RISC Core Torus Interconnect

Giuseppe Calderonio

A.Y 2022-2023

Abstract

A Chiesel-based implementation of a torus interconnect topology for data movement is presented in this report.
In particular, instead of using a standard hierarchical memory design that, in some domains can lead to sub-optimal performances, a bi-dimensional torus topology is proposed, where each matrix field is a Processing Element(PE), and prototyped via the Chipyard SoC generator as a tightly-coupled hardware co-processor; the main operation provided by the accelerator concerns data movement, indeed it has the possibility to exchange data with its neighbour through a command called "exchange".
In this way, each PE can be seen as an independent computing element, but at the same time whenever necessary can share data with other PEs, improving the overall data movement bandwidth. This project will show a toy example of a possible data-movement protocol, thus the implementation provides a bi-dimensional torus that can load and store, focusing on data exchange. The accelerator indeed has the possibility to store and load elements inside the memories with a broadcast approach, and it allow all the PEs to exchange data blocks to all their "neighbours", where a neighbour of a PE is any other PE physically linked to it through the torus topology. The performances of the operation do not depend on the number of elements of the matrix (like a common load, as reported later in this document), but only on the number of data blocks to exchange.

1 Introduction

In the context of high-performance processing systems, is becoming more and more common to design specific specialized hardware to solve hard tasks, and being able to design efficient specific-domain solutions that can be extended instead of re-designing them from scratch can reduce the implementation costs.
In particular, the domain of interest for this project is an environment where the problem involves a set of parallel computations organized as a matrix of elements interconnected in a torus topology, that need to share data but only between connected elements, and for each exchange it's necessary to exchange as much data as possible, so the required bandwidth becomes critical. An example of such an environment is represented by Molecular Dynamics (MD), a computer simulation method for analyzing the physical movements of atoms and molecules where computational power is a design constraint; spatial decomposition algorithms allow the load of numerically solving Newton's equations of motion for a system of interacting particles to be distributed among PEs that represent a chuck containing a subset of the total number of molecules, and then to exchange the results of computations among those chunks, and again until the simulation ends. This report presents the design choices and implementation of a torus interconnect structure, mainly focused on data movement, showing its characteristics, performances, strength and weaknesses.

1.1 Cache Hierarchy

Before discussing about the structure proposed, it's necessary to understand why a classical hierarchical memory approach may be suboptimal in order to address the problems of this specific domain.

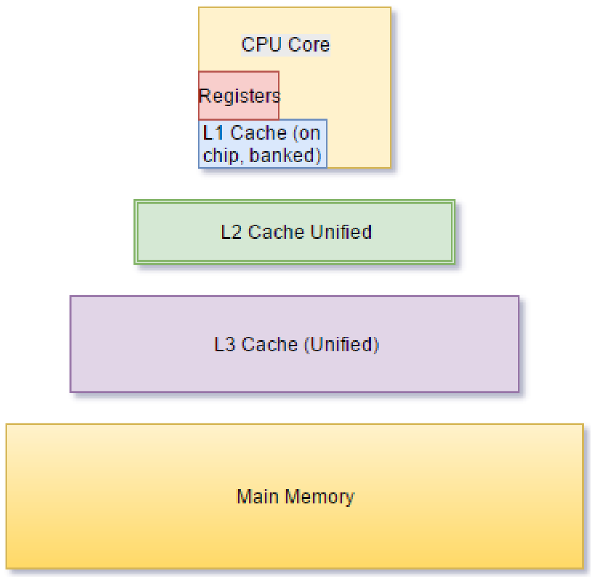

Figure 1: CACHE HIERARCHY: By Kbbuch - Own work, CC BY-SA 4.0

As shown in Figure 1, memories closer to the CPU are characterized by a lower access time, but can store a low amount of data, while going towards most distant memories with respect to the CPU, storing capacity increases, but latency increases.
For example, assuming a CPU running on a personal laptop, memory hierarchy could be divided in this way:

- L0: represent registers, the closest memory structure to the CPU
- L1: represent the First layer of cache, they are directly installed in the CPU, this means the access time is very low, usually 64KB/core
- L2: represents the second layer of cache, usually 256KB/core
- L3: represents the Third layer of cache, but this one can be shared among different CPUs, so it has to be bigger, usually 16MB
- Main memory: represents the higher layer, where all the data usually are stored

This is just an example, other implementations are possible, since domains an applications are many.

