All-to-all
High Performance Processors And Systems

Marco Colombo

Professor: Marco D. Santambrogio
Supervisor: Francesco Peverelli

30/06/2022
A.Y 2021-2022

POLITECNICO
MILANO 1863

Abstract

Many scientific applications, including the computation of the Fast Fourier Transform (FTT), require an all_to_all operation. Performing this operation with a general-purpose processor can be very costly since the memory hierarchy used in such processors is not optimal for the intensive exchange of data required by the all_to_all. However, this operation can be sped up by using a customized hardware accelerator specifically designed for this task. In this project, we present an implementation of such an accelerator, that implements a broadcast algorithm that further explores the parallelism in message transmission. Finally, we have validated the accelerator through RTL software simulation and compared the obtained performance with respect to the available bandwidth.

1 Introduction

All-to-all operation is a collective operation, where each processing element (PE) sends an individual message to every other PE. Collective communication involves global data movement and global control among a group of processors in a multicomputer. Several scientific applications exhibit the need for such communication patterns, such as matrix multiplication, LU-factorization, and basic linear algebra operations.

The goal of the project is to implement a hardware accelerator that speeds up the time required to perform this operation, considering a 2D mesh topology with edge size $n = \sqrt{p}$. To further explore the parallelism in message transmission, the accelerator implements an all_to_all broadcast algorithm, in which some degree of parallelism is achieved not only among the messages passing through different nodes but also among the messages passing through the same node.

In section 2 there is an introduction to the framework used to implement the accelerator and a brief explanation of how the broadcast algorithm works. Instead, in section 3 is shown the architecture of the accelerator itself and the instructions that can be performed with it. Finally, the performance of the new component and possible future works regarding the implementation of the accelerator are discussed, respectively, in section 4 and 5.

2 Background

2.1 Chipyard

Chipyard is an integrated design, simulation, and implementation framework for open-source hardware development. It has been developed at UC Berkeley. Chipyard is open-sourced online and is based on the Chisel and FIRRTL hardware description libraries, as well as the Rocket Chip SoC generation ecosystem. Chipyard brings together much of the work on hardware design methodology from Berkeley over the last decade as well as useful tools into a single repository that guarantees version compatibility between the projects and its submodules. A designer can use Chipyard to build and test a RISC-V-based system on a chip. This includes RTL development integrated with Rocket Chip, cloud FPGA-accelerated simulation with FireSim, and physical design with the Hammer framework.

2.2 RISC-V

RISC-V is an open standard instruction set architecture (ISA) based on established RISC principles. Unlike most other ISA designs, RISC-V is provided under open-source licenses that do not require fees to use. It was initially developed by the University of California Berkeley (UCB) and it started as a project for computer architecture research and education but is aiming also at becoming a standard for industrial implementations. An objective of the RISC-V project is to "be used as stable software development target", as stated in the RISC-V Instruction Set Manual. In fact, the standard defines fixed base integer ISAs in 32 and 64-bit versions plus some optional standard general-purpose extensions. The base integer instruction set was designed in order to include a small number of instructions and reduce the complexity of the hardware required for a minimal implementation. This allows versions to be developed that are suited to a range of applications from small, embedded microcontrollers to desktop personal computers and supercomputers with vector processors. Among all the ways in which it is possible to extend the ISA, the one used in this project is the use of one of the four opcodes

that are reserved for custom extensions (custom-0, custom-1, custom-2,custom-3). These four custom opcodes are the ones that the Rocket core processor uses in the RoCC interface.

This project is developed using **Chisel**, a hardware construction language that facilitates advanced circuit generation and design reuse for both ASIC and FPGA digital logic designs. It is based on top of Scala by adding hardware construction, providing designers with the power of a modern programming language to write complex, parameterizable circuit generators that produce synthesizable Verilog. This generator methodology enables the creation of re-usable components and libraries, such as the FIFO queue and arbiters in the Chisel Standard Library, raising the level of abstraction in design while retaining fine-grained control.

2.3 Rocket Chip

Rocket Chip is an open-source SoC generator designed by the Berkeley Architecture Research (BAR) group of the University of California Berkeley (UCB). Rocket Chip generator is used to configure and generate the Rocket Core. Rocket core is both a processor generator and a library of processor components. As a generator, it can produce a family of processor core designs, with different configuration parameters. The generated processors have the classical five-stage in-order scalar pipeline and can implement the base integer 32-bit RISC-V ISA as well as the 64-bit one, the one used in this project.

2.4 RoCC

The Rocket Custom Coprocessor Interface (RoCC) is an interface designed in order to extend the Rocket Core and allow easy decoupled communications between the core and the attached accelerator. The RoCC interface is divided into sub-interfaces. In particular, the cmd sub-interface connects the core with the accelerator and is used to send commands (the instructions) and the corresponding data to the coprocessor. With these commands the core can also request data from the coprocessor, this data can be sent by the accelerator to the core integer register file through the resp (response) interface. Both the command and the response interfaces are based on the Chisel Decoupled interface. This connection is based on a ready/valid protocol in which the sender drives the valid signal and the data and waits for the receiver to raise the ready signal. The data transfer is fired when both valid and ready are high at the same clock cycle. The core is connected with the accelerator through the Decoupled interface and can send commands to the accelerator by rising the valid signal when a legal custom instruction reaches the writeback stage.

The data bus of the command interface contains the following signals:

- inst the full 32-bit instruction
- rs1 a 64-bit (w.r.t config of this project) data bus holding the value of the register addressed by the rs1 field of the instruction
- rs2 a 64-bit (w.r.t config of this project) data bus holding the value of the integer register addressed by the rs2 field of the instruction

The inst signal is further divided into the fields specified by the RoCC custom instruction. The format of the RoCC instructions follows the R-type format of the riscv ISA, but the opcode section is bound to assume one of the four possible custom opcodes values. The RoCC coprocessor interface

- xd bit is set when inst_rd is a valid destination register: the core wants to receive data in the destination register pointed by inst_rd.
- xs1 bit is set when inst_rs1 is a valid source register: the core is sending the content of the first source register (inst_rs1) in the rs1 data bus.
- xs1 bit is set when inst_rs2 is a valid source register: the core is sending the content of the second source register (inst_rs2) in the rs2 data bus.

When a custom instruction with the inst_xd bit set arrives, the core will expect to receive, at some point, a result to be stored in the register pointed by inst_rd. This means that if a successive instruction uses that register before the value is produced and stored, the core will stall the pipeline to wait for the data from the accelerator. In this case, the accelerator drives the valid signal and the data, while the core only drives the ready signal. The data information includes the following buses:

- rd the five bits field specifying the destination register of the response, must be the same received with the command.
- data a 64-bit (w.r.t config of this project) bus with the data content to be written in the rd register.

