Head flit buffered into southvc 3 of r[0][1] (request)          20
Head flit buffered into westvc 3 of r[1][0] (request)          20
Sending body flits
Body flit buffered into localvc of r[0][0] (request)          30
Sending body flits
Body flit buffered into localvc of r[0][0] (request)          40
Body flit buffered into southvc of r[0][1] (request)          40
Body flit buffered into westvc of r[1][0] (request)          40
Sending body flits
Body flit buffered into localvc of r[0][0] (request)          50
Body flit buffered into southvc of r[0][1] (request)          50
Head flit buffered into southvc 5 of r[0][2] (request)          50
Body flit buffered into westvc of r[1][0] (request)          50
Sending body flits
Body flit buffered into localvc of r[0][0] (request)          60
Body flit buffered into southvc of r[0][1] (request)          60
Body flit buffered into southvc of r[0][2] (request)          60
Body flit buffered into westvc of r[1][0] (request)          60
Sending tail flit
Tail flit buffered into localvc of r[0][0] (request)          70
Body flit buffered into southvc of r[0][1] (request)          70
Body flit buffered into southvc of r[0][2] (request)          70
Body flit buffered into westvc of r[1][0] (request)          70
Head flit buffered into southvc 5 of r[1][1] (request)          70
Head flit buffered into westvc 5 of r[2][0] (request)          70
Notification message:0000000000000001          80
#
```

Figure 7.2: BSV simulation results for test case:1

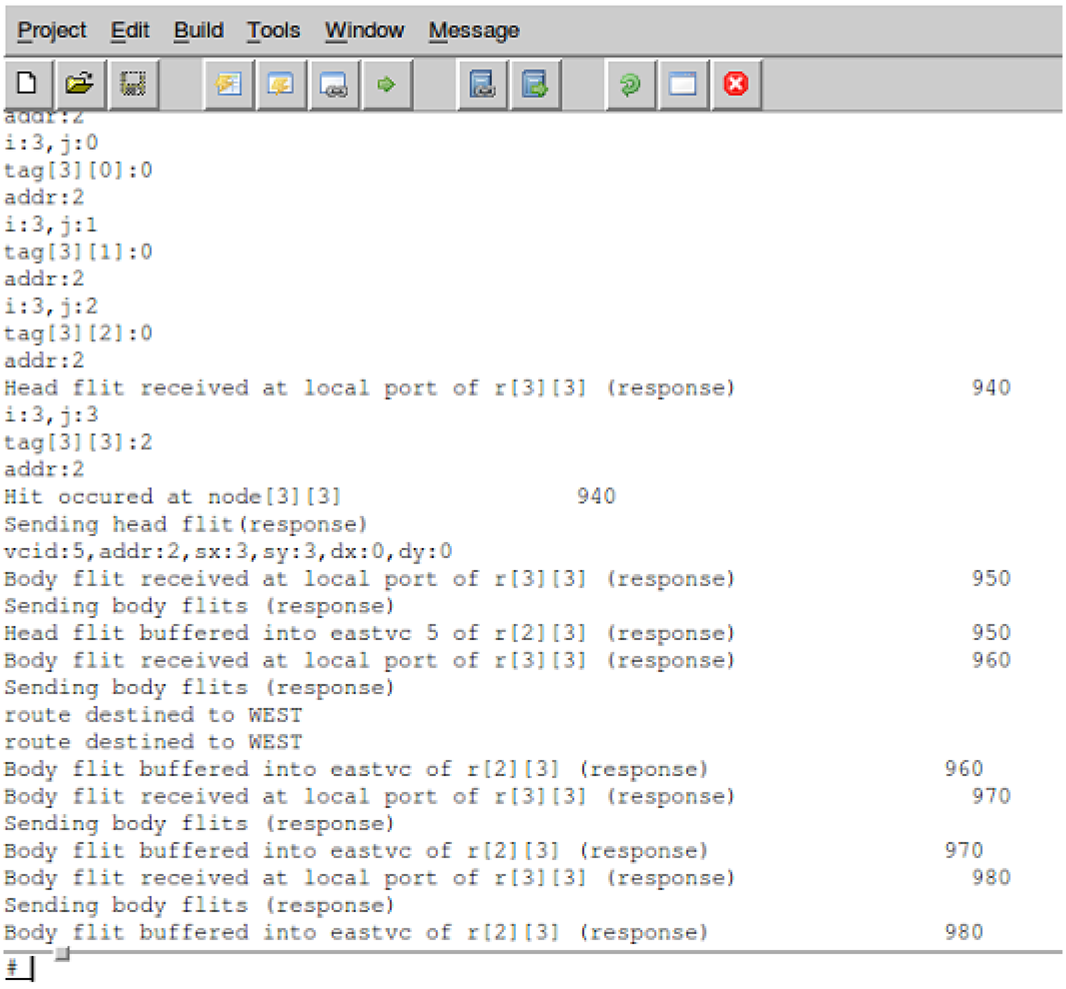
In figure 7.2 we can see that the headflit, four body flits and tail flit are send on to the local_out port of node (0,0). From there it passes on to the adjoining nodes by following the rules that were laid down for broadcasting.



