On the Design of a 3D Network-on-Chip for Many-core SoC

m5141153
Akram Ben Ahmed
Supervised by Prof. Ben Abdallah Abderazk
April 5, 2012

The Universityy of Aizu

Abstract

Global interconnects are becoming the principal performance bottleneck for high performance Systems-on-Chip (SoCs). Since the main purpose for this system is to shrink the size of the chip as smaller as possible while seeking at the same time for more scalability, higher bandwidth and lower latency. Conventional bus-based-systems are no longer reliable architecture for SoC due to a lack of scalability and parallelism integration, high latency and power dissipation, and low throughput. During this last decade, Network-on-Chip (NoC) has been proposed as a promising solution for future systems on chip design. It offers more scalability than the shared-bus based interconnection, allows more processors to operate concurrently.

Despite the higher scalability and parallelism integration offered by the Network-on-Chip (NoC) over the traditional shared-bus based systems, it's still not an ideal solution for future large scale Systems-on-Chip (SoCs), due to some limitations such as high power consumption, high cost communication, and low throughput. Recently, merging NoC to the third dimension (3D-NoC) has been proposed to deal with those problems, as it was a solution offering lower power consumption and higher speed.

In this this research, a 3D-NoC named OASIS (in short 3D-ONoC) has been designed to overcome the limitations of 2D-OASIS previously made in our research group. In this report we describe the 3D OASIS-NoC architecture in a fair amount of detail and present evaluation results and comparison between 3D and 2D OASIS.

Evaluation results show that despite the increasing hardware complexity, 3D-ONoC reduces the number of hops by 40% and also the average stall count by 74%. As a result the execution time improved by 36%. By increasing the traffic load with the Matrix

application, the execution time could be further enhanced from 36% obtained with one matrix multiplication to more than 41% with 1, 2, 3 and 4 matrix multiplications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Following *Moore's law*, the number of transistors kept increasing along the past few decades. That made shrinking the chip size while maintaining high performance possible. This technology scaling has allowed Systems-on-Chip (SoCs) [1, 2] systems to grow continuously in component count and complexity. Which significantly led to some very challenging problems such us power dissipation, resource management etc. In particular, the interconnection network starts to play a more and more important role in determining the performance and also the power consumption of the entire chip [3]. Those factors made conventional bus-based-systems and *P2P* no longer reliable architectures for SoC, due to the lack of scalability and parallelism integration, high latency and power dissipation, and low throughput.

Network-on-Chip [1, 4] was introduced as a promising method that can respond to these issues. Based on a simple and scalable architecture platform, NoC connects processors, memories and other custom designs together using switching packets on a hop-by-hop basis, in order to provide a higher bandwidth and higher performance. Figure1.1 (a) and Fig.1.1 (b) show one of the most well-known architectures which

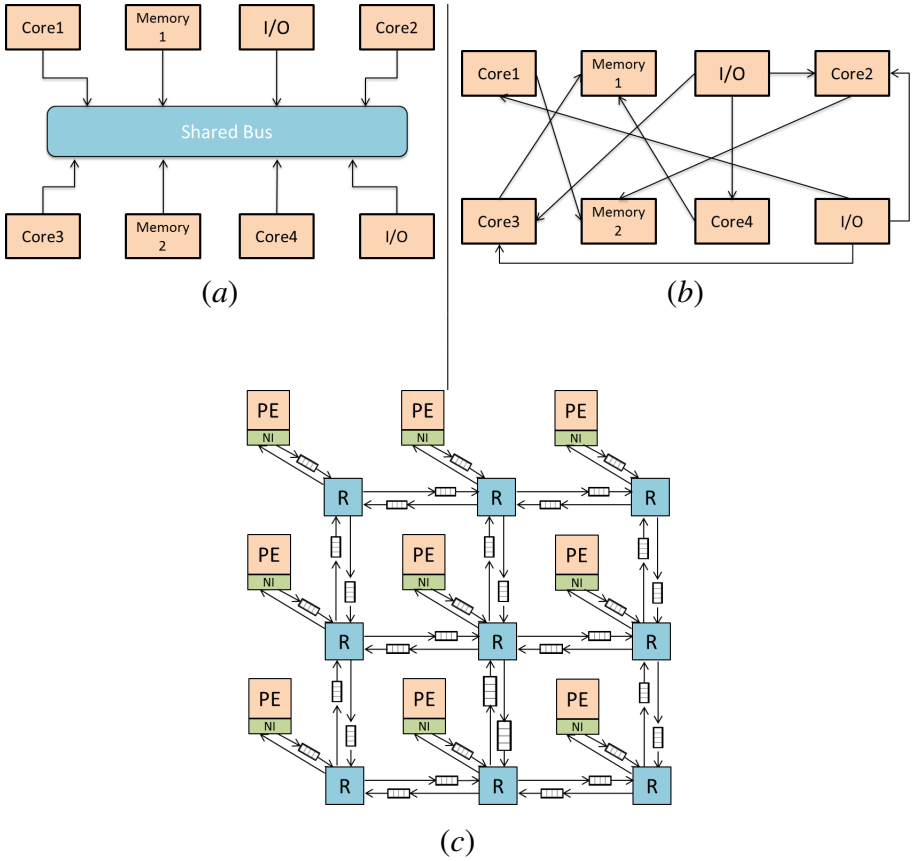

Figure 1.1: SoC architecture: (a) Shred-bus (b) Point-2-Point (c) NoC

are respectively Point-to-Point (*P2P*) and shared bus systems. As shown in Fig.1.1 (c), NoC architectures are based upon connecting segment (or wires) and switching blocks to combine the benefits of the two previous architectures while reducing their disadvantages, such us the large numbers of long wires in *P2P* and the lack of scalability in shared-bus systems.

1.2 Problems and Motivation

At the same time, future applications are getting more and more complex, demanding a good architecture to ensure a sufficient bandwidth for any transaction between

memories and cores as well as communication between different cores on the same chip. All this factors made NoC not enough reliable for future systems, especially when we talk about hundreds of cores. This limitation comes basically from the high diameter that suffers from NoC. The network's diameter is the number of hops that a flit traverses in the longest possible minimal path between a (source, destination) pair. The diameter is important for the NoC design since a large network diameter has a negative impact on the worst case routing latency in the network. For all these facts, the seek for optimizing NoC-based architecture becomes more and more necessary, and many researches have been conducted to achieve this goal in various approaches, such as developing fast routers [5, 6, 7, 8] or designing new network topologies [9, 10, 11].

One of these proposed solutions was merging the Network-on-Chip to the third dimension. In the past few years, three dimensional integrated circuits (3D-ICs) [12] have attracted a lot of attention as a potential solution to resolve the interconnect bottleneck. A three dimensional chip is a stack of multiple device layers with direct vertical interconnects tunneling through them [13, 14]. Researches made so far have shown that 3D-ICs can achieve higher packing density due to the addition of a third dimension to the conventional two-dimensional layout; and thanks to the reduced average interconnect length, 3D-ICs can achieve higher performance. Besides that, this reduction of total wiring, a lower interconnect power consumption can be obtained [15, 16], not forget to mention that circuitry is more immune to noise with 3D-ICs [12]. This may offer an opportunity to continue performance improvements using CMOS technology with smaller form factors, higher integration densities and supporting the realization

of mixed-technology chips [17]. As *Topol et al* in [16] stated, 3D-IC can improve the performance even in absence of scalability. Combining the NoC structure with the benefits of the 3D integration leads us to present 3D-NoC as a new architecture. This architecture responds to the scaling demands for future SoC, exploiting the short vertical links between the adjacent layers that can clearly enhance the system performance. This combination may provide a new horizon NoC design to satisfy the high requirements of future large scale applications.

One of the important design steps that should be taken care of while designing an 3D-NoC is to implement an efficient router, as it is the backbone of any NoC architecture. The router performance depends on many factors and techniques, such as the traffic pattern, the router pipeline design and the network topology. As *Feihui et al* in [18] mentioned, among these three factors we have less control over the traffic patterns compared with the topology and the pipeline design. Following this logic, and assuming the topology choice was already taken, one of the most important router enhancements that can be done is to improve the pipeline design, and then reducing the router delay. By reducing the pipeline delay, not only we decrease the per-hop delay, but also the whole network latency will be reduced.

On the other hand, the pipeline design is strongly associated with the routing algorithm adopted by the design. Routing is the process of determining the path that a flit should take between one source and one destination node. Routing algorithm can classified into minimal or non-minimal, depending on whether flits traveling from source to destination always use the minimal possible path or not. Minimal routing

schemes are shorter and require less complex hardware, but allowing non-minimal routes increases the path diversity and decreases the network congestion. Also the routing algorithms can be adaptive, where routing decisions are made based on the network congestion status and other information about network links or buffer occupancy of the neighboring nodes, or alternatively are deterministic. Although there are a large number of sophisticated adaptive routing algorithms, but they could require more complex implementation than that of the deterministic ones. That's why deterministic routing schemes has been adopted for 3D-NoC designs. One of the well-known and well used routing schemes used in 3D-NoCs is the Dimension Order Routing (DOR) XYZ algorithm. XYZ is a simple scheme, easy to implement and free of deadlock and lifelock. But on the other hand, it suffers from a non-efficient pipeline stage usage. This can introduce an additional packet latency which has an important effect on the router delay and eventually on the system overall performance. Enhancing this algorithm while keeping its simplicity may improve the system performance by reducing the packet delay.

Previously, in our research group, we proposed a 2D-NoC named OASIS [4, 19, 20]. Although 2D-OASIS-NoC has its advantages over the shared-bus based systems, it has also some limitations such as high power consumption, high cost communication, and low throughput.

Starting from all these facts, the main motivation of this work is to propose a 3D-NoC named 3D-OASIS-NoC which is an extension to our 2D-OASIS-NoC. 3D-OASIS-NoC uses our proposed efficient routing scheme named Look-ahead-XYZ (LA-XYZ). This algorithm improves the router pipeline design by parallelizing some stages

while taking advantage at the same time of the simplicity of the conventional XYZ. As a result, this routing scheme aims to enhance the router performance thereby achieving a low-latency design.

In this report, we present a complete architecture and design of 3D-OASIS-NoC in a fair amount of details. Evaluation results are also presented using real applications (JPEG encoder and Matrix Multiplication). We provide more details about the different components of 3D-OASIS-NoC including our proposed Look-ahead-XYZ routing scheme (LA-XYZ) and its ability to optimize the router pipeline design. We show how our design can present a better performance by reducing the congestion, decreasing the execution time and the power consumption when compared with the previously designed 2D-OASIS-NoC system.

1.3 Report organization

The rest of this report is organized as follow: In Chapter 2, we present some related works. Our proposed Look-ahead-XYZ routing algorithm (LA-XYZ) is described in Chapter 3, and then the architecture of the 3D-OASIS-NoC system is described in details in Chapter 4. Chapter 5 presents evaluation methodology and results. Finally, we end the report with concluding remarks and future works in Chapter 6.

Chapter 2

Related Works

In this chapter, we present some of the related works to 3D-NoC. Starting from those who focused on the benefits of 3D-NoC when compared with 2D designs, passing by those who investigated about the router architecture and routing algorithms dedicated for 3D-NoC.

2.1 3D-NoC versus 2D-NoC

3D-NoC is a widely studied research topic, and many related works have been conducted until now. Few of them focused on the benefits of the 3D-NoC architecture over the traditional 2D-NoC design. *Feero et al* [21] showed that 3D-NoC has the ability to reduce latency and the energy per packet by decreasing the number of hopes by 40% which is a basic and important factor to evaluate the system performance [21].

Pavlidis et al [22] analyzed the zero-load latency and power consumption, and demonstrated that a decrease of 62% and 58% in power consumption can be achieved with 3D-NoC when compared to a traditional 2D-NoC topology for a network size of N= 128 and N= 256 nodes, respectively, where N is the number of cores connected in the network. This power consumption reduction can simply be related to the reduction of number of hops, since a flit has less hops to traverse to go from one source to its

destination, and that includes less buffer access, less switch arbitration, and less link and crossbar traversal. All of these factors will eventually lead to decrease the power consumption.

2.2 3D-NoC router architecture

Another part of the researches focused on the router architecture. For example, *Li et al* [23] has modified the conventional 7x7 3D router using a shared bus as a communication interface between the different layers of the router, to create a *3D NoC-Bus Hybrid* router. This kind of routers reduces in fact the number of ports in each router from 7 to 6, but on the other hand flits wishing to travel from one layer to another should compete the access to the shared bus, since it's the only inter-layer communication interface. This may lead to undesirable performance degradation especially under a heavy inter-layer traffic.

Yan et al [24], also proposed another architecture for the the 3D-router, by implementing all the vertical links into a single 3D-crossbar. In this case, the router has only 5 ports since we dont need any more additional ports for the vertical connections. This technique reduces the inter-layer distance, and makes the travel between the different layers in one single hop possible. But this router also engenders a high router cost besides the implementation complexity of such router, which cannot be acceptable for some simple application that actually does not need such a complex router.

For all these facts, we adopted for our design, as most of the 3D-NoC designs use, the conventional 7x7 3D-router, as it is the lowest cost among the other architectures and also the simplest to implement showing several properties like regularity, concur-

rent data transmission, and controlled electrical parameters [25, 26]. All the benefits are acquired while making sure that this low cost and simple implementation does not affect the performance of our system.

2.3 3D-NoC routing algorithms

Many routing algorithms have been proposed for MPSoC networks but most of them focus only on 2D-network topologies. Among all the studies conducted for 3D-NoC few of them focused on routing algorithms. Between the few proposed ones, there are some custom routing schemes that aims to reduce the power consumption and thermal power which is a very challenge design for 3D-NoC systems. For instance, *Ramanujam et al* [27] presented an oblivious routing algorithm called randomized partially minimal (RPM) that aims to load balance the traffic along the network improving then the worst case scenario. RPM sends packets to a random layer first, then route them along their X and Y dimensions using either XY or YX routing with equal probability. Finally packets are sent to their final destination along the Z dimension.

In a quiet similar technique, *Chao et al* [28] addressed the thermal power problem in 3D-NoC, which is one of the most important issues in the 3D-NoC designs. Starting from the fact the upper layer in the network detains the highest thermal power in the design, they proposed a thermal aware downward routing scheme that sends first the traffic to a downer layer, routes along the X and Y dimension before sending the packets back up to their destination layer. This technique avoids communication in upper layers, where the thermal power is more important than the downer ones, and then may reduce the overall thermal power in the design. Thus, ensuring thermal safety while

guaranteeing less performance impact from temperature regulation.

Both of these two routing algorithms have their advantages in term of load balancing and thermal power reduction. But the routing used is not minimal, which effect in a direct way the number of hops. By adopting a non-minimal routing, the packet delay may increase in the system, especially when we talk about a large number of connected nodes.

To ensure a minimal path for flits when traveling the network while making the routing as simple as possible, the majority of the remaining 3D-NoC systems have been using the conventional minimal Dimension Order Routing (DOR) XYZ routing scheme. Other introduced a routing scheme based upon XYZ such as the case of *Tyagi* in [29] who extended a previous routing algorithm [30] called *BDOR* designated for 2D-NoC. *BDOR* forwards packets in one of two routes (XY- or YX-orders), depending on relative position of a source-destination pair, and that aims to improve the balance of paths along the network also when taking into account the destination.

XYZ routing scheme, and all the routing algorithms based upon it, is presented as a vertically balanced routing algorithm which has the best performance, since it's simple to implement, it is free of deadlock and lifelock, and also because packet ordering is not required [28, 31, 32]. On the other hand, it cannot always make the best use of each pipeline stage. For the simple reason that since the Switch Allocation stage (SA) is always dependent on the previous Routing Calculation (RC) one. This dependency can be explained by the fact that SA stage needs information about the desired output-port calculated from the RC stage, where the incoming flits should go through in order to pass to the next neighboring node. To solve this problem in 2D-NoC systems using

the Dimension Order Routing (DOR) XY routing scheme, a smart pipeline design can be adopted with the help of some advanced techniques like look-ahead routing [29]. This kind of routing has been used to reduce the pipeline stages in the router, by parallelizing some of these stages then reducing the router delay and then enhancing the system performance. Look-ahead routing has indeed been used with 2D-NoC but it hasn't been adopted for 3D Network-on-Chip architectures before.

A second problem that can be seen with a lot of conventional router using XYZ-based routing schemes, is in case of no-load traffic and when the input buffer is empty, the flit entering the router should be first stored in the input buffer before advancing the next RC stage even there is no any flit under process in the next stages. This unnecessary stall will increase the packet latency in the router, and its associated power consumption, adding a performance overhead to the whole system even in a light traffic case where the system is supposed to have a close-to-optimal performance since there is no congestion that may increase the latency. In order to face this problem, a technique called no-load bypass is used [33]. This technique allows the flit to advance to the RC stage in case where the buffer is empty. Then overlapping the unnecessary buffer writing stage (BW) then decreasing the router delay.

Previously in [34], a part of this research has been including architecture of a 3D Network-on-Chip architecture (named 3D-OASIS-NoC) based on a previously designed 2D-OASIS-NoC. The design's performance was evaluated using a simple application that randomly generates flits and sends them along the network. But real application could not be evaluated due to the absence of some components in the design such us the network interface. For that reason, a network interface has been added

to 3D-ONoC, the optimized version of 3D-OASIS-NoC, in order to make our system able to be evaluated with our real selected target applications (JPEG encoder and Matrix Multiplication).

Starting from all the facts already stated, in this report we present a complete architecture and design of 3D-OASIS-NoC. Also evaluation results are presented using real applications (JPEG encoder and Matrix Multiplication). We provide more details about the different components of 3D-OASIS-NoC including our proposed Look-ahead-XYZ routing scheme(LA-XYZ) and its ability to take advantage of the simplicity of the conventional XYZ algorithm, while improving the pipeline design of the 3D-NoC router then enhancing the overall performance. Our lookahead routing scheme means that each flit additionally carries one hot encoded *Next-Port* identifier used by the downstream router. The no-load bypass technique is also associated with LA-XYZ in order to get more pipeline improvement. We show how our design can present a better performance by reducing the congestion, decreasing the execution time and the power consumption when compared with the previously designed 2D-OASIS-NoC system.

Chapter 3

Look Ahead XYZ routing algorithm

In this section, the proposed Look Ahead XYZ routing algorithm (LA-XYZ) adopted for 3D-ONoC is shown. Its out-performance against the conventional Dimension Order Routing (DOR) XYZ algorithm is also explained in term of optimizing the router pipeline design that eventually leads to a performance enhancement.

Most of the 3D-NoC systems are based upon the Dimension Order Routing (DOR) XYZ algorithm. XYZ routes flits first along the X dimension, then along the Y and finally the flit is routed along the Z dimension to reach its destination. This process is done by comparing the address of the processing node with the destination node's address to determine the *Output-Port*:

- if *xdest* is larger than *xaddr* then *Output-Port* will be EAST. In the opposite case *Output-Port* will be WEST.

- if *ydest* is larger than *yaddr* then *Output-Port* will be NORTH, else *Output-Port* will be SOUTH.

- if *zdest* is larger than *zaddr* then *Output-Port* will be UP, and if this condition is not satisfied *Output-Port* will be DOWN.

- if *xdest* is equal to *xaddr*, *ydest* is equal to *yaddr* and *zdest* is equal to *zaddr* then *Output-Port* will be SELF.

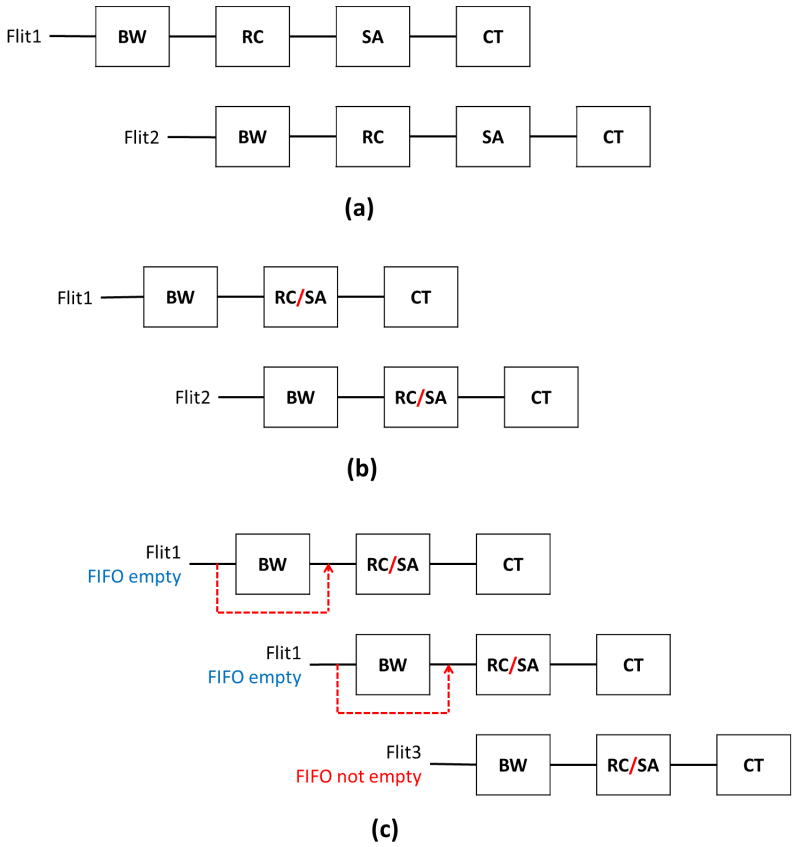

Figure 3.1: Router pipeline stages: (a) conventional XYZ (b) LA-XYZ (c) LA-XYZ with no-load bypass.