2.5 All-to-all communication algorithm [1]

In order to implement an All-to-all personalized exchange, namely, every node sends a distinct message to every other node, it is necessary to define the topology of the network and the algorithm that determines how the data exchange is performed. In particular, from now on we will consider a 2D mesh network topology. With the assumptions of full-duplex channels and all-port capability for each node, the internal structure of a node is composed of four pairs of first-in-first-out buffers: four input buffers and four output buffers, with each input buffer associated with an input channel of the node and each output buffer associated with an output channel of the node. Inside each node, there is also a buffer to store the messages broadcast from all other nodes in the network.

The goal is to achieve the maximum degree of parallelism in message transmission. Considering previous related work, there are mainly two algorithms that implement this operation:

- Saad and Schultz [2]: Each node sends the message along horizontal ring, and then vertical ring.
- Calvin, Perennes, and Trystram [3]: Recursive algorithm.

In these algorithms, some degree of parallelism is achieved among different nodes, but there is no time overlap for the messages passing through the same node. This algorithm further explores the parallelism in message transmission and achieves a good degree of parallelism among the messages passing through the same node. It is optimal in message transmission time and total communication delay is close to the lower bound of all-to-all broadcast. The algorithm is conceptually simple, and symmetrical for every message and every node. The basic idea is performing a controlled message flooding by using a specially designed spanning tree rooted at the source node (broadcast pattern), as shown in Figure 1. The broadcast from node $(x_0; y_0)$ can be logically divided into phases. In the first phase, the message originating from node $(x_0; y_0)$ reaches all its d-neighbors, for $d \geq 1$. Note that the number of phases in broadcast is equal to the diameter of the network. In the case of all_to_all broadcast, each node broadcasts its message to other nodes in a radiant way and the buffers in each node are used to resolve the channel contention. To achieve minimum communication delay, the algorithm allows overlapping of the switching time of one message and the transmission time of another message, and ensure that incoming messages arrived at a node are uniformly distributed to its four output buffers.

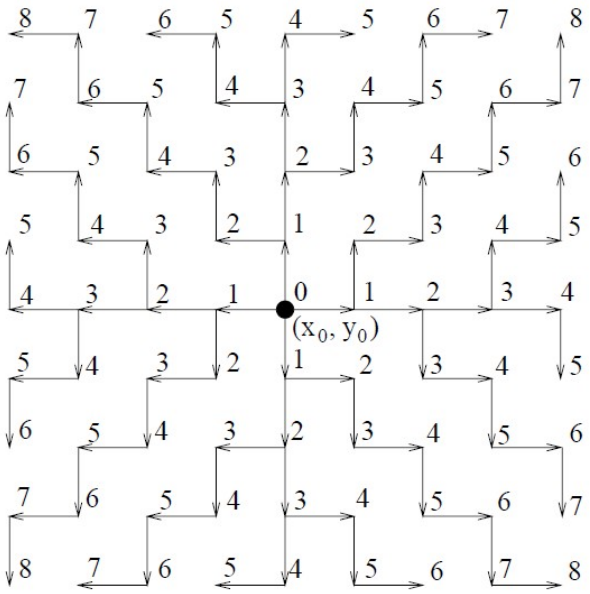


Figure 1: Broadcast pattern

Each node in an infinite mesh can be reached exactly once by using the broadcast pattern. For a mesh, the broadcast pattern is trimmed at the boundary of the mesh.

The following pseudocode gives a higher-level description of the algorithm. All-to-All Broadcast Algorithm:

```
All-to-All Broadcast Algorithm:
for each node (x, y) in the network do in parallel
  for each input and output buffer do in parallel
    case of an output buffer:
      repeat
        remove a message from the buffer;
        send it to corresponding neighboring node
          (enter into the input buffer of the neighbor);
      until no more message arrives at the buffer;
    end case;
    case of an input buffer:
      repeat
        remove a message from the buffer;
        extract the source address of the message, say, (x0, y0);
        copy the message content to the (x0, y0) entry of the
          buffer matrix;
        calculate the addresses of the neighbors to be
          multicast by using the broadcast pattern in Lemma 3;
        for a torus, perform the additional checks in Table 1;
        multicast the message to the output buffers connected
          to the corresponding neighbors;
      until no more message arrives at the buffer;
    end case;
  end for;
end for;
End
```

To give a more formal description of the broadcast pattern, the following functions need to be defined:

$$U(x) = \begin{cases} 1 & \text{if x} > 0 \\ 0 & \text{if x} < 0 \end{cases}$$

$$I(x) = \begin{cases} 1 & \text{if x} > 0 \\ \text{-1} & \text{if x} < 0 \end{cases}$$

$$mod_2(x) = xmod2$$

When a broadcast message originating from node $(x_0; y_0)$ reaches node (x; y), it continues to broadcast to the neighbors of node (x; y) as follows.

Case 1: $x = x_0$ and $y = y_0$. (x; y) broadcasts the message to all of its four neighbors $(x; y + 1), (x; y + 1), (x + 1; y), and (x + 1; y)$.

Case 2: Either $x = x_0$ or $y = y_0$ but not both. (x; y) multicasts the message to its two neighbors $(x_1; y_1)$ and $(x_2; y_2)$

2.1: $x = x_0$ and $y \neq y_0$, i.e. along the y-axis. $x_1, y_1, x_2, and y_2$ satisfy

$$x_1 = x + mod_2(y - y_0 + U(I(y - y_0)))$$

$$y_1 = y + I(y - y_0) * mod2(y - y_0 + 1 - U(I(y - y_0)))$$

$$x_2 = x - mod2(y - y_0 + U - I(y - y_0)))$$

$$y_2 = y + I(y - y0) * mod2(y - y0 + 1 - U - I(y - y0)))$$

2.2: $x \neq x_0$ and $y = y_0$, i.e. along the x-axis. $x_1, y_1, x_2, and y_2$ satisfy

$$x_1 = x + I(x - x0) * mod2(x - x0 + U(I(x - x0)))$$

$$y_1 = y + mod2(x - x0 + 1 - U(I(x - x0)))$$

$$x_2 = x + I(x - x0) * mod2(x - x0 + U(-I(x - x0)))$$

$$y_2 = y - mod2(x - x0 + 1 - U(-I(x - x0)))$$

Case 3: $x \neq x_0$ and $y \neq y_0$, (x; y) sends the message to its neighbor $(x_3; y_3)$ where x_3 and y_3 satisfy

$$x_3 = x + I(x - x0) * mod2((x - x0) + (y - y0) + U(I(x - x0) * I(y - y0))) \quad (1)$$

$$y_3 = y + I(y - y0) * mod2((x - x0) + (y - y0) + 1 - U(I(x - x0) * I(y - y0))) \quad (2)$$

3 Methods and Architecture

The accelerator has been created using Chipyard. In particular, the accelerator is a custom RoCC accelerator added to the existing Chipyard core RocketCore. It is mainly composed of a 2D mesh of processing elements, and the focus of the project is the way in which data are exchanged among the PEs, rather than the operations that each PE may perform on those data. Three operations are available: load, store, and all-to-all. Operations of load and store are implemented to allow testing, but they are not optimized. In the following part of this section, first, there is a description of the instruction set accepted by the accelerator, followed by a more detailed description of the accelerator itself and its components.