Although this approach is widely used and works well on average, in this specific domain of parallel computations where data need to be exchanged only to a subset of the whole system organized as a matrix, other solutions can be more efficient in terms of total bandwidth of data exchanged.
for this reasons, a new approach is proposed.

1.2 Torus Structure

In order to reach better performances in terms of bandwidth a new memory structure is proposed. This structure is based on a bi-dimensional matrix, where each element is a Processing element, having 5 physical memories and a well defined logic to access them, and it is connected with its neighbour through a set of queues.

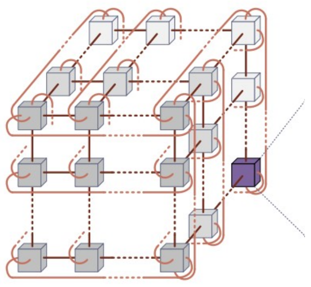

Figure 2: three-dimensional Torus Topology: [1]

Figure 2 shows a three-dimensional torus topology.
The main instruction provided by the ISA of the component is the exchange command "exchange(src, n_data, dest)" where:

- src is the source address (16 bits wide)
- n_data is the size of the data to exchange, where each memory block is 16 bits wide (16 bits wide)
- dest is the destination address (16 bits wide)

The effect of this command will be a broadcast of n_data blocks for each PE; an example is proposed to better understand at high level how the exchange works, considering each PE as a single logical memory, while as will be shown later in th report, each PE is an hardware component that manages 5 separate memories. For example, assuming this specific configuration of a bi-dimensional torus 2x2 where each connection is a queue (not represented for simplicity) :

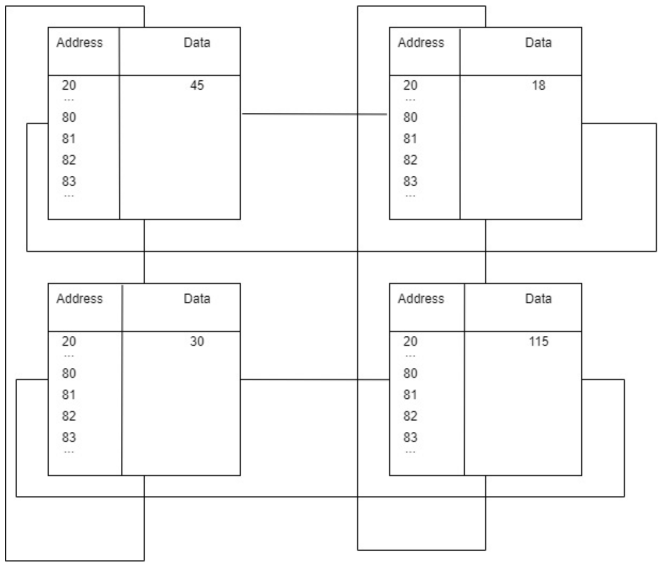

Figure 3: Configuration Before exchange

After applying the command "exchange(src = 20, n = 1 ,dst = 80)", in 2 clock cycles (n + 1) the resulting configuration will be:

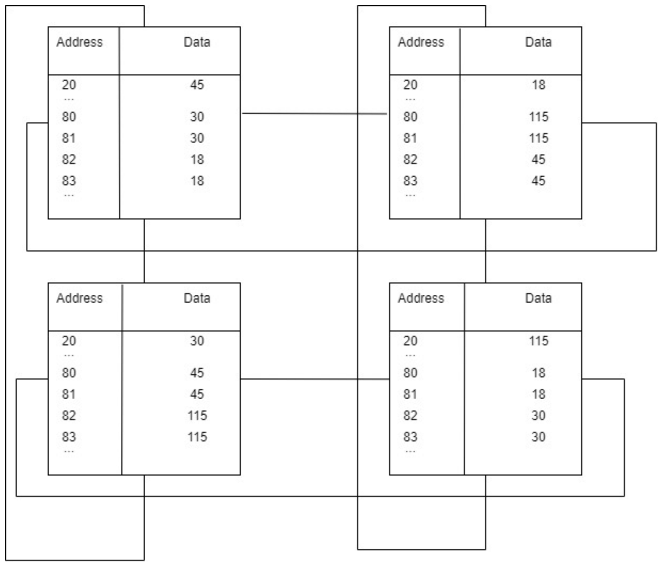

Figure 4: configuration after exchange (after 4 clock cycles)