The computed *Output-Port* issued from XYZ is sent then to the Switch Arbiter asking for grant to access the selected output-port. XYZ is a simple scheme, easy to implement and free of deadlock and lifelock. But on the other hand, it suffers from a non-efficient pipeline stage usage. Figure.3.1 (a) depicts a conventional router pipeline design based on XYZ scheme. As we stated at the end of Section 2, Virtual Channels are not taken into consideration for improving the performance of best-effort traffic,

and also for seek of simplicity, a packet is composed of one single flit.

Taking a closer look at Fig.3.1 (a), we can see that conventional XYZ-based router pipeline design contains 4 main pipeline stages: Buffer Writing (BW) where the incoming flit is stored in the input buffer, then in Routing Calculation stage (RC) destination address is fetched and decoded to determine the *Output-Port* direction. Information about the selected *Output-Port* are sent to the next stage, Switch Arbitration (SA), to resolve any competition between different requests from different input-ports. Finally the Crossbar traversal stage (CT) handles the transfer of the flit to the next neighboring node. This 4 pipelines router design increases the flit latency and its associated power consumption, since any flit should go through all these stages at each hop while traveling from source to destination. This can introduce a undesirable system overall performance degradation, especially when we talk about a large network size where the network diameter also increases, which might not satisfy the high requirements of some application.

In such kind of schemes, the pipeline stages are dependent on each others, and each one of them can make its computation unless it receives information from the previous stage. This dependency is especially seen between the (RC) and (SA) stages. To face this dependency problem, our proposed Look Ahead XYZ (LA-XYZ) optimizes the router pipeline design by parallelizing the (RC) and (SA) stages and then eliminating the dependency between them. LA-XYZ pre-computes the *Next-Port* direction of the downstream router and then embeds it in the flit. When arriving to the downstream node, this hot encoded *Next-Port* identifier will be used by the switch arbiter directly to ask the grant for using the selected output-port and reach the next neighboring node. At

Algorithm 1: LA-XYZ 1st phase: Assign next address

```
// Current node address
Input: X_cur, Y_cur, Z_cur
// Next port identifier
Input: Next-port
// Next node address
Output: X_next, Y_next, Z_next

// Evaluate Next node x_address
if (Next-port is EAST) then
  | X_next ← X_cur + 1;
else
  | if (Next-port is WEST) then
  |   | X_next ← X_cur − 1;
  | else X_next ← X_cur;
end

// Evaluate Next node y_address
if (Next-port is NORTH) then
  | Y_next ← Y_cur + 1;
else
  | if (Next-port is SOUTH) then
  |   | Y_next ← Y_cur − 1;
  | else Y_next ← Y_cur;
end

// Evaluate Next node z_address
if (Next-port is UP) then
  | Z_next ← Z_cur + 1;
else
  | if (Next-port is DOWN) then
  |   | Z_next ← Z_cur − 1;
  | else Z_next ← Z_cur;
end
```

the same time, when the the grant is computed in (SA), the (RC) calculates in parallel the direction of the *Next-Port* that will be used by the next downstream node. This parallel process reduces the pipeline stages from 4 to 3 with LA-XYZ as it explained in Fig.3.1 (b).

Algorithm 2: LA-XYZ 2nd phase: Define new Next-port

```
// Destination address
Input: X_dest, Y_dest, Z_dest
// Next node address
Input: X_next, Y_next, Z_next
// New next port for next node
Output: New-next-port

if (X_next is equal to X_dest) then
    if (Y_next is equal to Y_dest) then
        if (Z_next is equal to Z_dest) then
            New-next-port← LOCAL;
        else
            if (Z_next is smaller than Z_dest) then
                New-next-port← UP;
            else New-next-port← DOWN;
        end
    end
    else
        if (Y_next is smaller than Y_dest) then
            New-next-port← NORTH;
        else New-next-port← SOUTH;
    end
end
else
    if (X_next is smaller than X_dest) then
        New-next-port← EAST;
    else New-next-port← WEST;
end
```

LA-XYZ computation goes under two steps: *Assign next address* and *Define new Next-port*. As it illustrated in Algorithm.1, the first step fetches the *Next-Port* identifier from the incoming flit. Depending on the direction of this identifier, the address of the next downstream node can be predicted. This address is then used in the second

step (shown in Algorithm.2) by comparing it with the destination address of the flit which is also fetched from the flit head and then decoded. At the end of this process, informations about the *Next-Port* is issued then embedded again in the flit to be used as a source of information for the switch allocator in the downstream node.

For further optimization, the no-load bypass technique can be also associated with LA-XYZ. As it is shown in Fig.3.1 (c), the number of pipeline stages can be further minimized by overlapping the (BW) stage. In case where the input FIFO buffer is empty, the flit doesn't have to be stored in the input buffer but it continues its path straight to the (RC) and (SA) where the computation of both stages are still done in parallel. As a result, the number of pipeline stages can be further minimized from 3 to 2. Then, enhancing the system execution time, latency and power consumption, and especially the zero-load latency.

Since *LA-XYZ* is based upon *XYZ* routing, it is still a free of deadlock and live-lock routing algorithm. It is considered also as a minimal Dimension Order routing where each flit from any source and destination pair traverses the minimal number of hops from source to destination and where packet ordering is not required. As a result, LA-XYZ improves the router design while taking advantages of the simplicity of XYZ.

Chapter 4

3D-ONoC System Architecture

3D-ONoC is a scalable Network-on-Chip based on *Mesh* topology. The packets are forwarded among the network using *Wormhole-like* switching policy and then routed according to *Look-Ahead-XYZ* routing algorithm (LA-XYZ). As a flow control, 3D-ONoC adopts *Stall-Go* mechanism and *Matrix-Arbiter* as a scheduling technique.

The remaining parts of this chapter explain each component of 3D-ONoC system in a fair amount of details. We clarify also the reasons why some techniques has been chosen to be adopted for our design.

4.1 Topology

The 3D-ONoC system is based upon *Mesh* topology, where *x-addr*, *y-addr* and *z-addr* are attributed to each router and define its X, Y and Z coordinates respectively and its position along the network. Many topologies exist for the implementation of NoCs, some are regular (*Torus, tree-based*) and other irregular topologies are customized for some special application. We choose the Mesh topology for this design thanks to its several properties like regularity, concurrent data transmission, and controlled electrical parameters [25, 26].

Figure.4.1 shows a configuration example of 4x4x4 3D-ONoC design. We can see

in this figure that different layers are linked between each other via inter-layer channels. On the other side, each layer is composed of different switches which are connected to each other using some intra-layer links, each one of them is connected to one single processing element.

Code.4.1 illustrates the Verilog-HDL code in the 3D-ONoC top module that defines the mesh topology. the z-loop, y-loop and x-loop are used to define the dimensions of 3D-NoC. While the internal i-loop (line 17) is used to define the different input and output ports for each direction. For example, $i = 0$ refers to the local port where, the outputs and inputs of this port will be allocated later to the attached PE.

Taking the example of the Down port (line 31-39) , the output and input of this port are allocated to the UP port of the router situated just below the current router, which means the one in the the downer layer.

As it will be explained later, the unused ports should eliminated in order to reduce the area and power consumption. Continuing with the same DOWN port, it should be disabled when the router is located at the bottom of the topology, which means when z-pos=0. In this case, ad as it is illustrated in Code.4.1 (line 32-35), net-data-in and net-stop-in are assigned to 0.

Code 4.1: Verilog-HDL code defining the topology

```
1  generate
2  //z loop
3  for (z_pos=0; z_pos<Z_WIDTH; z_pos=z_pos+1) begin:z_loop
4
5     //y loop
6     for (y_pos=0; y_pos<Y_WIDTH; y_pos=y_pos+1) begin:y_loop
7
8       //x loop
9       for (x_pos=0; x_pos<X_WIDTH; x_pos=x_pos+1) begin:x_loop
10
11      router #(NOUT, FIFO_DEPTH, FIFO_LOG2D, FIFO_FULL_LVL) rtr(.clk(clk), .reset(
            reset),
12          .data_in(net_data_in[x_pos][y_pos][z_pos]), .data_out(net_data_out[x_pos][
                y_pos][z_pos]),
13          .stop_in(net_stop_in[x_pos][y_pos][z_pos]), .stop_out(net_stop_out[x_pos][
                y_pos][z_pos]),
```

```
14          .xaddr(x_pos[`L2NET_SIZE-1:0]), .yaddr(y_pos[`L2NET_SIZE-1:0]), .zaddr(z_pos
                [`L2NET_SIZE-1:0]));
15
16        //set up inter-router connections with correct boundary conditions
17        for (i=0; i<NOUT; i=i+1) begin:i0
18
19           //tile interface of router
20           if(i==0) begin
21        assign net_data_in[x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = data_in[(`
              WIDTH*X_WIDTH*z_pos*Y_WIDTH)+(`WIDTH*X_WIDTH*y_pos)+ (`WIDTH*(x_pos+1))-1:
              (`WIDTH*X_WIDTH*z_pos*Y_WIDTH)+(`WIDTH*X_WIDTH*y_pos)+(`WIDTH*x_pos)];
22        assign data_out[(`WIDTH* X_WIDTH*z_pos*Y_WIDTH)+(`WIDTH*X_WIDTH*y_pos)+(`WIDTH
              *(x_pos+1))-1: (`WIDTH*X_WIDTH*z_pos*Y_WIDTH) +(`WIDTH*X_WIDTH*y_pos)+ (`
              WIDTH*x_pos)] = net_data_out[x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i];
23
24        assign net_stop_in[x_pos][y_pos][z_pos][i] = stop_in[(X_WIDTH* z_pos * Y_WIDTH
              )+(X_WIDTH*y_pos)+x_pos];
25        assign stop_out[(X_WIDTH* z_pos * Y_WIDTH)+(X_WIDTH*y_pos)+x_pos] =
              net_stop_out[x_pos][y_pos][z_pos][i];
26           end
27  ...
28  ...
29
30  //down edge of router
31     if(i==6) begin
32        if(z_pos==0) begin
33           assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
34           assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
35        end else begin
36           assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                 net_data_out[x_pos][y_pos][z_pos-1][`WIDTH*(5+1)-1:`WIDTH*5];
37           assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos][y_pos][
                 z_pos-1][5];
38        end
39     end
40
41      end // for (i=0; i<NOUT-1; i=i+1)
42     end // block: x_loop
43    end // block: y_loop
44  end // block: z_loop
```

4.2 Switching policy

Considered as a very important choice for any NoC design, switching establishes the type of connection between any upstream and downstream node. It is important to deploy an efficient switching policy to ensure less blocking communication while trying to minimize the system complexity.

When it is related to packet switching, three main switching policies have been mostly used for NoC: *Store and Forward (SAF)*, *Virtual Cut Through (VCT)* and *Wormhole (WH)* [35].

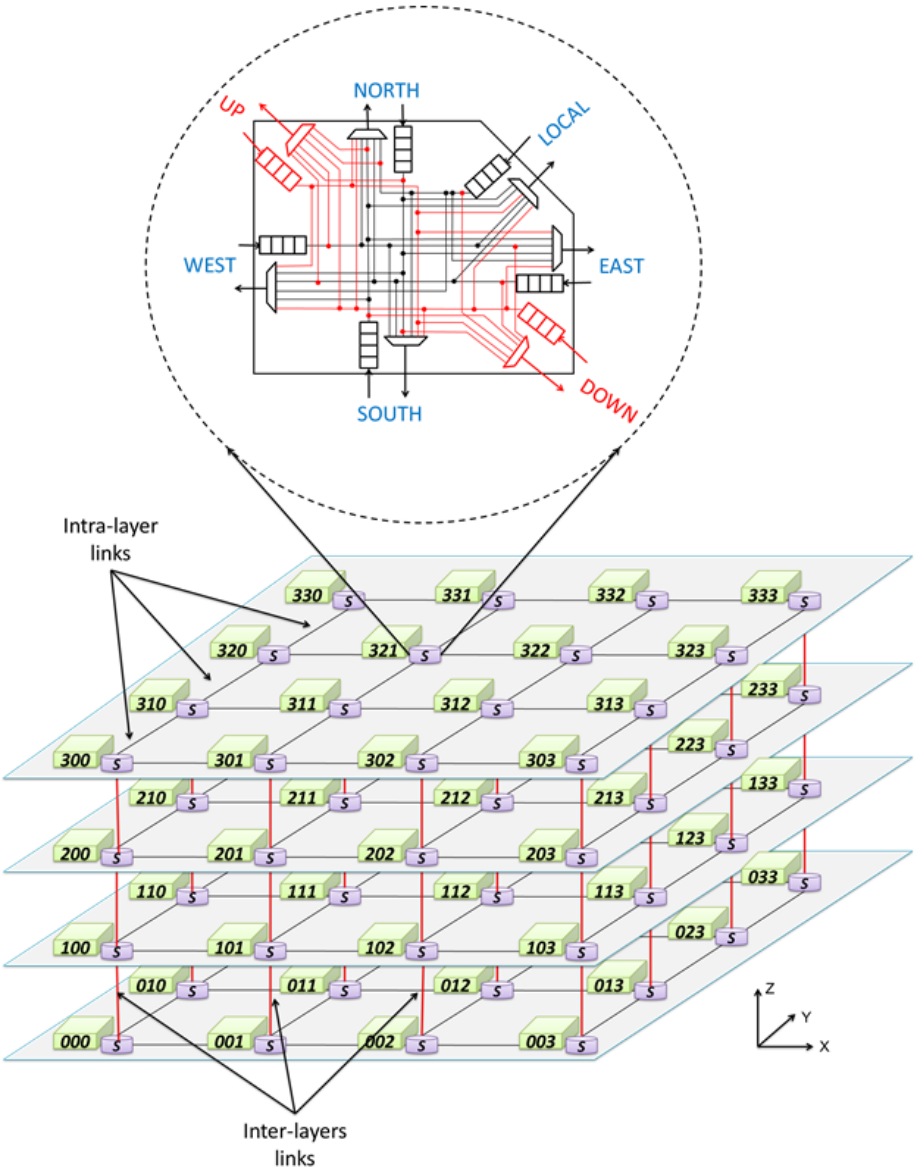

Figure 4.1: Configuration example of a 4x4x4 3D-ONoC mesh topology.

Code 4.2: Verilog-HDL code defining the flit structure

```
// Flit structure
‘define DATA              37:0
‘define TAIL              0
‘define NEXT_PORT         7:1
‘define XDEST             10:8
‘define YDEST             13:11
‘define ZDEST             16:14
‘define DATA              37:17
```

3D-ONoC adopts *Wormhole-like* switching and Virtual-Cut-Through forwarding method. The forwarding method which is chosen in a given instance depends on the level of packet fragmentation. For instance, each router in 3D-ONoC has input buffers which can store up to four flits by default. When a packet is divided into more than four flits, 3D-ONoC chooses Virtual-Cut-Through switching. When packets are divided into less than four flits, the system chooses Wormhole. In other words, when buffer size is greater than or equal to the number of flits, Virtual-Cut-Through is used, but when buffer size is less than or equal to the number of flits, Wormhole switching is employed. By combining the benefits of both switching techniques, packet forwarding can be executed in an efficient way while guaranteeing a small buffer size. As a result the system performance is enhanced while maintaining a reasonable area utilization and power consumption.

Figure 4.2 demonstrates the 3D-ONoC 81 bits flit format. The first bit indicates the *tail* bit informing the end of the packet. The next seven bits are dedicated to indicate the *Next-Port* that will be used by the *Look-Ahead-XYZ* routing algorithm to define the direction of the next downstream neighboring node where the flit will be sent to. Then, three bits are used to store destination information of each *xdest*, *ydest* and *zdest*. Having three bits for each destination field allows the network to have a maximum size of 8x8x8 3D-ONoC. But if the network size needs to be extended, the addresses fields

may also be increased to accommodate a larger network size. Finally the remaining 64 bits are dedicated to store the payload. Since 3D-ONoC is targeted for various applications, the payload size can be easily modified in order to respect the requirements of some specific applications. Code.4.2 shows the definition of the 3D-ONoC flit format. In addition, as we previously stated, the architecture does not provide for a separate head flit and every flit therefore identifies its destination X, Y, and Z addresses and carries an additional single bit to indicate whether its a tail flit or not.

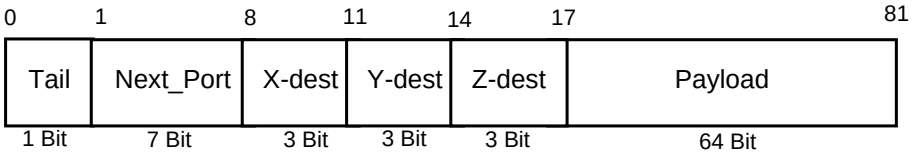

Figure 4.2: 3D-ONOC flit format.

4.3 Router architecture

The router is considered as the back-bone element in the whole 3D-ONoC design. The 3D-ONoC router architecture is based upon the 5x5 2D-ONoC router where, as shown in Fig.4.1, each switch has a maximum number of 7-input by 7-output port, where 4 ports are dedicated to connect to the neighboring routers in north, east, south and west direction using the intra-layer links. One port is used to connect the router to the local computation tile where the packet can be injected into or ejected from the network. The remaining two ports are added to connect the switch to the upper and downer layers to ensure the inter-layer communication. As a matter of fact, and as we previously stated, the number of ports depends on the position of the switch in the design, since we have to eliminate any unused links that have no connections with

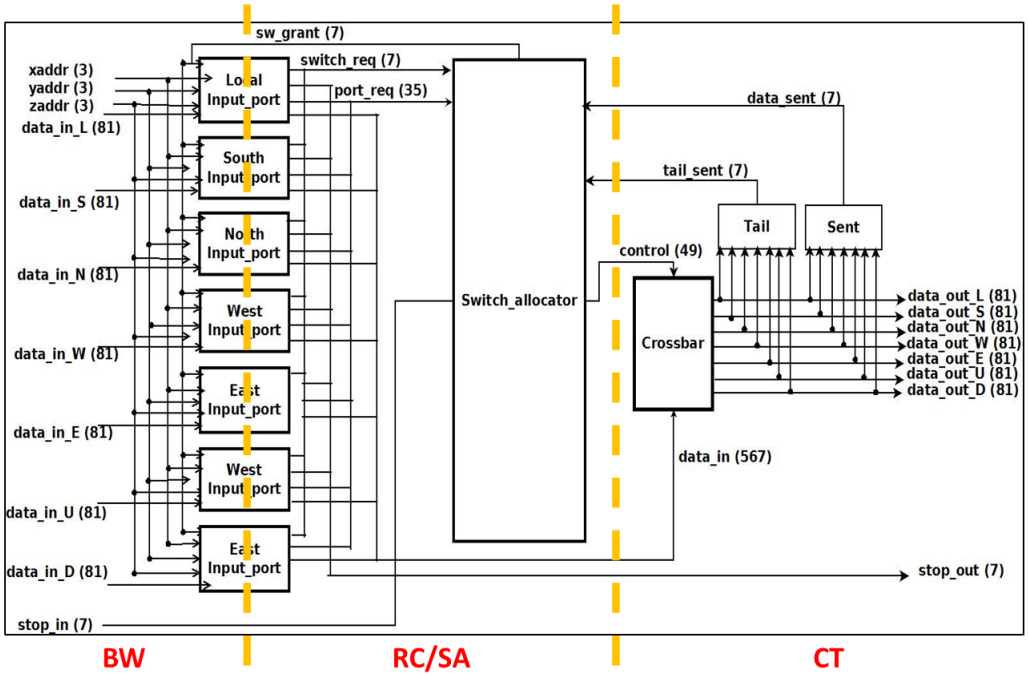

Figure 4.3: 3D-ONoC pipeline stages: Buffer writing (BW), Routing Calculation and Switch Allocation (RC/SA) and Crossbar Traversal stage (CT).

other switches in order to reduce power consumption. For example, as it is depicted in Fig.4.1, switch-000 have only four connected ports (north, east, up and local) and the remaining three ports (south, west and down) have been disabled since there are no connections to any neighboring routers along those directions.

Figure.4.3 represents 3D-ONoC switch architecture and that the routing process at each router can be defined by three main pipeline stages: Buffer writing (BW), Routing Calculation and Switch Allocation (RC/SA) and finally the Crossbar Traversal stage (CT).

Observing the Verilog HDL code for the *Router* module depicted in Code.4.3, 3D-ONoC contains seven *Input-port* modules for each direction represented in *input-port* module in line 4. This seven modules allocation are defined by the i-loop in line 2, where each value of i refers to the seven direction (Local, North, East, South, West,

Up, Down), and *NOUT* parameter in line 2 refers to the number of ports. The outputted *sw-req* signal defining the input port asking the grant and the output port requested defined by the *port-req* signal are sent from the seven input port to be an input port for the Switch allocator as it shown at line 19 and 20 of Code.4.3.

In addition to the *Switch-Allocator*, the *Crossbar* module is also defined (line 22-25). The crossbar circuit takes as input the *sw-cntrl* from the the switch allocator and *data-in* coming from the seven input ports.

Code 4.3: Verilog-HDL Code for Router

```
1  //instantiate input ports
2  for (i=0; i<NOUT; i=i+1) begin:il
3
4     input_port #(NOUT, FIFO_DEPTH, FIFO_LOG2D, FIFO_FULL_LVL) ip
5          (.clk(clk), .reset(reset),
6           .data_in(data_in[`WIDTH*(i+1)-1:`WIDTH*i]),
7           .data_out(cbar_data_in[`WIDTH*(i+1)-1:`WIDTH*i]),
8           .sw_req(sw_req[i]), .port_req(port_req[NOUT*(i+1)-1:NOUT*i]),
9           .sw_grant(sw_grant[i]), .stop_out(stop_out[i]),
10          .xaddr(xaddr), .yaddr(yaddr), .zaddr(zaddr));
11
12    assign data_sent[i] = |data_out[`WIDTH*i+`NEXT_PORT_END:`WIDTH*i+`NEXT_PORT_START
          ];
13    assign tail_sent[i] = data_out[`WIDTH*i];
14
15   end
16 endgenerate
17
18    sw_alloc #(NOUT) sw_allc(.clk(clk), .reset(reset),
19     .sw_req(sw_req), .stop_in(stop_in), .data_sent(data_sent), .tail_sent(tail_sent),
20     .port_req(port_req), .grant_out(sw_grant), .sw_cntrl(sw_cntrl));
21
22    crossbar #(NOUT, NOUT, `WIDTH) cbar(.clk(clk), .reset(reset),
23                .cntrl(sw_cntrl),
24                .data_in(cbar_data_in),
25                .data_out(data_out));
```

Now we analyze each component of the switch separately. Starting with the *Input-port*, the *Switch-Allocator* and finally *Crossbar* module.