3.1 ISA

As said before, there are three available operations. The main structure of the instructions is the following.

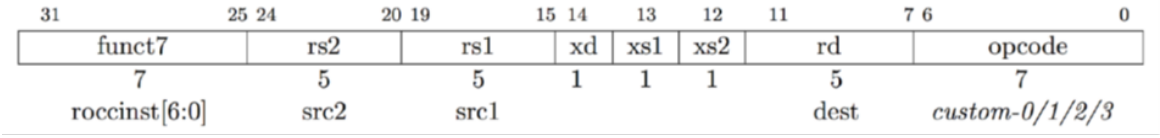


Figure 2: Custom operation

3.1.1 Load

This instruction allows to load into the memory of a specific PE 64 bits of data at a given address. To use this operation, its fields must be:

- **funct7** : 0000001.
- **rs1** : address of register which contains the 64 bit field rs1 of RoCCCommand.
- **rs2** : address of register which contains the 64 bit field rs2 of RoCCCommand.
- **rd** : address of register in which the 64 bit field data of RoCCResponse are written.
- **opcode** : custom-0 (0001011).

In particular, considering rs1 and rs2 fields (64 bits) of the RoCCCommand, the convention is:

- **rs1(63,0)** : data to be written into PE's memory.
- **rs2(15,0)** : x coordinate of the target PE.
- **rs2(31,16)** : y coordinate of the target PE.
- **rs2(63,32)** : memory address of the (x,y) PE at which rs1 will be written.

The value of the response field *data* is fixed to 32 because of testing reasons. However, this value is not representative, since this operation only has to write into PE's memory, and the response is only necessary to notify the processor that the operation has terminated.

3.1.2 Store

This instruction allows to retrieve from the memory of a specific PE, the data present at a given address. To use this operation, its fields must be:

- **funct7** : 0000010.
- **rs1** : not relevant.
- **rs2** : address of register which contains the 64 bit field rs2 of RoCCCommand.
- **rd** : address of register in which the 64 bit field data of RoCCResponse are written.
- **opcode** : custom-0 (0001011).

In particular, considering rs1 and rs2 fields (64 bits) of the RoCCCommand, the convention is:

- **rs1(63,0)** : not relevant.
- **rs2(15,0)** : x coordinate of the target PE.
- **rs2(31,16)** : y coordinate of the target PE.
- **rs2(63,32)** : memory address of the (x,y) PE at which response data are read.

The value of the response field *data* contains the 64 bits of data present at the specific address of the memory of the specific PE.

3.1.3 AllToAll

This instruction allows to perform the all-to-all operation. Its fields must be:

- **funct7** : 0000011.
- **rs1** : address of register which contains the 64 bit field rs1 of RoCCCommand.
- **rs2** : not relevant.
- **rd** : address of register in which the 64 bit field data of RoCCResponse are written.
- **opcode** : custom-0 (0001011).

In particular, considering rs1 and rs2 fields (64 bits) of the RoCCCommand the convention is:

- **rs1(15,0)** : DIM_N.
- **rs1(63,16)** : not relevant.
- **rs2(63,0)** : not relevant.

The value of the response data field is fixed to 35 because of testing reasons. However, this value is not representative, since this operation performs the all-to-all and the response is only necessary to notify the processor that it has terminated.

3.2 AllToAllAccelerator

This is the top-level component, which is used in the AllToAllConfig class to instantiate the accelerator. It extends module LazyRoCC, takes the object OpcodeSet as a parameter, and overrides the lazy val *module* with the module called AllToAllAcceleratorModule. The latter extends LazyRoCCModuleImp and takes the required implicit parameters. Since it extends LazyRoCCModuleImp, the IO consists of the interface RoCCIO. Signals of interrupt and exceptions of this interface are not used in this implementation. In particular, interrupt has a fixed value to false. This module, as shown in Figure 3, contains module AllToAllModule. The IOs of the two modules are properly connected to ensure the correct communication.

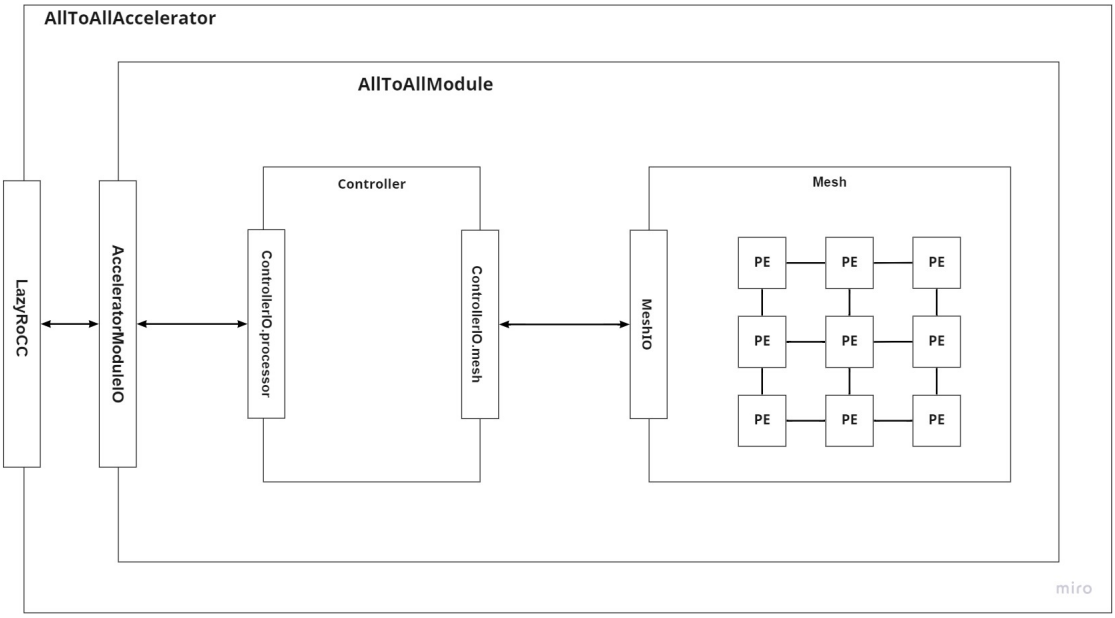


Figure 3: Accelerator

3.3 AllToAllModule

Although this module is logically very similar to the AllToAllAcceleratorModule, it is actually due to an important design choice of the accelerator's architecture. This module exposes as IO the interface AcceleratorModuleIO. This interface is very similar to RoCCCoreIO and is directly connected to the IO of AllToAllAcceleratorModule. This is particularly useful since it allows to perform IO testing using PeekPokeTester. Indeed, testing directly the accelerator with the actual LazyRoCC interface is not possible because of the implicit parameters, that are automatically passed only when the module is generated using the real configuration with the RocketChip core. This allows to easily and accurately test the module, by simulating the actual interface used by the processor. The constructor of this module has the following parameters:

- n : Int, it represents the size of the edge of the mesh that will be instantiated.
- cacheSize : Int, it represents the dimension in number of lines of 64 bits of the memory that each PE has.
- queueSize : Int, it represents the size of the input/output queues that each PE has on its edges.

