UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Design, Implementation and Analysis of Router Architectures and Network Topologies for FPGA-Based Multicore Systems

Author:

Boldbaatar Juvaa

Advisor:

José Flich Cardo

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Engineering

September 2015

Abstract

Design, Implementation and Analysis of Router Architectures and Network Topologies for FPGA-Based Multicore Systems

by

Boldbaatar Juvaa

Master of Science in Computer Engineering

Universitat Politècnica de València

This work involves the design, implementation and analysis of different 2D mesh-based network topologies for the Partitioned-Enabled Architecture for Kilocore Processors (PEAK) architecture. PEAK is a fully scalable architecture, implemented on an FPGA device, which uses advanced partitioning methods and coherence protocols to efficiently handle many core systems. In this work we implement several network topologies for the PEAK architecture and their associated routers.

Nowadays, manycore systems are rapidly evolving. In these systems, the network on-chip (NoC) is one of the defining components which influences performance. The NoC design relies on several structures and functions including the topology used and how data is routed on it. PEAK currently uses a 2D mesh topology with straightforward XY and LBDR routing approaches. It is obvious that if more efficient topologies are implemented, system performance will increase. Therefore several topologies and their satisfying router architectures need to be implemented and analyzed to evaluate their impact on system performance. Design, simulation, synthesis and implementation processes in this work are performed in Verilog by using Xilinx Vivado Suite software. For the validation in FPGA, an advanced multi-FPGA Prodesign board is targeted. Overheads in terms of FPGA resources are provided, together with initial performance analysis. The evaluation leads clustered topologies to become a good selection when trading performance and overheads for PEAK.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Dr J.Flich for the limitless support, guidance and advises that he has given to me.

I want to thank to rest of the GAP PEAK team members and lab colleagues at UPV for all of their kindness, help and support. Especially, many thanks to Jose Martinez and Rafael.

Special thanks to Carlos Calafate who was always available for me when I need assistance.

I would like to thank my wife who always encourages me and helps me, and my beautiful two daughters, for their patience while I completed this thesis. Finally, I want to thank to my parents for supported and motivated me.

Chapter 1

Introduction

Performance of traditional single core systems is limited by the clock frequency. Thermal problems like heating and power dissipation (the power wall problem) prevent engineers from increasing the clock frequency. At the same time, the increasing capacity in the number of transistors in emerging chip technologies, which follows the Moore's law [**1**], allows to benefit from the concept of putting multiple cores (but simpler ones) in one chip. Many slower cores show a better performance/power relationship than a monolithic single high-performant core. Although putting more cores in a single die is possible, it needs to be done taking into account new challenges so to achieve an efficient architecture. Several challenges are: (1) to keep consistency between all cores, (2) to achieve reliable interconnection between cores, and (3) to take full benefit from all cores when running applications. If technology continues to shrink, future systems will use more and more cores, thus demonstrating we effectively are in the manycore era.

At the time when microprocessors were composed of very few units building a single core, interconnection inside the chip was not a major concern and, thus, an issue worth not to consider. Mainly, a bus or several buses were used to provide the required connectivity between units inside the chip. When multicores emerge and more cores are added, a common bus is not longer a satisfying communication method. The reason is that tens of cores interconnected by a single bus easily lead the bus to become the bottleneck. Thus, increasing the number of cores in manycore chips requires more than a bus structure. That requirement pushes the Network on Chip (NoC) concept to become one of the most important issues in chip design. The NoC concept and its technology covers the fully functioning communication infrastructure for the chip with its own characterized concepts and techniques such as topologies, routing algorithms and switching techniques. The NoC research field is still growing as there is a need to develop more and more efficient and adapted networks.

This thesis deals with the implementation and analysis of NoCs for manycore systems. Well-known topologies are analyzed and adapted to an emulated manycore system, leading to new router models and routing algorithms, and ultimately leading to the unavoidable trade off between performance and cost. This thesis develops on top of the PEAK architecture, which is briefly introduced next.

1.1 Partitioned-Enabled Architecture for Kilocore Processors

The never satisfying demand of high performance computing in many industrial sectors such as space, avionics, car industry and many different research fields, requires highly efficient and powerful solutions. Partitioned-Enabled Architecture for Kilocore Processors (PEAK) has been recently proposed at UPV to address manycore architectures dealing with the demand of high performance computing by implementing a novel approach for NoCs.

PEAK is a manycore architecture designed for general purpose computing. It follows a tile-based approach which makes design easier and allows greater flexibility. The architecture has the ability to put some of its resources to sleep in order to save power and wake them up to get back on work when needed. That ability makes PEAK a candidate architecture addressing capacity computing where the target is to run as many small applications as possible. In order to achieve this, it enables partitioning methods for the NoC and the cache hierarchy. The current NoC in PEAK implements a 2D mesh topology. This topology enables connectivity between all the cores and has a low overhead impact when implemented. However, more advanced topologies may be designed for PEAK. This is the target in this thesis.

PEAK has several major components and significant properties. One important component is its partitioning capability which enables scalability of the architecture for efficiency and performance. The partitioning process reserves resources as groups and assigns them to every task or application. Another component is the coherence protocol. This protocol is responsible to guarantee every data shared between cores is kept coherent among all cores. Especially, manycore architectures have caches to store data which means the same data can be replicated at several places at the same time. The coherence protocol enables the simplified shared memory programming model.

Another key property of PEAK is its exposure to the software layer. Indeed, PEAK defines a lot of configurable registers and a protocol to access them. By doing this, the PEAK architecture can be fully exploited by an appropriate resource manager. PEAK has been coded in a hardware description language (HDL) and ported to a multi-FPGA prototyping system. This means the manycore system is emulated and not simulated. Anyway, the tools used to code the architecture allow behavioral simulations to be performed before going to emulation. In this thesis we take advantage of that previous step in order to analyze our implementations.

1.2 Structure of the Thesis

The document is structured as follows, after this introduction to PEAK architecture and motivation of this thesis, Chapter 2 starts with brief descriptions of some fundamental concepts of NoCs. This allows to a better understanding of the contributions of this thesis. Then, it follows the literature overview including four well-known examples of manycore architectures. Chapter 3 describes PEAK in greater detail as well the tools and programming models used in this thesis. Then, in Chapter 4 we introduce and describe the different topologies analyzed and implemented, together with their associated routing algorithms. We show results also in Chapter 4. Finally, we conclude in Chapter 5 with some conclusions and future work.

Chapter 2

Interconnection Network Background and Manycore Examples

In this chapter we provide an introduction to basic concepts of interconnection networks, focusing on NoCs. Then, we describe some exemplary cases of real manycore systems, focusing on the interconnect they implement. We will focus on real products from Kalray, Intel and Tilera. We will also include a recent research project which defines an architecture similar to PEAK.

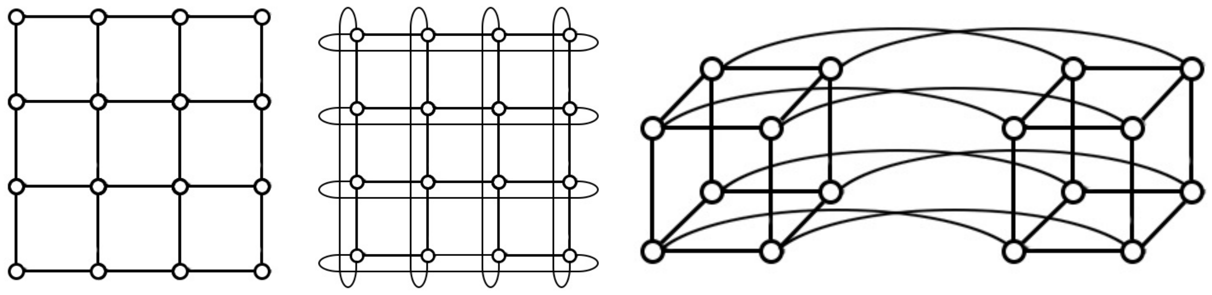
2.1 Some Fundamental Concepts of NoCs

Network on Chip defines how modules such as cores, registers and caches are connected and managed to transfer data over the chip. As technology advances, rapid growth of multiprocessor designs become a reality, and thus, demanding more sophisticated networks to achieve better performance and productivity. That demand makes NoCs a rapid developing field. Since NoC is a network, it has a topology, routing methods, flow control mechanisms and so on. In this section we introduce some basics of NoCs paying more attention to the topics that are more related to our work (namely topology and routing).

Topologies

Topology is a way to define which components or units are physically connected to each other. Before the NoC concept emerged many topologies were proposed for massively parallel processors. Many of those topologies have been proposed for the NoC environment, although few of them are suitable. Roughly speaking, topologies can be divided in two categories: direct and indirect topologies (hybrid systems mixing both also exist).

In direct topologies each node has a router to connect to others. The node, indeed, has embedded the router and has a direct and private link to reach the router. Routers, then connect between them providing shared paths between nodes. The term direct refers to the



(a) 4-ary 2-cube mesh (b) 4-ary 2-cube torus (c) 2-ary 4-cube hypercube

Figure 2.1: Three versions of k-ary n-cube topologies

fact that the node is directly connected to the router (they are tightly coupled). The most famous direct topology is the k-ary n-cube, where all the routers (and nodes) are structured in an n-dimensional field and each dimension has k routers. Therefore, the number of routers is k^n. Each pair of neighbor routers in each dimension is connected by an exclusive link. Each link is usually bidirectional. Figure 2.1 shows three versions of k-ary n-cubes, such as the mesh, torus, and hypercube [**2**]. In the torus network routers at the end of each dimension are connected with an added link, thus forming rings. In mesh variation those links do not exist. In hypercubes the number of routers per dimension is kept to 2. Therefore, dimensionality grows rapidly as network size is increased.

An important aspect of those topologies for NoCs is the fact that when implemented on top of a die the links showing larger lengths tend to introduce more complexity in the design and lead to poorer results. This means a simpler topology like the mesh version may perform better once implemented than the torus or hypercube versions (although those topologies exhibit better throughput and message latency numbers in principle). Another important aspect in those topologies is the bristling factor (b). It defines the number of nodes attached to a router. Indeed, in the previous examples, b was assumed to have a value of one. When the b value increases, the relative cost of the network (when compared with the overall system cost) decreases. However, performance of the network may suffer as the network has lower capacity. However, this only points the need to evaluate the trade off between performance and cost of such alternative designs. Figure 2.2 shows the case of a 4-ary 2-cube mesh with b=4. Thus, 64 nodes are connected through a network of 16 routers. The number of nodes in a k-ary n-cube with b factor is $k^n \times b$. This alternative design is appealing for NoCs and is being exploited by some current products (as we will see). We will exploit also this aspect in our designs.

In addition, in the literature we can find variations of k-ary n-cubes where extra links are added in order to provide more connectivity and performance. We will explore such topologies in Chapter 4.

In indirect topologies nodes are connected to routers but not all the routers have nodes attached to them. Therefore, routers and nodes are decoupled and used separately. The term indirect refers to the fact that nodes need to use an external router to communicate.

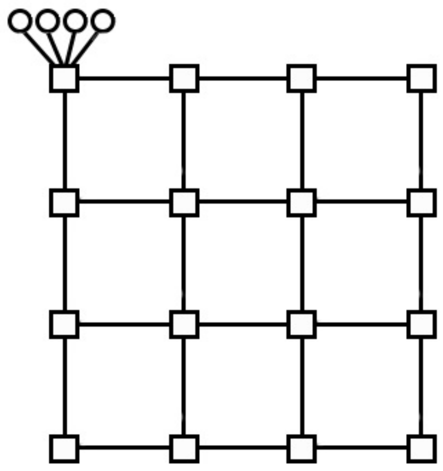
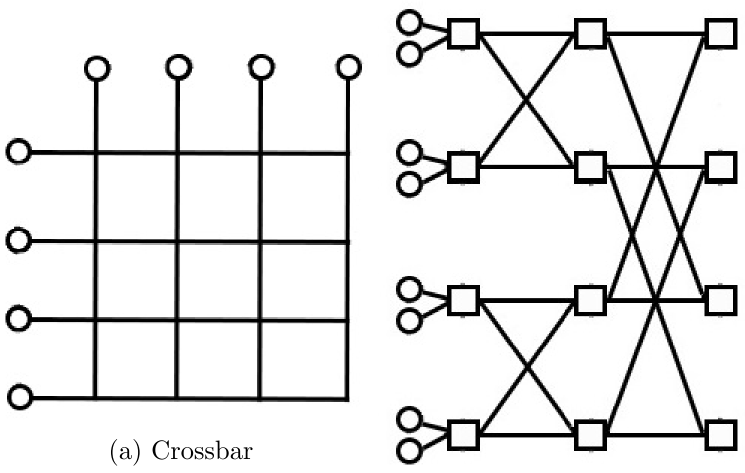


Figure 2.2: 4-ary 2-cube mesh with b=4. Boxes represent routers and circles represent nodes.



(a) Crossbar

(b) Fat Tree

Figure 2.3: Examples of indirect topologies. Boxes represent routers and circles represent nodes.

Two clear examples of such topologies are the crossbar and the fat-tree network (see Figure 2.3). A crossbar allows N nodes to communicate with N nodes allowing all possible direct connections. Indeed, is built as a matrix of switches where each switch communicates in a pair of nodes. This full connectivity allows maximum performance but does not scale with system size. Thus, a tradeoff topology is the multistage network (fat tree is a variant) where smaller crossbar routers are used and more than one is needed to allow two nodes to communicate. Indirect topologies are rarely considered for NoCs as they are too complex when mapped on a die.

Comparing direct topologies with indirect topologies, indirect ones show higher bisection

Table 2.1: Bisection of direct and indirect topologies for 64 nodes, measured as the number of bidirectional links crossing the bisection.

	2D mesh	**2D torus**	**FatTree**	**Crossbar**
Bisection Bandwidth	8	16	32	32

bandwidth. Bisection bandwidth is the bandwidth achieved at the smallest section that splits the network in two equal halves. In fat-trees the network achieves full bisection bandwidth in the sense the network does not behave as a bottleneck. Direct topologies are more appealing for NoCs, but with lower dimensionality (i.e. n=2). Indeed, in direct topologies nodes are spread over the chip and closely located to the attached router. Links are also homogeneous and have the same length. Contrary to this, in indirect topologies nodes are located on the periphery of the NoC and thus require longer links. Table 2.1 shows bisection (measured in number of bidirectional links crossing the bisection) of several direct and indirect topologies.

Switching

Once the network is structured by its topology, switching plays an important role as it instructs how data travels from source to destination. Switching techniques are implemented by using resources in the router in a coordinated and structured way. Although many designs exist, two became most widely adopted in interconnection networks for high-performance systems. Those have been also adopted for NoCs. We can mainly identify *wormhole* [**3**] and *virtual cut-through* [**3**] switching techniques.

Wormhole

Data or messages transferring through NoCs are divided into packets. Packets are then divided into flits - the smallest flow-controllable unit of data. Because wormhole (WH) switching takes into account flits, small sized buffers that can store only a few flits are implemented at routers thus saving resources. But nodes can still send very large messages. Thus, wormhole switching reserves the whole path for the message until its tail passes through. Only the header flit contains destination information and the remaining flits just follow their header so the flow is completely sequential. The routers must forward flits when the downstream router has free space to store a flit and even before the complete message arrives to the router. Because of path reservation, large contentions may occur in the network. Indeed, this is the main drawback of wormhole switching. To reduce such contention, virtual channels can be used.

Virtual Cut-Through

In Virtual Cut-Through (VCT) switching, flow control is applied at packet granularity instead of flit granularity. Thus, routers need to have large buffers to store complete packets. Flits, however, are forwarded downstream without waiting for the whole packet being buffered in the current router. Because of the use of large buffers, contention of the network is smaller, as compared to WH, thus exhibiting better performance.

Flow Control

Once the switching techniques are set in the applied topology, the flow control mechanism manages resources of every connected two devices (routers or nodes) in order to make successful data transmission and avoiding buffer overruns. Its main purpose is to avoid data loss caused by buffer overflow and to use resources as efficient as possible. We can, basically, identify two strategies: *stop&go* [**4**] and *credits* [**5**].

Stop&Go

Every buffer has two thresholds: one for *stop* and another for *go*. An status bit is used at the upstream router to keep the status of the flow. If the stop threshold is reached the router will send a stop signal to the upstream router to stop transmission. When the buffer reaches to go threshold it will send the go signal to start transmission again. An important aspect is when the receiver node sends the stop signal, as the sender router may still send flits until it receives the stop signal. Indeed, flits will be droped if the receiver router can not provide enough storage. To avoid this situation, stop and go thresholds need to be set based on the round trip latency of the link.

Credits

With credits the sender router receives credits from the downstream router. Whenever a credit is freed, the router sends back the credit to the upstream router. So, the number of credits is equal to the number of flits that the sender router can send to or the number of flits that the receiver router can store. Once the sender sends a flit it decrements its credit counter by one. The counter is incremented when a credit comes from the downstream router.