4.3.1 Input Port

Starting with the *Input-port* module represented in Fig.4.4 (and where the Verilog code is represented in Code.4.4), each one of the seven modules is composed of two main elements: *Input buffer* and the *Route* module.

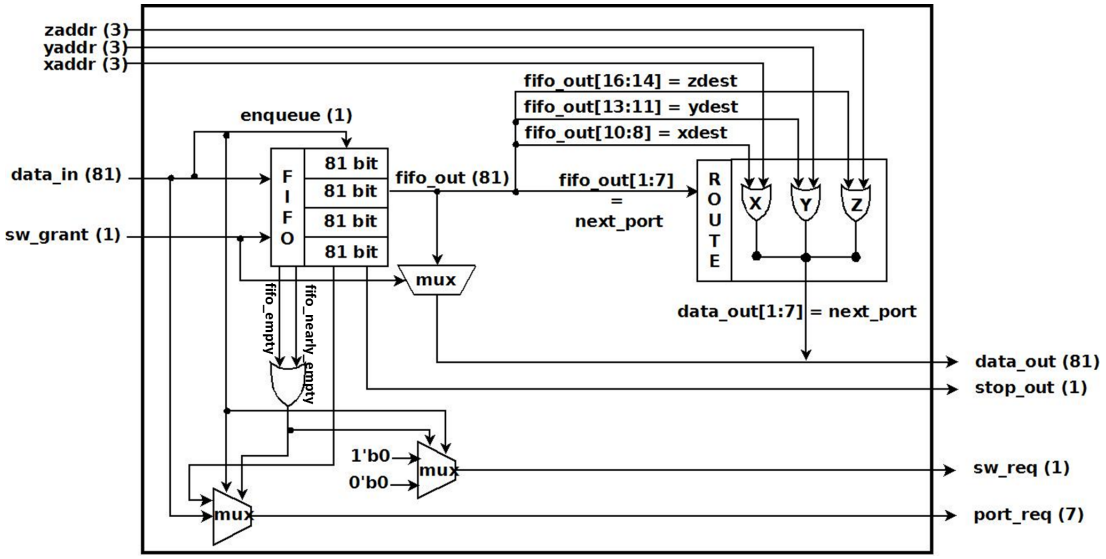

Figure 4.4: Input-port module architecture.

Code 4.4: Verilog-HDL Code for Input-port

```
1  //instantiate FIFO
2    fifo #(NOUT, FIFO_DEPTH, FIFO_LOG2D, FIFO_FULL_LVL) ff
3         (.data_in(data_in), .data_out(fifo_data_out),
4          .second_item_nextport(second_fifo_nextport),
5          .enqueue(enqueue), .dequeue(sw_grant),
6          .stop_out(stop_out), .nearly_empty(fifo_nearly_empty),
7          .empty(fifo_empty),
8          .clk(clk), .reset(reset));
9
10 //instantiate lookahead routing module
11   route #(NOUT) rr
12        (.xdest(fifo_data_out[`XDEST]), .ydest(fifo_data_out[`YDEST]),.zdest(
              fifo_data_out[`ZDEST]),
13        .xaddr(xaddr), .yaddr(yaddr), .zaddr(zaddr),
14        .nextport(fifo_data_out[`NEXT_PORT]), .new_nextport(lookahead_route));
```

Incoming 81 bits flits *data-in* from different neighboring switches, or from the connected computation tile, are first stored in the *Input buffer* and waiting to be processed. This step is considered as the first pipeline stage of the flit's life-cycle (BW). As it is illustrated in Code.4.5, arbitration between different flits is managed using FIFO queue technique. Each input buffer has by default four as depth, which means that it can host up to four 81 bits flits. Buffers occupy a significant portion of router area but can imply also increase in overall performance.

Code 4.5: Verilog-HDL Code for Input-FIFO-buffer

```
  always @(posedge clk) begin
    if (!reset) begin     //If out of reset
      if (enqueue) begin //Write a flit to the buffer
        fifo[tail_ptr] <= data_in;
        tail_ptr <= tail_ptr + 1;
      end
      if (dequeue) begin //Read a flit from the buffer
        head_ptr <= head_ptr + 1;
      end
//nearly full signal = stop_out,
      if (((tail_ptr + FULL_LVL[LOG2D-1:0] + 1'b1)==head_ptr) && enqueue && !dequeue)
          begin
        stop_out <= 1'b1;
      end
      if (((tail_ptr + FULL_LVL[LOG2D-1:0])==(head_ptr+1'b1)) && !enqueue && dequeue)
          begin
        stop_out <= 1'b1;
      end
      if ((tail_ptr + FULL_LVL[LOG2D-1:0])==head_ptr)begin
        if ((enqueue && !dequeue) || (!enqueue && dequeue))begin
          stop_out <= 1'b0;
        end
...
```

After being stored, the flit is fetched form the *FIFO* buffer and advanced to the next pipeline stage (RC/SA). The destination addresses (*xdest*, *ydest* and *zdest*) are then decoded in order to extract the information about the destination address in addition to the *Next-Port* pre-calculated in the previous upstream node. Those values are then sent to the *Route* circuit where La-XYZ routing scheme is executed to determine the *New-next-Port* direction for the next downstream node. At the same time the *Next-Port* identifier is also used to generate the request for the *Switch-Allocator* asking for grant to use the selected output port via *sw-req* and *port req* signals.

As we stated in Section.3, 3D-ONoC uses lookahead routing scheme *LA-XYZ* for fast routing. This scheme is based upon the dimension order (DOR) X-Y-Z static routing algorithm, where the X,Y and Z coordinates are satisfied in order. X-Y-Z routing is presented as the vertically balanced routing algorithm which has the best performance, since it's simple to implement, it is free of deadlock and live-lock, and also because packet ordering is not required. In addition to that each flit additionally

carries one hot encoded *Next-Port* identifier used by the downstream router. Since *LA-XYZ* is based upon *XYZ* routing, it is considered also as a minimal routing where each flit from any source and destination pair traverses the minimal number of hops.

To understand better how the *Next-Port* is decided, we designed the Verilog HDL code depicted in Code.4.6. As it is shown in this Code (line 1-12), the routing decision starts first by finding the next node's address. It is done by evaluating the actual *Next-Port* fetched from the flit, which gives a hint about which neighboring node the flit is going to be routed to and eventually knowing its exact address by incrementing *xaddr* or *yaddr* or *zaddr*. Depending on the resulted next address from the later step, the new *Next-Port* can be determined. As demonstrated between line 15 and 31 in Code.4.6, *LA-XYZ* compares the resulted next node's address (*next-xaddr*, *next-yaddr* and *next-zaddr*) and the destination addresses (*xdest*, *ydest* and *zdest*). At the end of the execution of this comparison, the new *Next-Port* (defined by *route* in Code.4.6) can be determined then embedded in the flit back again to be sent to the next node as Fig.4.4 illustrates.

Code 4.6: Verilog HDL implementation of LA-XYZ routing algorithm.

```
1   //assign next addresses
2     if (nextport == `EAST) next_xaddr = xaddr + 1'b1;
3     else if (nextport == `WEST) next_xaddr = xaddr - 1'b1;
4       else next_xaddr = xaddr;
5
6    if (nextport == `NORTH) next_yaddr = yaddr + 1'b1;
7     else if (nextport == `SOUTH) next_yaddr = yaddr - 1'b1;
8       else next_yaddr = yaddr;
9
10       if (nextport == `UP) next_zaddr = zaddr + 1'b1;
11    else if (nextport == `DOWN) next_zaddr = zaddr - 1'b1;
12       else next_zaddr = zaddr;
13
14  //evaluate next port
15  if (next_xaddr == xdest)
16  begin  if (next_yaddr == ydest)
17   begin   if (next_zaddr == zdest) route = `SELF;
18     else begin if(next_zaddr < zdest) route = `UP;
19              else route = `DOWN;
20          end
21     end
22     else  begin
```

```
23            if(next_yaddr < ydest) route = `NORTH;
24            else route = `SOUTH;
25          end
26      end
27      else  begin
28         if (next_xaddr < xdest) route = `EAST;
29           else route = `WEST;
30      end
31    end
```

If we take a look at Fig.4.1, and assume for example that a flit coming from switch-200 enters switch-201 (where the *xaddr*, *yaddr* and *zaddr* addresses are defined by 001, 000 and 001 respectively) trying to reach its destination node switch-313 (where the *xdest*, *ydest* and *zdest* addresses are defined by 011, 001 and 011 respectively). This flit caries "EAST" as a *nextport* identifier pre-calculated in the previous node (switch-200). According to the he first phase of the LA-XYZ algorithm, *next-xaddr= xaddr+1* which is the x-address of switch-202. In the second phase of the algorithm, *next-xaddr* is then compared with *xdest*. The comparison result will determine "EAST" as *route* (the new *Next-Port* for switch-202) which will be re-updated in the flit.

In order to enable the bypass technique, two signals are issued from the buffer to give information about the buffer occupancy status. These two signals are *fifo-empty* and *fifo-nearly-empty*. When the *fifo-empty* signal is issued, it means that the input buffer is empty and when an incoming flit arrives to the input port, it doesn't need to be stored in the buffer. Then overlapping the buffering stage and advancing to the next stage (RC and SA).

4.3.2 Switch Allocator

The *sw-req* and *port req* signals issued from each *Input-port* module, and giving information about the desired output-port, are transmitted to the *Switch-Allocator* module to perform the arbitration between the different requests. When more than two

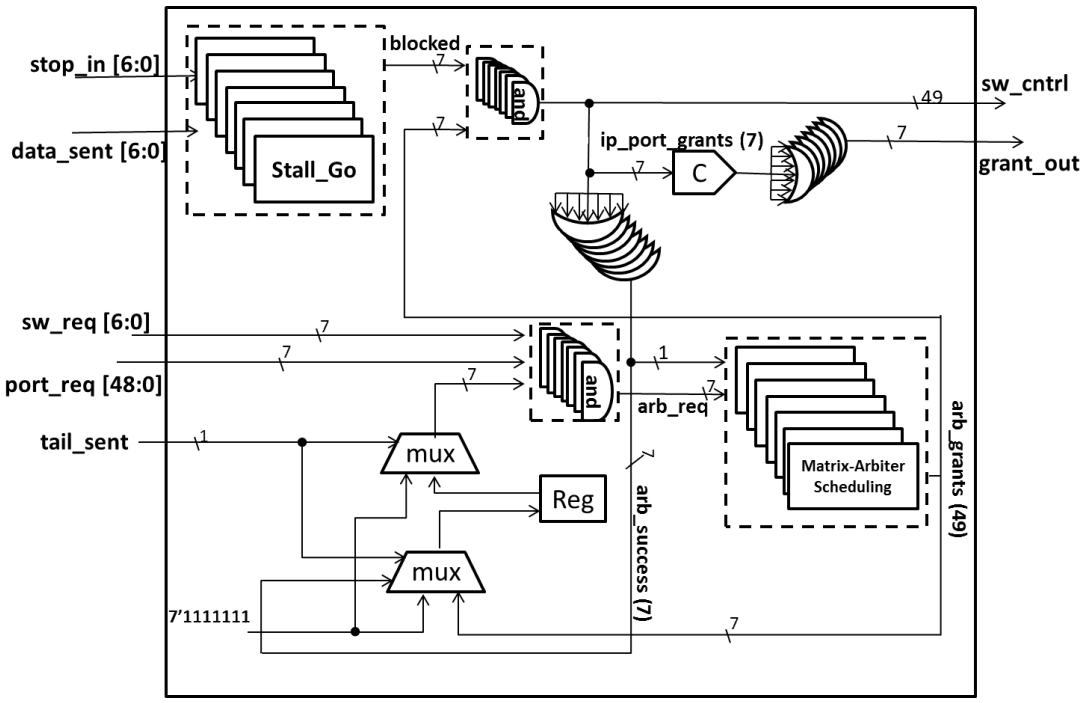

Figure 4.5: Switch allocator circuit.

input flits from different input-ports are requesting the same output-port at the same time, the *Switch-Allocator* manages to decide which output-port should be granted to which input-port, and when this grant should be allocated. This process is done in parallel with the routing computation done in *Input-port* to form the second pipeline stage.

As indicated in Fig.4.5, the switch allocator circuit has two output signals: one is *sw-cntrl* and the second one is *grant-out*. *sw-cntrl* contains all the information needed by the crossbar circuit about the scheduling result as it is explained later. On the other hand, the *grant-out* is sent back to the *Input-port* module and gives the grant to the appropriate input-port to send its data to the crossbar before reaching its next neighboring node. Figure4.5 shows that the switch allocator module is composed of two main components: *Stall-Go flow control* and *Matrix-Arbiter Scheduling*.

Stall-Go flow control module: Like the other flow control schemes, *Stall-Go* module manages the case of the buffer overflow. When the buffer exceeds its limitation on hosting flits (if the number of flits waiting for process are greater than the depth of the buffer), a flow control has to be considered to prevent from buffer overflow and eventually from packet dropping. Thus, allocating available resources to packets as they progress along their route. We chose *Stall-Go* flow control since it proves to be a low-overhead efficient design choice showing remarkable performance comparing to the other flow control schemes such us *ACK-NACK* or *Credit based* flow control. Like the other flow control schemes, *Stall-Go* module manages the case of the buffer

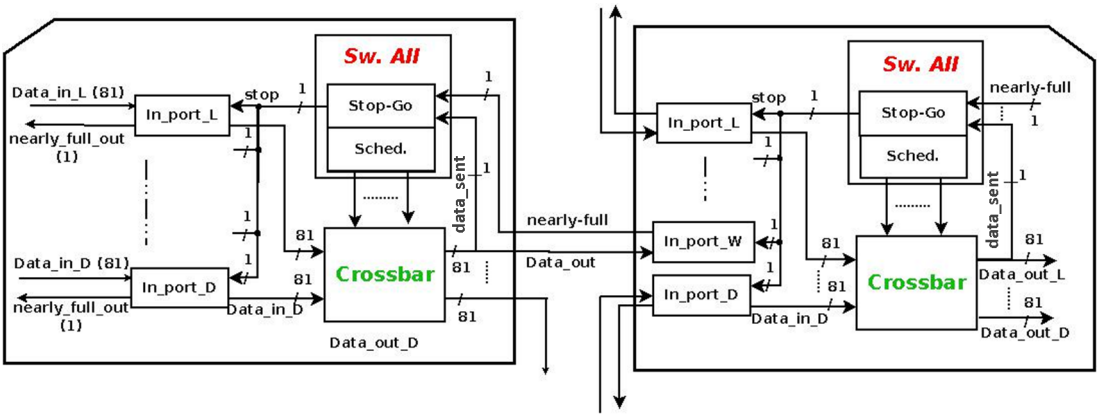

Figure 4.6: Stall-Go flow control mechanism.

overflow. When the buffer exceeds its limitation on hosting flits (if the number of flits waiting for process are greater than the depth of the buffer), a flow control has to be considered to prevent from buffer overflow and eventually from packet dropping. Thus, allocating available resources to packets as they progress along their route. We chose *Stall-Go* flow control since it proves to be a low-overhead efficient design choice showing remarkable performance comparing to the other flow control schemes such us *ACK-NACK* or *Credit based* flow control [36].

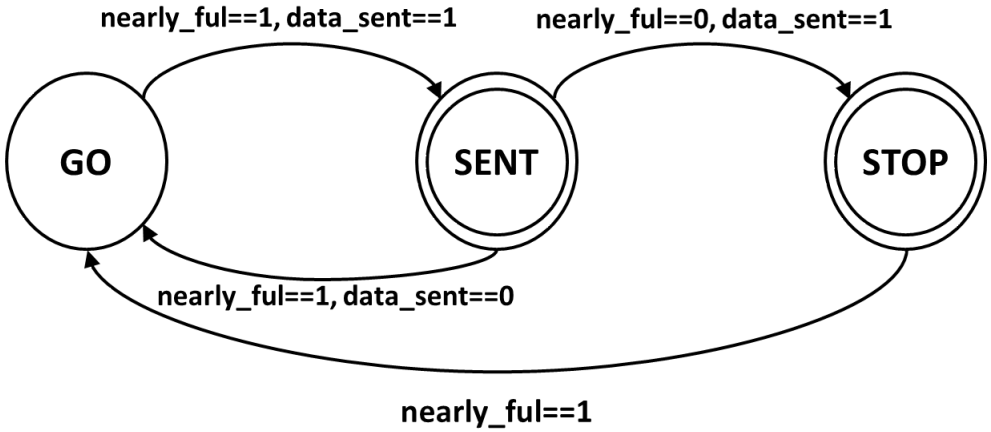

Figure 4.7: Stall-Go flow control State machine.

Code 4.7: Verilog HDL of the state machine decision

```
1    always @(posedge clk) begin
2
3   if (!reset) begin
4     if ((state==‘GO) && stop_in && data_sent)
5       state <= ‘SENT1;
6     if (state==‘SENT1) begin
7       if (stop_in && !data_sent)
8         state <= ‘GO;
9       if (!stop_in && data_sent)
10        state <= ‘STOP;
11    end
12
13  if ((state==‘STOP) && stop_in) // stop_in = nearly_full
14    state <= ‘GO;
15    end else
16    state <= ‘GO;
17  end
18
19   assign blocked = ( ((state==‘STOP) && !stop_in) || ((state==‘SENT1) && !stop_in &&
         data_sent) );
```

Stall-Go module, where the mechanism is represented in Fig.4.6, uses two control signals: *nearly-full* and *data-sent*. *nearly-full* signal is sent to the upstream node indicating that the input-buffer is almost full and only one slot is still available to host one last flit. After receiving this signal, the *FIFO* buffers suspend sending flits. The *data-sent* signal is issued when the flit is transmitted. Figure.4.7 represents the *Stall-Go* flow control state machine which aims to generate the *nearly-full* and *data-sent* signals. State *GO* indicates that the buffer is still able to host two or more flits. State

SENT indicates that the buffer can host only one more flit, and finally when we move to state *STOP*, it means that the buffer can not store anymore flits. The state machine is generated as indicated in Code.4.7 that shows the Verilog-HDL code explaining the main state transitions using *nearly-full* and *data-sent* signals.

Matrix-Arbiter scheduling module: The second component is the scheduling module. As shown in Fig.6, the input signals *sw-req* and *port-req* indicate the input-ports demanding the access, and which output-ports are they requesting respectively. Depending on these requests, the arbiter allocates the convenient output-port to its demander. Since 3D-ONoC transmits only one flit in every clock cycle, then when two input-ports or more are competing for the same output-port, the presence of a scheduling scheme is required in order to prevent from any possible conflict. The switch allocator in our design employs a least recently served priority scheme via the packet transmit layer. Thus, it can treat each communication as a partially fixed transmission latency [37], [38]. Matrix arbiter is used for a least recently served priority scheme.

In order to adopt Matrix arbiter scheduling for 3D-ONoC, we implemented a 6x6 scheduling-matrix. The scheduling module accepts all the requests from the different connected input-ports and their requested output-ports. Then it assigns priority for each request. In order to give the grant to the convenient input-port, the scheduling module verifies the scheduling-matrix, compares the priorities of the input-ports competing for the same output-port, and gives the grant to the one possessing the highest priority in the matrix. Following this basis, the scheduling module should make the input-port, which got the last grant to use the competed output-port, the lowest priority for the next round of arbitration, and then increases the priority of the rest of the remaining ports.

When there are no requests, the priority is unchanged. Based on these assumptions, we are sure that every input-port will be served and get the grant to use the output-port in a fair way.

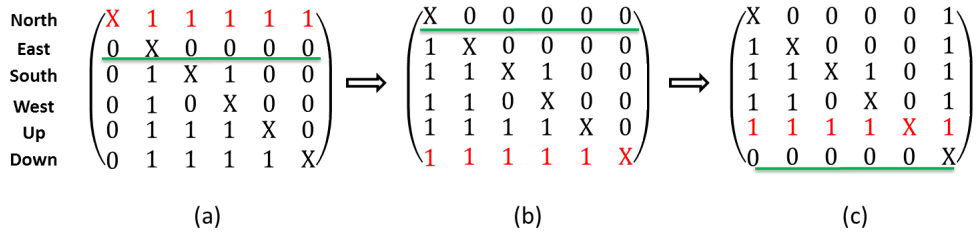

Figure 4.8: Scheduling-Matrix priority assignment.

Code 4.8: Matrix Arbiter code

```
1   generate
2   for (i=0; i<SIZE; i=i+1) begin:ol1
3     for (j=0; j<SIZE; j=j+1) begin:il1
4       if (j==i)
5         assign pri[i][j]=request[i];
6       else
7       if (j>i)
8         assign pri[i][j]=!(request[j]&&state[j*SIZE+i]);
9       else
10      assign pri[i][j]=!(request[j]&&!state[i*SIZE+j]);
11      end
12      assign grant[i]=&pri[i];
13  end
14  endgenerate
15
16  generate
17  for (i=0; i<SIZE; i=i+1) begin:ol2
18    for (j=0; j<SIZE; j=j+1) begin:il2
19      assign new_state[j*SIZE+i]=(success&&((state[j*SIZE+i]&&!grant[j])||(grant[i
            ]))||(!success&&state[j*SIZE+i]);
20      end
21  end
22  endgenerate
23
24  always@(posedge clk) begin
25    if (reset) state<=-1;
26    else begin
27    if (|request) state<=new_state;
28    end
29  end
```

Figure.4.8 illustrates a simple example of how our scheduling mechanism works. Each row of the matrix represents the competing input requests and their priorities. The scheduling-module starts by examining the priorities of each input-port request.