It contains the two main components of the accelerator: Controller and Mesh, and properly connects all the signals of their IO interfaces.

3.4 Controller

Controller is the module in charge of receiving signals from the processor and forwarding commands to the Mesh, which in turn contains all the PEs.

The IO interface of the controller (ControllerIO) is divided into two main parts: the first one is called ControllerIO.processor and the second one is called ControllerIO.mesh. ControllerIO.processor is the part in charge of receiving the commands from the processor and sending it the responses. It has the same signals as AcceleratorModuleIO interface. ControllerIO.mesh, instead, is the part of the Controller's interface in charge of managing the signals that Mesh module needs to perform the correct actions and to allow the Mesh to send responses. Both command and response are Decoupled interfaces in both the two cases described above.

Controller module mainly contains an FSM which implements the correct sequence of command/responses that the accelerator can receive/send to the processor. This module is necessary to allow a good decoupling between the structure of the commands/responses that must be exchanged with the processor and the ones used to communicate with the Mesh. This means that it is possible, for example, to easily change the opcodes of the operations or to change the interface used by the processor, without having to modify the structure of the Mesh and of all the PEs.

3.5 Mesh

Mesh is the module in charge of managing the PEs and generating the structure of the 2D mesh, with the correct connections among PEs and the connection of each PE to the Mesh itself. The IO interface is called MeshIO and has mainly three bundles of signals: command (cmd), response (resp), and the busy signal. Cmd and Resp are decoupled interfaces. When a command is received from the controller, it is sent in broadcast to all the PEs of the Mesh. In order to manage the responses of the PEs, there is a priority multiplexer that has as selector a signal called write_enable, raised by the PE that wants to write in output. In this way, the response forwarded from the Mesh to the Controller will have as payload the data from the desired PE. The busy signal of the whole Mesh is the logical OR of the busy signals of all the PEs. Similarly, MeshIO.cmd.ready and MeshIO.resp.valid are the logical AND of the corresponding PEs signals. This is because, in this implementation, it is possible to send a command to a PE only if all of them are ready. In particular, it is possible to perform load/store with only one PE at a time. This design choice is not only due to the fact that load/store operations are not the main goal of this project but also because of another important reason: since all-to-all operation needs to exchange a personalized message, having an operation that performs load/store with all PEs at the same time is not useful. In fact, in order to test the correct behavior of the all-to-all operation, a high level of granularity is necessary to prepare the memory with all the personalized messages and to check whether, after the operation, the memory contains the correct data in the correct position.

When instantiating Mesh module, also all the PEs constructors are invoked. In particular, each PE has an x coordinate and a y one. These coordinates are important for the correctness of the forwarding algorithm. Therefore, in Figure 4 there is an example, with n = 3, that shows the convention used to assign coordinates w.r.t. the position of the PE into the mesh. Notice that each couple (x,y) can be easily converted into a sequential number, which is shown in the figure into each PE.

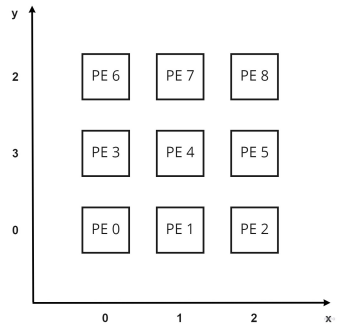


Figure 4: Coordinates convention

3.6 AllToAllPE

AllToAllPE is the most important module of the accelerator. It is the processing element that represents each node of the 2D mesh.

The constructor of this module has the following parameters:

- n : Int, it represents the size of the edge of the mesh that will be instantiated.
- cacheSize : Int, it represents the dimension in number of lines of 64 bits of the memory that each PE has.
- queueSize : Int, it represents the size of the input/output queues that each PE has on its edges.
- x : Int, it represents the x coordinate of the PE into the mesh.
- y : Int, it represents the y coordinate of the PE into the mesh.

The IO interface is very important since it defines the way in which each PE communicates with the adjacent ones w.r.t the mesh topology. It is divided into two main parts. The first one is related to the communication with the Mesh and it has the same fields (busy, cmd, resp) described in 3.5 . It is useful to send commands to the PEs and to receive their responses. The second part of the interface defines the interconnections between PEs. Each PE has four edges, and for each edge, there is a bidirectional connection with the adjacent PE. In particular, there is an **in** bundle and an **out** bundle. The **out** bundle is a Decoupled interface of type OutputPE, while **in** bundle is a Flipped Decoupled interface, still of type OutputPE. The interface OutputPE allows the exchange of one message or "packet", and each packet has the following fields:

- x_0 : ($\lceil \log_2 n \rceil$ bits) x coordinate of the PE at which the packet is generated.
- y_0 : ($\lceil \log_2 n \rceil$ bits) y coordinate of the PE at which the packet is generated.
- x_dest : ($\lceil \log_2 n \rceil$ bits) x coordinate of the PE at which the packet is sent.
- y_dest : ($\lceil \log_2 n \rceil$ bits) y coordinate of the PE at which the packet is sent.
- pos : (16 bits) the sequence number of the packet among the DIM_N packets.
- data : (64 bits) payload.

Each PE has an FSM that manages the commands and the responses, and implements the load operation, to write into its memory, and the store one, to read from its memory. However, those are simple operations and so we will not go into detail, but rather will be described the behavior of the all-to-all operation.

When an all-to-all command arrives at the PE (all the PEs belonging to the Mesh receive the signal at the same time), the operation starts and the busy field remains true until all the data exchange has been completed; at this point, the response valid signal is raised to inform the Mesh, and consequently the Controller, that the operation has been performed correctly. In order to understand how data are exchanged between PEs, it is important to show the memory layout. Each PE has its own register memory, composed of lines of 64 bits. The all-to-all operation expects data to be present in memory with a specific layout since there must be a correlation between the address at which data are stored and the coordinates of the PE that has to receive those data. Moreover, the all-to-all operation does not overwrite the part of memory that contains the messages to be sent, but data received from the other PEs of the mesh are stored starting from address $p * DIM_n - 1$, where $p = n^2$. Data are not stored in the same memory space because the arrival order of packets is not fixed and thus using two separate spaces ensures that it never happens that a line is overwritten before being sent. Figure 5 gives a graphical and intuitive representation of this layout.