Routing

Finally, we have network structures connected by the topology. Depending on which topology is used nodes have multiple paths to send data. This leads to the need to define routing. Routing makes decisions that result from an algorithm to select a path (as short as possible) to send data from a source to a destination. Usually, the routing method completely depends on the topology that is used.

Even though routing can be flexible in deciding the paths, there are some paths that are not available or suitable in order to avoid the following difficult situations. The *livelock* problem is a sort of a milestone in networking, when packets always move but never reach their destinations. Another situation is named *deadlock*, and occurs when several packets wait for each other holding resources forever. The routing algorithm can prevent both livelock and deadlock situations. Routing algorithms can be also categorized. For example, if the routing algorithm always selects the same path for each source-destination pair, the algorithm is called deterministic. One of the well known deterministic routing algorithms for k-ary n-cubes is Dimension-order routing (DOR) which is safe from deadlock and livelock situations. On the counterpart there are adaptive routing algorithms which may provide more than one different path for each source-destination pair.

DOR is used in k-ary n-cube topologies and is based on using links in an ordered way, thus avoiding deadlock conditions. Indeed, each dimension is crossed in an ordered way and following shortest paths. For example, in a 4-ary 2-cube mesh network, dimension X is crossed and once the message reaches the column of the destination node, the message is sent through the Y dimension, following minimal path. Notice that DOR can not be used in torus networks without other provisions to avoid deadlock. This is because physical rings exist in torus networks along the same dimension, and combination of messages could lead also to deadlocks. Virtual channels or more advanced flow control strategies (like the bubble flow control [**6**]) can be used. However, this means more resources and, thus, being more complex topologies at the end.

In this thesis we will use DOR routing and will propose variations when applied to alternative topologies (derived also from the mesh topology). We will not assume adaptive routing algorithm as they may introduce out of order delivery, which is not compatible with the coherence protocols implemented in PEAK.

Next we describe some real products and projects which use NoCs. This will allow us to put our work in perspective.

2.2 TSAR Project

TSAR (Tera Scale Architecture) [**7**] is a European funded project which aims to develop a coherent, scalable, manycore architecture reaching up to 4096 cores. TSAR uses a MIPS 32-bit processor and the DSPIN (Distributed, Scalable, Predictable, Interconnect Network) [**7**] NoC. The DSPIN network has a 5-port router and is based on regular 2D mesh topology with wormhole switching.

The main considered issue in TSAR is scalability, as this architecture is intended to integrate up to 4096 cores (even if the first prototype will contain only 128 cores). The second issue is power consumption, and all other technical solutions implemented in this project are driven by these two important issues.

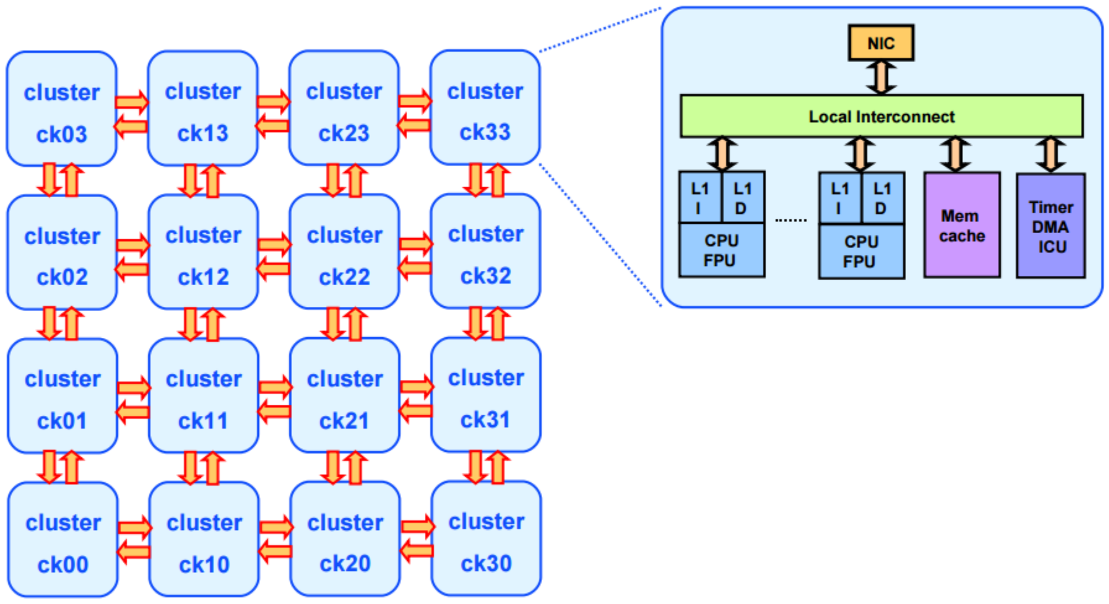


Figure 2.4: TSAR architecture overview (Image source: [**8**])

Architecture Overview

TSAR uses a simple 32-bit processor core with a single instruction issue RISC without any of the followings: superscalar features, out of order execution, branch prediction, speculative execution. TSAR is composed as a clustered architecture. Every cluster consists of up to 4 cores each with its own L1 cache.

Because of the scalability target, TSAR implements a logically shared but physically distributed memory based on NUMA (Non Uniform Memory Access) architecture where every processor can access every memory bank, but the access time, and the power consumption depends on the distance between the processor and the memory bank.

The TSAR project also implements the DHCCP protocol (Distributed Hybrid Cache Coherence Protocol) to maintain consistency for data between main memory and caches. The TSAR architecture is depicted in Figure 2.4.

Network On-Chip Approach

Generally two categories of interconnection networks are used in TSAR system: local interconnection network between units inside each cluster, and global interconnection network responsible for the communication between clusters.

Depending on the different purposes TSAR has three networks implemented: data read and write (Dnetwork); transfer coherence protocol information (Cnetwork); and external memory access in case of a miss on the memory cache (Xnetwork). The DSPIN network on chip (developed by the LIP6 laboratory) implements the Dnetwork and the Cnetwork. It implements a 2D mesh topology, and provides scalable bandwidth to the shared memory TSAR architecture [**7**]. It provides two fully separated virtual channels for the direct traffic and for the coherence traffic. For the D and C network the local interconnection uses a crossbar or a ring topology and the global interconnection is structured by a 2D mesh topology.

The Xnetwork between the memory cache and the external memory controller is implemented by a physically separated network using a 3D mesh topology. As we will see, the TSAR architecture is very similar to the current PEAK architecture. Indeed, the 2D mesh and DOR algorithm are used in both architectures. However, bristling with factor 4 is used in TSAR. We will adapt in our work the bristling factor to PEAK in this thesis.

2.3 Kalray

Kalray is a leading industrial player in chip industry which commercializes the Multi-Purpose Processor Array (MPPA) manycore technology. Kalray started in 2008 in Paris (France). Their first ever produced mainstream product is MPPA 256 integrated manycore processor composed of 16 clusters which have 16 cores on each cluster and connected by full duplex links using 2D torus topology. The core used in MPPA is a 32-bit VLIW (very long instruction word) processor.

With advancements of their MPPA processor architecture design, Kalray produces accelerators or coprocessors. The Kalray accelerator consists of 4 MPPA processors with 2 DDR3 memories for each processor all connected in the same daughterboard.

Architecture Overview

The MPPA processor chip integrates 16 compute clusters and 4 Input/Output subsystem controllers for managing Input/Output devices. Each Input/Output subsystem consists of SMP quadcore with a shared D-cache memory and DDR controller for accessing up to 64GB of external DDR3-1600 memory. The 16 compute clusters and the 4 Input/Output subsystems are connected by two NoCs with bidirectional links, one for data transfers (D-NoC), and the other for control information transfers (C-NoC). In the MPPA architecture the cluster is a basic unit. Each cluster contains 16 cores for processing purposes, one management core, shared memory and direct memory access unit. The total size of the shared memory is 2MB but it is organized as a 16 separate bank whose size is 128KB each. Figure 2.5 shows Kalray's MPPA architecture.

The MPPA processor is a 32 VLIW core with 7 stage pipeline. The core supports idle modes and wake on interrupt for energy savings. The core includes the main units such as arithmetic and logic unit, multiply and floating point units, load/store units, branch and controlling unit and registers.

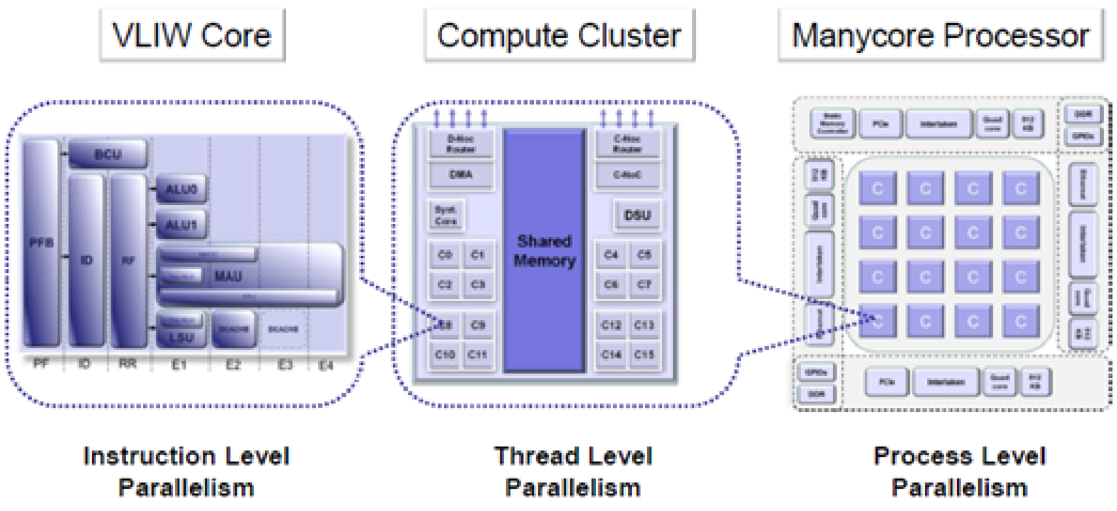


Figure 2.5: Kalray's MPPA architecture overview (Image source: [**9**])

NoCs implement a 2D torus topology. The network of MPPA 256 uses credit-based flow control and wormhole switching where the header flit contains all the routing information and other flits are all payloads. MPPA 256 has fully configurable Quality of Service (Qos) which provides bandwidth allocation in network environment. Also QoS plays big role in avoiding deadlock, by ensuring buffers never get filled at routers.

Processors on the accelerator board are connected by a network named NocX which refers to NoC extended version. NocX provides bandwidth of 40 Gb/s connection for multiple MPPA processors on the same board. Also, NocX allows to connect the MPPA processor or Kalray accelerator to external FPGA boards.

2.4 Intel Phi

Intel Xeon Phi is the first product based on Intel's MIC (Many Integrated Core) architecture [**10**]MIC can be seen as a symmetric multiprocessor architecture.

According to top 500 supercomputer list of June 2015 [**11**] Intel Xeon Phi was used in Tianhe-2 (MilkyWay-2) [**12**], the world's fastest supercomputer.

Intel has launched several production models for coprocessor, each with Xeon Phi processor and PCI bus to connect the host from a computer. The main component is the processor which consists of up to 61 Xeon Phi cores connected by a bidirectional ring network, up to 8 memory controllers that support GDDR5 and second generation PCI express system interface that supports up to 256-byte packet transmission. GDDR5 is a special type of DRAM that supports higher bandwidth. Every core includes 512-bit wide Vector Processor Unit (VPU) with Extended Math Unit (EMU).

Coprocessors have different system specifications varying on models. But generally its

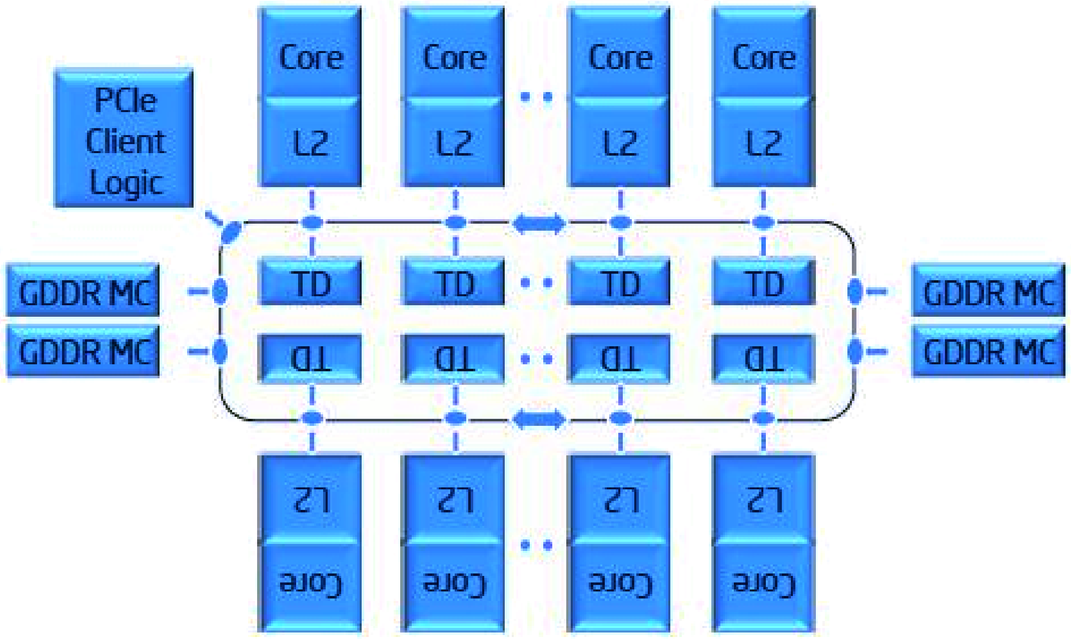


Figure 2.6: Intel Phi architecture overview (Image source: [**13**])

components are the processor, memory, PCI express interface. Depending on the cooling method coprocessors can be categorized into 2 categories named passive cooling and active cooling. Passive cooling coprocessor is dedicated to use in a supercomputing cluster or data center and active cooling coprocessor is suitable for desktop-like environments. Nowadays, Intel is offering three different coprocessor models: 3100 series has 57 cores, 5100 series has 60 cores and 7100 series has 61 cores.

Architecture Overview

Intel Xeon Phi processor is composed with computational section and input-output section. High speed bidirectional on-chip network with ring topology supports connection to all the cores, memory controllers and PCI express system input-output logics. The On-chip network implements shortest path algorithm to route in a ring. Each core has 512 kB second level cache. Phi has coherent management which keeps data consistent over the caches, named Tag Directory (TD) distributed mechanism. Figure 2.6 shows the Intel Phi processor architecture design.

Actually, the ring is a simple topology, where the cores connect with one another through one or multiple hops in both directions [**14**]. Intel Phi has three pairs of independent ring networks traveling in opposite directions (bidirectional). First pair, we may call it data ring is 64 bytes wide and dedicated to transfer data between the cores. Another pair is the address ring which is responsible for carrying addresses and read/write commands. The last pair of rings carries flow control and coherence messages, named acknowledgement ring. The ring topology (1-ary k-cube torus) has different properties: (1) its average number of hops is N/4 where N is the number of cores, (2), as the number of cores in the ring increases, the number of hops increases and so message's latency, (3) bottleneck can easily appear as the load increases in the ring, and (4) any link failure will lead to whole network to fail.

The ring uses shortest path algorithm to route the message. Once a message is on the ring there is no queuing or any turns so the message just needs to continue deterministically until it reaches its destination.

2.5 Tilera

Tilera, a tile-based processor architecture firstly launched in 2007 is nowadays owned by EZChip semiconductor company. First product was named Tile64 [**15**] which has 64 cores. Now they offer several products from processor TileGx9 with 9 cores to TileGx72 which contains 72 cores. Tilera is focused on high performance computing, digital media and cloud computing. Tilera equipped with NoC named Imesh which provides 5 way communication for each tile with static network support. Its coherent cache subsystem provides memory consistency and multicore scalability technology enables to use one or more cores as a processing group.

Architecture Overview

Here we describe the most well-known architecture from Tilera: its Tile64 architecture with 64 cores. Homogeneous tile (called full featured processor core) is composed of processor engine, cache engine and switch engine. One tile is capable of running an operating system.

The processor engine consists of a fetch unit, instruction decoder unit, issue logic unit, and general purpose and special purpose registers. Processor is 32-bit VLIW processor architecture with 3 execution pipelines to perform following operations: arithmetic and logical, bit manipulations, select operations, register reads/writes, memory maintenance operations, flow control and load/store.

The cache engine is responsible for caching of the instructions and data and also provides memory address translation. Cache subsystem consists of simple organization of dynamically distributed cache such as shared L2 cache. But depending on the Tilera processor model every cache organization is different. For example, model TILE64 has 8KB level 1 cache, TILE64pro provides 16KB level 1 cache. The switch engine is responsible for network communication to each tile. Switch engine contains multiple dynamic networks and a single static network. If tile is located just adjacent to the I/O device, switch engine directly connects to I/O device. Figure 2.7 shows Tilera many core architecture.