```
Head flit buffered into westvc 5 of r[2][0] (request)                  70
Notification message:0000000000000001                        80
Tail flit buffered into southvc of r[0][1] (request)                 80
Body flit buffered into southvc of r[0][2] (request)                 80
Head flit buffered into southvc 5 of r[0][3] (request)                 80
Tail flit buffered into westvc of r[1][0] (request)                80
Body flit buffered into southvc of r[1][1] (request)                 80
Body flit buffered into westvc of r[2][0] (request)                80
getting id
Body flit buffered into southvc of r[0][2] (request)                 90
Body flit buffered into southvc of r[0][3] (request)                 90
Body flit buffered into southvc of r[1][1] (request)                 90
Body flit buffered into westvc of r[2][0] (request)                90
i:0,j:1
tag[0][1]:0
addr:2
Tail flit buffered into southvc of r[0][2] (request)                100
Body flit buffered into southvc of r[0][3] (request)                100
Body flit buffered into southvc of r[1][1] (request)                100
Head flit buffered into southvc 5 of r[1][2] (request)                100
Body flit buffered into westvc of r[2][0] (request)               100
Body flit buffered into southvc of r[0][3] (request)                110
Body flit buffered into southvc of r[1][1] (request)                110
Body flit buffered into southvc of r[1][2] (request)                110
Body flit buffered into westvc of r[2][0] (request)               110
Body flit buffered into southvc of r[0][3] (request)                120
Tail flit buffered into southvc of r[1][1] (request)                120
Body flit buffered into southvc of r[1][2] (request)                120
Tail flit buffered into westvc of r[2][0] (request)               120
Head flit buffered into southvc 5 of r[2][1] (request)                120
Head flit buffered into westvc 5 of r[3][0] (request)                 120
#
```

Figure 7.3: BSV simulation results for test case:1

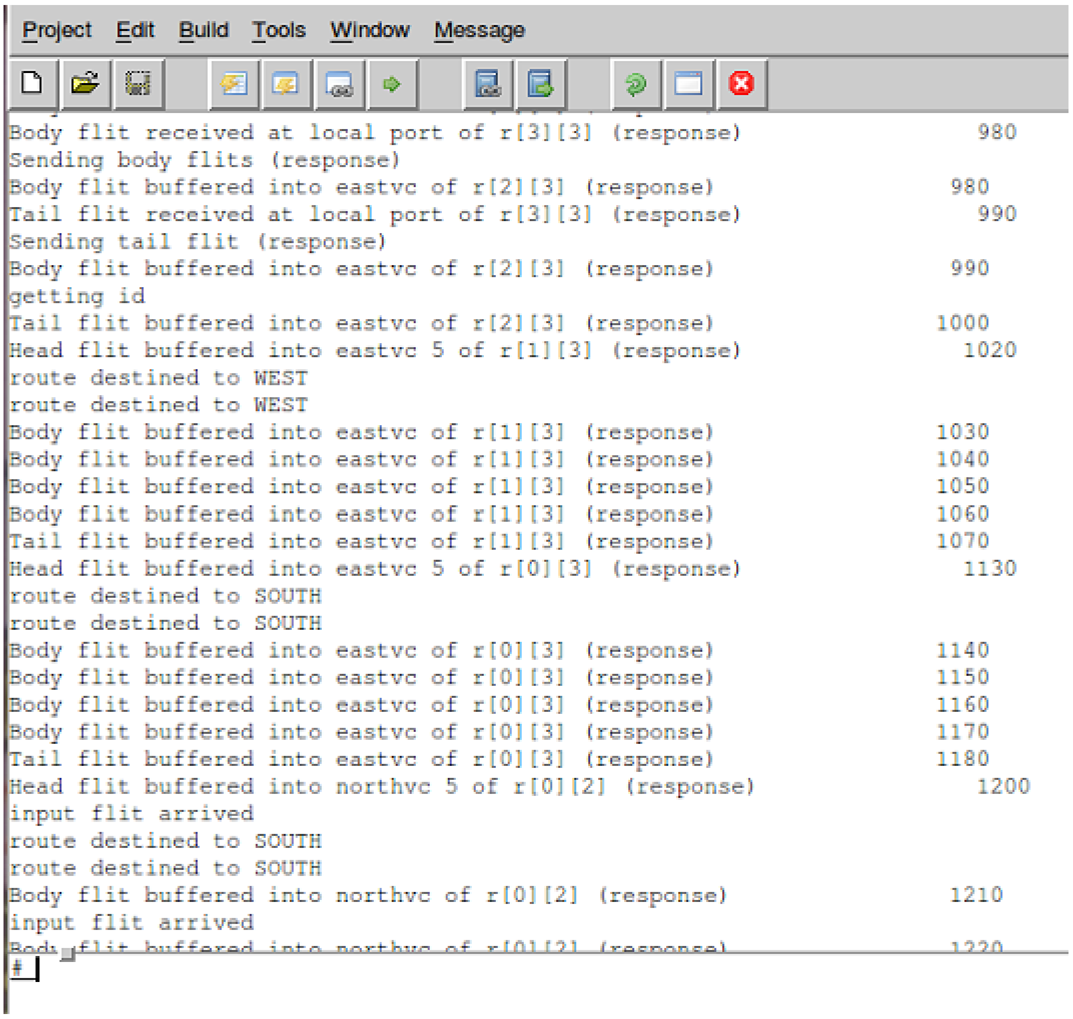
In figure 7.3 it can be seen that by this time the notification message is obtained from the notification network. The notification message is 0000000000000001 showing that a request from node (0,0) is in flight and all the nodes are waiting for it.



```
addr:2
i:3,j:0
tag[3][0]:0
addr:2
i:3,j:1
tag[3][1]:0
addr:2
i:3,j:2
tag[3][2]:0
addr:2
Head flit received at local port of r[3][3] (response)                 940
i:3,j:3
tag[3][3]:2
addr:2
Hit occured at node[3][3]                   940
Sending head flit(response)
vcid:5,addr:2,sx:3,sy:3,dx:0,dy:0
Body flit received at local port of r[3][3] (response)                 950
Sending body flits (response)
Head flit buffered into eastvc 5 of r[2][3] (response)                 950
Body flit received at local port of r[3][3] (response)                 960
Sending body flits (response)
route destined to WEST
route destined to WEST
Body flit buffered into eastvc of r[2][3] (response)                 960
Body flit received at local port of r[3][3] (response)                 970
Sending body flits (response)
Body flit buffered into eastvc of r[2][3] (response)                 970
Body flit received at local port of r[3][3] (response)                 980
Sending body flits (response)
Body flit buffered into eastvc of r[2][3] (response)                 980
#
```

Figure 7.4: BSV simulation results for test case:1

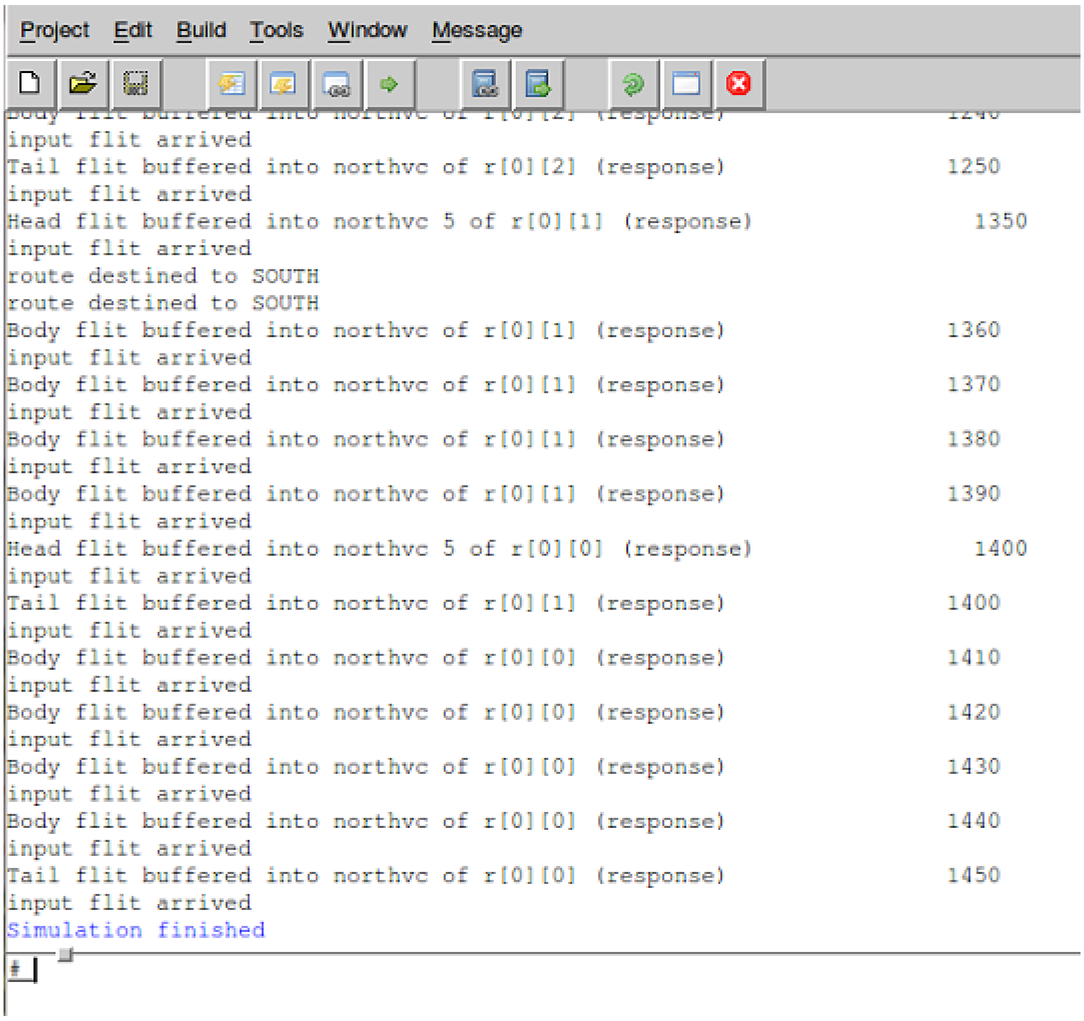
In figure 7.4 it can be seen that the tag value and the address matches at node (3,3). Therefore, response is send back from node (3,3).