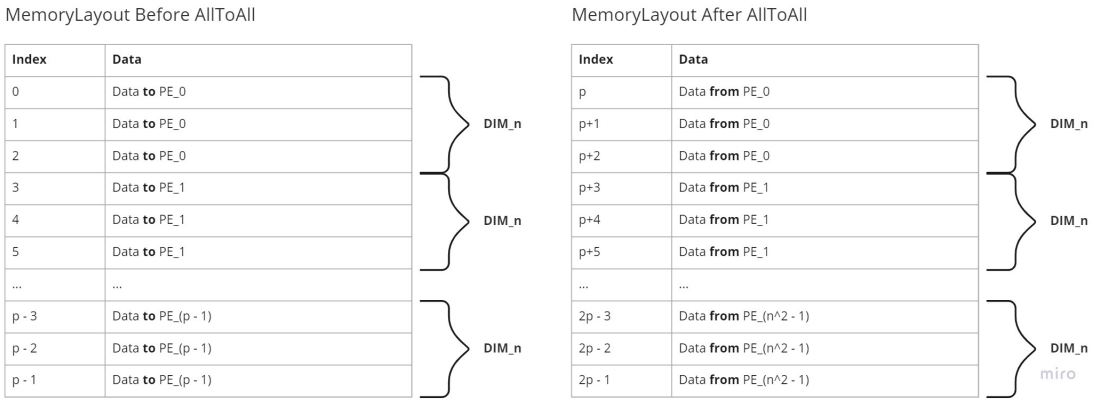


Figure 5: Memory Layout

In order to perform correctly the all-to-all operations, in each PE is embedded a slightly different version of the communication algorithm described in section 2.5, but maintaining the same general logic and the same SPT. The original algorithm is designed to perform all-to-all broadcast, while in this project we consider all-to-all personalized. In particular, the original algorithm does not have a mechanism to decide, given the coordinates of the source PE of the packet and the coordinates of the destination PE of the packet, to which of the possible neighbors the packet must be forwarded. In fact, the original algorithm only gives formulas to compute, given the coordinates of the source node of the packet the coordinates of the neighbor nodes that represents the next hop w.r.t. the SPT. In addition, during the initial phase (when source coordinates = current node coordinates) the algorithm does not specify a way to decide, given the coordinates of the destination node of the packet, to which of the four neighbors the packet must be forwarded, but only says to forward it to all neighbors. These two mechanisms are very important for an efficient implementation of the data exchange because they allow forwarding each packet only through the correct path, so not wasting any bandwidth. For this reason, two modules are used to fulfill those missing features: Dispatcher module and GenerationDispatcher module respectively.

Another important component used in the PE is the IndexCalculator module, which is further explained after, but it basically generates all the memory addresses from which the PE has to read data and the coordinates of the respective destination PEs.

When the all-to-all operation begins, each PE starts reading data from memory at addresses given by the IndexCalculator. For performance reasons, PE reads a block of four memory lines at a time and stores it into a vector of registers, together with the respective source x and y coordinates, the x and y destination coordinates (computed by GenerationDispatcher), and pos, so the entire packet is

ready to be sent in output. Each element of the vector also has a valid bit, which is set to true when data are read from memory, and it is updated (set to false), whenever the correspondent packet has been sent. This mechanism is implemented with a small FSM that continues to fill the registers, wait until all packets have been sent, and fill them again until the module IndexCalculator raises a signal called lastIndex. This tells the PE that all data have been correctly sent, so the FSM terminates. It is important to specify that there is no guarantee that the four messages preset simultaneously into the vector of registers have as direction to be forwarded to the four different edges of the PE. In other words, it can happen that multiple elements of the vector (also all of them), contain messages that must be enqueued into the same output queue. For this reason, a multiplexer is necessary to give only one message at a time. In particular, this component is a custom multiplexer called PEMux (see section 3.10).

Simultaneously, each PE receives the messages in input from the neighbors, and so it has to handle them. In particular, each side of the PE has a Dispatcher module, which tells the PE which is the direction (the neighbor) to which the received message must be forwarded, given the information contained in the message itself.

Since, at each clock cycle, the PE has to manage the packets coming from the four sides and also the ones generated by the PE itself, a mechanism to control the flow of all these data is necessary. To be specific, messages received from a side can potentially be forwarded to all the three remaining sides (or can be written into PE's memory in case currentPE = destPE). Notice that, due to the SPT, it never happens that a message is sent in output to the same side at which it has been received. Moreover, also packets read from memory can be potentially forwarded to all four edges. To manage this complexity, each output queue accepts as input the output of a round-robin arbiter, as described in Figure 7. Note that in Figure 7, for readability purposes, only the arbiter on the right side is shown, but the same happens also on the remaining sides. In this way, only one message per clock cycle is passed to the queue. Round robin behavior has been chosen to ensure that all the input queues and also the data generated from memory have the same probability to be enqueued in output, thus not prioritizing one of the four sources. This policy can be easily changed by modifying the logic of the arbiter, for example in case we want to prioritize data from memory instead of the ones that arrive from input queues. More in detail, the arbiter has four input channels: the first one is dedicated to data coming from memory, while the remaining three channels are dedicated to the three input queues that can potentially push data.

All the mechanism to fire the enqueue/dequeue is automatically managed by the decoupled interfaces that connect the different components, namely the ready and valid bits are automatically set depending on the state of the queues and depending on the actual direction to which messages must be sent.

Thanks to this architecture, even when the three input queues contain a message to be forwarded to the same output queue, and also the messages generated by memory must be forwarded to the same queue, the PE correctly handles the situation by letting pass one message per cycle. It is extremely important because, due to the big amount of packets that are generated by this operation, situations like the one just described can happen, and so they must be accurately managed to ensure the correct behavior of the all-to-all.

When the PE has no more messages neither into input queues nor into output queues, and when the data from memory have all been sent, the PE sets to false its own busy signal. When all the PEs becomes free, it means that all messages have reached the destination, thus the all-to-all operation is completed. After this, a response is generated to inform the Mesh, and consequently the controller and the processor, that the operation has correctly terminated. At this point, all PEs are free and so ready to receive new commands.

In figure 6, it is shown a high level schema of the part of the PE in charge of handling the generation of the messages. Notice that in the figure are shown only the connection with the right side of the PE for readability purposes, but similar connections are present also on the other three sides. In figure 7, instead, it is shown a high level schema of the part of the PE in charge of handling the forwarding of the messages. As before, only one side is shown, but similar connections are present also on the other three sides.