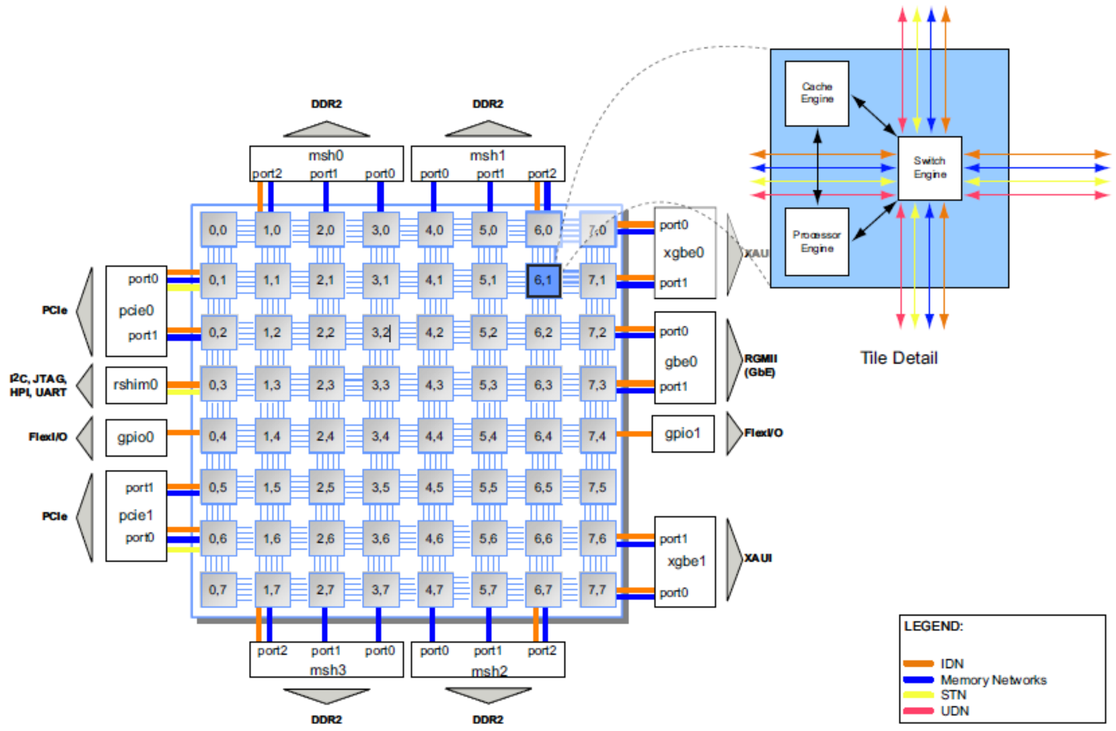


Figure 2.7: Tilera tile based architecture overview (Image source: [**15**])

Network On-Chip

Tilera's NoC, the Imesh is composed of multiple two-dimensional mesh networks to provide communication between tiles and I/O devices. Those networks dedicated to flow traffics like Memory System traffic, Cache System traffic, I/O traffic and software-based messages. The Imesh networks can be classified into two groups: Memory Networks and Messaging Networks. Memory Networks are responsible for handle all memory traffic such as cache misses, DDR2 requests, and so on. And Messaging Networks allow software to have control of the network and manually send messages between tiles and I/O devices. In additionally, Imesh has support of fully configurable static network. Instead of writing destination in flit header, routing is performed by value of special purpose register in every intermediate routers when static network is used.

The Imesh network implements a dimension-ordered routing policy. It uses XY routing algorithm but the TilePro (extended version of Tile64) family of processors allow each network to be configured to either route X dimension first or Y dimension first.

Main unit of data that transmits over the Imesh networks, is called "packets". Packets are divided into multiple N bit "flits", where N is the width of the network. Each packet contains a header flit designating the destination of the packet and the size of the packet, and a payload of data flits. Flow control between neighboring switch points is implemented via a credit based scheme. Each switch point has an input buffer that may hold three flits [**16**].

2.6 Conclusion

Table 2.2 shows the comparisons of the above mentioned architectures. Note that comparison is made with consideration only to NoC related main components including routing algorithm, network topologies, switching techniques and flow control mechanisms. The following conclusions are highlighted from this comparison.

In all the cases k-ary n-cube topologies are used, instead of using indirect topologies. Also, low dimensionality (2D or 1D) is used. This is due to the fact those topologies are floorplan aware and thus becomes easier to implement such topologies. Wormhole is also preferred, so buffer needs are low. In Intel Phi the message size equals the flit size thus reverting in VCT switching. For routing DOR is used in all the cases, demonstrating its simplicity when implemented.

Table 2.2: Comparison of architectures based on NoC features

Architecture	**Topology**	**Switching**	**Flow Control**	**Routing algorithm**
Intel Phi	Ring	VCT	N/A	Shortest Distance
Tsar project	2D Mesh	Wormhole	FIFO	XY routing
Tilera	2D Mesh	Wormhole	N/A	XY routing
Kalray	2D Torus	Wormhole	Credit based	N/A

In the next chapter we describe the PEAK architecture, where we develop our designs, as well as the methodology and tools used in this work.

Chapter 3

Methodology

In this chapter we present the methodology followed to accomplish the goals of this thesis. The chapter begins with detailed description of the PEAK architecture, which is the underline architecture we improve. Then, the chapter provides descriptive information about the hardware language used when coding several topologies and the software framework used to develop, simulate, synthesize and implement the modules.

3.1 PEAK Architecture

PEAK is a manycore architecture based on the tile concept. Tiles are designed, replicated, and connected, building the complete system. Each tile is composed mainly by eight modules: a core, one private 32KB cache for data (L1), 128KB shared cache (L2), one network interface (NI), three routers (R0, R1, R2) and a tile register (TR). Figure 3.1 depicts the PEAK architecture.

The MIPS architecture [**17**] is used for the core, which follows a pipelined design with in-order execution of instructions. The core has a private L1 cache for data (internally has one private cache for instructions). The NI module provides connectivity between modules within the tile. Basically, the NI is a buffered crossbar. The TR module is responsible for storing important configuration data of the tile and its components (described next). The L2 cache of the system is compounded of L2 banks on each tile, thus being a unified large L2 cache. An invalidation-based coherence protocol is implemented at the L2 cache level. The three routers provide connectivity between tiles (described next). Finally, main memory is accessed through the memory controller (MC) unit, which can be attached to any tile. Currently, one single MC is attached to tile 0.

Tile Register (TR)

The TR module is the main configuration component in PEAK. TR implements a set of registers to completely configure the behavior of the tile. The registers are divided in three

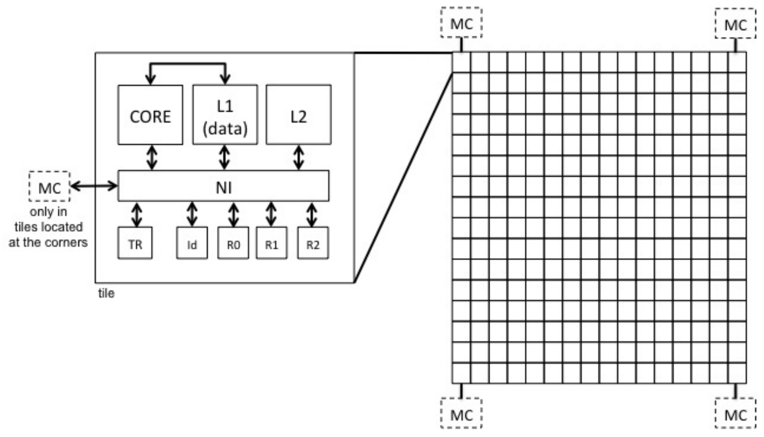


Figure 3.1: PEAK general architecture overview.

different categories: configuration registers, notification registers and statistics registers. By using configuration registers, PEAK can define the number of tiles of the system, set the core frequency, put the core to sleep, configure alternative coherence protocols, configure the routing algorithm in the router, enable broadcast support and gather network support, and even configure the memory controller attached to the tile. Notification registers are used by the applications running on top of PEAK. Notification registers allow applications to communicate each other and to synchronize. Statistic registers provide core, cache and network statistics. Cache misses and cache accesses can be tracked in real time. Tile registers are memory mapped to allow maximum flexibility as any core with privileges can access any TR from any tile in the system.

The Core

The PEAK core is a 32-bit pipelined in-order MIPS processor. It provides basic arithmetic and logic instructions. The processor is designed in 5 stages: fetch, decode, execution, memory access and write back. The data path is composed of the instruction cache (32KB, 64-byte blocks, direct mapping), the bank register for read operations (32 32-bit registers), the ALU, one L1D cache interface, one TR access interface, and the bank register for write operations. The control path is composed of a branch target buffer (BTB), the control unit (CU), the branch control unit and the data risks module [**18**]. Branch target buffer is used to cover conditional and unconditional branch instructions. Figure 3.2 shows the architecture of core.

Networks On Chip

Three networks named VN0, VN1 and VN2 are implemented for different purposes. VN0 and VN1 are used to transfer coherence protocol traffic between L1 and L2 modules. The VN2 network is dedicated to data transfer between the cores and TR/MC modules. 2D Mesh

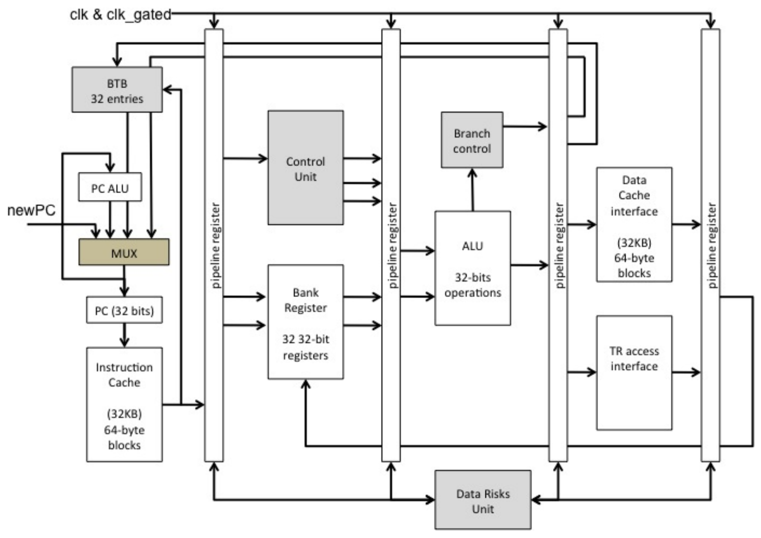


Figure 3.2: Core architecture.

topology is used in PEAK. Stop and Go flow control is implemented in the three networks. The router consists of 5 input ports (West, North, East, South and Local), buffer which can store four flits, routing unit, arbitration unit and the crossbar for output. A one flit which has size of 64 bits contains fields such as source address, destination address and payload. For VN0 and VN1, LBDR method performs routing. VN2 is the simplified router that uses XY routing algorithm. Router or switch architecture is shown in Figure 3.3.

Each input port consists in a pair of modules connected between them: the IBUFFER and ROUTING modules. These modules are in charge of storing incoming flits from the input port and computing the proper output port to forward the flits. The IBUFFER module implements a FIFO queue where flits are stored. The flit at the header is forwarded to the ROUTING module. Stop&Go is implemented in the IBUFFER module.

Each output port consists in a pair of modules, namely SA and XOP modules. The SA module performs switch arbitration and decides which input port is allowed at each cycle to send its flit through the associated output port. The XOP module just perfoms a multiplexing to select the proper input port granted.

All ROUTING modules are connected to all SA modules and XOP modules, with the exception of the input and output ports in the same dimension but in different direction (U turns are not possible). The PEAK router design has implemented 5 ports, namely North, East, West, South and Local port. Therefore, each input port (the ROUTING module) is connected to four output ports (e.g. North input port connected to East, South, West, and Local output ports). Notice that the figure only shows connections to only one output port (for the sake of clarity in the picture). The most important modules for our work are the ROUTING modules and the SA and XOP modules since we will need to add new topologies with more ports and new routing algorithms. Next, we describe these modules.

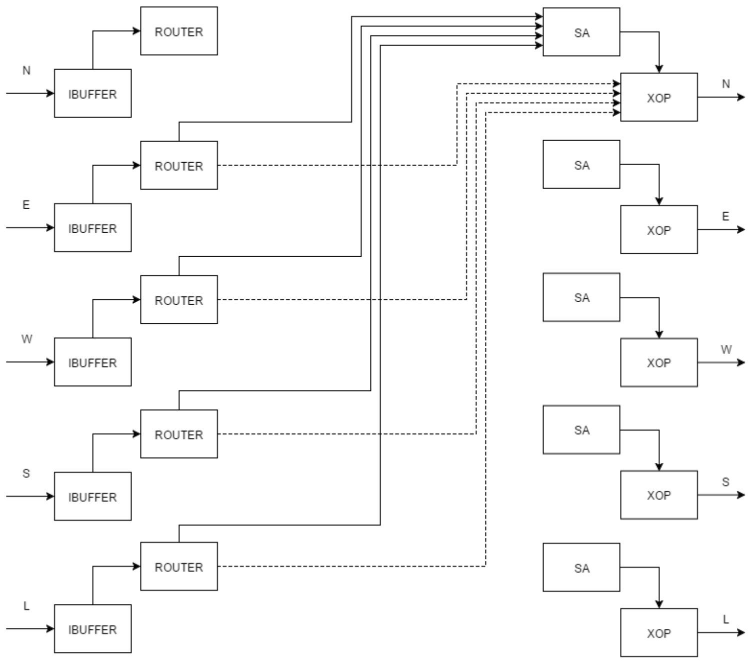


Figure 3.3: PEAK router architecture. Router's main components are the Ibuffer (buffer), Routing, SA (arbiter) and XOP (output).

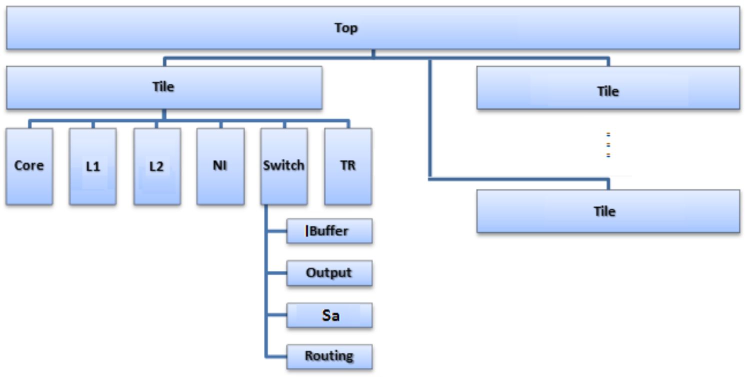


Figure 3.4: PEAK module hierarchy.

3.2 PEAK Modules

As previously commented, PEAK modules are implemented by Verilog language. It follows a hierarchical organization where modules are embedded into other modules. Figure 3.4 shows part of the modules and how they are related to each other. Specifically, the figure only shows the modules modified in this thesis.

The top module defines the whole PEAK system and instantiates all the tile modules of PEAK, providing also connection between tiles. Then, the tile module provides all the mod-

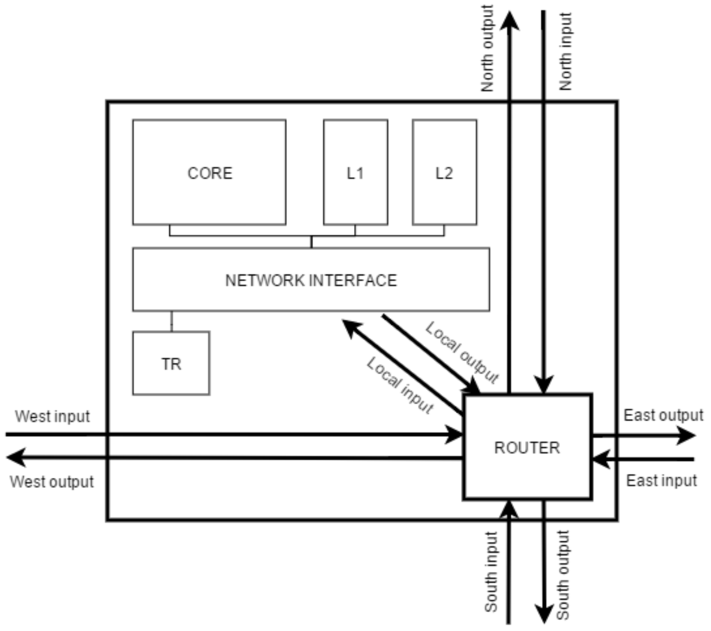


Figure 3.5: Structure of module tile with associated ports.

ules implemented at tile level, mainly the core, L1 cache, L2 cache bank, network interface, the three switches, and the tile register modules.