```
Body flit received at local port of r[3][3] (response)              980
Sending body flits (response)
Body flit buffered into eastvc of r[2][3] (response)              980
Tail flit received at local port of r[3][3] (response)              990
Sending tail flit (response)
Body flit buffered into eastvc of r[2][3] (response)              990
getting id
Tail flit buffered into eastvc of r[2][3] (response)             1000
Head flit buffered into eastvc 5 of r[1][3] (response)             1020
route destined to WEST
route destined to WEST
Body flit buffered into eastvc of r[1][3] (response)             1030
Body flit buffered into eastvc of r[1][3] (response)             1040
Body flit buffered into eastvc of r[1][3] (response)             1050
Body flit buffered into eastvc of r[1][3] (response)             1060
Tail flit buffered into eastvc of r[1][3] (response)             1070
Head flit buffered into eastvc 5 of r[0][3] (response)             1130
route destined to SOUTH
route destined to SOUTH
Body flit buffered into eastvc of r[0][3] (response)             1140
Body flit buffered into eastvc of r[0][3] (response)             1150
Body flit buffered into eastvc of r[0][3] (response)             1160
Body flit buffered into eastvc of r[0][3] (response)             1170
Tail flit buffered into eastvc of r[0][3] (response)             1180
Head flit buffered into northvc 5 of r[0][2] (response)             1200
input flit arrived
route destined to SOUTH
route destined to SOUTH
Body flit buffered into northvc of r[0][2] (response)             1210
input flit arrived
Body flit buffered into northvc of r[0][2] (response)             1220
#
```

Figure 7.5: BSV simulation results for test case:1

In figure 7.5 it is seen that the response is routed along the network according to the path given out by the XY routing algorithm. It is first routed along X direction and then along Y direction to finally reach the destination as shown in figure 7.6.

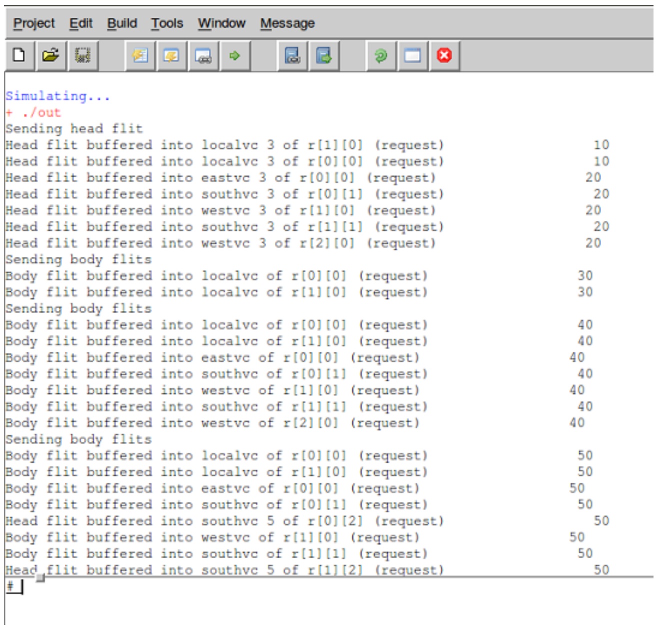


```
input flit arrived
Tail flit buffered into northvc of r[0][2] (response)          1250
input flit arrived
Head flit buffered into northvc 5 of r[0][1] (response)          1350
input flit arrived
route destined to SOUTH
route destined to SOUTH
Body flit buffered into northvc of r[0][1] (response)          1360
input flit arrived
Body flit buffered into northvc of r[0][1] (response)          1370
input flit arrived
Body flit buffered into northvc of r[0][1] (response)          1380
input flit arrived
Body flit buffered into northvc of r[0][1] (response)          1390
input flit arrived
Head flit buffered into northvc 5 of r[0][0] (response)          1400
input flit arrived
Tail flit buffered into northvc of r[0][1] (response)          1400
input flit arrived
Body flit buffered into northvc of r[0][0] (response)          1410
input flit arrived
Body flit buffered into northvc of r[0][0] (response)          1420
input flit arrived
Body flit buffered into northvc of r[0][0] (response)          1430
input flit arrived
Body flit buffered into northvc of r[0][0] (response)          1440
input flit arrived
Tail flit buffered into northvc of r[0][0] (response)          1450
input flit arrived
Simulation finished
```

Figure 7.6: BSV simulation results for test case:1

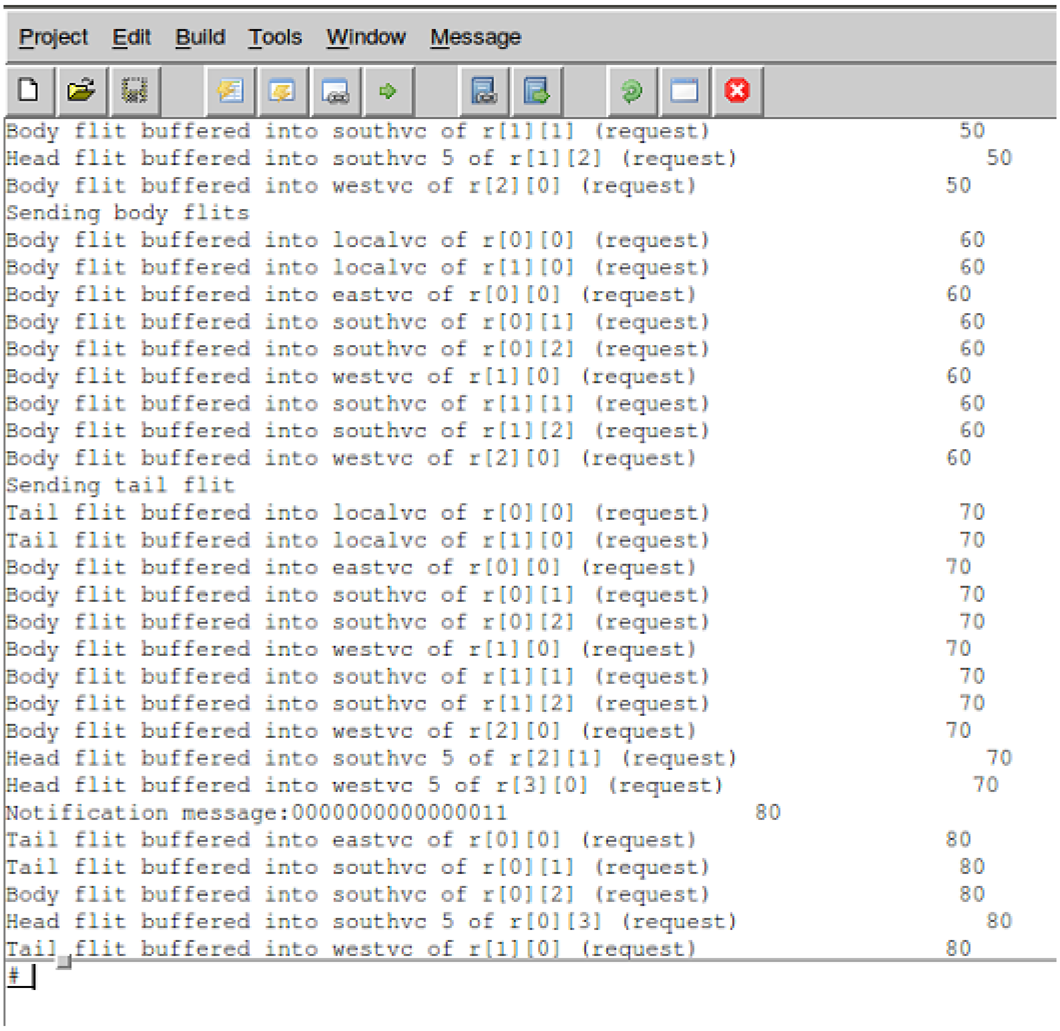
7.2.2 TEST CASE: 2

Now we consider the case when two requests are send from two different sources. In this example, a request is send from node (1,0) for which hit is supposed to occur at node (0,1). Another request is also send from node (0,0) for which hit should occur at node (0,2). The BSV simulation results are as shown:



```
Simulating...
+ ./out
Sending head flit
Head flit buffered into localvc 3 of r[1][0] (request)                10
Head flit buffered into localvc 3 of r[0][0] (request)                10
Head flit buffered into eastvc 3 of r[0][0] (request)                20
Head flit buffered into southvc 3 of r[0][1] (request)                20
Head flit buffered into westvc 3 of r[1][0] (request)                20
Head flit buffered into southvc 3 of r[1][1] (request)                20
Head flit buffered into westvc 3 of r[2][0] (request)                20
Sending body flits
Body flit buffered into localvc of r[0][0] (request)                30
Body flit buffered into localvc of r[1][0] (request)                30
Sending body flits
Body flit buffered into localvc of r[0][0] (request)                40
Body flit buffered into localvc of r[1][0] (request)                40
Body flit buffered into eastvc of r[0][0] (request)                40
Body flit buffered into southvc of r[0][1] (request)                40
Body flit buffered into westvc of r[1][0] (request)                40
Body flit buffered into southvc of r[1][1] (request)                40
Body flit buffered into westvc of r[2][0] (request)                40
Sending body flits
Body flit buffered into localvc of r[0][0] (request)                50
Body flit buffered into localvc of r[1][0] (request)                50
Body flit buffered into eastvc of r[0][0] (request)                50
Body flit buffered into southvc of r[0][1] (request)                50
Head flit buffered into southvc 5 of r[0][2] (request)                50
Body flit buffered into westvc of r[1][0] (request)                50
Body flit buffered into southvc of r[1][1] (request)                50
Head flit buffered into southvc 5 of r[1][2] (request)                50
#
```

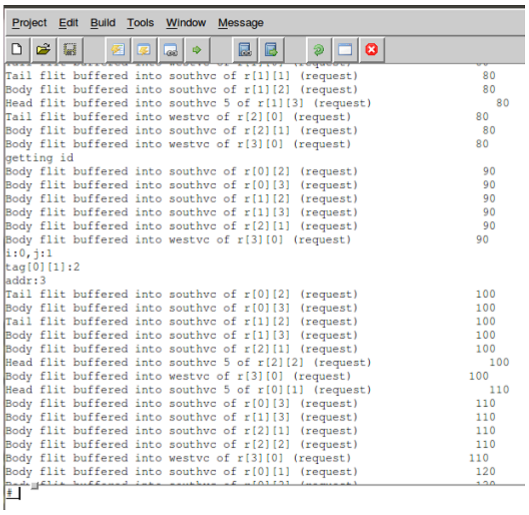
Figure 7.7: BSV simulation results for test case:2



```
Body flit buffered into southvc of r[1][1] (request)                  50
Head flit buffered into southvc 5 of r[1][2] (request)                  50
Body flit buffered into westvc of r[2][0] (request)                  50
Sending body flits
Body flit buffered into localvc of r[0][0] (request)                  60
Body flit buffered into localvc of r[1][0] (request)                  60
Body flit buffered into eastvc of r[0][0] (request)                  60
Body flit buffered into southvc of r[0][1] (request)                  60
Body flit buffered into southvc of r[0][2] (request)                  60
Body flit buffered into westvc of r[1][0] (request)                  60
Body flit buffered into southvc of r[1][1] (request)                  60
Body flit buffered into southvc of r[1][2] (request)                  60
Body flit buffered into westvc of r[2][0] (request)                  60
Sending tail flit
Tail flit buffered into localvc of r[0][0] (request)                  70
Tail flit buffered into localvc of r[1][0] (request)                  70
Body flit buffered into eastvc of r[0][0] (request)                  70
Body flit buffered into southvc of r[0][1] (request)                  70
Body flit buffered into southvc of r[0][2] (request)                  70
Body flit buffered into westvc of r[1][0] (request)                  70
Body flit buffered into southvc of r[1][1] (request)                  70
Body flit buffered into southvc of r[1][2] (request)                  70
Body flit buffered into westvc of r[2][0] (request)                  70
Head flit buffered into southvc 5 of r[2][1] (request)                  70
Head flit buffered into westvc 5 of r[3][0] (request)                  70
Notification message:0000000000000011                  80
Tail flit buffered into eastvc of r[0][0] (request)                  80
Tail flit buffered into southvc of r[0][1] (request)                  80
Body flit buffered into southvc of r[0][2] (request)                  80
Head flit buffered into southvc 5 of r[0][3] (request)                  80
Tail flit buffered into westvc of r[1][0] (request)                  80
#
```

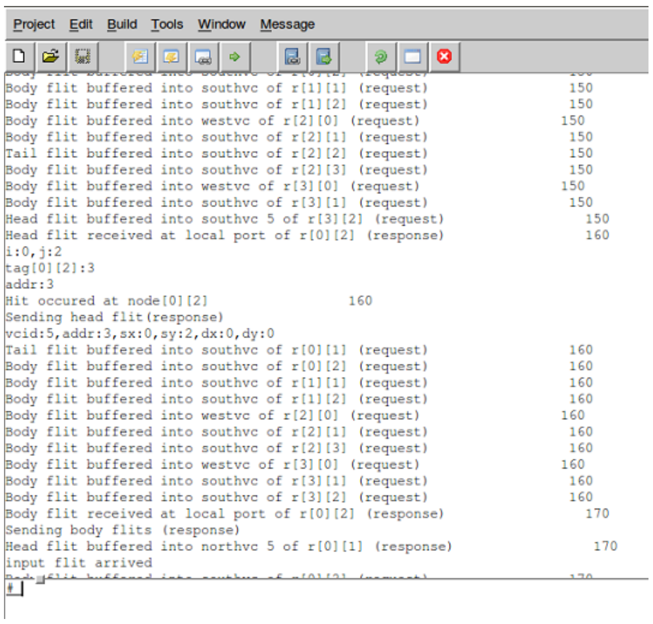
Figure 7.8: BSV simulation results for test case:2

In figure 7.8 it can be seen that the notification messages from both the sources have been merged together to give the notification message 0000000000000011. This indicates that requests are in flight from node 0 and node 1. All nodes will process the request from node 0 before it processes the request from node 1.