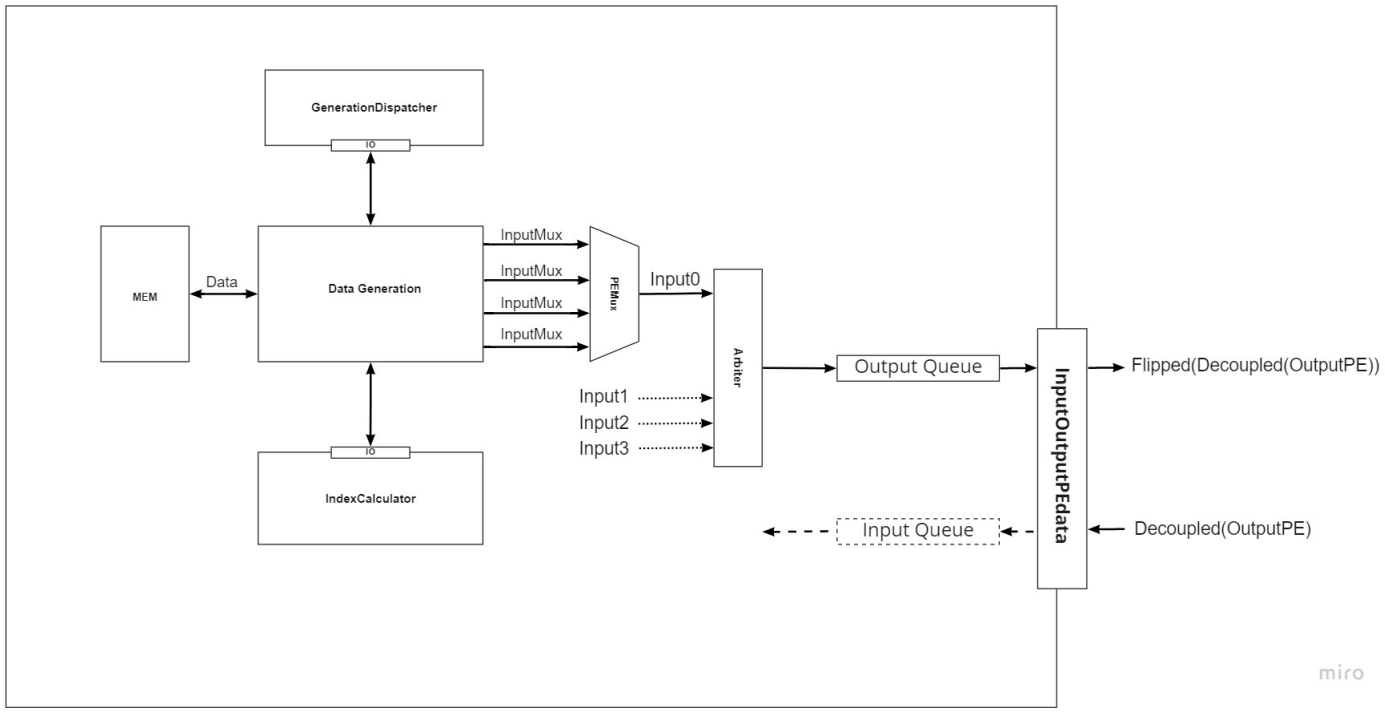


Figure 6: PE generation

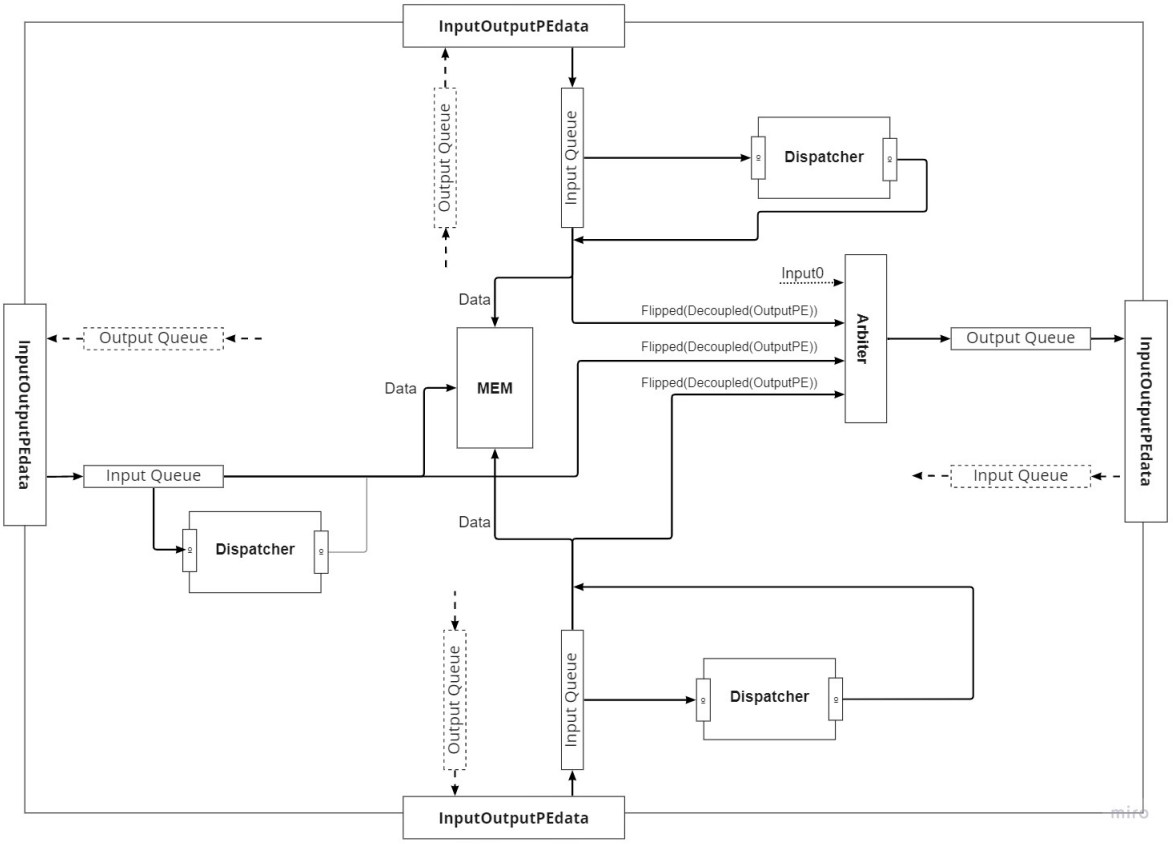


Figure 7: PE forwarding

3.7 IndexCalculator

This component is the one in charge of generating the memory addresses from which data put into the messages have to be read. Moreover, it also generates the correspondent x and y coordinates of the PE that must receive the message and the sequence number of the packet among the DIM_N ones. Each computed address also has a valid bit, this is because this component generates groups of four addresses (and relative information) at a time and, in case addresses to be generated are not multiple of four, the remaining addresses must have the valid bit set to false; in fact, those addresses are not representative of any correct memory line and if the valid signal was high, the message would be sent.

This module has three input signals. The first one is a reset signal: when it is raised, the component restarts the computation of the addresses from an initial state. The second signal is enable: when raised the module generates four new addresses and continues with the computation, when it is false the output of the module remains stable and unchanged. The last input signal is DIM_N and it is useful to parameterize the computation of the addresses w.r.t the DIM_N value contained in the all-to-all operation. This signal is sampled only when the reset signal is high, otherwise, it has no effects.

IndexCalculator component must guarantee two fundamental properties: all the correct addresses must be generated, and they must be generated only once, to guarantee that every message is sent once and only once. When all addresses have been generated, lastIndex signal is raised to inform the PE that all messages have been generated. The order in which addresses are generated may have an impact on the performance of the all-to-all operation, since, if the four destination PEs related to the four generated addresses have as next-hop w.r.t. the forwarding algorithm the four different directions (up, down, left, right), at each clock cycle it is possible to push the four messages to all the four output queues, without having bottlenecks. The implementation of such generation order, however, has not been addressed in this project. Nevertheless, this component is designed to be easily modified. In fact, to change the order of generation, it is enough to create another module with the same IO, and that guarantees the two properties mentioned above, and substitute the old module with the newer one in the code of PE module.