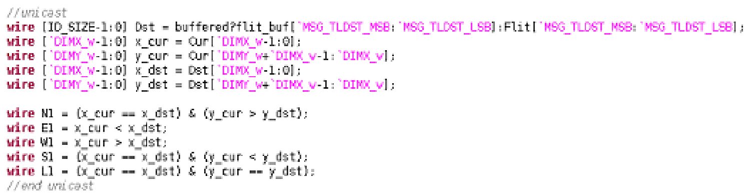
Figure 3.5 provides basic organizational plan of the tile module with all the ports associated to communicate with other tiles. The switch module includes all the modules to build a switch, namely the input buffer, the output crossbar, the routing module and the switch allocator module.

In the next section we provide brief explanation of modules *routing* and *arbiter* as an examples.

Module Routing

This module is in charge of deciding where to send the buffered flits. Depending on the routing algorithm used, the module routing reads destination address of the flit to calculate right next path. The module routing is provided with registers to store previous calculated route decision. Notice this is needed since we are assuming wormhole switching and all flits pass through the routing module. Only the header contains routing information. This module also supports broadcasting possibilities, thus our methods must respect and be compatible with broadcast traffic. Figure 3.6 shows some code part from module routing which is written in Verilog.

First 5 lines of code in figure 3.6 set values of wires named x_cur and y_cur which have the purpose of keeping current location of router and x_dst and y_dst which store destination



```
//unicast
wire [ID_SIZE-1:0] Dst = buffered?flit_buf[`MSG_TLDST_MSB:`MSG_TLDST_LSB]:Flit[`MSG_TLDST_MSB:`MSG_TLDST_LSB];
wire [`DIMX_w-1:0] x_cur = Cur[`DIMX_w-1:0];
wire [`DIMY_w-1:0] y_cur = Cur[`DIMY_w+`DIMX_v-1:`DIMX_v];
wire [`DIMX_w-1:0] x_dst = Dst[`DIMX_w-1:0];
wire [`DIMY_w-1:0] y_dst = Dst[`DIMY_w+`DIMX_v-1:`DIMX_v];

wire N1 = (x_cur == x_dst) & (y_cur > y_dst);
wire E1 = x_cur < x_dst;
wire W1 = x_cur > x_dst;
wire S1 = (x_cur == x_dst) & (y_cur < y_dst);
wire L1 = (x_cur == x_dst) & (y_cur == y_dst);
//end unicast
```

Figure 3.6: Code part shows basic routing logic.

location of buffered flit. These locations are defined by X and Y coordinates of a 2D mesh. We can see first level of route calculations from last 5 lines of codes from figure 3.6. For example we can see that if value of x_cur is less than value of x_dst, flit should go to east port. We can see flow diagram of this module in figure 3.7.

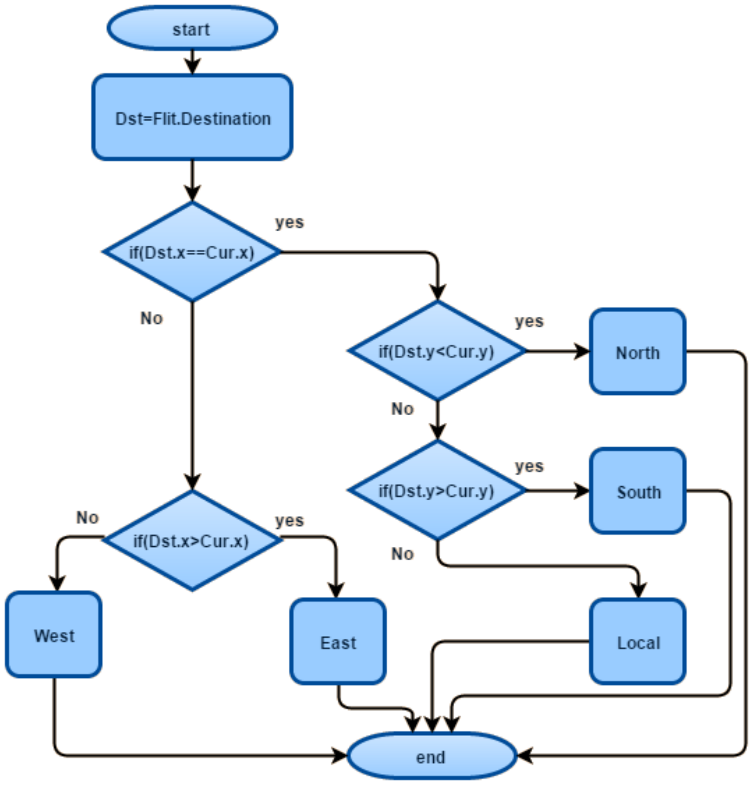
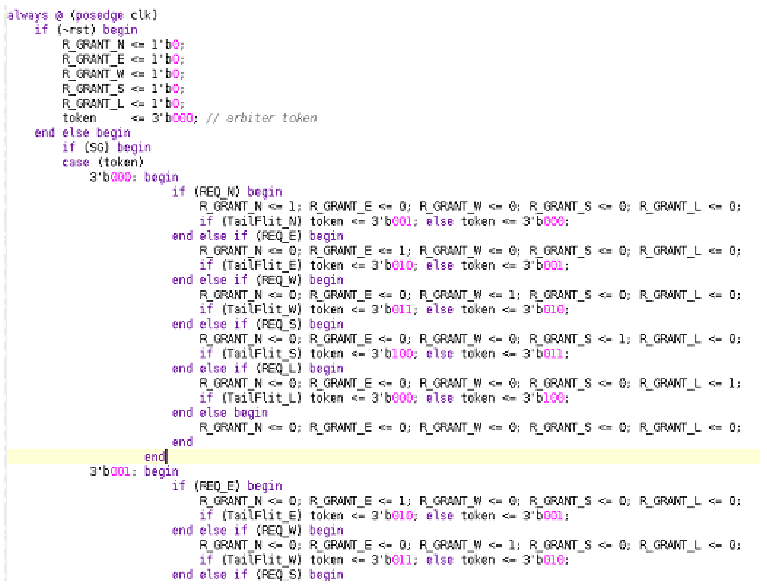


Figure 3.7: Flow diagram of module Routing.

Figure 3.7 shows following logic. If the X coordinate of the destination is the same of the X coordinate of the current switch, then the Y coordinate is inspected. In that case, it gives 3 options to check. First of all, if Y coordinate of current is higher that Y coordinate of destination switch, switch must route the flit to north port. And the second, if Y coordinate of destination is higher that Y coordinate of current, flit should be routed to south port.



```
always @ (posedge clk)
    if (~rst) begin
        R_GRANT_N <= 1'b0;
        R_GRANT_E <= 1'b0;
        R_GRANT_W <= 1'b0;
        R_GRANT_S <= 1'b0;
        R_GRANT_L <= 1'b0;
        token     <= 3'b000; // arbiter token
    end else begin
        if (SG) begin
        case (token)
            3'b000: begin
                        if (REQ_N) begin
                            R_GRANT_N <= 1; R_GRANT_E <= 0; R_GRANT_W <= 0; R_GRANT_S <= 0; R_GRANT_L <= 0;
                            if (TailFlit_N) token <= 3'b001; else token <= 3'b000;
                        end else if (REQ_E) begin
                            R_GRANT_N <= 0; R_GRANT_E <= 1; R_GRANT_W <= 0; R_GRANT_S <= 0; R_GRANT_L <= 0;
                            if (TailFlit_E) token <= 3'b010; else token <= 3'b001;
                        end else if (REQ_W) begin
                            R_GRANT_N <= 0; R_GRANT_E <= 0; R_GRANT_W <= 1; R_GRANT_S <= 0; R_GRANT_L <= 0;
                            if (TailFlit_W) token <= 3'b011; else token <= 3'b010;
                        end else if (REQ_S) begin
                            R_GRANT_N <= 0; R_GRANT_E <= 0; R_GRANT_W <= 0; R_GRANT_S <= 1; R_GRANT_L <= 0;
                            if (TailFlit_S) token <= 3'b100; else token <= 3'b011;
                        end else if (REQ_L) begin
                            R_GRANT_N <= 0; R_GRANT_E <= 0; R_GRANT_W <= 0; R_GRANT_S <= 0; R_GRANT_L <= 1;
                            if (TailFlit_L) token <= 3'b000; else token <= 3'b100;
                        end else begin
                            R_GRANT_N <= 0; R_GRANT_E <= 0; R_GRANT_W <= 0; R_GRANT_S <= 0; R_GRANT_L <= 0;
                        end
                    end
            3'b001: begin
                        if (REQ_E) begin
                            R_GRANT_N <= 0; R_GRANT_E <= 1; R_GRANT_W <= 0; R_GRANT_S <= 0; R_GRANT_L <= 0;
                            if (TailFlit_E) token <= 3'b010; else token <= 3'b001;
                        end else if (REQ_W) begin
                            R_GRANT_N <= 0; R_GRANT_E <= 0; R_GRANT_W <= 1; R_GRANT_S <= 0; R_GRANT_L <= 0;
                            if (TailFlit_W) token <= 3'b011; else token <= 3'b010;
                        end else if (REQ_S) begin
```

Figure 3.8: Code part of arbiter logic.

And third option is the case only both X and Y coordinates coincide between source and destination IDs then the local port is selected for routing.

However, if the X coordinate of the destination is not the same of the X coordinate of the current switch, flit will be routed along X coordinate to get destination. Thus it has to use west port or east port.

Module Arbiter

Module arbiter is one of the main components of the module switch. The main purpose of this module is to grant requested output ports by module routing. Generally arbiters granting mechanism based on arbitration algorithms like serve-longest-queue which prioritizes a longest queues and round robin which treats all requests have a equal priority.

The arbitration uses round robin algorithm. Module arbiter uses register named token to identify last granted output port. Last granted port has low priority in next cycle.

From figure 3.8 we can see that inside of a arbiter it has a switch case to check every value of token register. Also we can see arbiter logic completely from figure 3.9 which shows that flow diagram of module arbiter. Then inside of a each case it checks every possible requests with if else statement. Finally arbiter grants the requested port. After module arbiter grants the requested port of module routing flit can travel through that port by module output.

According to figure 3.9 we can see that how round robin arbiter works. Depending on the token it checks every requests in a different order. Then if request is true it grants that port. If tail_flit is true it keeps token unless it increases token value to accomplish round robin principles by changing the order (no one is always in the first or last. All are equally prioritized). If all requests are false, arbiter will not grant any port.

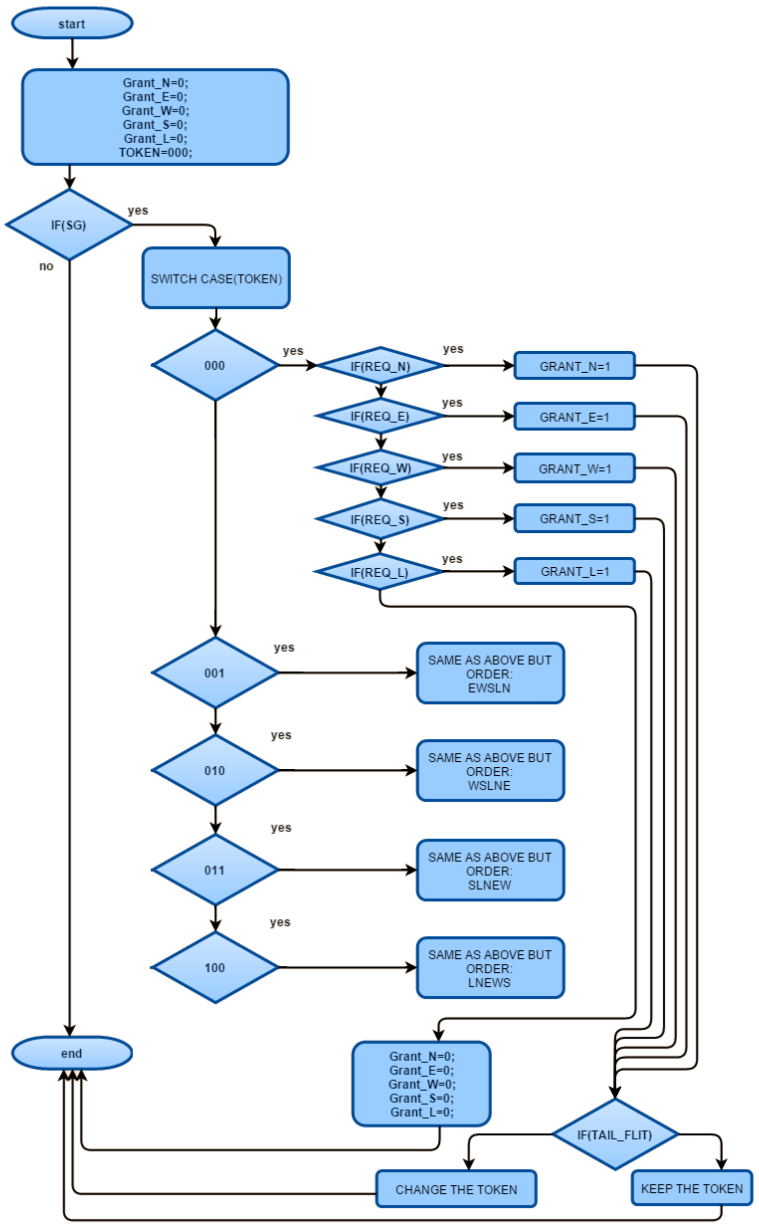
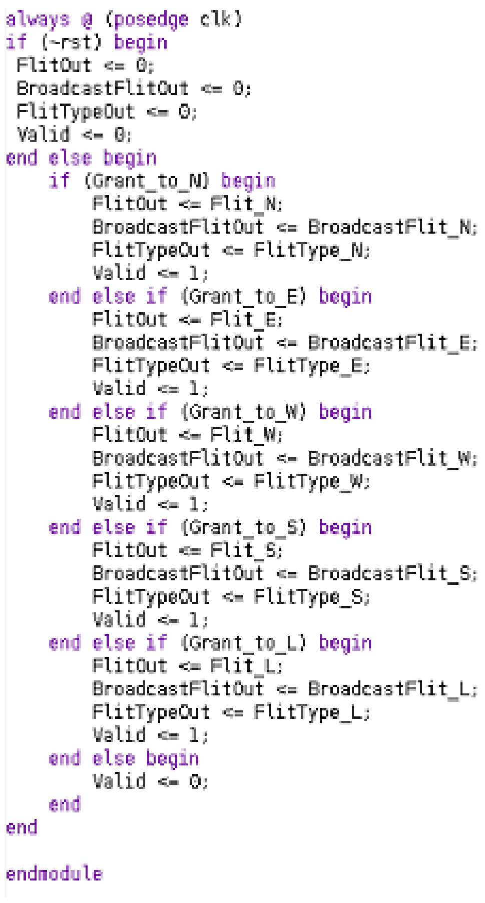


Figure 3.9: Flow diagram of module arbiter.

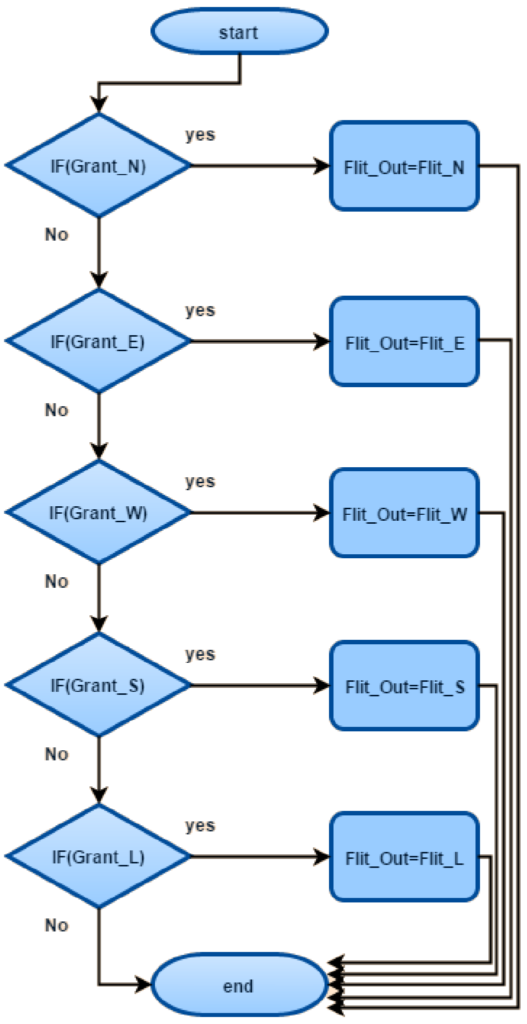
Module Output

Module output is a simple module which responsible for outs the flits that already routed and granted. This is the last module that flits must pass in the switch. This module has connection to module routing and at the same time it connects with module arbiter. As a input, output module takes flit from routing module and grant signal from arbiter. It uses conditional statements to check flits from routing module is granted or not.



```
always @ (posedge clk)
if (~rst) begin
 FlitOut <= 0;
 BroadcastFlitOut <= 0;
 FlitTypeOut <= 0;
 Valid <= 0;
end else begin
    if (Grant_to_N) begin
        FlitOut <= Flit_N;
        BroadcastFlitOut <= BroadcastFlit_N;
        FlitTypeOut <= FlitType_N;
        Valid <= 1;
    end else if (Grant_to_E) begin
        FlitOut <= Flit_E;
        BroadcastFlitOut <= BroadcastFlit_E;
        FlitTypeOut <= FlitType_E;
        Valid <= 1;
    end else if (Grant_to_W) begin
        FlitOut <= Flit_W;
        BroadcastFlitOut <= BroadcastFlit_W;
        FlitTypeOut <= FlitType_W;
        Valid <= 1;
    end else if (Grant_to_S) begin
        FlitOut <= Flit_S;
        BroadcastFlitOut <= BroadcastFlit_S;
        FlitTypeOut <= FlitType_S;
        Valid <= 1;
    end else if (Grant_to_L) begin
        FlitOut <= Flit_L;
        BroadcastFlitOut <= BroadcastFlit_L;
        FlitTypeOut <= FlitType_L;
        Valid <= 1;
    end else begin
        Valid <= 0;
    end
end

endmodule
```

(a) Verilog code



(b) Flow diagram

Figure 3.10: Module output

If the arbiter module grants requested port, output module puts routed flit to that output port. We can see this logic from figure 3.10a as Verilog code and also from figure 3.10b as a flow diagram.