```
Tail flit buffered into southvc of r[1][1] (request)          80
Body flit buffered into southvc of r[1][2] (request)          80
Head flit buffered into southvc 5 of r[1][3] (request)          80
Tail flit buffered into westvc of r[2][0] (request)          80
Body flit buffered into southvc of r[2][1] (request)          80
Body flit buffered into westvc of r[3][0] (request)          80
getting id
Body flit buffered into southvc of r[0][2] (request)          90
Body flit buffered into southvc of r[0][3] (request)          90
Body flit buffered into southvc of r[1][2] (request)          90
Body flit buffered into southvc of r[1][3] (request)          90
Body flit buffered into southvc of r[2][1] (request)          90
Body flit buffered into westvc of r[3][0] (request)          90
i:0,j:1
tag[0][1]:2
addr:3
Tail flit buffered into southvc of r[0][2] (request)          100
Body flit buffered into southvc of r[0][3] (request)          100
Tail flit buffered into southvc of r[1][2] (request)          100
Body flit buffered into southvc of r[1][3] (request)          100
Body flit buffered into southvc of r[2][1] (request)          100
Head flit buffered into southvc 5 of r[2][2] (request)          100
Body flit buffered into westvc of r[3][0] (request)          100
Head flit buffered into southvc 5 of r[0][1] (request)          110
Body flit buffered into southvc of r[0][3] (request)          110
Body flit buffered into southvc of r[1][3] (request)          110
Body flit buffered into southvc of r[2][1] (request)          110
Body flit buffered into southvc of r[2][2] (request)          110
Body flit buffered into westvc of r[3][0] (request)          110
Body flit buffered into southvc of r[0][1] (request)          120
#
```

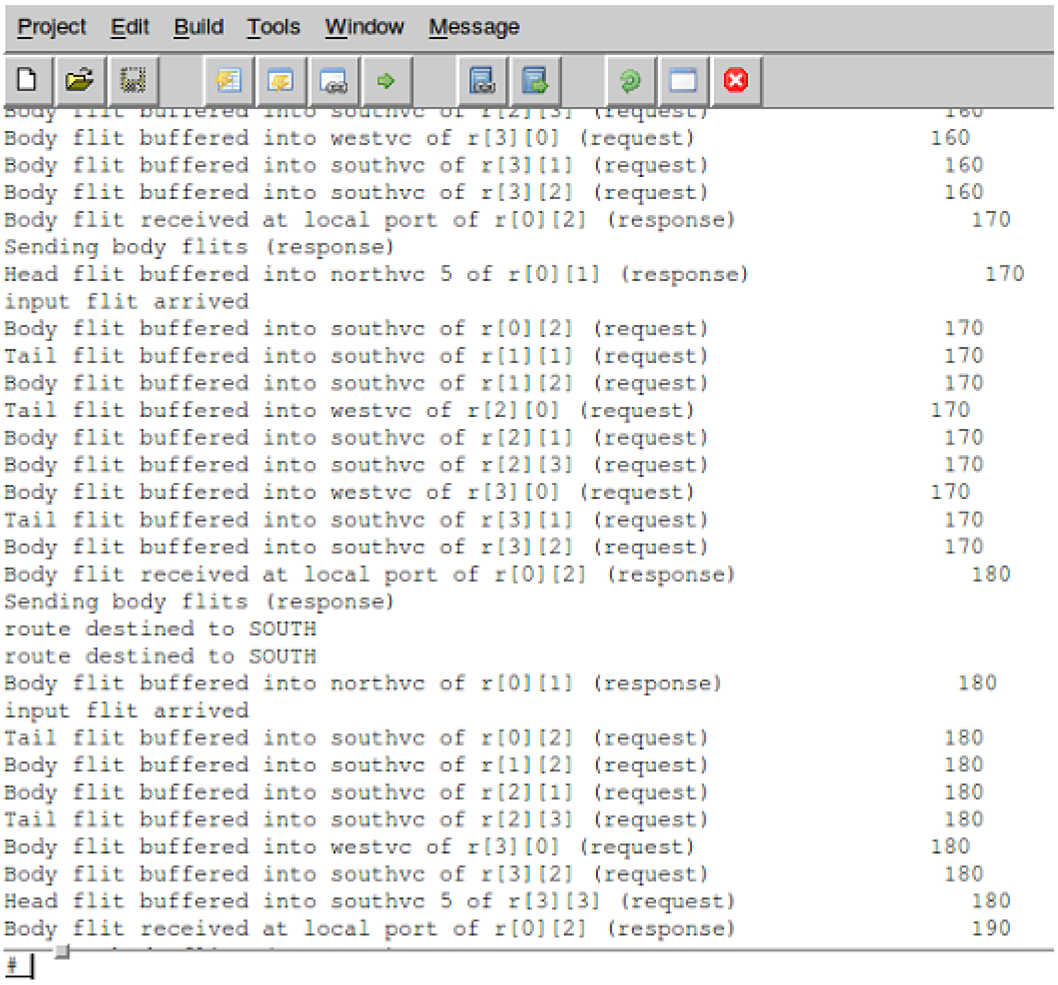
Figure 7.9: BSV simulation results for test case:2



```
Body flit buffered into southvc of r[1][1] (request)          150
Body flit buffered into southvc of r[1][2] (request)          150
Body flit buffered into westvc of r[2][0] (request)          150
Body flit buffered into southvc of r[2][1] (request)          150
Tail flit buffered into southvc of r[2][2] (request)          150
Body flit buffered into southvc of r[2][3] (request)          150
Body flit buffered into westvc of r[3][0] (request)          150
Body flit buffered into southvc of r[3][1] (request)          150
Head flit buffered into southvc 5 of r[3][2] (request)          150
Head flit received at local port of r[0][2] (response)          160
i:0,j:2
tag[0][2]:3
addr:3
Hit occured at node[0][2]          160
Sending head flit(response)
vcid:5,addr:3,sx:0,sy:2,dx:0,dy:0
Tail flit buffered into southvc of r[0][1] (request)          160
Body flit buffered into southvc of r[0][2] (request)          160
Body flit buffered into southvc of r[1][1] (request)          160
Body flit buffered into southvc of r[1][2] (request)          160
Body flit buffered into westvc of r[2][0] (request)          160
Body flit buffered into southvc of r[2][1] (request)          160
Body flit buffered into southvc of r[2][3] (request)          160
Body flit buffered into westvc of r[3][0] (request)          160
Body flit buffered into southvc of r[3][1] (request)          160
Body flit buffered into southvc of r[3][2] (request)          160
Body flit received at local port of r[0][2] (response)          170
Sending body flits (response)
Head flit buffered into northvc 5 of r[0][1] (response)          170
input flit arrived
#
```

Figure 7.10: BSV simulation results for test case:2

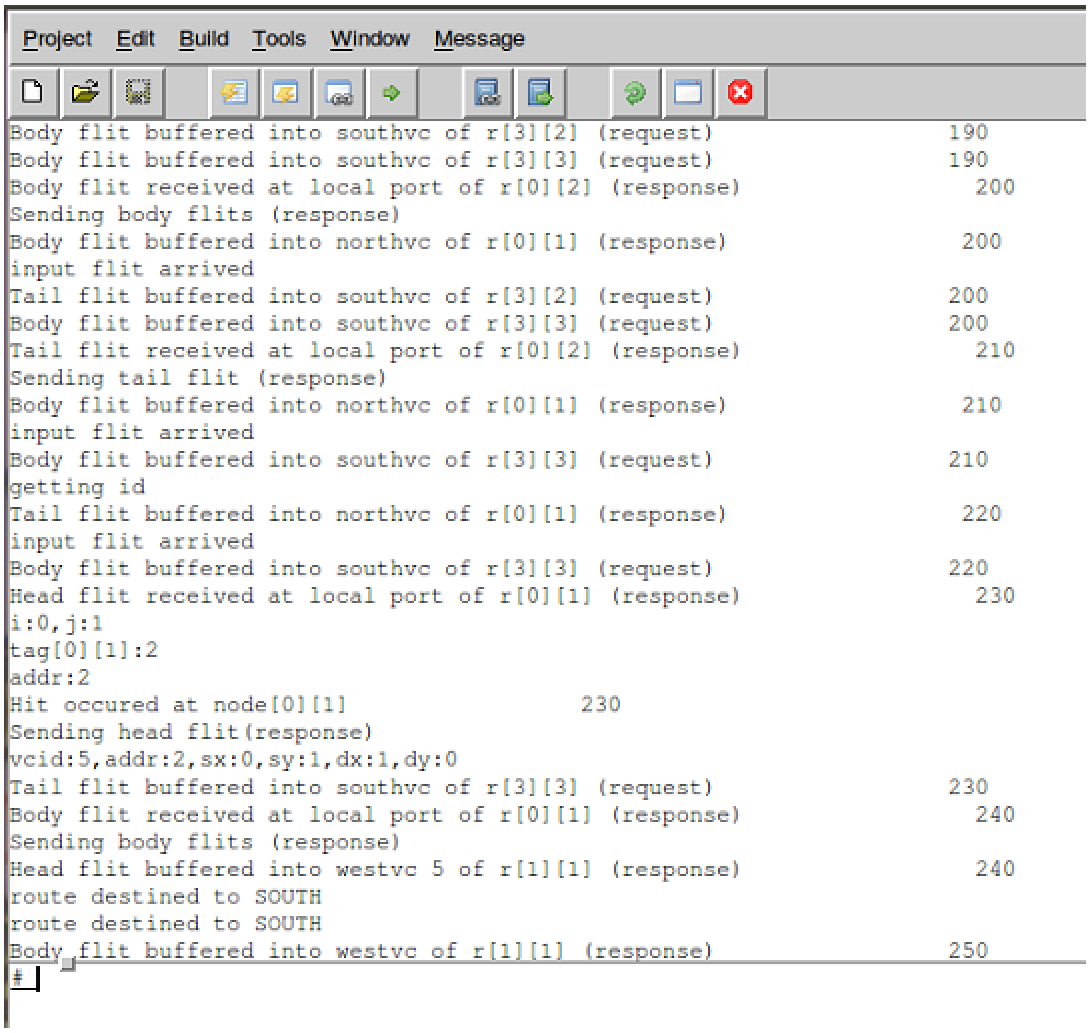
In figure 7.10 it can be seen that hit occurs at node (0,2) corresponding to the request that was send from node (0,0). Thus, the request from node (0,0) is processed before the request from node (1,0). The response is then send back to the source node. The response follows a route along the network as obtained from the XY routing algorithm.