3.8 GenerationDispatcher

This module allows computing the direction (up, down, left, right) to which the messages generated by the PE must be sent. The IO of this module takes as input the coordinates of the PE in which it is instantiated and the coordinates of the destination PE and sends in output the direction of the next hop. The idea behind the computation of the direction is the following: by observing Figure 1, it is possible to notice that there are four regions in the Cartesian plane delimited by the two bisectors and, given the destination coordinates, it is possible to easily identify the correct region to send the message to.

3.9 Dispatcher

This module allows computing the direction (up, down, left, right) to which the messages received in input must be forwarded. The IO of this module takes as input the coordinates of the PE in which it is instantiated, the coordinates of the PE in which the message is generated, and the coordinates of the destination PE, and sends in output the direction of the next hop. In addition, in case the destination PE is the current one, a signal called this_PE is raised, in order to inform the PE itself that the message must be stored in memory and not forwarded. This component actually implements the forwarding algorithm and the related broadcast pattern shown in Figure 1. The idea behind the computation of the direction is to consider the different scenarios in which a message could be. This is basically implemented by considering the four regions as different cases. Since the behavior of the algorithm is very similar for each region, we just consider the one that contains all the destination PEs of the packets forwarded on the right side of the starting PE (x_0, y_0). In particular, if the message is on the positive semiaxis x, if the x coordinate of the current PE is odd (easy to verify in hardware), if $x_{dest} <= y_{dest} + x_{PE}$) then forward on the upper side, else forward on the bottom side. Instead, if the message is not on the positive semiaxes x, so $x \neq x_0 and y \neq y_0$), the next hop can be easily computed by using the formulae 1 and 2. The computation of these formulas is optimized in hardware. Similar reasoning can be made for the other three regions.

3.10 PEMux

This module is basically a priority multiplexer that has as input the four packets generated by the PE and the valid bit related to each packet. If the valid bit is true, it means that the packet must be sent. As mentioned before, since multiple packets may be forwarded to the same output queue, this component is useful to send in input of the output queue only one packet at a time. The first message of the four ones with valid bit equals to true, is the one passed in output. If all valid bits are equal to false, a dummy message is passed in input. Notice that, in this case, no messages would be forwarded, since the decoupled interfaces will not fire. However, the dummy message is useful for testing reasons, to validate the behavior of the component.

4 Results

The custom accelerator has been successfully tested using both *Verilator* and *ChiselTest*.

Verilator is an open-source LGPL-Licensed simulator maintained by Veripool and can be used in the Chipyard framework to build and execute simulations. Chiseltest is the batteries-included testing and formal verification library for Chisel-based RTL designs.

The goal of the testing is to check the correctness of the all-to-all operation and to benchmark the component, in order to study its performance. Different tests have been executed. In particular, since the component is parametric w.r.t. the size of the mesh, it has been tested for different values of $n = \sqrt{p}$) and, for each value of n, the all-to-all operation has been performed with different values of DIM_N.

Considering the correctness of the operation, many tests have been performed by loading all the personalized messages in each PE, such that every single message is unique. This aspect is very important because the all-to-all operation moves a great quantity of data, and it is fundamental that each message is different from all the other ones, in order to be sure that, once the operation has been completed, all data have been moved to the correct PE and at the correct memory line.

Considering performance, for each test, the number of cycles required by the accelerator to perform the operation has been sampled.

To better understand the meaning of the reported results, some representative dimensions are explained in the following paragraph.

The total number of bits that must be exchanged during an all-to-all operation can be computed with the following formula:

$$TotBits = (Pkt_Header + Pkt_Payload) * DIM_N * p * (p - 1) \tag{3}$$

This is because each of the $n^2 = p$ PEs has to exchange p-1 (exclude data to the PE itself) multiplied by DIM_N packets.

The dimension in bits of the header of the packet varies depending on n, since the number of bits required to represent the coordinates of the PEs is $\lceil \log_2 n \rceil$ as shown in section 3.6, while the dimension of the payload is fixed to 64 bits. In formulae:

$$Pkt_{header} = \lceil \log_2 n \rceil * 4 + 16 \tag{4}$$

$$Pkt_{data} = 64 \tag{5}$$

$$Pkt = Pkt_{header} + Pkt_{data} \tag{6}$$

The total available bandwidth, considering the bidirectional connections among PEs, can be computed with the following formula:

$$AggregateBandwidth = 4 * (Pkt_{Header} + Pkt_{Payload}) * n * (n - 1) \tag{7}$$

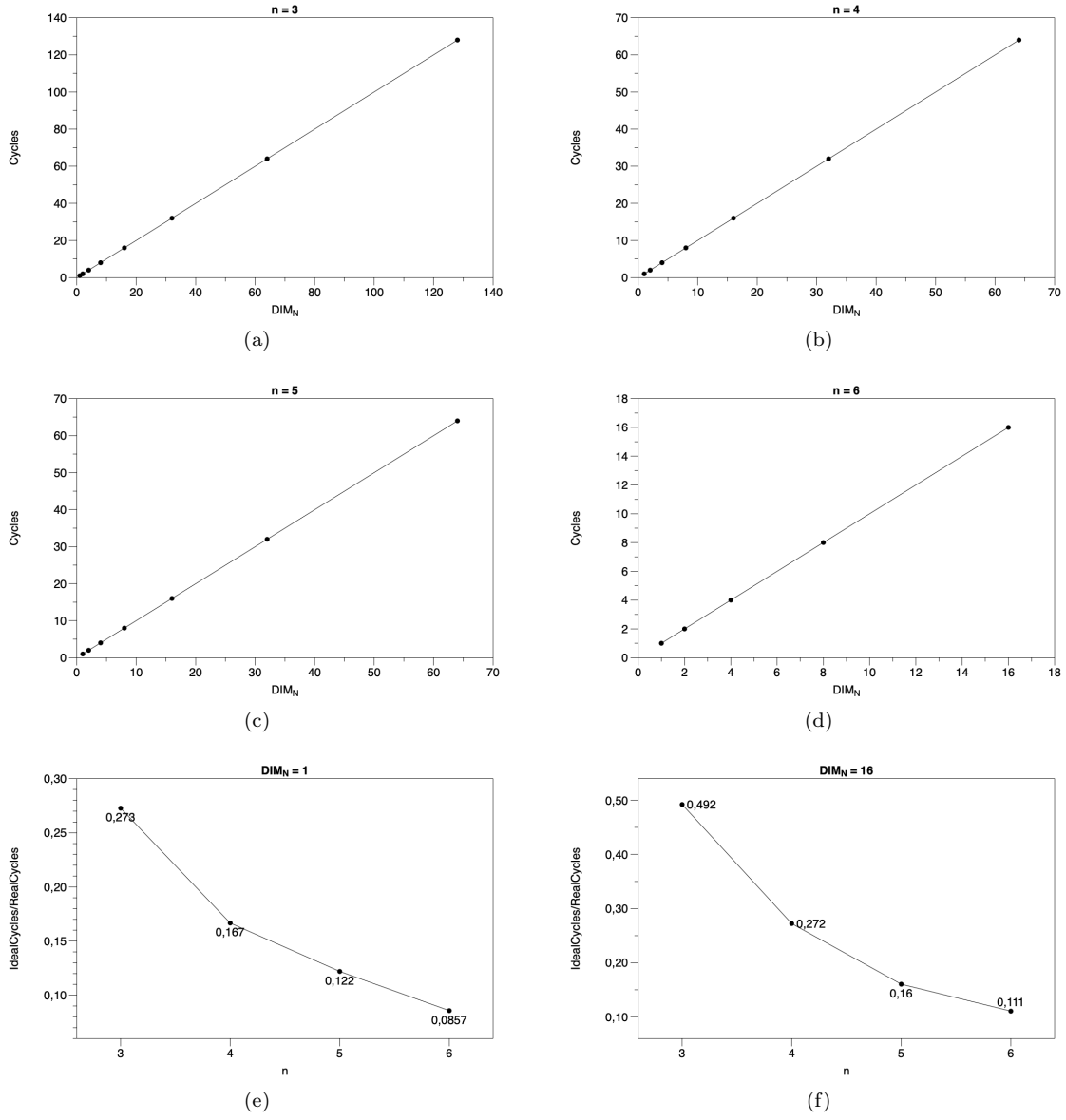
This is because considering the mesh topology, there are $2 * n * (n - 1)$ connections and each connection contains a number of bits equal to the size of the packet multiplied by two since it is bidirectional.