If north port is granted the output module outs the flit which comes from north port. If granted port is east port, flit which comes from east will out. And same for all other ports like west port, south port and local port.

3.3 Hardware Description Language: Verilog

A hardware description language (HDL) is similar to a typical computer programming language except its purpose. HDL is used to describe hardware rather than implement a program to be executed on a machine. Both Verilog HDL and VHDL (Very High Speed Integrated Circuit Hardware Description Language) languages are the most popular in the hardware industry.

In Verilog code, signals in the circuit are represented as variables in the source code, and logic functions are expressed by assigning values to these variables. Verilog source code is just a plain text. This feature, makes Verilog a portable and widely used, encourages sharing and reuse of Verilog-described circuits.[**19**].

It also allows faster development of new products in cases where existing Verilog code can be adapted for use in the design of new circuits. Verilog is based on C programming language so it has a similar syntax. Both structural and behavioral paradigms of integrated design circuit production are included in Verilog language. PEAK baseline architecture is coded mostly in Verilog.

Verilog or other hardware description languages have to provide two main objectives: simulation and synthesis.

Simulation is the process of verifying the functional characteristics of models at any level of abstraction [**20**]. For simulation we use special Verilog files named test benches. Actually test benches are not so different than normal Verilog file. But its purpose makes it special. We can say that test benches are Verilog modules to test other modules.

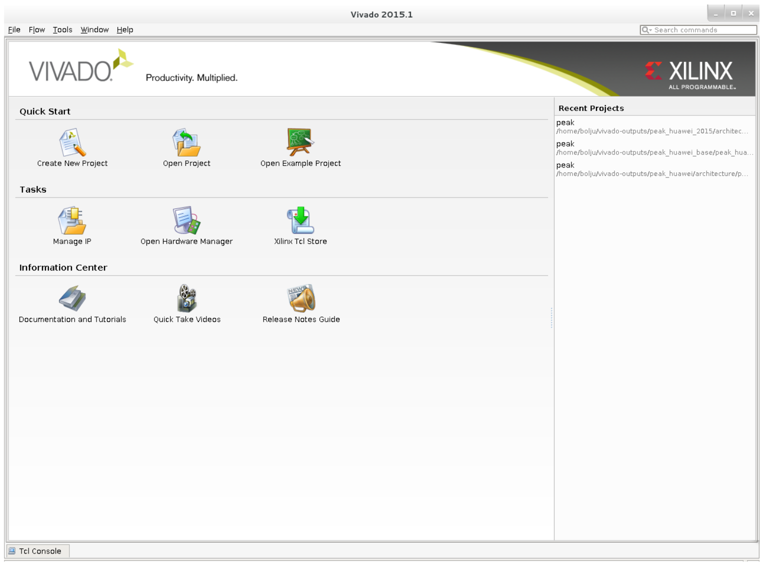
Synthesis is the process in which synthesis tool like design compiler takes RTL (register transfer level - model representation of a digital circuit design) in Verilog, target technology and constraints as input and maps the RTL to target technology primitives. During this process we have to set target board and constraints.

3.4 Vivado Design Suite

We use Vivado Design Suite software 2014.2 mostly and 2015.1 few times for design, simulation and implementation phases of our work. Vivado Design Suite is a powerful tool that includes all the necessary features to accomplish every process of hardware development such as simulation and synthesis. In figure 3.11 screenshot of Vivado Suite software is shown.

Design and Coding

The suite supports both VHDL and Verilog hardware description languages. Vivado provides a friendly source editor equipped with every necessary tool for efficiently write and edit codes. It also includes source viewer with four different types: hierarchical view, view by IP



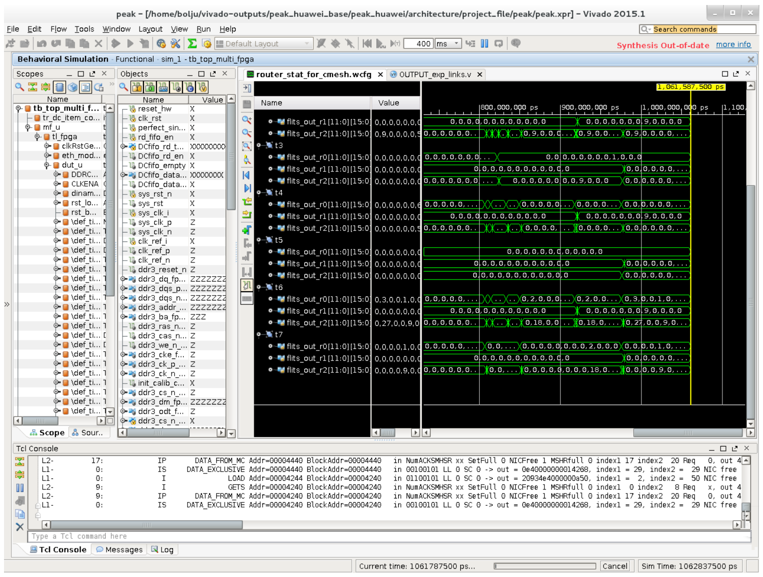
peak
/home/bolju/vivado-outputs/peak_huawei_2015/architec...
peak
/home/bolju/vivado-outputs/peak_huawei_base/peak_hua...
peak
/home/bolju/vivado-outputs/peak_huawei/architecture/p...

Figure 3.11: Screenshot of Xilinx Vivado Suite.

source, library view and by compile order. As well as, status windows named Messages, Log, Reports, Design runs and Tcl consoles are very useful tools in Vivado environment. Suite's dynamic sliding navigator pane contains all necessary configuration shortcuts and so on.

Simulation

In order to run simulations, simulation source (test benches) files must prepared. Test benches are just similar source files written by Verilog or other languages. The system allows to make simulation related configurations like choosing target simulator depending on the installed simulators, used language in sources to be simulated, directory path to save simulation related files such as settings and results and top module of project to be simulated. After simulation run, the system shows result as a waveform configuration file where every signal can be seen as variable lines with a value in time. Side windows named Scope and Objects are also provided to list the modules and their values. Modules can be addable from those windows to waveform to see more signals and removable from existing entries in waveform too. It enables users to create their own modified waveform and save it to see desired simulation. Some important menus are available to use to control simulation process. Those menus are dedicated to perform tasks like re-launching the simulation process, restart the simulation and set specific time for simulation. We can see screenshot of running simulation in Figure 3.12.



peak - [/home/bolju/vivado-outputs/peak_huawei_base/peak_huawei/architecture/project_file/peak/peak.xpr] - Vivado 2015.1

```
L2-          17:          IP      DATA_FROM_MC Addr=00004440 BlockAddr=00004440   in NumACKSMHSR xx SetFull 0 NICFree 1 MSHRfull 0 index1 17 index2  20 Req   0, out 4
L1-           0:          IS   DATA_EXCLUSIVE Addr=00004440 BlockAddr=00004440   in 00100101 LL 0 SC 0 -> out = 0e40000000014268, index1 = 29, index2 =  29 NIC free
L1-           0:           I             LOAD Addr=00004244 BlockAddr=00004240   in 01100101 LL 0 SC 0 -> out = 20934e4000000a50, index1 =  2, index2 =  50 NIC free
L2-           9:           I             GETS Addr=00004240 BlockAddr=00004240   in NumACKSMHSR xx SetFull 0 NICFree 1 MSHRfull 0 index1  0 index2   8 Req   x, out 4
L2-           9:          IP      DATA_FROM_MC Addr=00004240 BlockAddr=00004240   in NumACKSMHSR xx SetFull 0 NICFree 1 MSHRfull 0 index1 17 index2  20 Req   0, out 4
L1-           0:          IS   DATA_EXCLUSIVE Addr=00004240 BlockAddr=00004240   in 00100101 LL 0 SC 0 -> out = 0e40000000014268, index1 = 29, index2 =  29 NIC free
```

Current time: 1061787500 ps...   Sim Time: 1062837500 ps

Figure 3.12: Vivado Suite during simulation.

Chapter 4

Topology Implementation and Evaluation

In the previous chapter we described the tools and methods used. In this chapter we show the details of the implemented topologies and the performance achieved. First, we describe each topology designed and implemented on the PEAK system by using Vivado software suite with Verilog hardware description language. Then, we compare each topology performance and resource usage. Routing and switch arbitration modifications for selected topologies are described as well when needed.

4.1 New Topologies for PEAK

In this Section we describe all the new topologies implemented for PEAK. These topologies derive from the baseline 2D mesh topology. We add new links and new logic functionality. We start with the topologies that use more connectivity by using more links. Then, we focus on topologies with concentrated nodes (bristling factor b set to 4).

DMESH: 2D Mesh with Diagonal Links

2D Mesh with diagonal links (DMESH) is a mesh-based topology with additional diagonal links between switches. Figure 4.2b illustrates a DMESH. Every node (tile) has four links connecting the four tile neighbors at one hop distance on each direction and dimension, plus four diagonal links connecting tiles to neighbors at one hop distance at each quadrant. Notice that edge and border nodes (tiles) have not all the diagonal links implemented.

Diagonal links can be very useful when the destination node is located at a different row and column from the source tile. By using a diagonal link the destination tile can be accessed by a lower number of (logical) hops. However, care must be taken when designing the routing algorithm in order to avoid deadlocks. Indeed, we can not use XY routing with non-diagonal links and randomly use diagonal links. One clear case would be four messages routed along diagonal links building a cycle. Each message is using one diagonal link and requesting a non-diagonal one. Figure 4.1 shows the case. Message m_1 is using X- and X-Y+ diagonal link and requests X+ link. The X+ link is used by m_2 which is using also link X+Y+ and requests link Y+. That link is being used by m_3 which also uses X+Y- and requests Y- link. Finally, m_4, which uses Y- and X-Y-, is requesting link X- which is used by m_1. Therefore, a cycle arises and no message makes forward progress, leading to a deadlock condition.

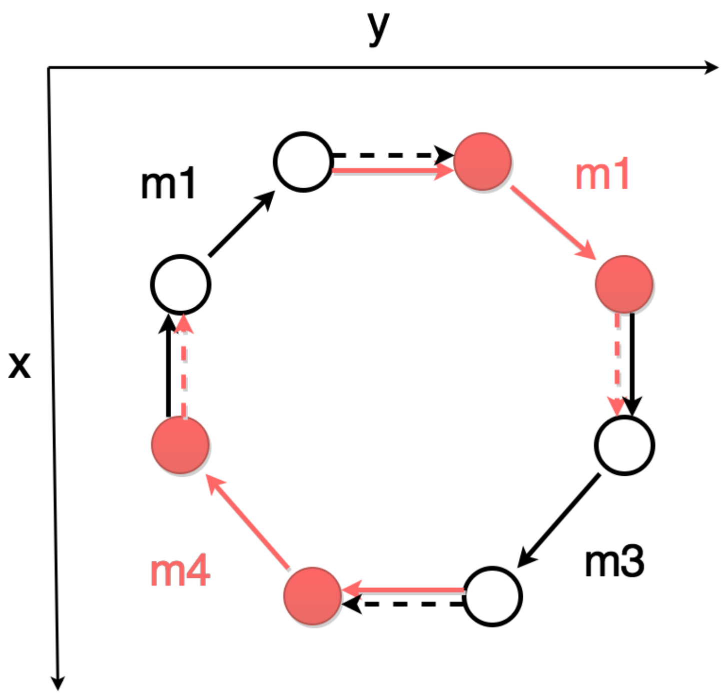
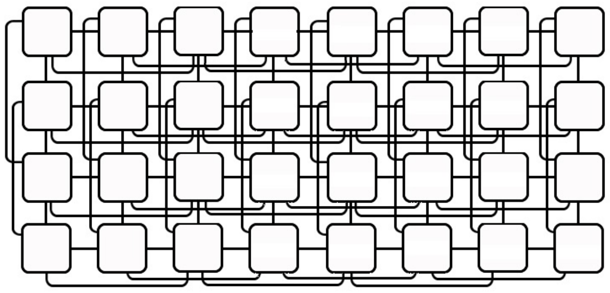


Figure 4.1: Circular requests create deadlock condition when using diagonal links

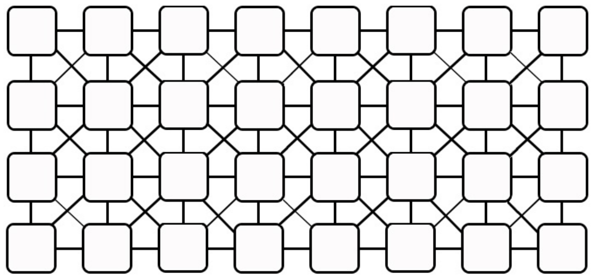
Implementation Details of DMESH Topology on PEAK

In order to implement DMESH we modified different modules in PEAK. Next we describe major modifications performed.

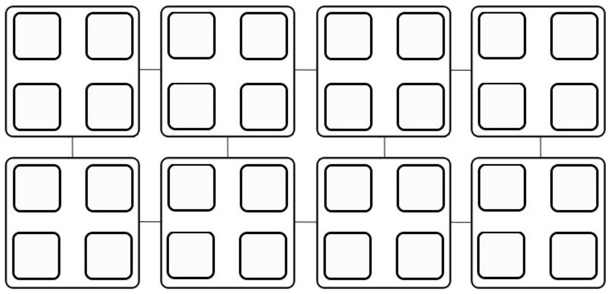
Module Top. At this level links between tiles are defined and instantiated. Thus, we instantiate new diagonal links based on a tile's position in the grid. These new links are named North-West, North-East, South-West and South-East. In the best case, one tile can have 9 links defined, including the local link. However, other tiles may have a lower number.



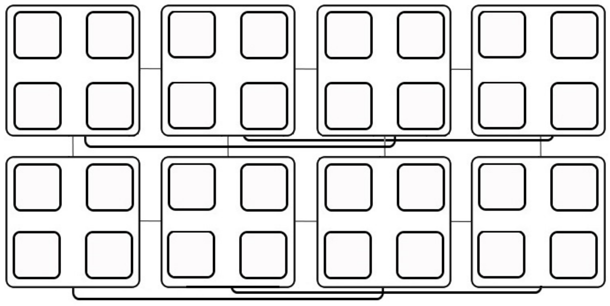
(a) EMESH links



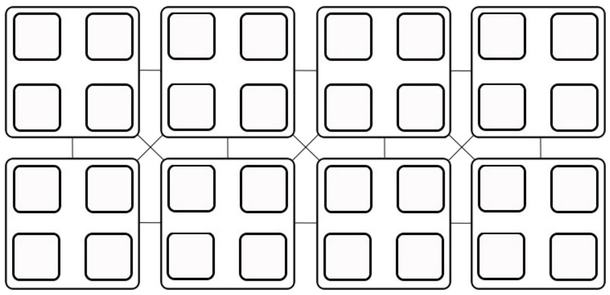
(b) DMESH links



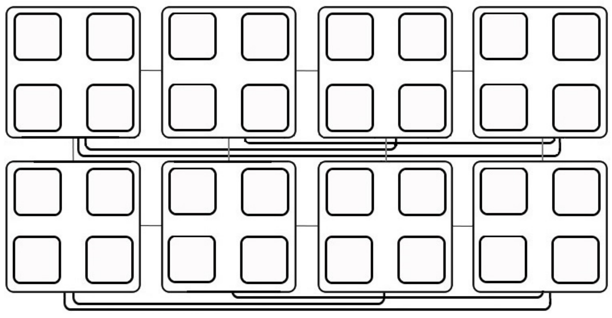
(c) CMESH



(d) CEMESH



(e) CDMESH



(f) FBFLY

Figure 4.2: Implemented NoC Topologies, each has 32 cores

Module Tile. As tiles are homogeneous in PEAK, we add four more ports in order to provide full diagonal connectivity. Notice that we instantiate input and output ports separately. As we have three networks in PEAK we provided ports to each network. This means twelve input and twelve output ports added to the tile module. Inside the tile module we connect the new ports to the three switches instantiated inside the module. These switches need to be adapted.