After the highest priority input is served, the arbiter updates the scheduling-matrix by making the request which got the last grant, the lowest priority for the next round of arbitration, by inversing its row and column.

The matrix shown in Fig.4.8 (a) illustrates the initial scheduling-matrix where *North*, *Up* and *Down* input-ports are asking the grant to eject their flits to the *Local* port. Observing this figure, the *North* request (highlighted in red) has higher priorities compared with the remaining two requests. As a result the Arbiter gives the grant to the *North* request. Then *North* becomes the lowest priority (as it is underlined by a green line) and the remaining two requests priorities are incremented. In the next round (Figure.4.8 (b)), *Down* seems to have a higher priority than the *Up* request. The arbiter then gives the grant to *Down* and make its priority the lowest. Finally, as it is shown in Fig.4.8 (c), the *Up* request having the highest priority among the others, is giving the grant to eject its data to the requested output port. Code.4.8 depicts the Verilog HDL code for the implementation for the Matrix arbiter.

4.3.3 Crossbar

The switch allocator, sends the issued *control* signal to the crossbar circuit to complete the third and final Crossbar Traversal pipeline stage (CT), where information about the selected input port and the *Next-Port* are embedded, and then stored in the *sw-cntrl-reg* register as it is shown in Fig.4.9. After that, the crossbar fetches these information, receives the data from the FIFO buffer of the selected input-port. Then, it allocates the appropriate channel for transmission to the decoded *Next-Port*. Finally, the crossbar sends the flit to its destination as illustrated in Fig.4.9.

When all the flits are transmitted, the *tail* bit informs the switch allocator via a

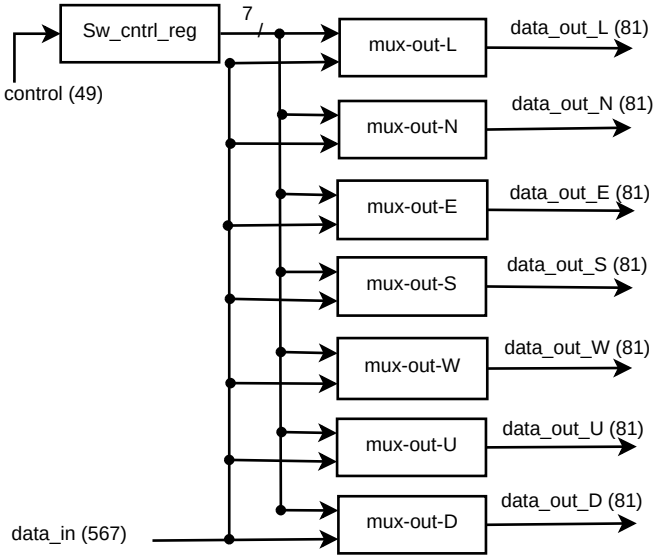

Figure 4.9: Crossbar circuit.

Code 4.9: Code for Crossbar circuit

```
1  //crossbar.v
2  generate
3    for (i=0;i<NOUT;i=i+1) begin:output_loop
4    mux_out #(NIN, WIDTH) cbar_mux(.cntrl(cntrl_reg[NIN*(i+1)-1:NIN*i]), .data_in(
         data_in), .data_out(data_out[WIDTH*(i+1)-1:WIDTH*i]));
5    end
6  endgenerate
7
8  //mux_out
9  generate
10   //loop over each bit of data
11   for (i=0;i<WIDTH;i=i+1) begin:bit_loop
12     assign data_out[i] = mux(cntrl, data_bits[i]);
13     //loop over each input channel
14     for (j=0;j<n_in;j=j+1) begin:input_loop
15       assign data_bits[i][j] = data_in[WIDTH*j+i];
16     end
17   end
18 endgenerate
19
20 function mux;
21   input [n_in-1:0] cntrl;
22   input [n_in-1:0] data_in;
23   integer i;
24
25 begin
26   mux = 0;
27   for (i=0; i<n_in; i=i+1) begin
28     if(cntrl[i] == 1'b1) mux = data_in[i];
29   end
30 end
31 endfunction // mux
```

tail-sent signal that the packet transmission is completed and can free the used channel

so it can be exploited by another packet. Code.7.7 depicts the Verilog HDL code for the implementation for the Crossbar circuit.

4.4 Network interface

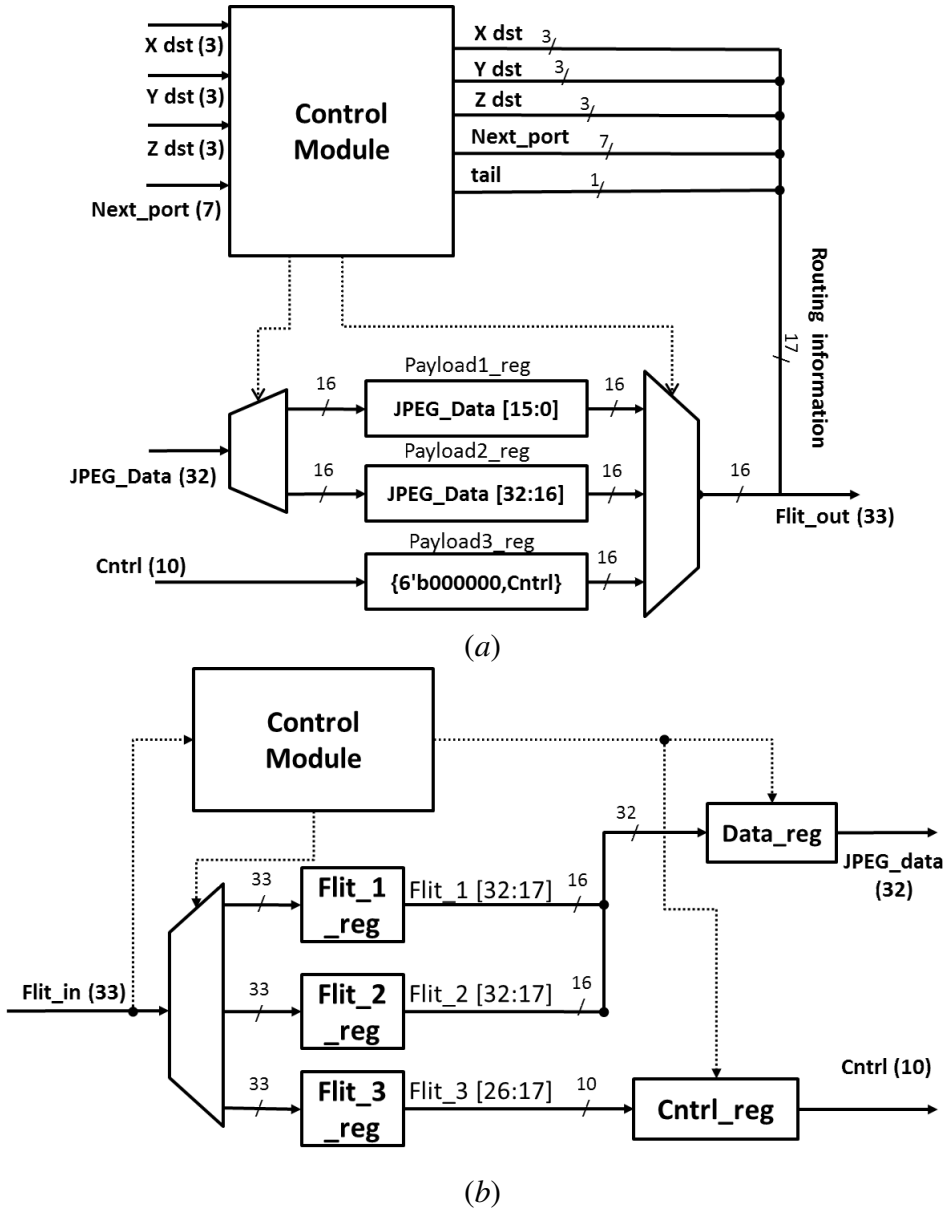

Figure 4.10: Network Interface Architecture: (a) Transmitter (b) Receiver

In order to enable real applications to be run on 3D-ONoC, we added a Network

Interface (NI) to every router as a medium interface between the different PEs (Processor, memory, I/O etc...) that can be connected, and our network. In this research, we tested 3D-ONoC using JPEG encoder application [39]. For that reason, we designed both *Transmitter* and *Receiver* NI in every switch of our network. We set the packet size to 99 bits which includes three 33 bits flits. Each flit contains 17 bits defining the routing information (*xdst*, *ydst*, *zdst*, *Next-Port* and *tail*) and the remaining 16 bits are dedicated for the payload.

Figure.4.10(a) shows the architecture of the *Transmitter-NI*. It receives a 32 bits data from the JPEG module that will be divided into two portions representing the payload of the two first flits of the packet. The payload of the third flit contains the 10 bits control signal from the JPEG module, and the remaining six bits are unused. As shown in Fig.4.10 (a) , a *Control Module* manages the fits generation. It adds the convenient destination addresses and *Next-Port* direction to each flit, and marks the end of the packet by adding the (*tail* bit to the third final flit. The generated flits are then injected into the network. The Verilog HDL implementation of the *Transmitter-NI* is depicted in Code.4.10.

Code 4.10: Verilog-HDL sample code for the sending NI

```
module NI_02_send (clk, rst, enable, data_in, flit);
   input        clk, rst;
   input      enable;
   input [23:0]   data_in;

   output reg [32:0]    flit;

    always @(state)begin
      case(state)

  `f0:begin
     if(cntrl) begin
      next_state <= `f1;
    flit <= 33'hz;
     end
     else next_state <= `f1;
  end

  `f1:begin
```

```
20      next_state <= `f3;
21      flit[0]     <= `header;
22      flit[7:1]   <= `EAST;
23      flit[16:8]  <= `dest_03;
24      flit[32:17] <= data_in[23:8];
25    end
26
27    `f2:begin
28      if(cntrl) begin
29      next_state <= `f3;
30      flit[0]     <= `header;
31      flit[7:1]   <= `EAST;
32      flit[16:8]  <= `dest_03;
33      flit[32:17] <= data_in[23:8];
34      end
35      else next_state <= `f2;
36    end
37
38    `f3:begin
39      next_state <= `f4;
40      flit[0]     <= `header;
41      flit[7:1]   <= `EAST;
42      flit[11:8]  <= `dest_03;
43      flit[24:17] <= data_in[7:0];
44      flit[32:25] <= 0;
45
46    end
47    `f4:begin
48      next_state <= `f2;
49      flit[0]     <= `tail;
50      flit[7:1]   <= `EAST;
51      flit[16:8]  <= `dest_03;
52      flit[17]    <= enable;
53      flit[32:18] <=  0;
54
55    end
56    default:next_state <= `f0;
57        endcase
58     end
```

On the other side, the *Receiver-NI* receives the incoming three flits of each packet ejected from the network, and then stores them into three temporary registers. After that, as it is shown in Fig.4.10 (b), the 16 bits payload of the first and second flit are fetched form the temporary registers, reassembled together and finally stored in the *Data-reg* register.

Controlled by another *Control Module*, the complete 32 bits resulted Data and the 10 bits control signals, are fetched the sent to their attached JPEG module after the complete packet is received. The Verilog HDL implementation of the *Transmitter-NI* is depicted in Code.4.11

Code 4.11: Verilog-HDL sample code for the receiving NI

```
module NI_03_rec (clk, rst, flit, data_out, enable);

//input output
   input        clk, rst;
   input [32:0]       flit;
   output reg [23:0]      data_out;
   output reg         enable;
   //reg
   reg [15:0]     data_high;
   reg [7:0]      data_low;
   reg          ena;

   reg [1:0]      state;
   reg [1:0]      next_state;
   reg [32:0]     preflit;

  reg [23:0]      pre_data_out;
   //state
   always @(posedge clk)begin
      if(rst==1)state <= `f0;
      else begin
   preflit <= flit;
   state <= next_state;
      end
   end

   //state
   always @(state or flit)begin
      case(state)

  `f0:begin
     if(flit!=preflit)begin
        next_state <= `f1;
        data_high <= 0;
        data_low <= 0;
     end
     else next_state <= `f1;
  end
  `f1:begin
     if(flit!=preflit)begin
        next_state <= `f2;
        data_high <= flit[32:17];
     end
     else next_state <= `f1;
  end
  `f2:begin
     if(flit!=preflit)begin
        next_state <= `f3;
        data_low <= flit[24:17];
     end
     else next_state <= `f2;
  end
  `f3:begin
     if(flit!=preflit)begin
        next_state <= `f1;
        ena <= flit[17];
      pre_data_out <= {data_high, data_low};
     end
     else next_state <= `f3;
  end
  default:next_state <= `f0;
      endcase
   end
```

Based on this network interface, another one has been designed to satisfy the requirements of another application that we used for evaluating 3D-ONoC, which is Matrix-Multiplication. We chose the matrix multiplication as one of our evaluating target, since it is wildly used in scientific application. Due to its large multi-dimensional data array, it is extremely demanding in computation power and meanwhile it is potential to achieve its best performance in a parallel architecture and doesnt involve synchronization [40]. All of these reasons make the Matrix-Multiplication a very suitable application to evaluate 3D-ONoC and show its outperforming performance against 2D-ONoC.

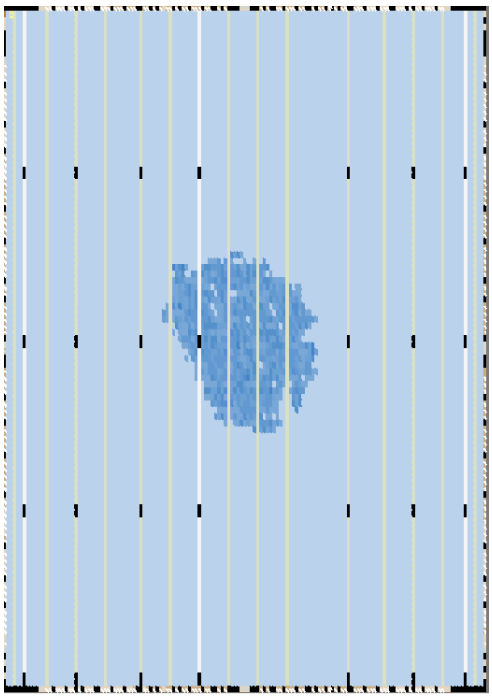

Figure 4.11: Chip floor plan for a 2x2x2 3D-ONoC.

By the end of this chapter, we presented the main components of our Mesh based 3D-ONoC system. We explained how the packets are forwarded among the network using *Wormhole-like* switching and Virtual-Cut-Through switching policies. We also

give more details about the router components including the hardware implementation of our proposed *Look-Ahead-XYZ* routing algorithm (LA-XYZ). For the flow control, we demonstrated that 3D-ONoC adopts *Stall-Go* mechanism in the Switch Allocator and how this flow control efficiently avoids dropping packets. Examples about the *Matrix-Arbiter* scheduling technique are also provided to show its ability to serve all the request in a fair way. Figure.4.11 shows the chip floor plan for a 2x2x2 3D-ONoC for the Altera Stratix III EP3SL150F1152C2 chip, and Figure.4.12 shows the RTL view of the same 2x2x2 3D-ONoC system. Both of these figures are generated using the QUARTUS II tool after succeeding the correct compilation of the system.

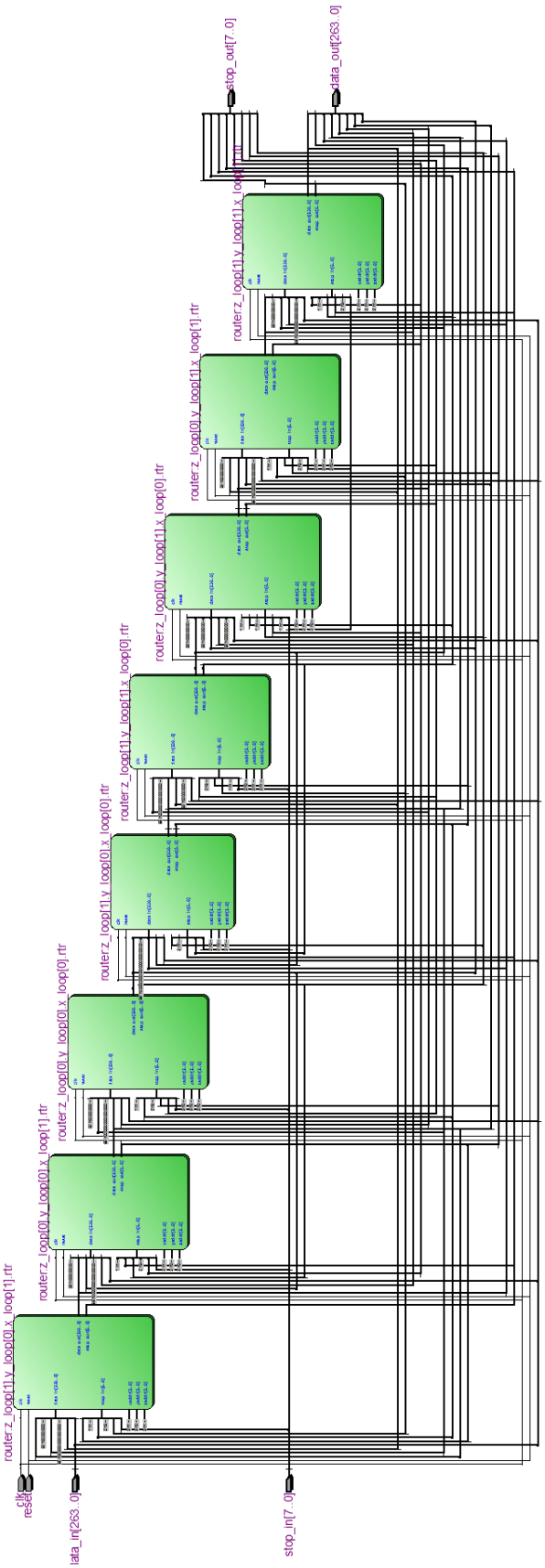

Figure 4.12: RTL view of 2x2x2 3D-ONoC.

Chapter 5

Evaluation

Using the JPEG encoder and the Matrix-multiplication applications, in this chapter we evaluate the hardware complexity of 3D-ONoC in term of area utilization, power consumption (static and dynamic) and clock frequency. The performance evaluation is also done by analyzing the execution time, the number of hops and also the number of stall after the execution of the both of the application. All the results obtained are analyzed and compared with 2D-ONoC.

5.1 Evaluation methodology

5.1.1 JPEG encoder

Starting with the JPEG encoder application, which is a well-known application that is widely used application by many researchers. Including some parallel processing, JPEG might be a good application to evaluate the performance of NoC.

For instance, we took into consideration the task implementation shown in Fig.5.1.For additional analysis, we made further divisions to the *Y:d-q-h*, *Cb:d-q-h*, *Cr:d-q-h* and *FIFO* modules, and the resulted task graph is illustrated in Fig.5.2. This extension aims to increase the network size and deploy more parallel execution of the different modules of the application, and then can take advantage of the scalability and the reduced

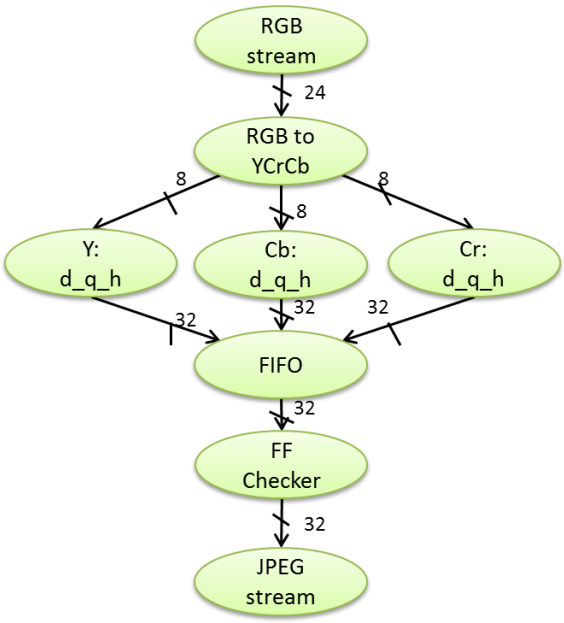

Figure 5.1: Task graph of the JPEG encoder

number of hops offered by our design.

As we analyze the modified task graph represented in Fig.5.2, we noticed that the communication bandwidth between *DCT*, *Quantization* and *Huffman* modules are very high (640 bits) compared with those found between the different other modules of the application (8, 24 and 32 bits). This bandwidth gap will cause unbalanced traffic distribution especially when implemented on hardware, since we will increase the link size in addition to the size and number of flits in the packet format, causing higher latency and thermal power problem. All these factors, will eventually decrease the overall performance of our system, instead of enhancing it.

For all the reasons previously stated, we will implement the first task graph represented in Fig.5.1 and we randomly mapped the tasks into 2D-ONoC (2x4) and 3D-ONoC (2x2x2) as shown in Fig.5.3 (a) and Fig.5.3 (b) respectively.