As it is possible to see from Figure 4, the data has been broadcasted (even if this is a specific corner case in which the data coming from up and down sides are equal, and the same for left and right sides). The latency of the operation depends only on the parameter n = 1, and not on the number of PEs, as it will be shown later in the report

Main benefits of this approach are:

- bandwidth: the amount of data that can flow in a queue, is configurable at design time, at is half of the width supported by the channels (could be expanded to the total width, as described in future work section)
  for example, if the architecture works with 32 bits, the bandwidth of a single channel is of 16 bits per cycle
  considering that the number of PEs is n, and the neighbours are 4 for each of them, and that could be sent 16 bits/cycle, the potential exchange of data is much more powerful.
- 2D Structure: instead of handling the connections with a bus, data is exchanged among neighbours, avoiding potential latency

2 Metodology

This section will present the technical tools used for the development of this project.

2.1 Chipyard

Chipyard is a framework for designing and evaluating SoC hardware. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip.

2.1.1 Generators

A Chipyard generator is an implemented hardware component that can execute operaions and be simulated / synthesized.

- Processor cores : a set of RISC-V is available and implemented in Chipyard; for the torus accelerator development, an in order RISC-V core has been used to simulate and test it.
- Accelerators: Some ready to use accelerators are provided in the Chipyard github repository (for instance, A fixed-function accelerator for the SHA3 hash function, or a matrix-multiply accelerator targeting neural-networks called Gemmini).
  The project is developed as a Chipyard hardware accelerator.

2.2 RoCC interface

In order to facilitate the development process, Chipyard offers a well defined interface called "RoCC interface" (RoCC stands for RocketChip Coprocessor), that allows designers to integrate with one of the provided cores an accelerator, implement it and test it.

The ISA of the interface is structured as follows :

- customX rd, rs1, rs2, funct

The X will be a number between 0 and 3, determining the opcode of the instruction; 0 and 3 are not casual numbers, indeed the CPU can support up to 4 accelerators, and each of them should be disambigued. The fields rd, rs1, and rs2 are respectively the register numbers of the destination register, first source register and second source register of the operation, where the register content can be set at compile time in the framework (for this project for example, each register is 32 bits wide). Funct is a signal used to distinguish between instruction belonging to the same accelerator.

Chipyard offers also a C library containing macros in order to properly call an instruction on the accelerator, that has been used in the testing part of the project. The main macro is shown below :

- ROCC_INSTRUCTION_DSS(X, rd, rs1, rs2, funct)

It represents exactly the RoCC interface structure, with the diffrence that :

- X and funct have to be hardcoded
- rd has to be an integer variable that will contain the result of the operation
- rs1 and rs2 have to be integer variables containing the content of the registers that we want to give as source operands

Once the macro has properly managed our operation in the binary of the code, at runtime the RoCC interface will decode the operation and communicate with the accelerator to execute the instruction. The ISA between the RoCC interface and a generic accelerator, to be as generic as possible, provides many signals for both integers and floating pointer operations; for the scope of this project however, floating pointers have not been considered, reducing the real used ISA to this set of signals :