Module Switch (and its derived internal modules). At switch level, new logic needs to be implemented. More ports have been added to the switch module to provide flit transfers. This means more buffers created to serve flits from diagonal connections. In particular, an input buffer (IBUFFER) is instantiated for each new input port and a crossbar output port (XOP) is instantiated for each new output port. Connections between input buffers and new output ports have been created. We can see switch module structure from figure 4.3. We can see in the figure in red the new added modules and connections.

One additional aspect is the switch arbiter at each output port (previous ports and new ports). Each output port may potentially forward flits from any input port and this means arbitration needs to be changed. The current arbiter in PEAK is a round-robin arbiter which arbitrates between all input ports (excluding the port with the same dimension and in opposite direction). In order to support new ports, we extended the round-robin arbiter with new inputs. Figure 4.4 shows the previous arbiter and the new one. Notice that no priority is given to any input port.

The most important module with major modifications is the routing module. This module has been replicated for each input port as it is in charge of routing flits from a specific input port. As commented previously, XY routing is no valid any more since it may introduce deadlocks when using diagonal links. The approach we follow to avoid deadlocks and still route messages is to consider diagonal links as a subset of links that can not be arbitrarily mixed with non-diagonal links. Indeed, we define an order of usage of both types of links. If all messages use diagonal links and then request non-diagonal links, but then never request again diagonal links then no deadlock can occur. Non-diagonal links are used using XY routing as before. Diagonal links, however, are used with a minimal path policy.

When a message is injected into the network all links can be used (diagonal ones and non-diagonal ones). Obviously, diagonal links have more priority since they provide shorter paths. The message is then routed and selects the most suitable link to take. After some cycles, the message arrives to the next switch. There, the same routing strategy applies if the message came through a diagonal link, thus, the message considers both types of links. If diagonal links are not available (or are not profitable for the message), then the message selects a non-diagonal link, following XY rule. From that moment, the message is routed through non-diagonal links until it reaches final destination. By doing this, we are considering non-diagonal links as escape paths that are used when diagonal ones are not available or are not profitable for the message.

In order to implement the new routing approach, we add two new signals to every routing

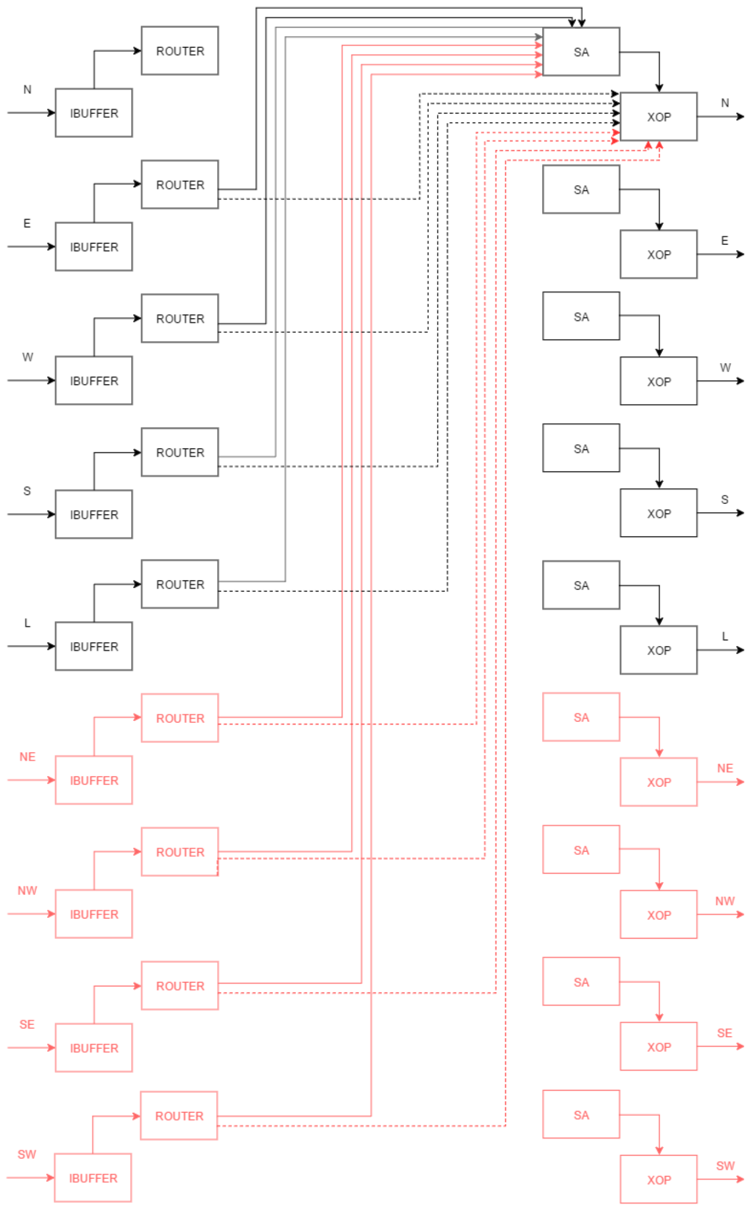
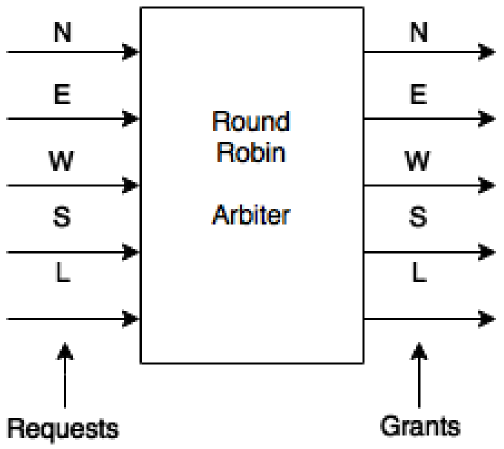
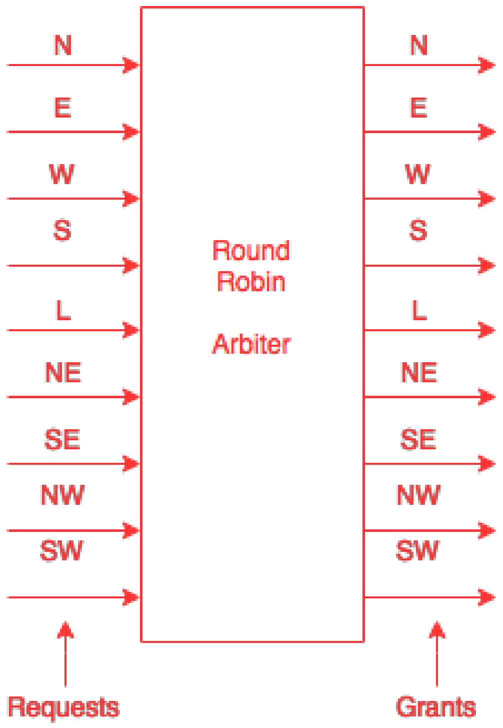


Figure 4.3: Router architecture for DMESH

module named *isDiagonal* and *isLocal*. The *isDiagonal* signal is used to identify that the current routing module is associated with a diagonal port. Similarly, the *isLocal* port defines a current routing module is associated with a local port. DMESH new routing module interface is shown in figure 4.5.



(a) Previous arbiter



(b) DMESH arbiter

Figure 4.4: Previous and DMESH arbiter

EMESH: 2D Mesh with Express Links

If we add some express links to 2D mesh topology, it will become 2D mesh with express links topology (we refer to this topology as EMESH). The express link connects the current node (tile) to tiles at two hops distance in the same dimension and direction, thus bypassing the neighbor tile. Thus, by using express links, we can save one hop in every routing module. Easily, we can deduce that two hops in a 2D mesh equals to one hop in EMESH. Figure 4.2a shows the mesh topology with express links.

Implementation Details of EMESH on PEAK

With EMESH, every tile may have up to four express links to transfer data besides four normal mesh links and a local link. The express links are used following the XY rule, therefore, being compatible with non-express links. This makes routing deadlock-free since there will be no cycles. No message will use an Y channel (either express or normal) and then request an X channel (either express or normal).

To implement EMESH we follow a similar approach as performed for the DMESH topology. At the top module we defined and instantiated new wires building the new links. In

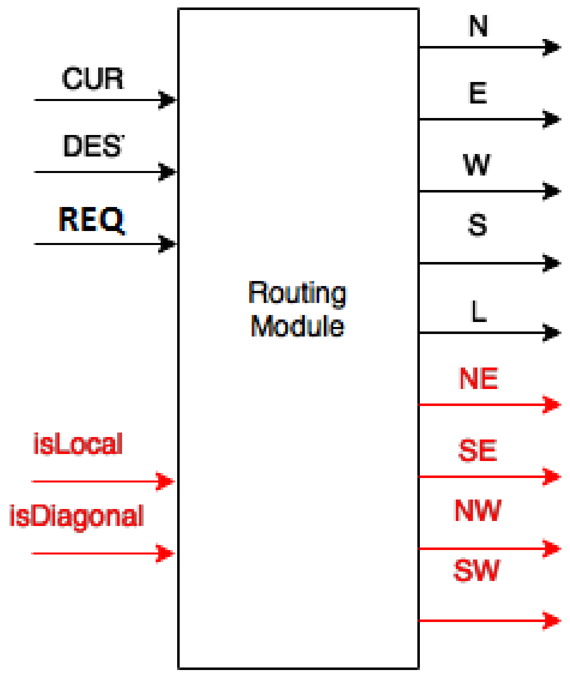


Figure 4.5: Router module interface of DMESH

total, up to 24 ports (taking into account we have three networks and ports are unidirectional). Depending on the tile position the number of ports used varies. Similarly, all the 24 ports have been instantiated at the tile level (tiles are homogeneous) and connected properly to the three internal switches.

The switch and its associated modules have been also changed to host the EMESH topology in PEAK. New input buffers have been used, new arbiters instantiated to the new output ports, new XOP modules at the output ports and the proper wiring between the input and output ports. In addition, the arbiters have been redefined to host new inputs.

Finally, the routing module has been modified to consider the new express channels. Those channels, coded as NN, EE, WW, and SS have higher priority meaning they are selected when the destination is at least two hops away from the current tile. For this, we added a comparator inside the routing module. The express channels follow the XY rule, meaning the X ports (either normal or express) need to be canceled before considering Y ports (either normal or express). Figure 4.6 shows the algorithm implemented in the module.

CMESH: 2D Concentrated Mesh

The concentrated mesh (CMESH) is a special kind of mesh with a lower number of switches and links. The concentration factor (bristling factor) defines how many tiles share the same

```
Current_X = Current_Switch.X;
Current_Y = Current_Switch.Y;
Destination_X = Buffered_Flit.Destination.X;
Destination_Y = Buffered_Flit.Destination.Y;

IF Current_X == Destination_X THEN
    IF Curent_Y- 1 > Destination_Y THEN route to NorthNorth
    ELSE IF  Curent_Y- 1 == Destination_Y THEN route to North
    ELSE IF Curent_Y+ 1 < Destination_Y THEN route to SouthSouth
    ELSE IF  Curent_Y+ 1 == Destination_Y THEN route to South
    ELSE IF  Curent_Y == Destination_Y THEN route to Local
ELSE IF Current_X+ 1 < Destination_X THEN route to EastEast
ELSE IF Current_X+ 1 == Destination_X THEN route to East
ELSE IF Current_X- 1 > Destination_X THEN route to WestWest
ELSE IF Current_X- 1 == Destination_X THEN route to West
```

Figure 4.6: Routing algorithm for EMESH

switch. In our case, four tiles share a switch (bristling factor set to four). Switches connect between them as in the original mesh topology. The CMESH topology shortens the network diameter in comparison with the original mesh topology. Figure 4.2c shows the CMESH topology for 32 tiles.

Implementation Details of CMESH on PEAK

Implementing CMESH has deep implications on the PEAK architecture. Indeed, one of the first issues to address is the tile identification system. Unique IDs are used for each tile to, first, identify the tile resources, and second, to correctly route messages within the network. Now that we have in CMESH more than one tile per router how IDs are labelled and used becomes crucial. The current PEAK system has defined the maximum supported number of tiles to 256. This means, the ID is sized to 8 bits. In a regular mesh (or the previous DMESH and EMESH topologies) all the ID bits are used at the routing level to decide the destination tile position within the grid. Now, because of the concentration factor, we need to differentiate between tiles and clusters of tiles. For a bristling factor of four, groups of four tiles share the same switch. Therefore, they are in the same cluster. The 8 bits of the ID field are then used in a different manner. The two least significant bits of the ID are used to select the tile within the cluster and the six most significant remaining bits are used to route messages between clusters. This leads to a consecutive numbering of tiles within a cluster. Thus, tiles 0, 1, 2, and 3 are in cluster 0, tiles 4, 5, 6, and 7 are in cluster 1, and so on until we reach tiles 252, 253, 254 and 255 in cluster 63.

Also, the complete tile is replicated inside the cluster. In order to provide such clustering approach, we define a new module in PEAK termed ctile (clustered tile). This structure includes four tiles and the routing infrastructure needed. In particular, the core, the NI, the L1D, the L2 bank, and the tile register module are replicated and instantiated in ctile. Also, the three routers are embedded into ctile and connected properly to the four NIs. Figure 4.8 shows the original tile structure and the new ctile structure.

Each core in tile has connection with its own L1 cache, L2 cache and Tile Register provided by its own Network Interface. We can consider them as a one sub-tile each has its own unique ID and address. Thus, tile of concentrated mesh or we can say concentrated tile consists of four sub-tiles. We can see 4x2 sized Cmesh topology with 32 cores in Figure 4.1(c).

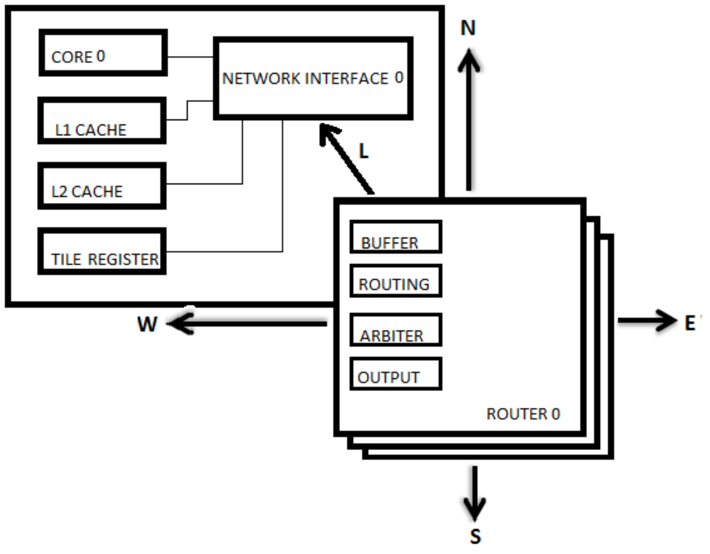
This topology requires many major changes at many levels (top level, tile level, new ctile module, switch module and its components). First, the top module needs to be modified in order to provide support to four IDs per ctile module. The top module, instead of instantiating a set of tiles, now it instantiates a set of ctiles and each ctile module embeds four tiles. Links between ctiles do not change as the topology is still a 2D mesh. The ctile module is a new module which instantiates four different (and internally connected) tiles and the associated switches.

The switch module and its associated modules have also major modifications. In detail, three new local ports are defined and instantiated to host the three new tiles connected to the router. This means a pair of IBUFFER and RT modules per new input port and a pair of SA and XOP modules per new output port. In addition, the RT module has been adapted to discriminate internally whether the message needs to be routed to a different ctile or delivered internally. In the later case, the RT module needs to select the proper local port to be used as output port. This is performed based on the current ID of the ctile (notice that this means the upper 6 bits of the ID field) and the destination ID. The following algorithm describes the RT module implementation.