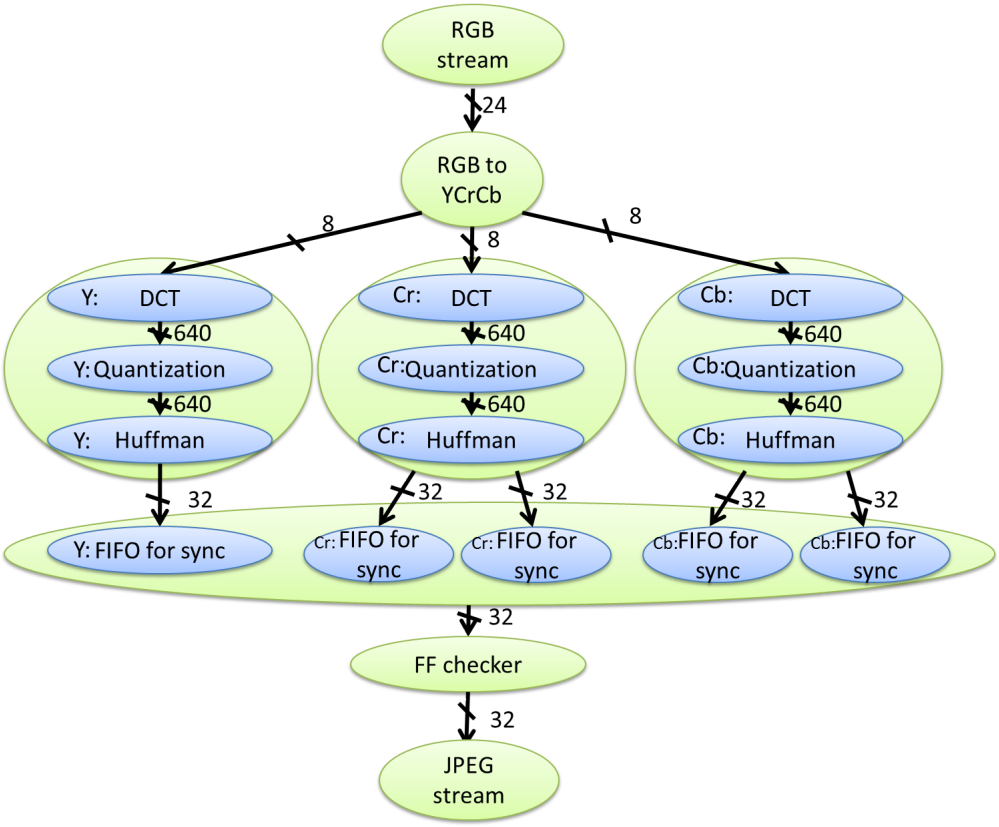

Figure 5.2: Extended task graph of the JPEG encoder

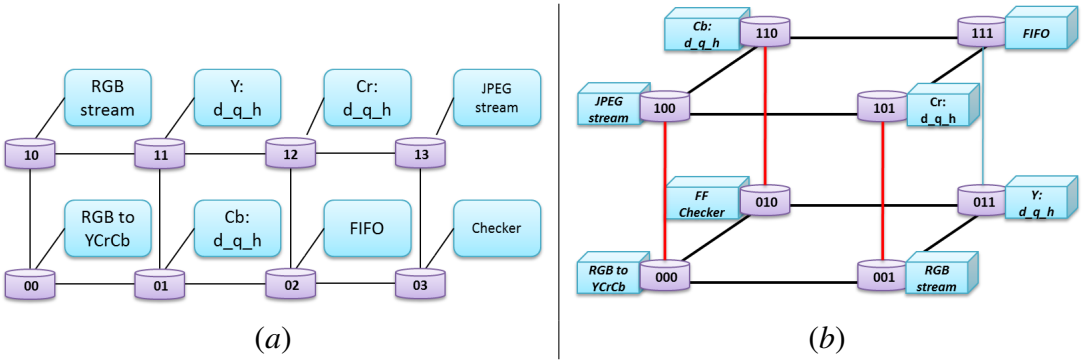

Figure 5.3: JPEG encoder mapped onto: (a) 2x4 2D-ONoC (b) 2x2x2 3D-ONoC

5.1.2 Matrix multiplication

$$\begin{pmatrix} A11 & \cdots & A1k \\ \vdots & \ddots & \vdots \\ Ai1 & \cdots & Aik \end{pmatrix} \mathrm{X} \begin{pmatrix} B11 & \cdots & B1j \\ \vdots & \ddots & \vdots \\ Bk1 & \cdots & Bkj \end{pmatrix} = \begin{pmatrix} R11 & \cdots & R1j \\ \vdots & \ddots & \vdots \\ Ri1 & \cdots & Rij \end{pmatrix}$$

Figure 5.4: Matrix multiplication example: The multiplication of an *i*x*k* matrix *A* by a *k*x*j* matrix *B* results in an *i*x*j* matrix *R*.

First we assume that an *i*x*k* matrix *A* has *i* rows and *k* columns, where A_{ik} is an element of *A* at the *i*-th row and *k*-th column. As it demonstrated in Fig.5.4, an *i*x*k* matrix *A* can be multiplied by a *k*x*j* matrix *B* to obtain an *i*x*j* matrix *R*. Figure.5.5 presents how the matrix *R* can be obtained according to Formula 4.1.

$$R_{i,j} = \sum_{n=0}^{k-1} A_{i,n}.B_{n,k} \tag{5.1}$$

When implemented onto 3D-ONoC, and for seek of convenience or without loss in generality, we can assume that all the matrices are square and having *n*x*n* size. In 3D-ONoC, each element of the three matrices is assigned to a computation module which is connected to one router. As a result the number of routers connected to the network is the sum of all the elements of three matrices which is equal to $3n^2$. Each

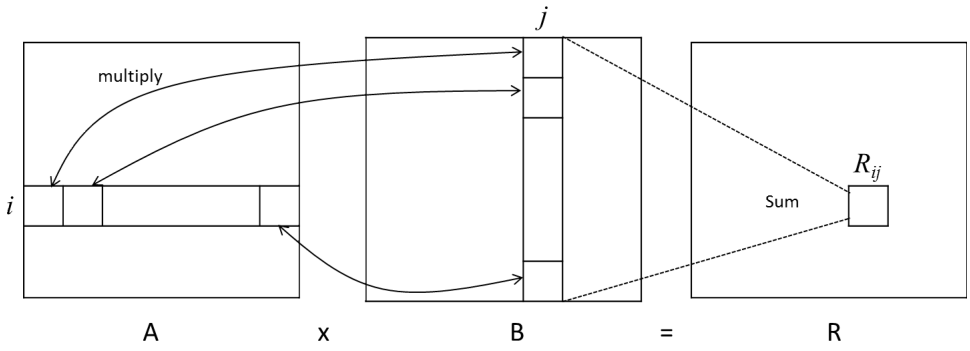

Figure 5.5: Simple example demonstrating the Matrix multiplication calculation.

element of the matrix *B* receives *n* flits from *n* different elements of the matrix *A* in order to make the multiplication. Then, each element of the matrix *B* sends *n* flits to *n* different elements of the matrix *R* where all the received values are summed then the final resulted value is outputted. In total $2n^3$ flits travel the network for a *nxn* square matrix multiplication.

As we previously stated at the beginning of this chapter, we want to evaluate the number of hops traversed by all the flits generated by the Matrix application. For this matter we define:

$$3D\_Hops_i = |x\_dest_i - x\_src_i| + |y\_dest_i - y\_src_i| + |z\_dest_i - z\_src_i| \tag{5.2}$$

Where $3D\_Hops_i$ is the number of hops consumed for one single flit $i \in \{0,1,2,....,2n^3-1\}$ (the set of all flits), traveling from one source node (where the address is defined by *x_dest*, *y_dest* and *z_dest*) to its destination node (*x_src*, *y_src* and *z_src*). As a result, we can say that the number of hops consumed by an *nxn* square matrix multiplication can be defined by:

$$3D\_Total\_Hops = \sum_{k=0}^{2n^3-1} 3D\_Hops_k \tag{5.3}$$

According to Formula 4.2 and 4.3, the number of hops for 2D-ONoC can be then extracted and defined as follow:

$$2D\_Hops_i = |x\_dest_i - x\_src_i| + |y\_dest_i - y\_src_i| \tag{5.4}$$

$$2D\_Total\_Hops = \sum_{k=0}^{2n^3-1} 2D\_Hops_k \tag{5.5}$$

For the evaluation, we took the case of 3x3, 4x4 and finally a 6x6 matrix multiplication. For each one of these three cases, two mapping approaches has been taken into consideration. For instance, we take the example of 3x3 matrix multiplication. We randomly mapped the elements of the three matrices into 2D-ONoC (3x9) and 3D-ONoC (3x3x3) using an optimistic mapping approach as presented in Fig.5.6 (a). In this mapping we tried to make the communication distance as close as possible, in order to reduce the number of hops which eventually will lead to decrease the latency. Figure.5.6 (b), on the other hand, illustrates a pessimistic task mapping approach. The second approach tries to increase the communication path of the different flits traversing the network.

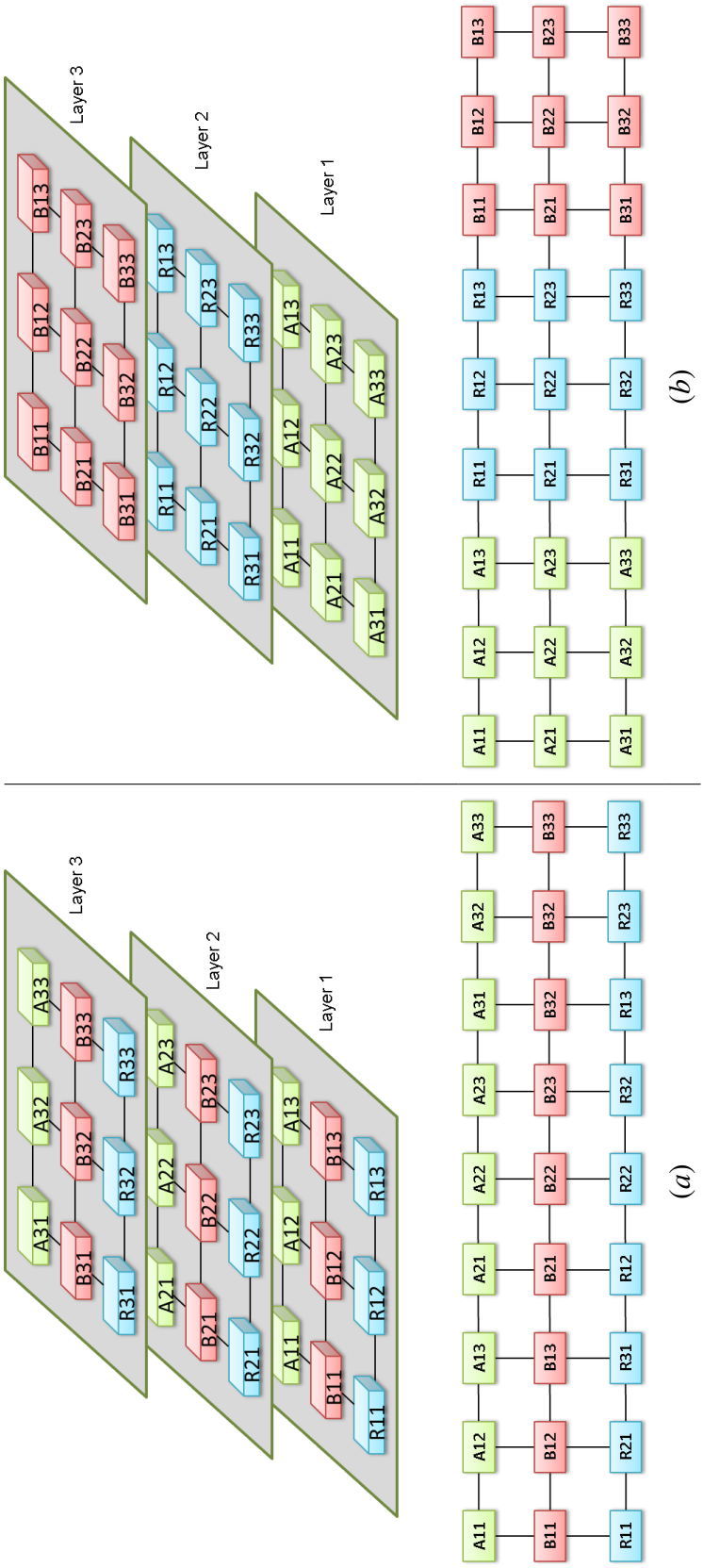

Figure 5.6: 3x3 matrix multiplication using (a) optimistic and (b) pessimistic mapping approaches

In order to obtain an easier and more accurate evaluation both of 3D-ONC is implemented in Verilog HDL. We evaluated and compared the hardware complexity in terms of area, power consumption (static and dynamic) and clock frequency and also the performance in term execution time, the number of hops, and also we counted the number of *stop-signal* generated from our *Stall-Go* flow control mechanism. All the evaluation results obtained for 3D-ONoC are than compared to 2D-ONoC system.

We chose the Stratix III FPGA as a target device and then the synthesis was done by the Quartus II software, which both are provided by Altera inc.. We used *PowerPlay Power Analyzer* tool in QuartusII in order to evaluate the power consumption generated. This design approach results in more accurate speed, area and power consumption evaluation. The use of FPGA is a very convenient choice for our design, thanks to its simplicity and the ability of reconfigurability. In addition to that, it provides faster simulation than the traditional software emulation while maintaining a cheaper cost than implementing with real processors. Table.5.1 presents the parameters used for the synthesis of 3D-ONoC design

5.2 Evaluation results

5.2.1 Hardware complexity evaluation

As we previously stated, the goal of this section is to provide a hardware evaluation for our 3D-ONoC including area, power consumption, and clock frequency when simulated with both JPEG encoder and Matrix multiplication applications.

Table.5.2 illustrates the hardware evaluation results obtained. The results show that the logic utilization of 3D-ONoC is increased by an average of 37% compared to the 2D design. The increased number of ALUTs can be explained by the fact that the

Table 5.1: Simulation parameters.

Parameters		2D-ONoC	3D-ONoC
Network Size (Mesh)	JPEG	2x4	2x2x2
	Matrix (3x3)	3x9	3x3x3
	Matrix (4x4)	6x8	4x4x3
	Matrix (6x6)	9x12	6x6x3
Packet size	JPEG	3 flits	3 flits
	Matrix	1 flit	1 flit
Flit size	JPEG	30 bits	33 bits
	Matrix	35 bits	30 bits
Header size	JPEG	12 bits	17 bits
	Matrix	14 bits	17 bits
Payload size	JPEG	16 bits	16 bits
	Matrix	21 bits	21 bits
Buffer Depth		4	4
Switching		Wormhole-like	Wormhole-like
Flow control		Stall-Go	Stall-Go
Scheduling		Matrix-Arbiter	Matrix-Arbiter
Routing		LA-XY	LA-XYZ
Target Device		Altera Stratix III	Altera Stratix III

3D-ONoC router has two additional ports and a larger crossbar than 2D-ONoC. The additional number of ports incurs additional buffers, which is costly in term of area.

In term of clock speed 3D ONoC under-performs the 2D-ONoC architecture by 16% on average due to the increased hardware complexity. While the power static consumption is increased with 3D-ONoC with almost 14% for the same additional hardware reasons, the dynamic power on the other hands is decreased in average of 16% while executing JPEG and the two mapping approaches foe each of the three matrix multiplications. As a conclusion, the total power consumption is decreased with nearly 1.4%.

Many factors affect the dynamic power in FPGA, such us capacitance charging, supply voltage and clock frequency. Since the first two factors are the same for both

Table 5.2: 3D-ONoC hardware complexity compared with 2D-ONoC.

Application	Area (ALUTs)		Power(mW)						Speed(MHz)	
	2D	3D	2D			3D			2D	3D
			Static	Dynamic	Total	Static	Dynamic	Total		
JPEG	28.401	30.382	811.63	4.27	815.9	769.13	4.01	773.14	193.8	160.72
Matrix 3x3	18.012	30.954	969.84	332	1301.84	1032.14	260	1292.14	158.73	130.01
Matrix 4x4	36.393	61.157	1073.52	495.2	1568.72	1055.65	410	1452.65	146.56	101.41
Matrix 6x6	89.576	144.987	1113.29	580	1693.29	1051.06	450.2	1501.26	98.85	98.1

3D and 2D ONoC designs, and only the clock frequency is different between them, we can say that the reduction of the clock frequency had an impact on the reduction of the dynamic power. Besides that the clock frequency reduction, we believe that the reduction of number of hops (that will be explained in the next section) also plays an important role in the reduction of dynamic power. In fact, when the number of hops is reduced it means that the flit has less hops, shorter path which eventually means less buffering, routing and scheduling. All these factors lead to reduce the dynamic power when using 3D-ONoC when compared with 2D system.

5.2.2 Performance analysis evaluation

For the performance evaluation, we run each of the four applications. Then we evaluated the execution time, the number of hops and the number of *stop-signal* of each one of them after verifying the correctness of the resulted data.

Starting with the execution time, we run each of the four applications on 3D-ONoC and 2D-ONoC. Figure.5.7 demonstrates the execution time results. Taking a closer look at the JPEG application results, we may see that there is a slight improvement of 1.4% with 3D-ONoC when compared with the 2D architecture. This slight improvement can be explained by many reasons.

First, JPEG is a small application which we could map into only eight nodes. That

is a quiet small number to exploit the benefits of a 3D-NoC. Seconds, when observing the task graph of JPEG (previously shown in Fig.5.1), JPEG has indeed some tasks working in parallel(*Y:d-q-h*, *Cb:d-q-h* and *Cr:d-q-h*), but at the same time we can see that *FIFO* module is dependent of those three tasks. Another reason is, the JPEG computation modules involve heavy computation. This leads to decrease the clock frequency of the entire system in a very inconvenient way for 3D-ONoC. The performance of 3D-ONoC is then hided and can't be taken advantage of. All of those reasons have an important impact on the performance of the 3D-ONoC. JPEG might be a very appropriate application to show the out performance of NoC over the traditional interconnect systems (such us bus-based system or P2P), but when we talk about 3D-ONoC that is targeted for hundreds of cores which is dedicated to a large number of cores with higher parallelism tasks.

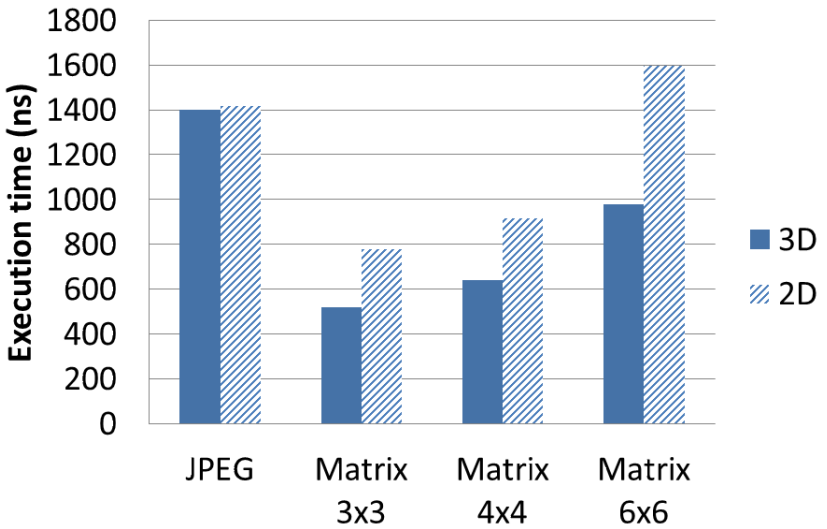

Figure 5.7: Execution time comparison between 3D and 2D ONoC.

On the other part, when evaluated with the Matrix multiplication application, 3D-ONoC shows a greater performance and decreases the execution time for about 35%,

33% and 41% for each of 3x3, 4x4 and 6x6 matrix respectively. In total 3D-ONoC reduces the execution time for one single Matrix multiplication to up to 36% when compared with 2D-ONoC. As we stated previously, due to the fact that the Matrix multiplication has a larger data array, higher number of parallel tasks with less dependency between them, Matrix multiplication shows greater performance than JPEG. While the JPEG is mapped onto 8 nodes only, the matrix multiplication can reach the 108 nodes for the 6x6 matrix size. These factors are very suitable to show the performance enhancement when adopting 3D-ONoC. This enhancement can be related to the reduction of number of hops that offers 3D-ONoC.