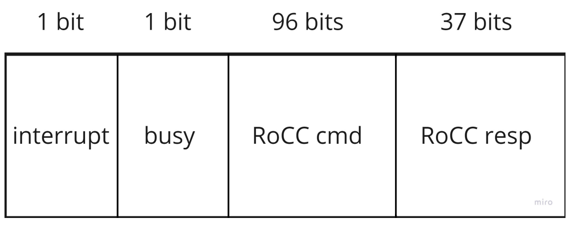

Figure 5: High level RoCC cmd ISA

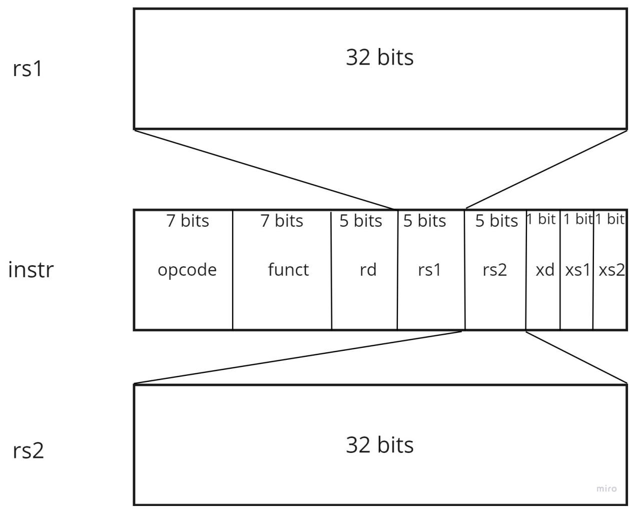

Figure 6: High level RoCC cmd ISA

A rocc cmd contains both the instruction coming from the processor and the content of the source operands.
The flags xd, xs1, xs2 are true when the correspondent register holds valid data.

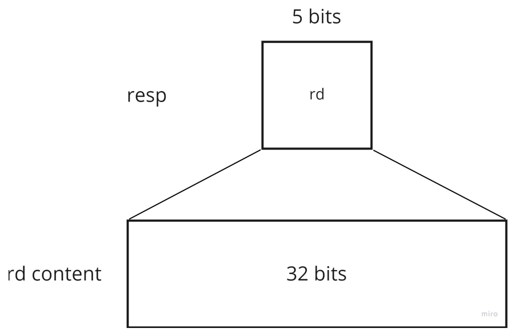

Figure 7: High level RoCC resp ISA

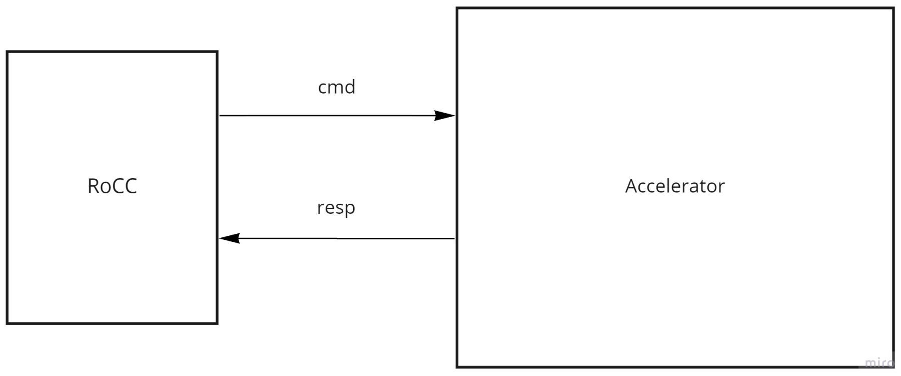

Figure 8: High level communication overview

In figure 8 is briefly shown how the RoCC interface communicates with the accelerator.

2.3 Verilator

Verilator is an open source tool which converts Verilog code in behavioral models in C or C++ language. This implies that is possible to simulate the accelerator (once fully synthesized) directly from the C code; in particular, it has been used specifically for the testing part of the project.

3 Architectural Description

This section will focus mainly on the accelerator structure, giving an overview of the high level components and how they are interconnected, then they will be analyzed separately, with a special focus on design choices made, implementation and internal structure.

3.1 ISA

This section will give an overview on the possible commands supported by the torus accelerator; although it is not complete, since this would require the complete knowledge of the hardware structure, a brief description is presented to facilitate next sections understanding (each operation is broadcasted by the controller to each PE if it is a valid one):

- store : mem(rs2) := rs1
  - rs1 : 32 bits signal, containing in the first 16 bits the immediate value to be stored
  - rs2 : 32 bits signal, containing in the first 16 bits the memory address
  - funct : when set to ”1” (and the command is valid), a store command is triggered
- load : out_queue := mem(rs2)
  - rs2 : 32 bits signal, containing in the first 16 bits the memory address
  - funct : when set to ”0” (and the command is valid), a load command is triggered