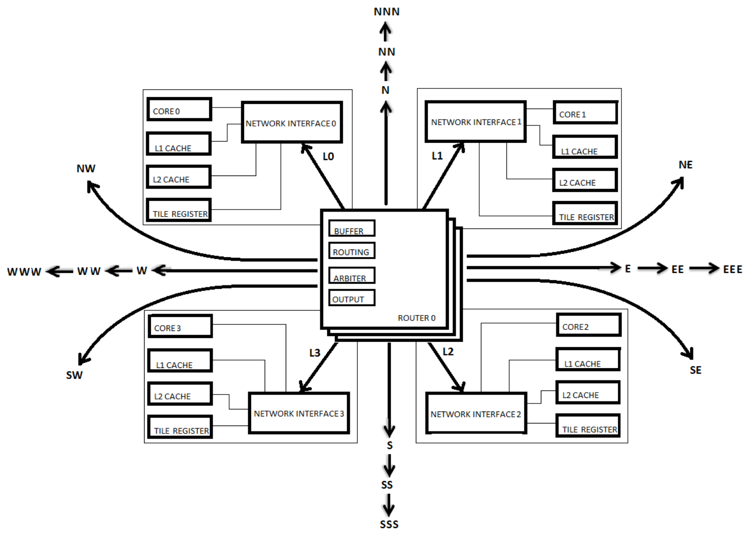
```
Current_X = Current_Switch.X;
Current_Y = Current_Switch.Y;
Destination_X = Buffered_Flit.Destination.X;
Destination_Y = Buffered_Flit.Destination.Y;
Destination_Local = Buffered_Flit.Destination.Local;

IF Current_X == Destination_X THEN
     IF Curent_Y > Destination_Y THEN route to North
     ELSE IF  Curent_Y < Destination_Y THEN route to South
     ELSE IF Curent_Y == Destination_Y THEN
          IF Destination_Local == 0 THEN route to Local0
          ELSE IF Destination_Local == 1 THEN route to Local1
          ELSE IF Destination_Local == 2 THEN route to Local2
          ELSE IF Destination_Local == 3 THEN route to Local3
ELSE IF Current_X < Destination_X THEN route to East
ELSE IF Current_X > Destination_X THEN route to West
```

Figure 4.7: Routing algorithm for CMESH

Finally, the arbiter has been adapted to host eight requests (from N, E, W, S, L0, and the new L1, L2, and L3).



(a) Tile structures with main components



(b) Ctile structures with main components

Figure 4.8: Structures of current tile and ctile

C{DE}MESH: Concentrated Mesh with Diagonal/Express Links

We can combine the previous topologies to create more advanced ones. This is the case when we mix the clustered topology with either diagonal or express links. This leads to the CDMESH (concentrated mesh with diagonal links) and to the CEMESH (concentrated mesh with express links). The complexity of these new topologies is additive and relies on the fact of combining the ctile module implemented for the CMESH topology and adding the resources used in the DMESH and EMESH topologies. Figures 4.2e and 4.2d show both topologies.

FBFLY: Flattened Butterfly Topology

Figure 4.2f shows the Flattened butterfly topology for a 32-core system. In this topology, FBFLY, every node has one exclusive link to every other node along the same row and column. In addition, a bristling factor of four is used. This topology allows to reach any node in just two (logical) hops maximum. However, resources to implement the topology seem high.

We implement this topology as is a close variation of the previous ones. With this topology we close the spectrum of alternative topologies PEAK may have, taking as a background the underlying 2D mesh topology.

Implementation Details of FBFLY on PEAK

The first modifications for FBFLY are in the use of ctile module instead of tile. This module is adapted to the new situation with more links to connect to other ctiles. Indeed, now have, for a 32-node system, eight ctiles and thus four new links to connect between ctiles. These new links are termed EE, EEE, WW, WWW. The switch module, therefore, needs to be improved with four new input buffer modules and four new RT modules. The same applies for the output ports (four new XOP and SA modules).

The RT module needs to be adapted as well, since now a message may request up to four new output ports. Also, those output ports need to be selected with the XY rule in mind so to avoid deadlocks. In addition, EEE/WWW ports have higher priority than EE/WW ports. The same applies between EE/WW ports and E/W ports. Minimal paths are enforced. The following algorithm shows the RT module behavior.

Finally, the arbiter is adapted to host the new requests from the new ports.

4.2 Performance and Implementation Results

In this section we show and analyze the achieved results. This section has two differentiated parts. The first part (overhead analysis) shows the synthesis results of the topologies, with a focus at different levels: switch, tile and top. The target FPGA board (V2000T) is selected for the analysis, and we will see how all the topologies fit the resources for a target 32-core system. The second part (performance analysis) shows the performance achieved by the different topologies. To build this part we opted for simulation results with Vivado when running successfully the PEAK manager and an instance of a matrix multiplication (mmul) application. We simulate two different cases for mmul: with 32 cores and with 8 cores.

```
Current_X = Current_Switch.X;
Current_Y = Current_Switch.Y;
Destination_X = Buffered_Flit.Destination.X;
Destination_Y = Buffered_Flit.Destination.Y;
Destination_Local = Buffered_Flit.Destination.Local;


IF Current_X == Destination_X THEN
    IF Curent_Y > Destination_Y THEN route to North
    ELSE IF  Curent_Y < Destination_Y THEN route to South
    ELSE IF Curent_Y == Destination_Y THEN
        IF Destination_Local == 0 THEN route to Local0
        ELSE IF Destination_Local == 1 THEN route to Local1
        ELSE IF Destination_Local == 2 THEN route to Local2
        ELSE IF Destination_Local == 3 THEN route to Local3
ELSE IF Current_X +1 == Destination_X THEN route to East
ELSE IF Current_X -1 == Destination_X THEN route to West
ELSE IF Current_X +2 == Destination_X THEN route to EastEast
ELSE IF Current_X -2 == Destination_X THEN route to WestWest
ELSE IF Current_X +3 == Destination_X THEN route to EastEastEast
ELSE IF Current_X -3 == Destination_X THEN route to WestWestWest
```

Figure 4.9: Routing algorithm for FBFLY

Overhead Analysis

To obtain overheads of the topologies we use the synthesis tool of Vivado suite software and a 32-core system implementation. This leads to 8×4 configurations for MESH, DMESH and EMESH topologies, and 4×2 configurations for CMESH, CDMESH, CEMESH, and FBFLY topologies. We first analyze switch overhead, then tiles overheads, and finally overall system overheads.

Switch Overheads

We have synthesized all the seven switch architectures, one for each of 7 topologies. Notice that they differ in the switch radix and the internal logic to support the different routing modules, mainly. We provide the breakdown of FPGA resources per component, mainly the input buffer module (IB), routing module (RT), arbiter module (SA), and crossbar output module (OUT). All these components are associated to an input or output port in the switch, so they represent the resources needed for only one instance of such module. The switch, later will be evaluated with all the needed modules, replicated per port.

Figure 4.10 shows the results. According to those results we can see that the OUT module has more resources demand than any other. This happens in all the topologies and mostly in LUTs resources. This is due to the use of an internal 64-bit register to store the flit at the output channel and how this is implemented in Vivado. Anyway, we see that all the modules increase in resources (mainly LUTs) except IB module. In particular, the RT module increases when moving from the MESH topology to any other topology. This increase in size is, however, small and lower always than 50% overhead. This increase is due to the new routing functionality needed to support new output ports in the new topologies. The SA module also increases in area and its increase is proportional to the number of input

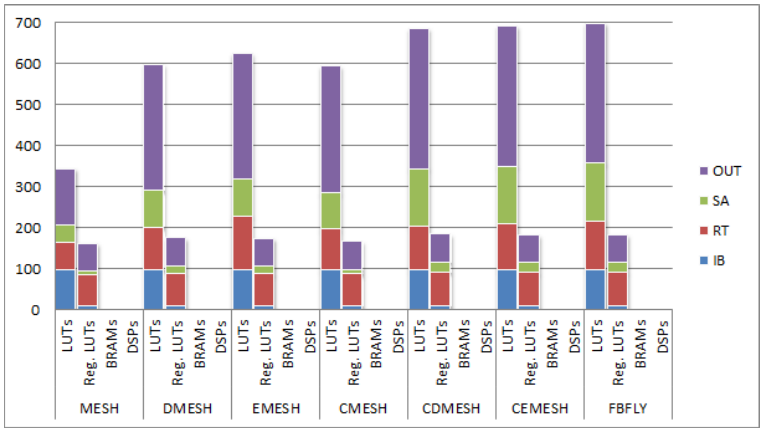


Figure 4.10: Switch components overheads for the different topologies to be supported.

ports coded in the new topologies. For instance, for DMESH we see an almost 100% increase in LUTs when compared to MESH topology. Indeed, we are moving from a 5-port switch to a 9-port switch. The increase is higher in CDMESH, CEMESH, and FBFLY topologies. In such topologies we have up to 150% increase in LUTs and this is because we are moving from a 5-port switch up to a 12-port switch (for the CDMESH topology). Therefore, in LUTs we see a linear increase that somehow was expected. The OUT module also increases almost linearly with the switch degree. Therefore, we do not see any deficiency introduced by our designs, when compared to the baseline PEAK architecture. For register LUTs we see almost the same results regardless topologies and we can also see that BRAMs and DSPs are not used by the switch components.

Figure 4.11 shows now the total resource needs to implement a complete switch for each analyzed topology. We can see an interesting result here. For LUTs, the CMESH topology exhibits lower resources than DMESH and EMESH. This is mainly due to the lower number of ports coded in the topology but not exclusively by this fact. Indeed, results are somehow in accordance with the number of ports implemented in the different switches to support the topologies. Figure 4.12 shows the resources broken down by port (dividing resources by the number of implemented ports). As we can see, differences between implementations are smaller when considered the port granularity. However, there are major differences still. The MESH topology is still the most efficient one at port level. However, the rest fall within 100% increase in resources at port level. This is mainly in LUTs. The higher radices of the switch leads to more alternative switching paths which necessarily leads to higher resources. However, overheads are not as large as expected when considering port granularity.

Tile Overheads

Now we analyze the overheads of the tiles embedding the corresponding switch versions and the clustered approach (ctile) for some configurations. It is worth noticing the tile structure

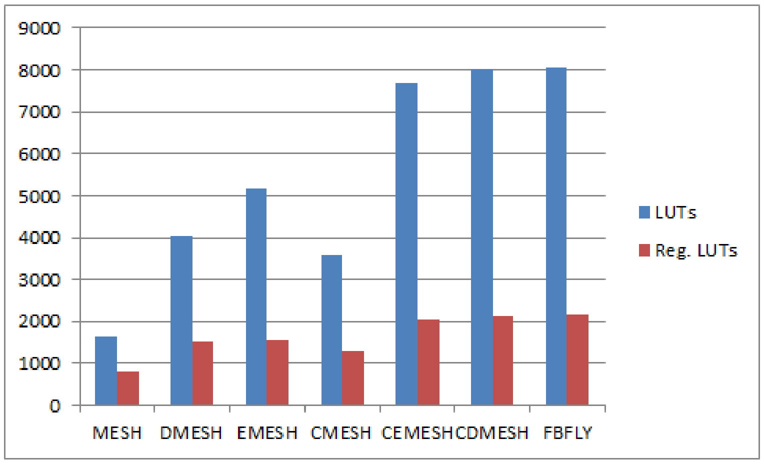


Figure 4.11: Switch overheads for the different topologies to be supported.

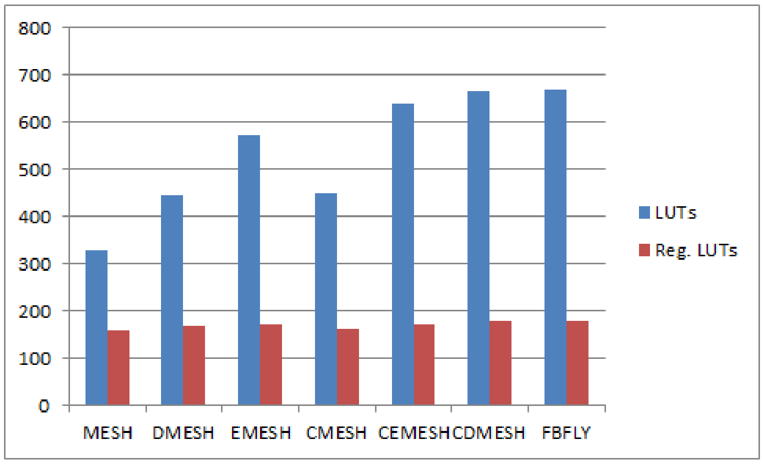


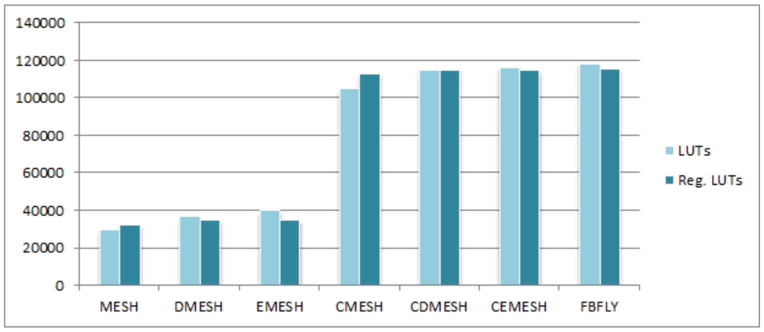
Figure 4.12: Port overheads for the different topologies to be supported.

(and now also the ctile structure) is the basic building block in PEAK that is repeated to build the complete system. Thus, it is important to analyze its overhead.

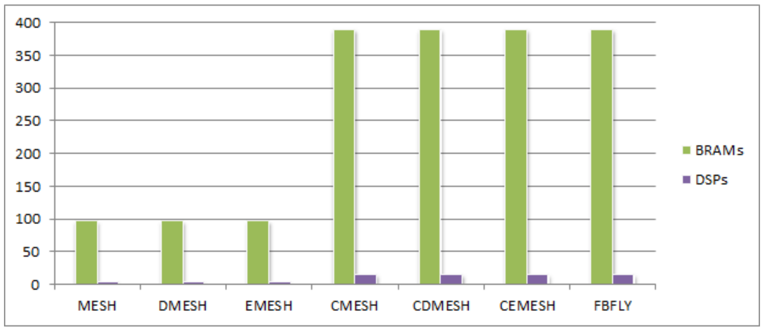
Figure 4.13 shows the obtained tile overhead results in resources. Because of the similar results we obtained, we categorize results in two categories: mesh-based and cmesh-based.

Mesh-based tiles (those for MESH, DMESH, and EMESH) have much less overhead in any type of FPGA resource when compared to cmesh-based tiles (those for CMESH, CDMESH, CEMESH, and FBFLY). Indeed, we see an approximate factor of resource utilization of four between mesh-based and cmesh-based tiles. This corresponds to the bristling factor.

If we focus on the mesh-based tiles, we can see that MESH tile has less resources in LUTs and slightly in register LUTs. This is due to the simpler switch as the switch radix is lower. However, here we can see that the switch overhead is much lower when compared at tile



(a) Tile overhead LUTs and Registers



(b) Tile overhead BRAMs and DSPs

Figure 4.13: Tile overheads for the different topologies to be supported.

granularity. The cores and caches play a major role in resource demand. Therefore, we can deduce that overhead of topologies with diagonal and express channels is not significant at tile level. Anyway, we could see still a 15% increase in LUTs when moving from MESH to EMESH.

If we focus on cmesh-based topologies, we can see that the same trend occurs, being the CMESH topology the one with less resource demand, specially in LUTs. However, the percentage difference here is much smaller as the tiles have now four fully deployed cores with the associated cache hierarchy components. Therefore, we can deduce also those topologies do not play a significant role in resource usage when compared between them.

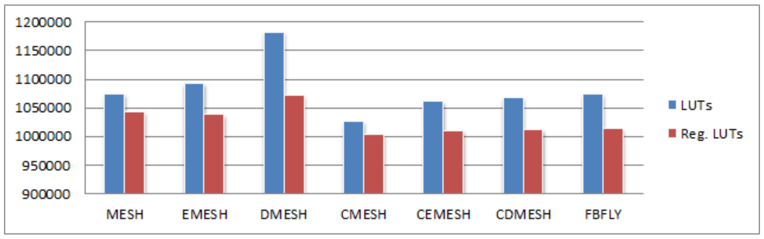
The main conclusion we can obtain from those results is that it is worth using topologies with high radixes as the network resource needs are not high. When compared with connectivity and provided bandwidth, thus, the best topology is the FBFLY one as its resource needs are similar to the baseline CMESH topology.

For the BRAMs and DSPs resources we see also the factor of four from mesh-based topologies to cmesh-based ones. Also, resource needs for the different topologies do not change significantly.

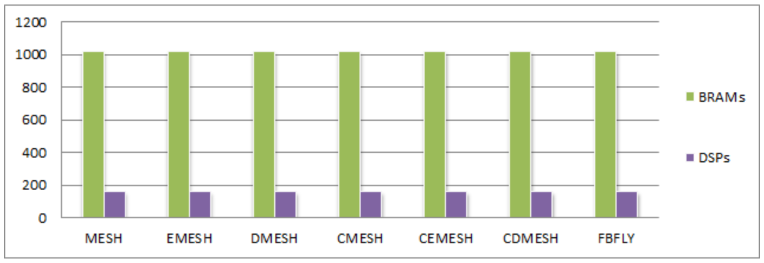
Topology Overheads

Up to now we have compared the switch architectures and the tiles for the seven topologies. Now we compare 7 tops (top is the module name which implemented a certain topology). According to figure 4.14 (b) BRAMs and DSPs usage are exact same in every top. Now, we provide final synthesis results when considering the whole PEAK system instantiated for each of the seven topologies. This is an important analysis since it puts the seven topologies under the same number of cores configuration. We target a 32-core system which currently fits in the target FPGA device.