```
Body flit buffered into southvc of r[2][3] (request)           160
Body flit buffered into westvc of r[3][0] (request)           160
Body flit buffered into southvc of r[3][1] (request)           160
Body flit buffered into southvc of r[3][2] (request)           160
Body flit received at local port of r[0][2] (response)           170
Sending body flits (response)
Head flit buffered into northvc 5 of r[0][1] (response)           170
input flit arrived
Body flit buffered into southvc of r[0][2] (request)           170
Tail flit buffered into southvc of r[1][1] (request)           170
Body flit buffered into southvc of r[1][2] (request)           170
Tail flit buffered into westvc of r[2][0] (request)           170
Body flit buffered into southvc of r[2][1] (request)           170
Body flit buffered into southvc of r[2][3] (request)           170
Body flit buffered into westvc of r[3][0] (request)           170
Tail flit buffered into southvc of r[3][1] (request)           170
Body flit buffered into southvc of r[3][2] (request)           170
Body flit received at local port of r[0][2] (response)           180
Sending body flits (response)
route destined to SOUTH
route destined to SOUTH
Body flit buffered into northvc of r[0][1] (response)           180
input flit arrived
Tail flit buffered into southvc of r[0][2] (request)           180
Body flit buffered into southvc of r[1][2] (request)           180
Body flit buffered into southvc of r[2][1] (request)           180
Tail flit buffered into southvc of r[2][3] (request)           180
Body flit buffered into westvc of r[3][0] (request)           180
Body flit buffered into southvc of r[3][2] (request)           180
Head flit buffered into southvc 5 of r[3][3] (request)           180
Body flit received at local port of r[0][2] (response)           190
```

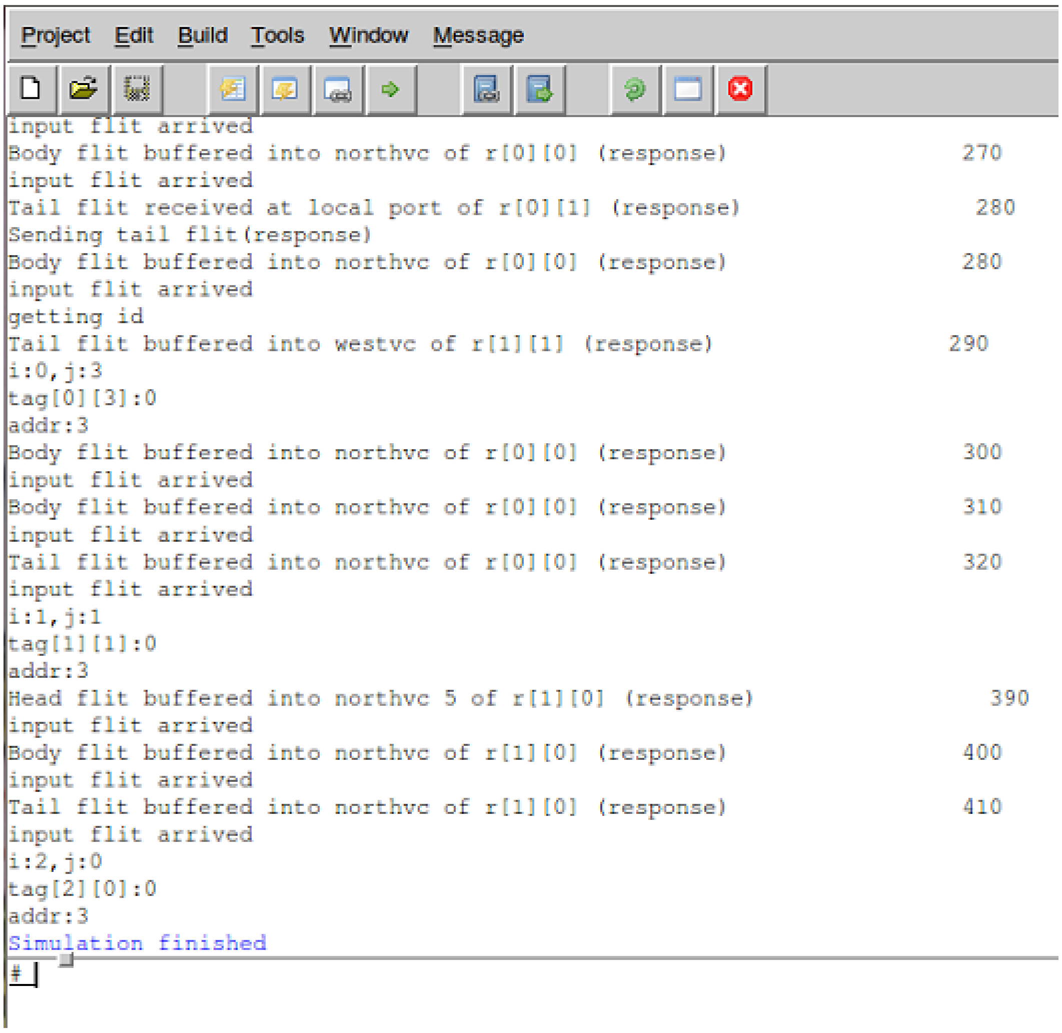
Figure 7.11: BSV simulation results for test case:2

In figure 7.12 it can be seen that hit occurs at node(0,2) corresponding to the request that ws send from node (1,0). The response is then send back from this node to node (1,0).