In order to evaluate the results, we consider a lower bound of number of steps that the algorithm must perform before terminating and compare it with the values obtained by testing the component. In particular, considering the mesh topology and the forwarding algorithm, in an ideal case in which

the bandwidth of the channels between PEs is unlimited, given a PE i, all other PEs would receive all the messages from i in a number of clock cycles equal to the Manhattan distance from PE i to the furthest one from i itself. This is because, if the bandwidth is unlimited, when the message arrives at the furthest destination, all the other messages have been received too, since their destination is closer. Considering i the PE on a corner of the mesh, this distance is equal to $2 * n$. In addition, this value is multiplied by the value of DIM_N, so we obtain that an ideal number of clock cycles would be $IdealCycles = 2 * n * DIM_N$.

The following table shows the obtained results. The tests have been performed with a configuration such that input and output queues were big enough not to represent a bottleneck for the algorithm. Notice that the last column contains the value $IdealCycles/RealCycles$, and it indicates how the performance of the accelerator are close to the one that is possible to obtain considering the lower bound of steps described above. The higher this value, the better the performance.

n	**DIM_N**	**Pkt_header**	**Pkt_data**	**Tot[bytes]**	**RealCycles**	**IdealCycles/RealCycles**
3	1	24	64	792	22	0,273
3	2	24	64	1584	35	0,343
3	4	24	64	3168	57	0,421
3	8	24	64	6336	99	0,485
3	16	24	64	12672	195	0,492
3	32	24	64	25344	377	0,509
3	64	24	64	50688	784	0,490
3	128	24	64	101376	1493	0,514
4	1	24	64	2640	48	0,167
4	2	24	64	5280	78	0,205
4	4	24	64	10560	132	0,242
4	8	24	64	21120	249	0,257
4	16	24	64	42240	470	0,272
4	32	24	64	84480	912	0,281
4	64	24	64	168960	1818	0,282
5	1	28	64	6900	82	0,122
5	2	28	64	13800	137	0,146
5	4	28	64	27600	253	0,158
5	8	28	64	55200	492	0,163
5	16	28	64	110400	998	0,160
5	32	28	64	220800	1959	0,163
5	64	28	64	441600	3875	0,165
6	1	28	64	14490	140	0,086
6	2	28	64	28980	238	0,101
6	4	28	64	57960	453	0,106
6	8	28	64	115920	880	0,109
6	16	28	64	231840	1736	0,111



Considering the plots a,b,c,d, we can conclude that, given a fixed value of n, the number of cycles required by the all_to_all operation to complete is directly proportional to the value of DIM_N. In particular, the ratio between the number of cycles and DIM_N is almost constant: if DIM_N doubles, the number of cycles (almost) doubles too. The fact that, by passing from $DIM_N = 1$ to $DIM_N = 2$, the number of cycles is less than doubled, can be explained by considering that, when operation starts, there is a sort of startup overhead, which is due to the fact that with the actual IndexCalculator, there is a high probability that the four computed destinations belong to the same region of the broadcast STP. The higher the value of DIM_N, the lower the impact of this delay.

Considering instead plots e and g, it is evident that performance decrease fast w.r.t. value of n. This is because of the mesh topology itself. In fact, the number of messages that must be exchanged is $O(n^4)$, while the available bandwidth is $O(n^2)$, so performance in terms of clock cycles decrease quadratically w.r.t. the value of n, and it confirms the data obtained with tests. However, considering the all-to-all operation, the mesh topology is a good trade-off between the number of connections among the PEs and the performance that is possible to achieve. To improve this result, a different topology is required, and in particular a topology with more connections and consequently more available bandwidth.

5 Future work

A possible future work is to validate the design of the custom hardware accelerator through FPGA prototyping, also to further explore the scalability of this component by testing configurations with values of n higher than six.

In addition, it is possible to test the behavior of the accelerator using different types of memories into the PEs. In fact, considering the actual implementation each PE has a register memory, but it could be improved by using RAM memories.

Another possible improvement, as already explained in section 3.7, can be to change the order in which addresses are generated by the IndexCalculator module. In particular, the new order should be such that the four next hops, related to the four generated addresses, are in the four different directions as many times as possible. In this way, each PE could better exploit the available bandwidth, especially in the initial phase of the all-to-all operation.

6 Conclusions

In this project, we have proposed a new hardware accelerator that implements a personalized all-to-all operation, considering a 2D mesh topology, where each node is a processing element. The broadcast algorithm has been selected to achieve a high degree of parallelism, not only among different nodes but also among messages passing through the same node. In addition, this accelerator is parametric to the size of the mesh. Finally, the results proved that the 2D mesh topology scale well w.r.t. the value of DIM_N, but the available bandwidth does not increase proportionally with respect to the quantity of data to be exchanged with increasing n.

References

[1] Yuanyuan Yang and Jianchao Wang. Efficient all-to-all broadcast in all-port mesh and torus networks. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 290–299, 1999.

[2] Youcef Saad and Martin H Schultz. Data communication in parallel architectures. *Parallel Computing*, 11(2):131–150, 1989.

[3] C. Calvin, S. Perennes, and D. Trystram. All-to-all broadcast in torus with wormhole-like routing. In *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 130–137, 1995.