Figure 4.14 shows the overhead of the seven topologies compared. According to the previous tile-based analysis, the FBFLY was the best topology. However, according to the top analysis now, the best topology is the CMESH one followed by the ones with added links, CEMESH and CDMESH, and lastly the FBFLY. Overheads, however, are not large.



(a) Overhead of top by LUTs and Registers



(b) Overhead of top by BRAMs and DSPs

Figure 4.14: Overhead of top modules

Performance Analysis

In order to obtain performance results we have used the behavioral simulation capabilities of Vivado design suite. We target 32 cores which is the largest system size that currently is supported in PEAK. This leads to 8×4 configuration when using tiles (MESH, DMESH, and EMESH), and 4×2 configuration when using ctiles (CMESH, CDMESH, CEMESH, and FBFLY). The matrix multiplication code is listed next. The MMUL code is a simple program. Each instantiated process takes a row and computes all the elements of the row for the final matrix. The row is accessed via a protected section via a semaphore. For the simulations we assume a 40×40 matrix size. We show results here for matrix multiplications with 32 cores (results with 8 cores showed a similar trend and are not shown here). Following is the MMUL code written by programming language C.

```c
char matrixA[rows][cols]; char matrixB[rows][cols];
char matrixC[rows][cols]; unsigned int sem; int row; //initializing arrays and varaibles to be used

void mmul(int argc, int *argv) //function to set mmul
{
peak_print_str("Starting matrix multiplication (master)...\n"); //starting mmul

row = 0; peak_unlock(&sem); //row starts from 0

unsigned int this_tile = peak_get_tile_id();

unsigned int cores = peak_get_notification_bits();
unsigned int task_id = peak_get_task_id(this_tile);
peak_print_str("cores assigned: "); peak_print_hex(cores); peak_print_str("\n");
unsigned int i = MAX_NUM_CORES;
while (cores) { //performing mmul in every assigned core
if ((cores & 0x00000001) && (i!=this_tile)) {
peak_print_str("launching on core ");
peak_print_int(i);
peak_print_str("\n");
peak_assign_pc_address_core(i, mmul_func);
peak_set_task_id(i, task_id);
peak_unfreeze_core(i);
}
i=i-1; cores = cores >> 1;}

peak_print_str("launching on this core\n");
mmul_func(); }

void mmul_func() //main work function
{
int i, j, pos;

peak_print_str("starting matrix multiplication (slave)...\n");

while (1) {  //starting loop
peak_lock(&sem);
if (row == rows) { //checks wether all the rows processed or not
peak_print_str("end of mmul\n"); //if all rows finished
unsigned int this_tile = peak_get_tile_id();
peak_set_notification_bit(0, this_tile);
peak_unlock(&sem);
peak_freeze_local_core();
} else {  //case of row calculation did not finish
i = row;
row = row + 1;} //incrementing row number
peak_unlock(&sem);

peak_print_str("processing row "); peak_print_int(i); //Printing number of row which being processed
peak_print_str("\n");

void init_matrixA() //Initializing array A. And set 0 to all items.
{
peak_print_str("initializing matrix A\n");
int i, j;
for (i=0;i<rows;i++) {
```

```
for (j=0;j<cols;j++) {
matrixA[i][j] = 0;}}}

void init_matrixB() //Initializing array B. And set 0 to all items.
{
peak_print_str("initializing matrix B\n");
int i, j;
for (i=0;i<rows;i++) {
for (j=0;j<cols;j++) {
matrixB[i][j] = 0;}}}
```

Matrix Multiplication Performed by 32 Cores

Figure 4.15 shows the total execution time of the matrix multiplication when using all 32 cores.[1] We can see that the MESH topology (the baseline topology) is the one which takes more time to run the matrix multiplication. When using diagonal links (DMESH) and express links (EMESH) we see a decrease in execution time. However, most of the reduction comes with all the concentrated topologies. Those topologies achieve a reduction in execution time when compared to the non-concentrated topologies. Therefore, the trend of increasing connectivity or providing concentration helps in obtaining higher performance.

When looking at absolute numbers we can see that execution time reduction is not very large. Indeed, difference between the MESH and CDMESH topologies (exhibiting the highest and lowest execution time, respectively varies only in 1%. This small reduction is due to the low usage of the NoC by the application. Indeed, this occurs as the data set (the two source matrices and the destination matrix) fit in the cache hierarchy and therefore many accesses to the matrix end up in hit accesses to the L1D cache. The 1% execution reduction is due only to the small percentage of the traffic when cold misses occur. Nonetheless, this is a clear indication there is an impact on performance by the more advanced and sophisticated topologies. It is expected also that with more intensive applications (or applications with larger data sets) topology will play an increasing role in determining the execution time of the application.

Looking at results we may deduce CDMESH is the best topology in our current system and configuration. However, if we take into account also implementation results the choice could be different. Indeed, the FBFLY topology may become a better topology. First, its execution time does not differ much from the CDMESH topology. And second, its implementation overhead is almost similar with the one of CDMESH. One thing that needs to be taken into account is that the tested system size (32 cores) does not allow FBFLY to fully deploy its potential, since it is implemented only with two rows of ctiles. A fully deployed FBFLY topology would mean 64 cores in a 4×4 grid.

Performance results were not extended with more benchmarks as they were not available in PEAK at the time of coding the topologies. A complete suite of topology analysis benchmark is under development at this stage.

[1]Note that one core does not run the mmul application as it is running the PEAK manager. Therefore, in total there are 31 cores running the matrix multiplication

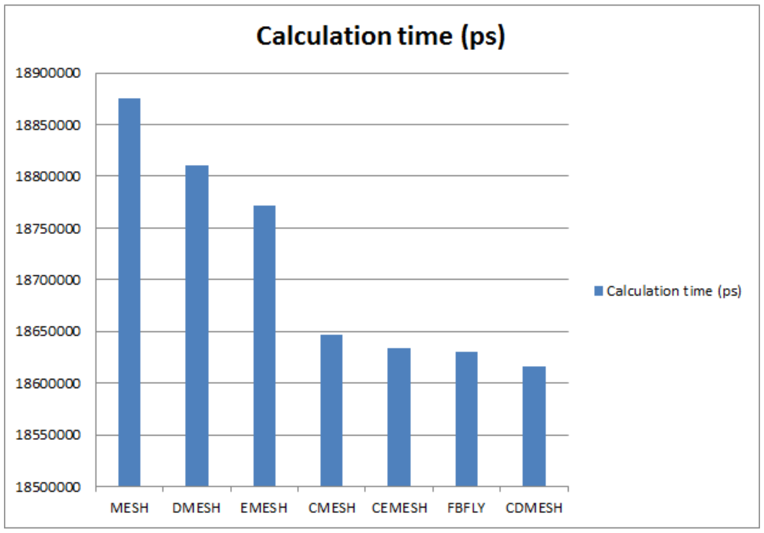


Figure 4.15: Calculation time

As previously stated, we also run the mmul application in 8 cores configuration (with the same target 32-core systems). We expected execution time would be significantly reduced as the eight cores would be confined inside 2 ctiles. However, due to the low usage of the network, execution time was not significantly impacted in this situation. Anyway, the same slight differences were appreciated.

In order to obtain a more insightful explanation of the performance results, we plot the number of flits injected by each possible port in all the configurations. We assume the complete execution of the MMUL application.

From Table 4.1 we can see the number of flits crossed over the ports of the topologies. It also shows the percentage with respect the baseline MESH case. We need to consider the memory controller (MC) is connected to tile (or ctile) zero. Some ports are more used than others due to data location. For example, as a general trend, the east output port is more used than others. Also because of the limited network size (8×4 or 4×2), the number of cores (32) and the used topologies, some ports are just never used. For example, we have only 2 rows in FBFLY, thus NNN and SSS ports are never used. Notice also that the same message that travels through two hops is counted twice in the number of flits that crossed links. Thus, higher numbers means also larger distances when comparing overall number of injected flits for different topologies. Indeed, concentrated topologies exhibit a much smaller number of injected flits due to locality exploitation and reduced hop distance. This will have an impact also on energy and power consumption.

Figure 4.16 shows the percentages of crossed flits normalized to the MESH baseline case. According to results, the FBFLY topology reduces hop distance by a 60% when compared to the MESH case. Indeed, concentrated topologies significantly reduce hop distance. DMESH and EMESH did also reduce hop distance but to a lower extent, thus not justifying their

Table 4.1: Total flits crossed over all routers. Depending on the topology, some ports are empty valued which means not attached.

		MESH	DMESH		EMESH		CMESH		CEMESH		CDMESH		FBFLY	
	Ports	Total	Total	(%)	Total	(%)	Total	(%)	Total	(%)	Total	(%)	Total	(%)
1	N	15830	3646	23.03	5675	35.84	6021	38.03	6021	38.03	1224	7.73	6021	38.03
2	E	101334	67488	66.59	25144	24.81	42182	41.62	14930	14.73	29975	29.58	8218	8.10
3	W	37685	25865	68.63	11795	31.29	14647	38.86	6371	16.90	10397	27.58	4587	12.17
4	S	42196	8794	20.84	14445	34.23	14359	34.02	14359	34.02	2699	6.39	11411	27.04
5	L/L0	54191	54129	99.88	54129	99.88	12759	23.54	12759	23.54	12759	23.54	14549	26.84
6	NN	-	-	-	5061	-	-	-	0	-	-	-	-	-
7	EE	-	-	-	37663	-	-	-	13626	-	-	-	7408	-
8	WW	-	-	-	12904	-	-	-	4138	-	-	-	2119	-
9	SS	-	-	-	13889	-	-	-	0	-	-	-	-	-
10	NE	-	3752	-	-	-	-	-	-	-	1375	-	-	-
11	NW	-	8399	-	-	-	-	-	-	-	3422	-	-	-
12	SE	-	30030	-	-	-	-	-	-	-	10832	-	-	-
13	SW	-	3339	-	-	-	-	-	-	-	828	-	-	-
14	EEE	-	-	-	-	-	-	-	-	-	-	-	6947	-
15	WWW	-	-	-	-	-	-	-	-	-	-	-	1784	-
16	L1	-	-	-	-	-	17902	-	17911	-	17911	-	17911	-
17	L2	-	-	-	-	-	11279	-	11279	-	11279	-	11279	-
18	L3	-	-	-	-	-	12180	-	12180	-	12180		12180	-
	Total	251236	205442	81.77	180705	71.92	131329	52.27	113574	45.20	114881	45.72	104414	41.56

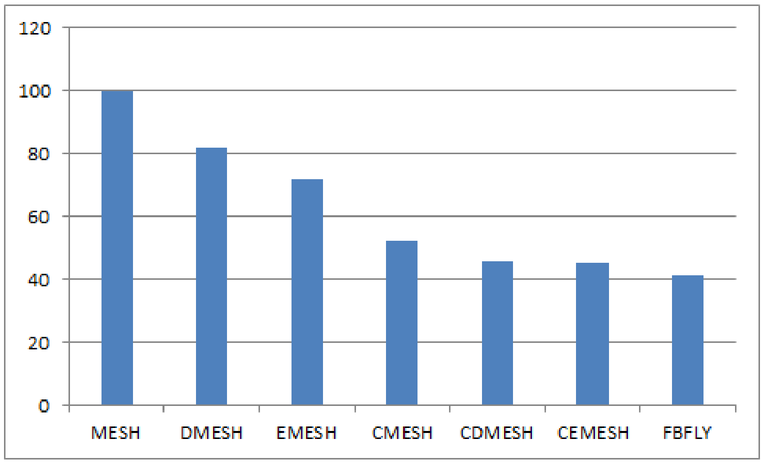


Figure 4.16: Total flits crossed in percentages.

higher costs. We can, thus, deduce, concentrated topologies offer a good tradeoff between performance and implementation costs for PEAK.

Chapter 5

Conclusion

Our thesis main goal was to implement mesh based variations of topologies and to design and implement appropriate router architectures for those topologies. Then, to evaluate best ones by analysis. In this work we implemented up to 6 different topologies for the NoC used in PEAK architecture. During implementation we also designed and implemented different router architectures for each topology. We performed simulation and obtained synthesis results to evaluate the topologies in terms of performance and overhead. Based on those results, we obtain the following conclusions:

- All targeted topologies have been successfully implemented, synthesized and simulated.
- From the point of view of performance the best topologies are the ones based on concentrated tiles, like cmesh with diagonal links and cmesh with flattened butterfly. The added connectivity provided to those topologies allows to achieve more flexibility when choosing network paths. Execution time is reduced, although not to a high extent.
- High radix routers are an efficient choice for NoCs. This means more connection paths are provided. However, benefits in terms of area and performance are lower than the ones achieved by using concentrated tiles.
- According to ctile, we can say that bristling factor is useful in big networks with more nodes. It can save wires, routers and even it can be helpful to deal with addressing problem of network with many nodes.

This project was a challenging one not because of the complexity of the implemented topologies, but because of the complexity of the PEAK architecture and its definition. Many new signal protocols and connection patterns had to be learnt from scratch and thus posed major challenge to us. In addition, the non-trivial size of the system (32 cores; fully deployed) led to very time consuming (both in time and in memory resources of the PC) simulation processes. Detecting bugs was challenging as the simulation time took millions of cycles to reach where the bug reproduced.

This project opens the doors of PEAK to become more flexible and more powerful when defining and using topologies. All the topologies experienced in this project will be added to the main version of the PEAK architecture and will be adjusted in a parametrizable and configurable environment, so the designer can just indicate which topology the design must use.

As future work, this work will evolve to a deeper analysis of the topologies with explicit NoC related benchmarks which will test the topologies in the full range of traffic, from very low load to saturation. Also, as future work, virtual channels and QoS metrics and mechanisms will be added on top of these topologies.

Bibliography

[1] Gordon E. Moore, ”Cramming more components onto integrated circuits”, *Electronics*, 1965

[2] Jie Chen, Cheng Li and Paul Gillard, ”Network-on-Chip (NoC) Topologies and Performance: A Review”, *NECEC 2011 forum*, 2011, http://necec.engr.mun.ca/ocs2011/viewabstract.php?id=41,

[3] J. Duato and A. Robles et al., ”A comparison of router architectures for virtual cut-through and wormhole switching in a NOW environment”, *Journal of Parallel and Distributed Computing*, pp. 224–253, 2001

[4] Ankur Agarwal et al., ”Survey of Network on Chip (NoC) Architectures and Contributions”, *Engineering, Computing and Architecture*, 2009

[5] Rostislav (Reuven) Dobkin, ”Credit-based Communication in NoCs”, http://eecourses.technion.ac.il/049036/Reuven%20Dobkin%20Credits%2024apr08-print.pdf”, 2007

[6] Lizhong Chen and Timothy M. Pinkston, ”Worm-Bubble Flow Control”, IEEE, *The 19th IEEE International Symposium on High Performance Computer Architecture*, pp. 366 - 377, 2013

[7] TSAR Project, https://www-soc.lip6.fr/trac/tsar

[8] Alain Greiner, ”TSAR: a scalable shared memory many-cores architecture with global cache coherence”, http://www.mpsoc-forum.org/previous/2009/slides/Greiner.pdf, 2009

[9] Duco van Amstel and Benot Dupont de Dinechin, ”Guaranteed Services on the Network-on-Chip of a Manycore Processor”, http://2014.rtas.org/wp-content/uploads/Amstel.pdf, 2014

[10] Jim Jeffers, ”Intel Many Integrated Core Architecture: An Overview and Programming Models”, https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL_Elec_Struct_WS_02062012.pdf, 2012

[11] www.top500.org, ”Supercomputer list of June 2015”, http://top500.org/lists/2015/06/, 2015

[12] www.top500.org, ”Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P”, http://top500.org/system/177999, 2015

[13] George Chrysos, ”Intel Xeon Phi Coprocessor - the Architecture, Intel Corporation”, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[14] Rezaur Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools The Guide for Application Developer*, Appress Media, 2013

[15] Tilera Corporation, ”TILE PROCESSOR ARCHITECTURE OVERVIEW FOR THE TILEPRO SERIES”, http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf”, 2013

[16] Tilera Corporation, ”Tilera Tile Processor User Architecture Manual”, 2011

[17] MIPS, http://imgtec.com/mips/architectures/

[18] J. Flich et al., ”PEAK: Partitioned-Enabled Architecture for Kilocore Processors”, 2014

[19] Stephen Brown and Zvonko Vranesic, ”Fundamentals of Digital Logic with Verilog Design”, McGraw-Hill, 2014

[20] Deepak Kumar Tala, ”Verilog Tutorial”, http://www.asic-world.com