Code 5.1: Verilog-HDL code for hops number count

```
1     for (i=1;i<=3;i=i+1)begin
2       for (j=1;j<=3;j=j+1)begin
3         for (k=1;k<=3;k=k+1)begin
4           #200000
5   // ***************Hop count from A to B***************
6           if ((A_adress [i][j][2:0])>(B_adress [j][k][2:0]))
7           Total_hops= Total_hops+ ((A_adress [i][j][2:0])-(B_adress [j][k][2:0]));
8           else
9           Total_hops= Total_hops+ ((B_adress [j][k][2:0])-(A_adress [i][j][2:0]));
10
11          if ((A_adress [i][j][5:3])>(B_adress [j][k][5:3]))
12          Total_hops= Total_hops+ ((A_adress [i][j][5:3])-(B_adress [j][k][5:3]));
13          else
14          Total_hops= Total_hops+ ((B_adress [j][k][5:3])-(A_adress [i][j][5:3]));
15
16          if ((A_adress [i][j][8:6])>(B_adress [j][k][8:6]))
17          Total_hops= Total_hops+ ((A_adress [i][j][8:6])-(B_adress [j][k][8:6]));
18          else
19          Total_hops= Total_hops+ ((B_adress [j][k][8:6])-(A_adress [i][j][8:6]));
20
21  // ***************Hop count from B to R***************
22          if ((B_adress [i][j][2:0])>(R_adress [k][j][2:0]))
23          Total_hops= Total_hops+ ((B_adress [i][j][2:0])-(R_adress [k][j][2:0]));
24          else
25          Total_hops= Total_hops+ ((R_adress [k][j][2:0])-(B_adress [i][j][2:0]));
26
27          if ((B_adress [i][j][5:3])>(R_adress [k][j][5:3]))
28          Total_hops= Total_hops+ ((B_adress [i][j][5:3])-(R_adress [k][j][5:3]));
29          else
30          Total_hops= Total_hops+ ((R_adress [k][j][5:3])-(B_adress [i][j][5:3]));
31
32          if ((B_adress [i][j][8:6])>(R_adress [k][j][8:6]))
33          Total_hops= Total_hops+ ((B_adress [i][j][8:6])-(R_adress [k][j][8:6]));
34          else
35          Total_hops= Total_hops+ ((R_adress [k][j][8:6])-(B_adress [i][j][8:6]));
36        end
37      end
38    end
```

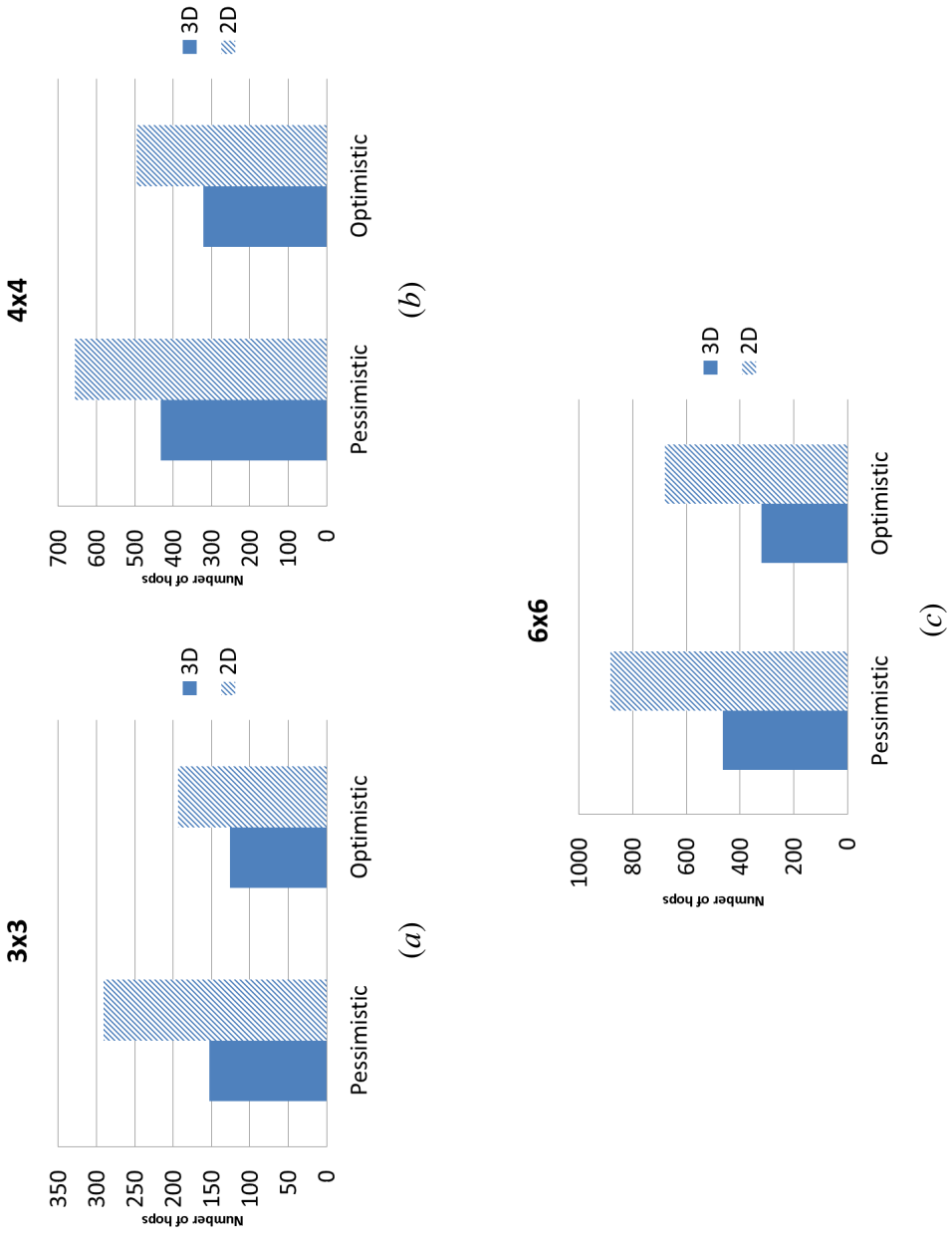

Figure 5.8: Average number of hops comparison for both pessimistic and optimistic mapping: (a) 3x3 (b) 4x4 (c) 6x6.

Figure.5.8 show the variation of the number of hops between 3D-ONoc and 2D-ONoC with 3x3, 4x4 and 6x6 matrix multiplication using pessimistic and optimistic mapping. The number of hops can be calculated using the Verilog code depicted in Code.5.1. This portion of code is added to the test bench that performs the calculation.

When we analyze this figure, we may see that 3D-ONoC reduces the number of hops compared with the 2D system with an average percentage of 42%, 31% and 47% 3x3, 4x4 and 6x6 matrices respectively having a total number of hops reduction of 40% over the 2D architecture. This can significantly reduce the execution time, since flits have fewer hops to traverse to reach their destination.

Another reason contributing on the performance of 3D-ONoC is the reduction of the traffic congestion. This can be seen by observing the *Stall-Go* flow control and the number of *stop-signal* generated by each Matrix Multiplication. To execute this calculation, we added a small portion of code (Code.5.2) at the end at the end of the 3D-ONoC module, that uses the *net-stop-out* signal issued from the flow control and calculates the total stall count.

Code 5.2: Verilog-HDL code defining the flit structure

```
1  // 3D-ONoC top module: network.v
2
3  ...
4  ...
5   always @(reset) begin
6   if (reset) count <=  0;
7   end
8
9
10  always   @(net_stop_out)
11   begin : stop
12       for (j=0;j<Y_WIDTH;j=j+1)begin
13         for (k=0;k<X_WIDTH;k=k+1)begin
14           for (l=0;l<NOUT;l=l+1)begin
15           if (net_stop_out[k][j][l]) count = count+1;
16           end
17         end
18       end
19 end
```

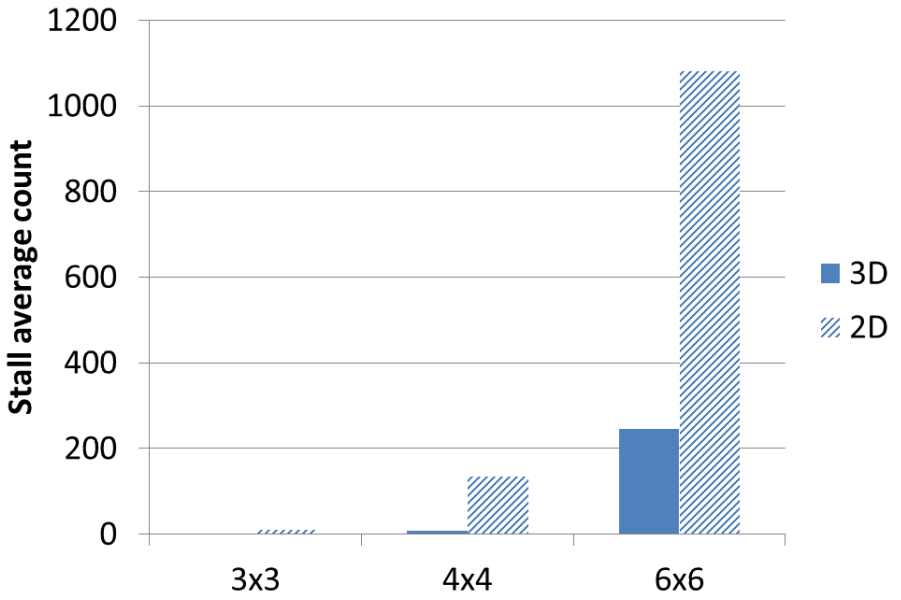

Figure 5.9: Stall average count comparison between 3D and 2D ONoC.

As a matter of fact when observing Fig.5.9, we can see that the stall count increase linearly when we increase the matrix which is related to the number of flits traveling the network. Even 3D-ONoC can reach up to 77% of stall count reduction over the 2D design with 6x6 Matrix multiplication, the stall count impact cannot be clearly seen with 3x3 and 4x4 calculation. This can simply explained by the fact that we are calculating a single matrix multiplication which generates only 54 and 128 flits for 3x3 and 4x4 matrix size respectively. This small number of flits was not enough to cause any trafiic congestions in 3D-ONoC. For that reason, we decide to extend the evaluation to calculate not only one Matrix multiplication but also to calculate 2, 3 and 4 different matrices at the same. This aims to increase the number of flits traveling the network at the same time to cause congestion. Then we evaluate again the average stall count.

Figure.5.10, depicts the average stall count of both 3D and 2D ONoC when imple-

mented with 1, 2, 3 and 4 matrix multiplications. When analyzing this figure, the stall count has been dramatically decreased to 94%, 67% and 59% in average for 3x3, 4x4 and 6x6 matrix Multiplication respectively. In total 3D-ONoC reduces the stall count to up to 74%.

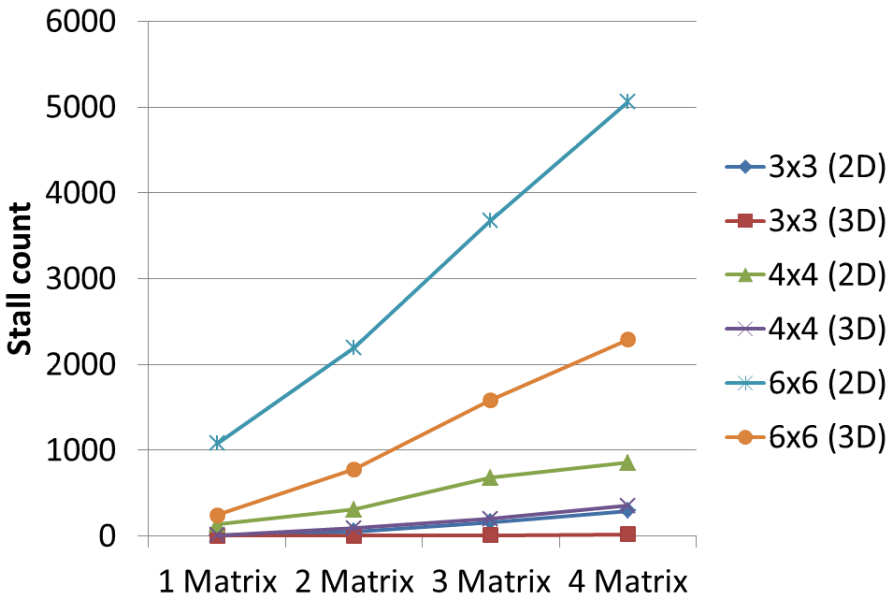

Figure 5.10: Stall average count comparison between 3D and 2D ONoC with different traffic loads.

After calculating the stall number, we want to see the impact of increasing the traffic congestion on the execution time. So evaluate again the execution time of each Matrix size when performing 1, 2, 3 and 4 matrix multiplications. The result obtained are shown in Fig.5.11 reduces the execution time to 36%, 39% and 47% for 3x3, 4x4 and 6x6 matrix Multiplication respectively. Then improving the total execution time reduction from 36%, obtained in the first experience with one matrix multiplication, to more than 41% when evaluated with heavier traffic load.

As the results mentioned above, 3D-ONoC take advantage of its ability to reduce

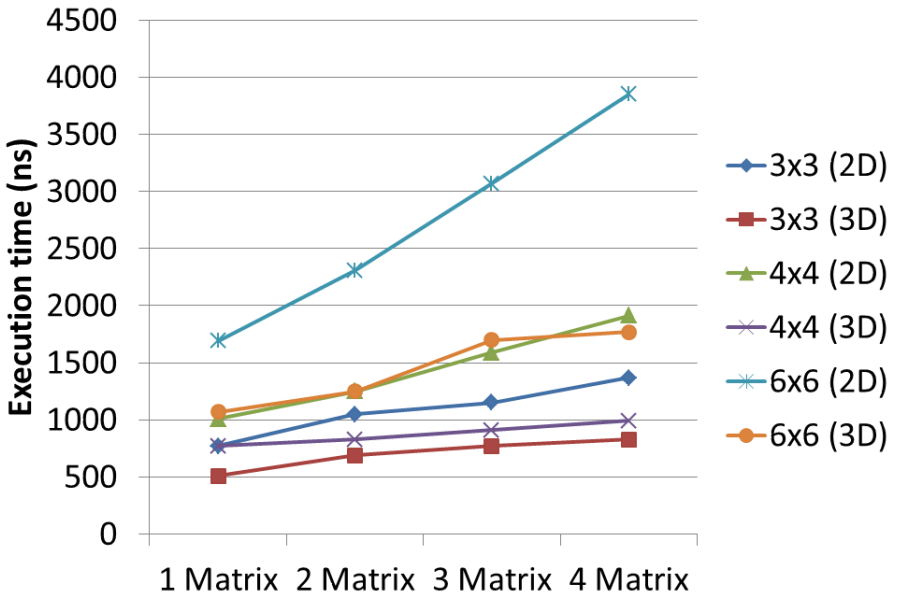

Figure 5.11: Execution time comparison between 3D and 2D ONoC with different traffic loads.

the number of hops to enhance the performance. In addition, since 3D-ONoC router has two additional input-output ports, flits traveling the network have better routing choices which eventually will decrease the congestion that can be caused when using 2D-ONoC, having an important impact on the overall performance of the system. Not forget to mention, this will improve the traffic balance along the whole network which plays a very crucial role on the thermal power dissipated from the design.

Chapter 6

Conclusion and Future Work

3D-ONoC is a natural extension of the 2D-ONoC design previously developed by our group. In this report we present a hardware design for 3D-OASIS Network-on-Chip (3D-ONoC) including complete details about the main components of the design. We also present a preliminary hardware and performance evaluation results using JPEG encoder Matrix multiplication applications.

Evaluation results show that in term of speed 3D-ONoC under-performs 2D-ONoC architecture with 16% observing a 37% area utilization penalty and a slight improvement of 1.4% in total power consumption. Despite the increasing hardware complexity, 3D ONoC shows an improvement in term of execution time by reducing the delay to 28% in overall compared to the 2D architecture.

We explained that by the fact that 3D-ONoC decreases the number of hops by 40% and also the average stall count to 74%. In a second experience we proved that by increasing the traffic load with the Matrix application, we can enhance the execution time reduction from 36% obtained with one matrix multiplication to more than 41% with 1, 2, 3 and 4 matrix multiplications.

As a future work, we will try to optimize the routing algorithm in order to enhance the performance of our design. We will try also to optimize the router architecture, especially the input buffers which is one of the most important reason of the area penalty. This aims to obtain an enhanced design of 3D-ONoC that increase the performance while keeping the hardware cost balanced and reasonable. Also, a thermal power study should be done to observe how 3D-ONoC deals with such important performance requirement.

References

[1] A. Habibi,M. Arjomand, H. Sarbazi-Azad , Multicast-Aware Mapping Algorithm for On-chip Networks, 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, Feb 2011 pp. 455-462 .

[2] G. Leary, Karam S. Chatha, Design of NoC for SoC with Multiple Use Cases Requiring Guaranteed Performance, 23rd International Conference on VLSI Design, January 2010 pp. 200-205 .

[3] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multicore Architectures: Understanding Mechanisms, Overheads and Scaling. Proc. of the 32nd Int. Sym. on Comp. Arch., pp. 408-419, Madison, USA, 2005.

[4] A. Ben Abdallah, M. Sowa, Basic, Network-on-Chip Interconnection for Future Gigascale MCSoCs Applications: Communication and Computation Orthogonalization, Proc. of The TJASSST2006 Symposium on Science, DEC. 2006.

[5] J. Kim, D. Park, T. Theocharides, V. Narayanan, C. Das. A Low Latency Router Supporting Adaptivity for On-Chip Interconnects. Proc. of the 42nd Conf. on Design Auto., pp. 559-564, 2005.

[6] J. Kim, C. Nicopoulos, D. Park, V. Narayanan, M. S. Yousif, and C. R. Das. A Gracefully Degrading and Energy-Efficient Modular Router Architecture for On-Chip Networks. Proc. of the 33rd Int. Sym. on Comp. Arch., pp. 138-149, 2006.

[7] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha. Express Virtual Channels: Towards the Ideal Interconnection Fabric. Proc. of the 34th Int. Sym. on Comp. Arch., pp. 150-161, 2007.

[8] R. Mullins, A. West, and S. Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. Proc. of the 31st Int. Sym. on Comp. Arch., pp. 188-197, 2004.

[9] W. J. Dally. Express Cubes: Improving the Performance of kary-n-cube Interconnection Networks. IEEE Trans. on Computers, 40(9):1016-1023, 1991.

[10] J. Kim, J. Balfour, and W. J. Dally. Flatterned Butterfly Topology for On-Chip Networks. Proc. of the 40th Int. Sym. on Microarchitecture, pp. 172-182, 2007.

[11] U. Y. O. and R. Marculescu. Its a Small World After All: NoC Performance Optimization via Long-Range Link Insertion. IEEE Trans. on VLSI Sys., 14(7):693-706, July 2006.

[12] G. Philip, B. Christopher, and P. Ramm, Handbook of 3D Integration: Technology and Applications of 3D Integrated Circuits, Wiley-VCH, 2008.

[13] S. Das et al. Technology, Performance, and Computer Aided Design of Three-Dimensional Integrated Circuits. In Proc. International Symposium on Physical Design, 2004.

[14] P. Morrow, M. Kobrinsky, S. Ramanathan, C.-M. Park, M. Harmes, V. Ramachandrarao, H. Park, G. Kloster, S. List, and S. Kim. Wafer-Level 3D Interconnects Via Cu Bonding. In Proc. the 21st Advanced Metallization Conference, Oct. 2004.

[15] J. Joyner, P. Zarkesh-Ha, and J. Meindl. A stochastic global net-length distribution for a three-dimensional system-on-chip(3D-SoC). In Proc. 14th Annual IEEE International ASIC/SOC Conference, Sept. 2001.

[16] A. W. Topol, J. D. C. La Tulipe, L. Shi, D. J. Frank, K. Bernstein, S. E. Steen, A. Kumar, G. U. Singco, A. M. Young, K. W. Guarini, and M. Ieong, Three-dimensional integrated circuits, IBM Journal of Research and Development, vol. 50, no. 4/5, pp. 491506, July 2006.

[17] L. P. Carloni, P. Pande, and Y. Xie, Networks-on-chip in emerging interconnect paradigms: Advantages and challenges, In Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS09), San Diego, CA, May 2009, pp. 93-102.

[18] F. Li, C. Nicopoulos, T. D. Richardson, Y. Xie, N. Vijaykrishnan, M. T. Kandemir: Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. ISCA 2006: 130-141

[19] K. Mori, A. Ben Abdallah, K. Kuroda, Design and Evaluation of a Complexity Effective Network-on-Chip Architecture on FPGA, Proc. of The 19th Intelligent System Symposium (FAN 2009), pp.318-321, Sep. 2009.

[20] K. Mori, A. Esch, A. Ben Abdallah, K. Kuroda, Advanced Design Issues for OASIS Network-on-Chip Architecture, IEEE Proc. of the 5th International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA-2010), Nov. 2010, pp. 74-79.

[21] B. Feero, P. Pratim Pande, Performance Evaluation for Three-Dimensional Networks-on-Chip, Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 9th-11th May 2007, pp. 305-310.

[22] V. F. Pavlidis, E.G. Friedman, 3-D Topologies for Networks-on-chip, IEEE Transactions on VLSI Systems, Oct. 2007, pp. 1081-1090.

[23] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir. Design and management of 3D chip multiprocessors using network-in-memory. ACM SIGARCH Computer Architecture News, 34(2):130?141, 2006.

[24] S. Yan and B. Lin. Design of application-specific 3D networks-on-chip architectures. In Proceedings of International Conference of Computer Design, pages 142149, Oct. 2008.

[25] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing", in Proc.19th Ann. Int'l Symp. Computer Architecture, May 1992, pp. 278-287.

[26] J. Hu and R. Marculescu, Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures, in Proc. DATE'03, 2003, pp. 688-693.

[27] R. S. Ramanujam and B. Lin, Near-optimal oblivious routing on threedimensional mesh networks, in Proc. IEEE Int. Conf. Comp. Design, Lake Tahoe, CA, 2008.

[28] C. H. Chao, K. Y. Jheng, H. Y. Wang, J. C. Wu, and An-Yeu Wu, "Traffic- and thermal-aware run-time thermal management scheme for 3D NoC systems," in Proc. ACM/IEEE Int. Symp. Networks-on-Chip (NoCS), Grenoble, France, May 2010, pp. 223-230.

[29] S. TYAGI, EXTENDED BALANCED DIMENSION ORDERED ROUTING ALGORITHM FOR 3D-NETWORKS, Centre for Development of Advance Computing, Noida, (U.P.), India International Conference on Parallel rocessing Workshops, pp 499-506, 2009 http://www.iacqer.com/Proceedings

[30] J. M. Montaana, M. Koibuchi, H. Matsutani, H. Amano, Balanced Dimension-Order Routing for k-ary n-cubes, Department of Information and Computer Science,Keio University, Yokohama, Japan, International Conference on Parallel rocessing Workshops, pp 499-506, 2009

[31] K. Lahiri, A. Raghunathan, and S. Dey, Efficient Exploration of the SoC Communication Architecture Design Space, in Proc. IEEE/ACM ICCAD'00, 2000, , pp. 424-430.

[32] K. Dev, Multi-Objective Optimization using evolutionary Algorithms, John Wiley and Sons Ltd, 2002, pp. 245-253.