```
Body flit buffered into southvc of r[3][2] (request)                190
Body flit buffered into southvc of r[3][3] (request)                190
Body flit received at local port of r[0][2] (response)                200
Sending body flits (response)
Body flit buffered into northvc of r[0][1] (response)                200
input flit arrived
Tail flit buffered into southvc of r[3][2] (request)                200
Body flit buffered into southvc of r[3][3] (request)                200
Tail flit received at local port of r[0][2] (response)                210
Sending tail flit (response)
Body flit buffered into northvc of r[0][1] (response)                210
input flit arrived
Body flit buffered into southvc of r[3][3] (request)                210
getting id
Tail flit buffered into northvc of r[0][1] (response)                220
input flit arrived
Body flit buffered into southvc of r[3][3] (request)                220
Head flit received at local port of r[0][1] (response)                230
i:0,j:1
tag[0][1]:2
addr:2
Hit occured at node[0][1]                230
Sending head flit(response)
vcid:5,addr:2,sx:0,sy:1,dx:1,dy:0
Tail flit buffered into southvc of r[3][3] (request)                230
Body flit received at local port of r[0][1] (response)                240
Sending body flits (response)
Head flit buffered into westvc 5 of r[1][1] (response)                240
route destined to SOUTH
route destined to SOUTH
Body flit buffered into westvc of r[1][1] (response)                250
#
```

Figure 7.12: BSV simulation results for test case:2



```
input flit arrived
Body flit buffered into northvc of r[0][0] (response)              270
input flit arrived
Tail flit received at local port of r[0][1] (response)              280
Sending tail flit(response)
Body flit buffered into northvc of r[0][0] (response)              280
input flit arrived
getting id
Tail flit buffered into westvc of r[1][1] (response)              290
i:0,j:3
tag[0][3]:0
addr:3
Body flit buffered into northvc of r[0][0] (response)              300
input flit arrived
Body flit buffered into northvc of r[0][0] (response)              310
input flit arrived
Tail flit buffered into northvc of r[0][0] (response)              320
input flit arrived
i:1,j:1
tag[1][1]:0
addr:3
Head flit buffered into northvc 5 of r[1][0] (response)              390
input flit arrived
Body flit buffered into northvc of r[1][0] (response)              400
input flit arrived
Tail flit buffered into northvc of r[1][0] (response)              410
input flit arrived
i:2,j:0
tag[2][0]:0
addr:3
Simulation finished
#
```

Figure 7.13: BSV simulation results for test case:2

OBSERVATION:

1. Time taken for the request to reach all nodes = 28 clock cycles

2. Maximum latency for hit to occur = 94 clock cycles

3. Minimum latency for hit to occur = 10 clock cycles

4. Maximum latency for response to reach destination node = 46 clock cycles

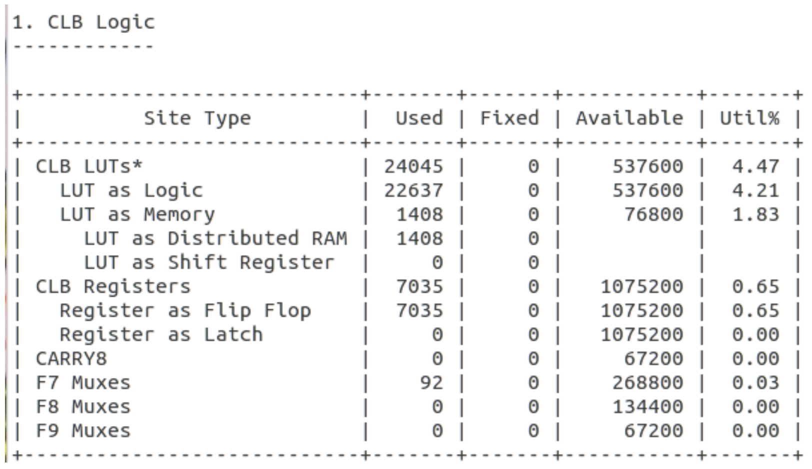
5. Minimum latency for response to reach destination node = 6 clock cycles

7.3 SYNTHESIS

Further the design is compiled to generate the verilog files which are required for the EDA tool to complete the logic synthesis as discussed in the design flow diagram. We not only receive an optimized netlist after logic synthesis but also reports for area and timing which are required for analysis.

The synthesis tool accepts the verilog files of the design and runs the synthesis algorithm for logic minimization. The synthesis culminates with generation of synthesized design schematic and detailed synthesis report with hardware units used in the final design. It is possible to selectively visualize the flow of the signals and the modules of interest making it convenient for the designer to verify the correctness of the design.

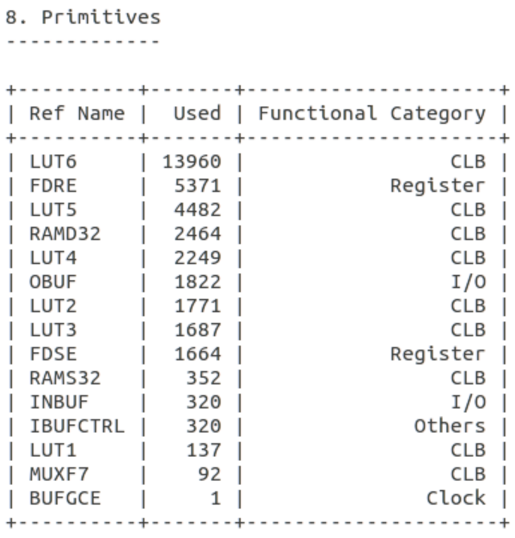
The design ws synthesised for a clock period of 5.3 ns. The worst slack obtained is 0.011 ns.The maximum frequency of operation is 188 MHz.