[33] L. Xin and C.-s. Choy, A Low-latency NoC Router with Lookahead Bypass, in IEEE Int. Symp. pn Circuits and Systems (ISCAS), 2010, pp.39813984.

[34] A Ben Ahmed, A. Ben Abdallah, K. Kuroda, Architecture and Design of Efficient 3D Network-on-Chip (3D NoC) for Custom Multicore SoC, IEEE Proc. of BWCCA-2010, Nov. 2010.

[35] M. S. Rasmussen, "Network-on-Chip in Digital Hearing Aids", Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, IMM-Thesis-2006-76, 2006.

[36] A. Pullini , F. Angiolini , D. Bertozzi and L. Benini, Fault tolerance overhead in network-on-chip flow control schemes, Proceedings of the 18th annual symposium on Integrated circuits and system design, Florianolpolis, Brazil, September 04-07, 2005, pp.224 - 229

[37] B. T. Gold. "Balancing Performance, Area, and Power in an On-Chip Network.", Master's thesis, Department of Electrical and Computer Engineering, Virginia Tech, August 2004.

[38] Z, Fu and X. Ling "The design and implementation of arbiters for Network-on-chips." IEEE, Industrial and Information Systems (IIS), 2010 2nd International Conference, vol. 1, p. 292-295, 2010

[39] J. Rosethal, JPEG Image Compression Using an FPGA, Master of Science in Electrical and Computer Engineering, University of California Santa Barbara DEC. 2006.

[40] Z. WANG and O. HAMMAMI. "A 24 Processors System on Chip FPGA Design with Network on Chip".

Chapter 7

3D-OASIS Network-on-Chip with JPEG encoder and Matrix Multiplication Verilog-HDL code

3D-OASIS Network-on-Chip with JPEG encoder and Matrix multiplication is implemented in the VerilogHDL code. This appendix includes:

- 2x2x2 3D-ONoC top module *network.v*, that represents the system topology.
- 3D-ONoC implemented with JPEG encoder. This code includes 3D-ONoC topology connected with the different modules of JPEG encoder via the appropriate transmitter and receiver Network Interfaces.
- 3D-ONoC implemented with 3x3 Matrix multiplication (AxB=R) including:
  - One element of Matrix A that sends its own value to three different elements of Matrix B by adding a tag to each flit to be distinguished later in Matrix B.
  - One element of Matrix B that receives three different values from three different elements from Matrix A and perform the multiplication with its own value. Then depending on the tag received with the flit, this Matrix B element sends its the multiplied value to three different elements of Matrix R.
  - One element of Matrix R that receives three different values from three different elements from Matrix B and perform the addition before issuing the final resulted value.
  - The system top module that handles connecting the different matrices elements with 3D-ONoC system.
  - The test-bench file that initialize the different control signals, calculates the number of hops and that receives the resulted Matrix R.

Code 7.1: Code for 3D-OASIS-NoC Top Module

```
1  /*
2   *
3   * Top level network - Instantiates X_WIDTH x Y_WIDTH  x Z_WIDTH
4   * wormhole routers and interconnects them appropriately
5   *
6  */
7
8  `ifndef VCS
9   `include "defines.v"
10 `endif
11
12 module network(clk, reset,
13           data_in, data_out,
14           stop_in, stop_out);
15
16    //network size
17    parameter X_WIDTH = 2;
18    parameter Y_WIDTH = 2;
19    parameter Z_WIDTH = 2;
20
21    //router parameters
22    parameter NOUT          = 7;
23    parameter FIFO_DEPTH    = 4;
24    parameter FIFO_LOG2D    = 2;
25    parameter FIFO_FULL_LVL = 2;
26
27
28    input                                clk, reset;
29
30    input [X_WIDTH*Y_WIDTH*Z_WIDTH*`WIDTH-1:0]  data_in;
31    input [X_WIDTH*Y_WIDTH*Z_WIDTH-1:0]         stop_in;
32
33    output [X_WIDTH*Y_WIDTH*Z_WIDTH*`WIDTH-1:0] data_out;
34    output [X_WIDTH*Y_WIDTH*Z_WIDTH-1:0]        stop_out;
35
36
37    wire [(`WIDTH*NOUT)-1:0]          net_data_out [X_WIDTH-1:0][Y_WIDTH-1:0][Z_WIDTH
          -1:0];
38    wire [(`WIDTH*NOUT)-1:0]          net_data_in  [X_WIDTH-1:0][Y_WIDTH-1:0][Z_WIDTH
          -1:0];
39    wire [NOUT-1:0]           net_stop_out [X_WIDTH-1:0][Y_WIDTH-1:0][Z_WIDTH-1:0];
40    wire [NOUT-1:0]           net_stop_in  [X_WIDTH-1:0][Y_WIDTH-1:0][Z_WIDTH-1:0];
41
42
43    genvar i, x_pos, y_pos, z_pos;
44
45
46    generate
47
48    //z loop
49    for (z_pos=0; z_pos<Z_WIDTH; z_pos=z_pos+1) begin:z_loop
50
51       //y loop
52       for (y_pos=0; y_pos<Y_WIDTH; y_pos=y_pos+1) begin:y_loop
53
54    //x loop
55    for (x_pos=0; x_pos<X_WIDTH; x_pos=x_pos+1) begin:x_loop
56
57
58
59
60       router #(NOUT, FIFO_DEPTH, FIFO_LOG2D, FIFO_FULL_LVL) rtr(.clk(clk), .reset(
             reset),
61          .data_in(net_data_in[x_pos][y_pos][z_pos]), .data_out(net_data_out[x_pos][
                y_pos][z_pos]),
62          .stop_in(net_stop_in[x_pos][y_pos][z_pos]), .stop_out(net_stop_out[x_pos][
                y_pos][z_pos]),
```

```
63          .xaddr(x_pos[`L2NET_SIZE-1:0]), .yaddr(y_pos[`L2NET_SIZE-1:0]), .zaddr(z_pos
                [`L2NET_SIZE-1:0]));
64
65
66      //set up inter-router connections with correct boundary conditions
67      for (i=0; i<NOUT; i=i+1) begin:i0
68
69          //tile interface of router
70          if(i==0) begin
71      assign net_data_in[x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = data_in[(`
            WIDTH*X_WIDTH*z_pos)+(`WIDTH*Y_WIDTH*z_pos)+(`WIDTH*X_WIDTH*y_pos)+(`WIDTH
            *(x_pos+1))-1:(`WIDTH*X_WIDTH*z_pos)+(`WIDTH*Y_WIDTH*z_pos)+(`WIDTH*X_WIDTH
            *y_pos)+(`WIDTH*x_pos)];
72      assign data_out[(`WIDTH*X_WIDTH*z_pos)+(`WIDTH*Y_WIDTH*z_pos)+(`WIDTH*X_WIDTH*
            y_pos)+(`WIDTH*(x_pos+1))-1:(`WIDTH*X_WIDTH*z_pos)+(`WIDTH*Y_WIDTH*z_pos
            )+(`WIDTH*X_WIDTH*y_pos)+(`WIDTH*x_pos)] = net_data_out[x_pos][y_pos][z_pos
            ][`WIDTH*(i+1)-1:`WIDTH*i];
73
74      assign net_stop_in[x_pos][y_pos][z_pos][i] = stop_in[(X_WIDTH*z_pos)+(Y_WIDTH*
            z_pos)+(X_WIDTH*y_pos)+x_pos];
75      assign stop_out[(X_WIDTH*z_pos)+(Y_WIDTH*z_pos)+(X_WIDTH*y_pos)+x_pos] =
            net_stop_out[x_pos][y_pos][z_pos][i];
76          end
77
78          //north edge of router
79          if(i==1) begin
80      if(y_pos==Y_WIDTH-1) begin
81          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
82          assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
83      end else begin
84          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                net_data_out[x_pos][y_pos+1][z_pos][`WIDTH*(3+1)-1:`WIDTH*3];
85          assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos][y_pos+1][
                z_pos][3];
86      end
87          end
88
89          //east edge of router
90          if(i==2) begin
91      if(x_pos==X_WIDTH-1) begin
92          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
93          assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
94      end else begin
95          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                net_data_out[x_pos+1][y_pos][z_pos][`WIDTH*(4+1)-1:`WIDTH*4];
96          assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos+1][y_pos][
                z_pos][4];
97      end
98          end
99
100         //south edge of router
101         if(i==3) begin
102     if(y_pos==0) begin
103         assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
104         assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
105     end else begin
106         assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                net_data_out[x_pos][y_pos-1][z_pos][`WIDTH*(1+1)-1:`WIDTH*1];
107         assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos][y_pos-1][
                z_pos][1];
108     end
109         end
110
111         //west edge of router
112         if(i==4) begin
113     if(x_pos==0) begin
114         assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
115         assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
```

```
116       end else begin
117          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                 net_data_out[x_pos-1][y_pos][z_pos][`WIDTH*(2+1)-1:`WIDTH*2];
118          assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos-1][y_pos][
                 z_pos][2];
119       end
120          end
121          //up edge of router
122          if(i==5) begin
123       if(z_pos==Z_WIDTH-1) begin
124          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
125          assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
126       end else begin
127          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                 net_data_out[x_pos][y_pos][z_pos+1][`WIDTH*(6+1)-1:`WIDTH*6];
128          assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos][y_pos][
                 z_pos+1][6];
129       end
130          end
131
132          //down edge of router
133
134
135              if(i==6) begin
136       if(z_pos==0) begin
137          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] = 0;
138          assign net_stop_in [x_pos][y_pos][z_pos][i] = 1'b1;
139       end else begin
140          assign net_data_in [x_pos][y_pos][z_pos][`WIDTH*(i+1)-1:`WIDTH*i] =
                 net_data_out[x_pos][y_pos][z_pos-1][`WIDTH*(5+1)-1:`WIDTH*5];
141          assign net_stop_in [x_pos][y_pos][z_pos][i] = net_stop_out[x_pos][y_pos][
                 z_pos-1][5];
142         end
143        end
144
145     end // for (i=0; i<NOUT-1; i=i+1)
146
147    end // block: x_loop
148
149   end // block: y_loop
150
151  end // block: z_loop
152
153 endgenerate
154
155 endmodule // network
```

7.1 3D-ONoC with JPEG encoder

Code 7.2: 3D-OASIS Network-on-Chip with JPEG encoder top module

```
1  `ifndef VCS
2   `include "defines.v"
3  `endif
4
5  module 3D-NoC-JPEG (clk,clk1,reset,
6                      RGB_stream, enb_in,
7                      end_of_file_signal,
8                      JPEG_bitstream, data_ready,
9                      end_of_file_bitstream_count,
10                     eof_data_partial_ready);
11
12          // input and output
13    input          clk;      // For NoC and NI
14    input          clk1;     // For JPEG modules
15    input          reset;
16    input    [23:0]    RGB_stream;
17    input          enb_in;
18   input   end_of_file_signal;
19
20   output  [31:0]    JPEG_bitstream;
21   output          data_ready;
22   output  [4:0]   end_of_file_bitstream_count;
23   output          eof_data_partial_ready;
24
25 // clock registers
26
27    reg      [3:0]   i;
28
29
30 // ********************************
31 // ***************Wires***************
32 // ********************************
33
34   // Network
35    wire   [32:0]    data_in_13,data_in_12,data_in_11,data_in_10,data_in_03,data_in_02
          ,data_in_01,data_in_00;
36    wire   [32:0]  w_13,w_12,w_11,w_10,w_03,w_02,w_01,w_00;
37
38    wire     [7:0]   stop_out_N;
39    wire     [7:0]   stop_in_N;
40
41   // RGBtoYCrCb module
42    wire   [23:0]    RGB2YCBCR_in;
43    wire         enable_in;
44
45    wire   [23:0]    RGB2YCBCR_out;
46    wire         enable_out;
47
48   // Y d_q_h module
49    wire     [7:0]   data_Y;
50    wire         enb_Y;
51
52   wire    [31:0]  JPEG_Y;
53    wire         ready_Y;
54    wire     [4:0]   orc_Y;
55    wire         eobo_Y;
56    wire         eobe_Y;
57
58   // Cr d_q_h module
59    wire     [7:0]   data_Cr;
60    wire         enb_Cr;
61
62   wire    [31:0]  JPEG_Cr;
```

```
63    wire          ready_Cr;
64    wire     [4:0]   orc_Cr;
65    wire          eobe_Cr;
66
67   // Cb d_q_h module
68    wire     [7:0]   data_Cb;
69    wire          enb_Cb;
70
71   wire    [31:0]  JPEG_Cb;
72    wire          ready_Cb;
73    wire     [4:0]   orc_Cb;
74   wire          eobe_Cb;
75
76   // output to Y
77   wire [31:0] y_data_in;
78    wire       y_data_ready;
79    wire [4:0]   y_orc;
80    wire       y_eof_output;
81    wire       y_eof_empty;
82
83   // output to Cb
84   wire [31:0] Cb_data_in;
85    wire     Cb_data_ready;
86    wire [4:0]   Cb_orc;
87    wire       Cb_eof_empty;
88
89   // output to Y
90   wire [31:0] Cr_data_in;
91    wire       Cr_data_ready;
92    wire [4:0]   Cr_orc;
93    wire       Cr_eof_empty;
94
95   // output from FIFO
96   wire  [31:0]  JPEG_out;
97   wire      data_ready_out;
98   wire  [4:0] orc_reg_out;
99
100  // output to Checker
101  wire [31:0] Che_JPEG_data;
102   wire       Che_data_ready;
103   wire [4:0]   Che_orc;
104
105  wire [31:0] che_JPEG_bitstream1;
106  wire      che_data_ready_1;
107  wire [4:0]  che_end_of_file_bitstream_count;
108  wire      che_ff_eof_data_partial_ready;
109
110
111 network network  (.clk(clk),
112             .reset(reset),
113             .data_in({data_in_13,data_in_12,data_in_11,data_in_10,data_in_03,
                        data_in_02,data_in_01,data_in_00}),
114             .data_out({w_13,w_12,w_11,w_10,w_03,w_02,w_01,w_00}),
115             .stop_in(stop_in_N),
116             .stop_out(stop_out_N)
117             );
118
119 // RGB stream sending to RGB2YCBCR
120 NI_02_send NIsend02(.clk(clk),
121             .rst(reset),
122             .enable(enb_in),
123             .data_in(RGB_stream),
124             .flit(data_in_02)
125             );
126
127 // RGB2YCBCR receiving from RGB stream
128 NI_03_rec NIrec03  (.clk(clk),
129             .rst(reset),
```

```
130             .flit(w_03),
131             .data_out(RGB2YCBCR_in),
132             .enable(enable_in)
133             );
134 
135 // RGB2YCBCR module
136 RGB2YCBCR RGB   (.clk(clk1),
137             .rst(reset),
138             .enable(enable_in),
139             .data_in(RGB2YCBCR_in),
140             .data_out(RGB2YCBCR_out),
141             .enable_out(enable_out)
142             );
143 
144 // RGB2YCBCR sending to Y, Cr, and Cb
145 NI_03_send NIsend03(.clk(clk),
146             .rst(reset),
147             .enable(enable_out),
148             .data_in(RGB2YCBCR_out),
149             .flit(data_in_03)
150             );
151 
152 // Y receiving from RGB2YCBCR
153 NI_00_rec NIrec00 (.clk(clk),
154             .rst(reset),
155             .flit(w_00),
156             .data_out(data_Y),
157             .enable(enb_Y)
158             );
159 // Y_dqh  module
160 yd_q_h Y_dqh    (.clk(clk1),
161             .rst(reset),
162             .enable(enb_Y),
163             .data_in(data_Y),
164             .JPEG_bitstream(JPEG_Y),
165             .data_ready(ready_Y),
166             .y_orc(orc_Y),
167             .end_of_block_output(eobo_Y),
168             .end_of_block_empty(eobe_Y)
169             );
170 
171 // Y sending to FIFO
172 NI_00_send NIsend00(.clk(clk),
173             .rst(reset),
174             .y_data_in(JPEG_Y),
175             .y_data_ready(ready_Y),
176             .y_orc(orc_Y),
177             .y_eof_output(eobo_Y),
178             .y_eof_empty(eobe_Y),
179             .flit(data_in_00)
180             );
181 
182 // Cb receiving from RGB2YCBCR
183 NI_11_rec NIrec11  (.clk(clk),
184             .rst(reset),
185             .flit(w_11),
186             .data_out(data_Cb),
187             .enable(enb_Cb)
188             );
189 
190 // Cb_dqh module
191 cbd_q_h Cb_dqh    (.clk(clk1),
192             .rst(reset),
193             .enable(enb_Cb),
194             .data_in(data_Cb),
195             .JPEG_bitstream(JPEG_Cb),
196             .data_ready(ready_Cb),
197             .cb_orc(orc_Cb),
```

```
198             .end_of_block_empty(eobe_Cb)
199             );
200
201 // Cb sending to FIFO
202 NI_11_send NIsend11(.clk(clk),
203             .rst(reset),
204             .Cb_data_in(JPEG_Cb),
205             .Cb_data_ready(ready_Cb),
206             .Cb_orc(orc_Cb),
207             .Cb_eof_empty(eobe_Cb),
208             .flit(data_in_11)
209             );
210
211 // Cr receiving from RGB2YCBCR
212 NI_01_rec NIrec01  (.clk(clk),
213             .rst(reset),
214             .flit(w_01),
215             .data_out(data_Cr),
216             .enable(enb_Cr)
217             );
218
219 // Cr_dqh module
220 crd_q_h Cr_dqh    (.clk(clk1),
221             .rst(reset),
222             .enable(enb_Cr),
223             .data_in(data_Cr),
224             .JPEG_bitstream(JPEG_Cr),
225             .data_ready(ready_Cr),
226             .cr_orc(orc_Cr),
227             .end_of_block_empty(eobe_Cr)
228             );
229
230 // Cr sending to FIFO
231 NI_01_send NIsend01(.clk(clk),
232             .rst(reset),
233             .Cr_data_in(JPEG_Cr),
234             .Cr_data_ready(ready_Cr),
235             .Cr_orc(orc_Cr),
236             .Cr_eof_empty(eobe_Cr),
237             .flit(data_in_01)
238             );
239
240 // FIFO receiving from Y, Cb, Cr
241 NI_10_rec NIrec10  (.clk(clk),
242             .rst(reset),
243             .flit(w_10),
244             .y_data_in(y_data_in), .y_data_ready(y_data_ready), .y_orc(y_orc), .
                    y_eof_output(y_eof_output), .y_eof_empty(y_eof_empty),
245             .Cb_data_in(Cb_data_in), .Cb_data_ready(Cb_data_ready), .Cb_orc(Cb_orc),
                    .Cb_eof_empty(Cb_eof_empty),
246             .Cr_data_in(Cr_data_in), .Cr_data_ready(Cr_data_ready), .Cr_orc(Cr_orc),
                    .Cr_eof_empty(Cr_eof_empty)
247             );
248
249 // FIFO receiving from Y, Cb, Cr
250 fifo_out FIFO   (.clk(clk1),
251              .rst(reset),
252              .cb_JPEG_bitstream(Cb_data_in), .cr_JPEG_bitstream(Cr_data_in), .
                     y_JPEG_bitstream(y_data_in),
253              .cb_orc(Cb_orc), .cr_orc(Cr_orc), .y_orc(y_orc),
254              .cb_data_ready(Cb_data_ready), .cr_data_ready(Cr_data_ready), .
                     y_data_ready(y_data_ready),
255              .end_of_block_output(y_eof_output),
256              .cb_eob_empty(Cb_eof_empty), .cr_eob_empty(Cr_eof_empty), .y_eob_empty(
                     y_eof_empty),
257              .JPEG_bitstream(JPEG_out), .data_ready(data_ready_out), .orc_reg(
                     orc_reg_out));
258
```

```
259
260 // FIFO sending to Checker
261 NI_10_send NIsend10(.clk(clk),
262             .rst(reset),
263             .JPEG_data(JPEG_out),
264             .data_ready(data_ready_out),
265             .orc(orc_reg_out),
266             .flit(data_in_10)
267             );
268
269 // Checker receiving from FIFO
270 NI_12_rec NIrec12  (.clk(clk),
271             .rst(reset),
272             .flit(w_12),
273             .JPEG_data(Che_JPEG_data),
274             .data_ready(Che_data_ready),
275             .orc(Che_orc)
276             );
277 // FF_checker module
278
279  ff_checker ff_check (.clk(clk1),
280               .rst(reset),
281                .end_of_file_signal(end_of_file_signal),
282                .JPEG_in(Che_JPEG_data),
283                .data_ready_in(Che_data_ready),
284                .orc_reg_in(Che_orc),
285                .JPEG_bitstream_1(che_JPEG_bitstream1),
286                .data_ready_1(che_data_ready1),
287                .orc_reg(che_end_of_file_bitstream_count),
288                .eof_data_partial_ready(che_eof_data_partial_ready));
289
290
291 // Checker sending to JPEG_bitstream
292 NI_12_send NIsend12(.clk(clk),
293             .rst(reset),
294             .JPEG_data(che_JPEG_bitstream1),
295             .data_ready(che_data_ready1),
296             .orc(che_end_of_file_bitstream_count),
297             .eof_data_partial_ready(che_eof_data_partial_ready),
298             .flit(data_in_12)
299             );
300
301 // JPEG_bitstream receiving from Checker
302 NI_13_rec NIrec13  (.clk(clk),
303             .rst(reset),
304             .flit(w_13),
305             .JPEG_data(JPEG_bitstream),
306             .data_ready(data_ready),
307             .orc(end_of_file_bitstream_count),
308             .eofr(eof_data_partial_ready)
309             );
310 endmodule
```

7.2 3D-ONoC with 3x3 Matrix Multiplication

Code 7.3: Verilog-HDL sample code For Matrix A

```
1
2    //**** This module represents one element of the first Matrix A ****//
3  //**** It sends 3 different flits to 3 different elements of Matrix B ****//
4
5  `ifndef VCS
6   `include "defines.v"
7  `endif
8
9  module Matrix (clk, reset,
10            Matrix_value_A,
11            next_port1,dest1,
12            next_port2,dest2,
13            next_port3,dest3,
14            tag,
15            data_out);
16
17 input                    clk, reset;
18 input     [`Matrix_WIDTH-1:0]   Matrix_value_A;
19 input     [6:0]           next_port1,next_port2,next_port3;
20 input     [8:0]           dest1,dest2,dest3;
21 input     [2:0]           tag;
22 output reg  [`WIDTH-1:0]      data_out;
23
24 reg [3:0]       state;
25
26    always @(posedge clk)begin
27       if(reset==1)
28     state <= `f1;
29       else begin
30     if (state==4'b0100) data_out <= 0;
31     else state <= state+1;
32     end
33
34      case(state)
35
36       `f1:begin
37         if(reset==0)
38           begin
39           data_out[0]    <= 1'b1;
40           data_out[`NEXT_PORT]   <= next_port1;
41           data_out[`DEST]  <= dest1;
42           data_out[`DATA_Mat] <= Matrix_value_A;
43           data_out[`TAG] <= tag;
44           end
45         else state <= `f1;
46       end
47
48       `f2:begin
49         data_out[0]    <= 1'b1;
50         data_out[`NEXT_PORT]   <= next_port2;
51         data_out[`DEST]  <= dest2;
52         data_out[`DATA_Mat] <= Matrix_value_A;
53         data_out[`TAG] <= tag;
54       end
55
56       `f3:begin
57         data_out[0]    <= 1'b1;
58         data_out[`NEXT_PORT]   <= next_port3;
59         data_out[`DEST]  <= dest3;
60         data_out[`DATA_Mat] <= Matrix_value_A;
61         data_out[`TAG] <= tag;
62       end
63
```

```
64        `f4:begin
65         data_out <= 0;
66        end
67        default:state <= `f4;
68
69      endcase
70
71    end
72  endmodule
```

Code 7.4: Verilog-HDL sample code For Matrix B

```
1
2         //**** This module represents one element of the second Matrix B ****//
3       //**** It receives 3 different flits to 3 different elements of Matrix A ****//
4                //**** verifies the tag, make the multiplication****//
5       //**** sends the resulted multiplication to 3 different elements of Matrix R
            ****//
6
7  `ifndef VCS
8   `include "defines.v"
9  `endif
10
11 module Matrix3 (clk, reset,
12           Matrix_value_B, data_in,
13           next_port1,dest1,
14           next_port2,dest2,
15           next_port3,dest3,
16           data_out);
17
18 input                  clk, reset;
19 input      [`Matrix_WIDTH-1:0] Matrix_value_B;
20 input      [`WIDTH-1:0]    data_in;
21 input      [6:0]         next_port1,next_port2,next_port3;
22 input      [8:0]         dest1,dest2,dest3;
23
24 output reg  [`WIDTH-1:0] data_out;
25
26
27 always @(data_in)begin
28      if(!data_in) data_out <= 0;
29     else begin
30         if(data_in[`TAG]==3'b000) begin
31           data_out[0]    <= 1'b1;
32           data_out[`NEXT_PORT]   <= next_port1;
33           data_out[`DEST]  <= dest1;
34           data_out[`DATA_Mat] <= Matrix_value_B*data_in[`DATA_Mat];
35           data_out[`TAG] <= data_in[`TAG];
36         end
37         else begin
38             if(data_in[`TAG]==3'b001) begin
39               data_out[0]    <= 1'b1;
40               data_out[`NEXT_PORT]   <= next_port2;
41               data_out[`DEST]  <= dest2;
42               data_out[`DATA_Mat] <= Matrix_value_B*data_in[`DATA_Mat];
43               data_out[`TAG] <= data_in[`TAG];
44             end
45             else begin
46               data_out[0]    <= 1'b1;
47               data_out[`NEXT_PORT]   <= next_port3;
48               data_out[`DEST]  <= dest3;
49               data_out[`DATA_Mat] <= Matrix_value_B*data_in[`DATA_Mat];
50               data_out[`TAG] <= data_in[`TAG];
51             end
52         end
53     end
54 end
55 endmodule
```

Code 7.5: Verilog-HDL sample code For Matrix R

```
1
2          //**** This module represents one element of the resulted Matrix R ****//
3        //**** It receives 3 different flits to 3 different elements of Matrix B ****//
4                 //**** verifies the tag, make the final addition****//
5
6  `ifndef VCS
7   `include "defines.v"
8  `endif
9
10 module Matrix2 (clk, reset,
11           data_in,
12           Matrix_value_R);
13
14 input                  clk, reset;
15 input     [`WIDTH-1:0]    data_in;
16
17 output reg  [`Matrix_WIDTH-1:0] Matrix_value_R;
18
19
20    always @(posedge clk)begin
21   if(reset==1) Matrix_value_R <= 0;
22   else
23     if (data_in) begin
24       Matrix_value_R <= Matrix_value_R+data_in[`DATA_Mat];
25     end
26   end
27 endmodule
```

Code 7.6: 3D-OASIS Network-on-Chip with 3x3 Matrix Multiplication top module

```
1
2
3  `ifndef VCS
4   `include "defines.v"
5  `endif
6
7  module System (clk,reset,
8            val_R11,val_R12,val_R13,
9            val_R21,val_R22,val_R23,
10           val_R31,val_R32,val_R33);
11
12 input                  clk, reset;
13 output  [`Matrix_WIDTH-1:0]   val_R11,val_R12,val_R13,val_R21,val_R22,val_R23,val_R31
       ,val_R32,val_R33;
14
15
16    wire [(`WIDTH-1):0] w_00,w_01,w_02,w_03,w_04,w_05,w_06,w_07,w_08;
17    wire [(`WIDTH-1):0] w_10,w_11,w_12,w_13,w_14,w_15,w_16,w_17,w_18;
18    wire [(`WIDTH-1):0] w_20,w_21,w_22,w_23,w_24,w_25,w_26,w_27,w_28;
19
20    wire [(`WIDTH-1):0] data_in_08,data_in_07,data_in_06,data_in_05,data_in_04,
          data_in_03,data_in_02,data_in_01,data_in_00;
21    wire [(`WIDTH-1):0] data_in_18,data_in_17,data_in_16,data_in_15,data_in_14,
          data_in_13,data_in_12,data_in_11,data_in_10;
22    wire [(`WIDTH-1):0] data_in_28,data_in_27,data_in_26,data_in_25,data_in_24,
          data_in_23,data_in_22,data_in_21,data_in_20;
23
24   wire [`X_WIDTH*`Y_WIDTH*`Z_WIDTH-1:0] stop_out;
25   wire [`X_WIDTH*`Y_WIDTH*`Z_WIDTH-1:0] stop_in;
26
27  network network( .clk(clk),
28              .reset(reset),
29              .data_in({data_in_28,data_in_27,data_in_26,data_in_25,data_in_24,
                    data_in_23,data_in_22,data_in_21,data_in_20,
30                     data_in_18,data_in_17,data_in_16,data_in_15,data_in_14,data_in_13,
                           data_in_12,data_in_11,data_in_10,
31                     data_in_08,data_in_07,data_in_06,data_in_05,data_in_04,data_in_03,
                           data_in_02,data_in_01,data_in_00}),
32              .data_out({w_28,w_27,w_26,w_25,w_24,w_23,w_22,w_21,w_20,
33                      w_18,w_17,w_16,w_15,w_14,w_13,w_12,w_11,w_10,
34                      w_08,w_07,w_06,w_05,w_04,w_03,w_02,w_01,w_00}),
35              .stop_in(stop_in),
36              .stop_out(stop_out));
37
38 // ***********************************************
39 // ***************Matrix_1   Layer_1***************
40 // ***********************************************
41
42 Matrix A11  (.clk(clk),
43         .reset(reset),
44         .Matrix_value_A(`val_A11*(!reset)),
45         .next_port1(`UP),.dest1(`dest_B11),
46         .next_port2(`EAST),.dest2(`dest_B12),
47         .next_port3(`EAST),.dest3(`dest_B13),
48         .tag(3'b000),
49         .data_out(data_in_06));
50
51 Matrix A12  (.clk(clk),
52         .reset(reset),
53         .Matrix_value_A(`val_A12*(!reset)),
54         .next_port1(`WEST),.dest1(`dest_B21),
55         .next_port2(`SOUTH),.dest2(`dest_B22),
56         .next_port3(`EAST),.dest3(`dest_B23),
57         .tag(3'b000),
58         .data_out(data_in_07));
59
60
```

```
61  Matrix A13  (.clk(clk),
62          .reset(reset),
63          .Matrix_value_A(`val_A13*(!reset)),
64          .next_port1(`WEST),.dest1(`dest_B31),
65          .next_port2(`WEST),.dest2(`dest_B32),
66          .next_port3(`SOUTH),.dest3(`dest_B33),
67          .tag(3'b000),
68          .data_out(data_in_08));
69
70
71  Matrix A21  (.clk(clk),
72          .reset(reset),
73          .Matrix_value_A(`val_A21*(!reset)),
74          .next_port1(`NORTH),.dest1(`dest_B11),
75          .next_port2(`EAST),.dest2(`dest_B12),
76          .next_port3(`EAST),.dest3(`dest_B13),
77          .tag(3'b001),
78          .data_out(data_in_03));
79
80
81  Matrix A22  (.clk(clk),
82          .reset(reset),
83          .Matrix_value_A(`val_A22*(!reset)),
84          .next_port1(`WEST),.dest1(`dest_B21),
85          .next_port2(`UP),.dest2(`dest_B22),
86          .next_port3(`EAST),.dest3(`dest_B23),
87          .tag(3'b001),
88          .data_out(data_in_04));
89
90  Matrix A23  (.clk(clk),
91          .reset(reset),
92          .Matrix_value_A(`val_A23*(!reset)),
93          .next_port1(`WEST),.dest1(`dest_B31),
94          .next_port2(`WEST),.dest2(`dest_B32),
95          .next_port3(`SOUTH),.dest3(`dest_B33),
96          .tag(3'b001),
97          .data_out(data_in_05));
98
99
100 Matrix A31  (.clk(clk),
101         .reset(reset),
102         .Matrix_value_A(`val_A31*(!reset)),
103         .next_port1(`NORTH),.dest1(`dest_B11),
104         .next_port2(`EAST),.dest2(`dest_B12),
105         .next_port3(`EAST),.dest3(`dest_B13),
106         .tag(3'b010),
107         .data_out(data_in_00));
108
109
110 Matrix A32  (.clk(clk),
111         .reset(reset),
112         .Matrix_value_A(`val_A32*(!reset)),
113         .next_port1(`WEST),.dest1(`dest_B21),
114         .next_port2(`NORTH),.dest2(`dest_B22),
115         .next_port3(`EAST),.dest3(`dest_B23),
116         .tag(3'b010),
117         .data_out(data_in_01));
118
119
120 Matrix A33  (.clk(clk),
121         .reset(reset),
122         .Matrix_value_A(`val_A33*(!reset)),
123         .next_port1(`WEST),.dest1(`dest_B31),
124         .next_port2(`WEST),.dest2(`dest_B32),
125         .next_port3(`UP),.dest3(`dest_B33),
126         .tag(3'b010),
127         .data_out(data_in_02));
128
```

```
129
130 // ***********************************************
131 // ***************Matrix_2   Layer_2***************
132 // ***********************************************
133
134 Matrix2 R11 (.clk(clk),
135         .reset(reset),
136         .data_in(w_16),
137         .Matrix_value_R(val_R11));
138
139 Matrix2 R12 (.clk(clk),
140         .reset(reset),
141         .data_in(w_17),
142         .Matrix_value_R(val_R12));
143
144
145 Matrix2 R13 (.clk(clk),
146         .reset(reset),
147         .data_in(w_18),
148         .Matrix_value_R(val_R13));
149
150
151 Matrix2 R21 (.clk(clk),
152         .reset(reset),
153         .data_in(w_13),
154         .Matrix_value_R(val_R21));
155
156
157 Matrix2 R22 (.clk(clk),
158         .reset(reset),
159         .data_in(w_14),
160         .Matrix_value_R(val_R22));
161
162
163 Matrix2 R23 (.clk(clk),
164         .reset(reset),
165         .data_in(w_15),
166         .Matrix_value_R(val_R23));
167
168
169 Matrix2 R31 (.clk(clk),
170         .reset(reset),
171         .data_in(w_10),
172         .Matrix_value_R(val_R31));
173
174
175 Matrix2 R32 (.clk(clk),
176         .reset(reset),
177         .data_in(w_11),
178         .Matrix_value_R(val_R32));
179
180
181 Matrix2 R33 (.clk(clk),
182         .reset(reset),
183         .data_in(w_12),
184         .Matrix_value_R(val_R33));
185
186 // ***********************************************
187 // ***************Matrix_3   Layer_3***************
188 // ***********************************************
189
190 Matrix3 B11 (.clk(clk),
191         .reset(reset),
192         .Matrix_value_B(`val_B11*(!reset)),
193         .data_in(w_26),
194         .next_port1(`DOWN),.dest1(`dest_R11),
195         .next_port2(`SOUTH),.dest2(`dest_R21),
196         .next_port3(`SOUTH),.dest3(`dest_R31),
```

```
197         .data_out(data_in_26));
198
199 Matrix3 B12 (.clk(clk),
200         .reset(reset),
201         .Matrix_value_B(`val_B12*(!reset)),
202         .data_in(w_27),
203         .next_port1(`EAST),.dest1(`dest_R12),
204         .next_port2(`EAST),.dest2(`dest_R22),
205         .next_port3(`EAST),.dest3(`dest_R32),
206         .data_out(data_in_27));
207
208
209 Matrix3 B13 (.clk(clk),
210         .reset(reset),
211         .Matrix_value_B(`val_B13*(!reset)),
212         .data_in(w_28),
213         .next_port1(`DOWN),.dest1(`dest_R13),
214         .next_port2(`SOUTH),.dest2(`dest_R23),
215         .next_port3(`SOUTH),.dest3(`dest_R33),
216         .data_out(data_in_28));
217
218
219 Matrix3 B21 (.clk(clk),
220         .reset(reset),
221         .Matrix_value_B(`val_B21*(!reset)),
222         .data_in(w_23),
223         .next_port1(`NORTH),.dest1(`dest_R11),
224         .next_port2(`DOWN),.dest2(`dest_R21),
225         .next_port3(`SOUTH),.dest3(`dest_R31),
226         .data_out(data_in_23));
227
228
229 Matrix3 B22 (.clk(clk),
230         .reset(reset),
231         .Matrix_value_B(`val_B22*(!reset)),
232         .data_in(w_24),
233         .next_port1(`NORTH),.dest1(`dest_R12),
234         .next_port2(`DOWN),.dest2(`dest_R22),
235         .next_port3(`SOUTH),.dest3(`dest_R32),
236         .data_out(data_in_24));
237
238
239 Matrix3 B23 (.clk(clk),
240         .reset(reset),
241         .Matrix_value_B(`val_B23*(!reset)),
242         .data_in(w_25),
243         .next_port1(`NORTH),.dest1(`dest_R13),
244         .next_port2(`DOWN),.dest2(`dest_R23),
245         .next_port3(`SOUTH),.dest3(`dest_R33),
246         .data_out(data_in_25));
247
248
249 Matrix3 B31 (.clk(clk),
250         .reset(reset),
251         .Matrix_value_B(`val_B31*(!reset)),
252         .data_in(w_20),
253         .next_port1(`NORTH),.dest1(`dest_R11),
254         .next_port2(`NORTH),.dest2(`dest_R21),
255         .next_port3(`DOWN),.dest3(`dest_R31),
256         .data_out(data_in_20));
257
258
259 Matrix3 B32 (.clk(clk),
260         .reset(reset),
261         .Matrix_value_B(`val_B32*(!reset)),
262         .data_in(w_21),
263         .next_port1(`NORTH),.dest1(`dest_R12),
264         .next_port2(`NORTH),.dest2(`dest_R22),
```

```
265         .next_port3(`DOWN),.dest3(`dest_R32),
266         .data_out(data_in_21));
267
268
269 Matrix3 B33 (.clk(clk),
270         .reset(reset),
271         .Matrix_value_B(`val_B33*(!reset)),
272         .data_in(w_22),
273         .next_port1(`NORTH),.dest1(`dest_R13),
274         .next_port2(`NORTH),.dest2(`dest_R23),
275         .next_port3(`DOWN),.dest3(`dest_R33),
276         .data_out(data_in_22));
277
278
279 endmodule //   TOP_test
```

Code 7.7: Test-bench for 3x3 Matrix multiplication

```
1
2
3  `ifndef VCS
4   `include "defines.v"
5  `endif
6
7
8  module Test;
9
10 wire [`Matrix_WIDTH-1:0]  val_R11,val_R12,val_R13;
11 wire [`Matrix_WIDTH-1:0]  val_R21,val_R22,val_R23;
12 wire [`Matrix_WIDTH-1:0]  val_R31,val_R32,val_R33;
13
14 reg clk;
15 reg reset;
16
17 reg [8:0]          A_adress [3:1][3:1];
18 reg [8:0]          B_adress [3:1][3:1];
19 reg [8:0]          R_adress [3:1][3:1];
20
21 reg [9:0] Total_hops;
22
23 integer i,j,k;
24   System sys (.clk(clk),
25           .reset(reset),
26           .val_R11(val_R11),.val_R12(val_R12),.val_R13(val_R13),
27           .val_R21(val_R21),.val_R22(val_R22),.val_R23(val_R23),
28           .val_R31(val_R31),.val_R32(val_R32),.val_R33(val_R33));
29
30    always #100000 clk = ~clk;
31
32    initial begin
33       #0
34       clk = 0;
35       reset = 1'b1;
36
37     Total_hops <= 0;
38     // ***********************************************
39     // **************Matrix_1   Layer_1***************
40     // ***********************************************
41     A_adress [1][1] =`dest_A11 ;
42     A_adress [1][2] =`dest_A12 ;
43     A_adress [1][3] =`dest_A13 ;
44     A_adress [2][1] =`dest_A21 ;
45     A_adress [2][2] =`dest_A22 ;
46     A_adress [2][3] =`dest_A23 ;
47     A_adress [3][1] =`dest_A31 ;
48     A_adress [3][2] =`dest_A32 ;
49     A_adress [3][3] =`dest_A33 ;
50     // ***********************************************
51     // **************Matrix_2   Layer_2***************
52     // ***********************************************
53     R_adress [1][1] =`dest_R11 ;
54     R_adress [1][2] =`dest_R12 ;
55     R_adress [1][3] =`dest_R13 ;
56     R_adress [2][1] =`dest_R21 ;
57     R_adress [2][2] =`dest_R22 ;
58     R_adress [2][3] =`dest_R23 ;
59     R_adress [3][1] =`dest_R31 ;
60     R_adress [3][2] =`dest_R32 ;
61     R_adress [3][3] =`dest_R33 ;
62
63     // ***********************************************
64     // **************Matrix_3   Layer_3***************
65     // ***********************************************
66     B_adress [1][1] =`dest_B11 ;
67     B_adress [1][2] =`dest_B12 ;
```

```
    B_adress [1][3] =‘dest_B13 ;
    B_adress [2][1] =‘dest_B21 ;
    B_adress [2][2] =‘dest_B22 ;
    B_adress [2][3] =‘dest_B23 ;
    B_adress [3][1] =‘dest_B31 ;
    B_adress [3][2] =‘dest_B32 ;
    B_adress [3][3] =‘dest_B33 ;


      #2000000
      reset = 1'b0;


    for (i=1;i<=3;i=i+1)begin
      for (j=1;j<=3;j=j+1)begin
        for (k=1;k<=3;k=k+1)begin
          #200000
  // ***************Hop count from A to B***************
          if ((A_adress [i][j][2:0])>(B_adress [j][k][2:0]))
          Total_hops= Total_hops+ ((A_adress [i][j][2:0])-(B_adress [j][k][2:0]));
          else
          Total_hops= Total_hops+ ((B_adress [j][k][2:0])-(A_adress [i][j][2:0]));

          if ((A_adress [i][j][5:3])>(B_adress [j][k][5:3]))
          Total_hops= Total_hops+ ((A_adress [i][j][5:3])-(B_adress [j][k][5:3]));
          else
          Total_hops= Total_hops+ ((B_adress [j][k][5:3])-(A_adress [i][j][5:3]));

          if ((A_adress [i][j][8:6])>(B_adress [j][k][8:6]))
          Total_hops= Total_hops+ ((A_adress [i][j][8:6])-(B_adress [j][k][8:6]));
          else
          Total_hops= Total_hops+ ((B_adress [j][k][8:6])-(A_adress [i][j][8:6]));

  // ***************Hop count from B to R***************
          if ((B_adress [i][j][2:0])>(R_adress [k][j][2:0]))
          Total_hops= Total_hops+ ((B_adress [i][j][2:0])-(R_adress [k][j][2:0]));
          else
          Total_hops= Total_hops+ ((R_adress [k][j][2:0])-(B_adress [i][j][2:0]));

          if ((B_adress [i][j][5:3])>(R_adress [k][j][5:3]))
          Total_hops= Total_hops+ ((B_adress [i][j][5:3])-(R_adress [k][j][5:3]));
          else
          Total_hops= Total_hops+ ((R_adress [k][j][5:3])-(B_adress [i][j][5:3]));

          if ((B_adress [i][j][8:6])>(R_adress [k][j][8:6]))
          Total_hops= Total_hops+ ((B_adress [i][j][8:6])-(R_adress [k][j][8:6]));
          else
          Total_hops= Total_hops+ ((R_adress [k][j][8:6])-(B_adress [i][j][8:6]));
        end
      end
    end

      #20000000
finish;

   end // initial begin
endmodule //   TOP_test
```