```
1. CLB Logic
------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| CLB LUTs*                  | 24045 |     0 |    537600 |  4.47 |
|   LUT as Logic             | 22637 |     0 |    537600 |  4.21 |
|   LUT as Memory            |  1408 |     0 |     76800 |  1.83 |
|     LUT as Distributed RAM |  1408 |     0 |           |       |
|     LUT as Shift Register  |     0 |     0 |           |       |
| CLB Registers              |  7035 |     0 |   1075200 |  0.65 |
|   Register as Flip Flop    |  7035 |     0 |   1075200 |  0.65 |
|   Register as Latch        |     0 |     0 |   1075200 |  0.00 |
| CARRY8                     |     0 |     0 |     67200 |  0.00 |
| F7 Muxes                   |    92 |     0 |    268800 |  0.03 |
| F8 Muxes                   |     0 |     0 |    134400 |  0.00 |
| F9 Muxes                   |     0 |     0 |     67200 |  0.00 |
+----------------------------+-------+-------+-----------+-------+
```

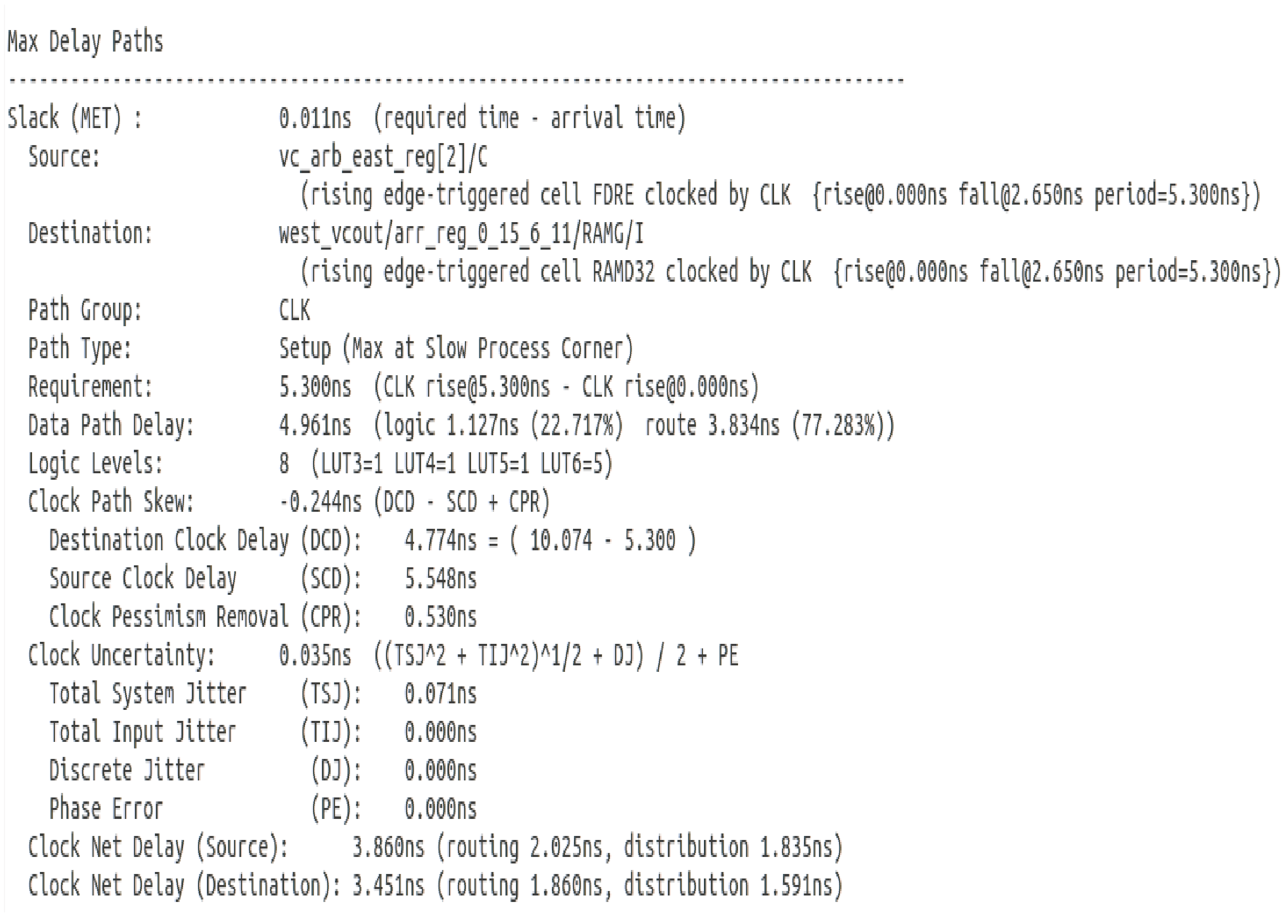
Figure 7.14: Utilisation report after synthesis



```
8. Primitives
-------------

+----------+-------+---------------------+
| Ref Name |  Used | Functional Category |
+----------+-------+---------------------+
| LUT6     | 13960 |                 CLB |
| FDRE     |  5371 |            Register |
| LUT5     |  4482 |                 CLB |
| RAMD32   |  2464 |                 CLB |
| LUT4     |  2249 |                 CLB |
| OBUF     |  1822 |                 I/O |
| LUT2     |  1771 |                 CLB |
| LUT3     |  1687 |                 CLB |
| FDSE     |  1664 |            Register |
| RAMS32   |   352 |                 CLB |
| INBUF    |   320 |                 I/O |
| IBUFCTRL |   320 |              Others |
| LUT1     |   137 |                 CLB |
| MUXF7    |    92 |                 CLB |
| BUFGCE   |     1 |               Clock |
+----------+-------+---------------------+
```

Figure 7.15: Utilisation as primitive blocks after synthesis



```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.011ns  (required time - arrival time)
  Source:                 vc_arb_east_reg[2]/C
                            (rising edge-triggered cell FDRE clocked by CLK  {rise@0.000ns fall@2.650ns period=5.300ns})
  Destination:            west_vcout/arr_reg_0_15_6_11/RAMG/I
                            (rising edge-triggered cell RAMD32 clocked by CLK  {rise@0.000ns fall@2.650ns period=5.300ns})
  Path Group:             CLK
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            5.300ns  (CLK rise@5.300ns - CLK rise@0.000ns)
  Data Path Delay:        4.961ns  (logic 1.127ns (22.717%)  route 3.834ns (77.283%))
  Logic Levels:           8  (LUT3=1 LUT4=1 LUT5=1 LUT6=5)
  Clock Path Skew:        -0.244ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.774ns = ( 10.074 - 5.300 )
    Source Clock Delay      (SCD):    5.548ns
    Clock Pessimism Removal (CPR):    0.530ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
  Clock Net Delay (Source):      3.860ns (routing 2.025ns, distribution 1.835ns)
  Clock Net Delay (Destination): 3.451ns (routing 1.860ns, distribution 1.591ns)
```

Figure 7.16: Timing report

Chapter 8

CONCLUSION AND FUTURE WORK

An ordered mesh network interconnect has been implemented by making use of a virtual channel router architecture so that unordered requests could be processed in a global order by making use of the notification network. This has provided a solution to the problem of ordering requests without any complexity. Further work can be carried out in order to create a more optimized network.

A single-cycle pipeline optimization can be achieved to reduce the network latency and buffer read/write power, by implementing lookahead bypassing, wherein, a lookahead containing control information for a flit is sent to the next router during that flit's switch traversal stage. At the next router, the lookahead performs route-computation and tries to pre-allocate the crossbar for the approaching flit. Lookaheads are prioritized over buffered flits.

The received requests are processed in the order determined by the notification network. Therefore, there is a possibility of the network running into a deadlock, as the request that the node is awaiting might not be able to enter the node because the buffers in the node and routers enroute are all occupied by other requests. To prevent this scenario, one reserved virtual channel (rVC) can be added to each router, reserved for the coherence request with source id equal to the expected source id of the node.

Also, we can enforce a mechanism where requests from the same source also can be ordered with respect to each other. Since requests are identified by source id alone, the main network must ensure that a later request does not overtake an earlier request from the same source. To enforce this two requests at a particular input port of a router cannot have the same source id. At each output port, a source id tracker table can be used which keeps track of the source id of the request in each virtual channel at the next router.

Bibliography

[1] Bluespec Inc., Bluespec System Verilog Reference Guide, Revision 30 July 2014.

[2] Rishiyur S. Nikhil and Kathy R. Czeck, "BSV by Example", 2010.

[3] Bhavya K. Daya et al, "SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering", ISCA, Jun 2014.

[4] Mark Hill, "On-Chip networks", University of Wisconsin Madison.

[5] Shubhangi D Chawade, Mahendra A Gaikwad and Rajendra M Patrikar, "Review of XY Routing Algorithm for Network-on-Chip Architecture", International Journal of Computer Applications (0975 – 8887), Volume 43– No.21, April 2012.

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

[7] William Dally, "Virtual channel flow control", ISCA, 1990.

[8] Shakti Series Processors.ppt, RISE Lab.

[9] John L. Hennesy and David A. Patterson, "Computer Architecture: A Quantitative Approach", 5th edition, 2011.