  The load, since for 1 load there are n data to be returned where n = #PEs, does not directly return the value, but it buffers it in an output queue, waiting for another operation called ”get load” that will dequeue one element (so to empty the buffer n ”get load” command need to be issued).
- get load : operation that has to be issued after a load, and returns a 32 bit signal that was previously stored in the first element of the output queue.
  ISA : get load → rd := buffer()
  - funct : when set to ”2” (and the command is valid), a get load command is issued
- exchange : most important operation that when issued, each PE for the next n cycles loads from its internal memory the value at address ”source + i” and broadcasts it to all its neighbours, and at the same time stores (always in its internal memory) the values coming from its neighbors at address ”dest + i”
  ISA : exchange(src, n, dest)
- src : signal representing the source address from which starting to load and broadcast
- dest : signal representing the destination address in which storing results coming from neighbors
- n : number of memory elements (all of 16 bits each) to be exchanged

3.2 Torus Accelerator

3.2.1 Interface

The interface of the accelerator is directly connected with the RoCC interface, so its structure has already been discussed in the RoCC dedicated section

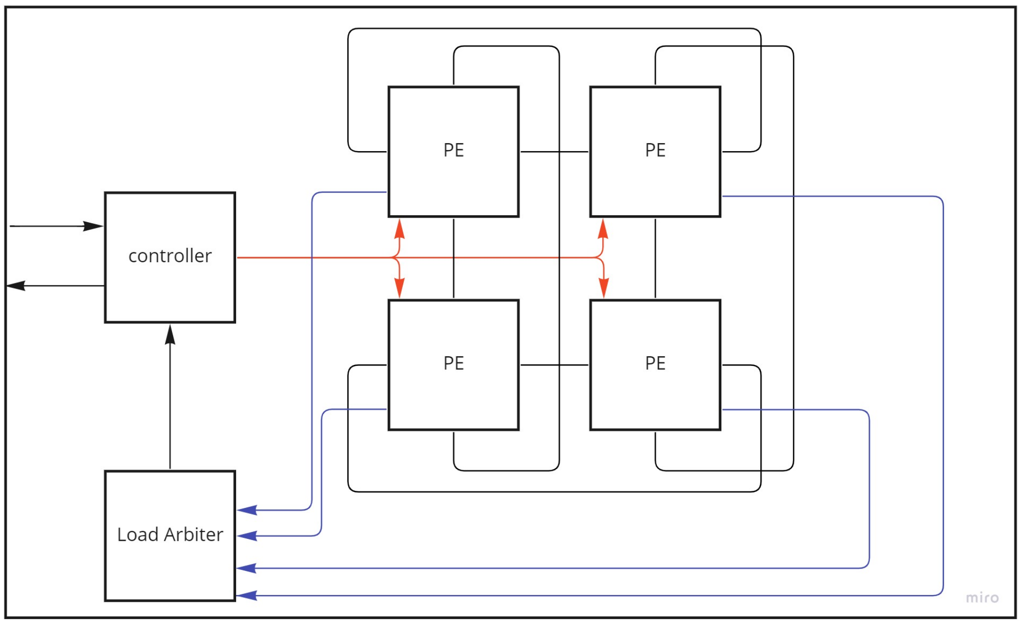

Figure 9: Torus representation with 4 PEs

3.2.2 Components

As it's possible to see in Figure 9, the main components of the system are: .

- Controller : component that handles the interface with the processor, decodes inputs and handles outputs, and broadcasts commands decoded to all the PEs. When the CPU issues a new valid command (so when the opcode of the operation matches with the opcode of the accelerator), the controller decodes the operation, and if the operation was a "get load", the controller dequeues an element from its output queue and returns it as a response in "data" field, otherwise enqueues the command in an input queue, that is directly connected with al the PEs.
- PE : processing element, containing 5 memories and the logic to exchange data with neighbours (for visualization purposes, in figure 9 inputs from the controller are represented by orange arrows, neighbour connections are represented with black lines, outputs are represented with blue arrows).
- Load Arbiter : component used to map n data coming from n PEs into 1 data; in fact it takes n clock cycles to forward all the data to the controller(called Load Arbiter because it is used only to return results after a load command).

3.3 Controller

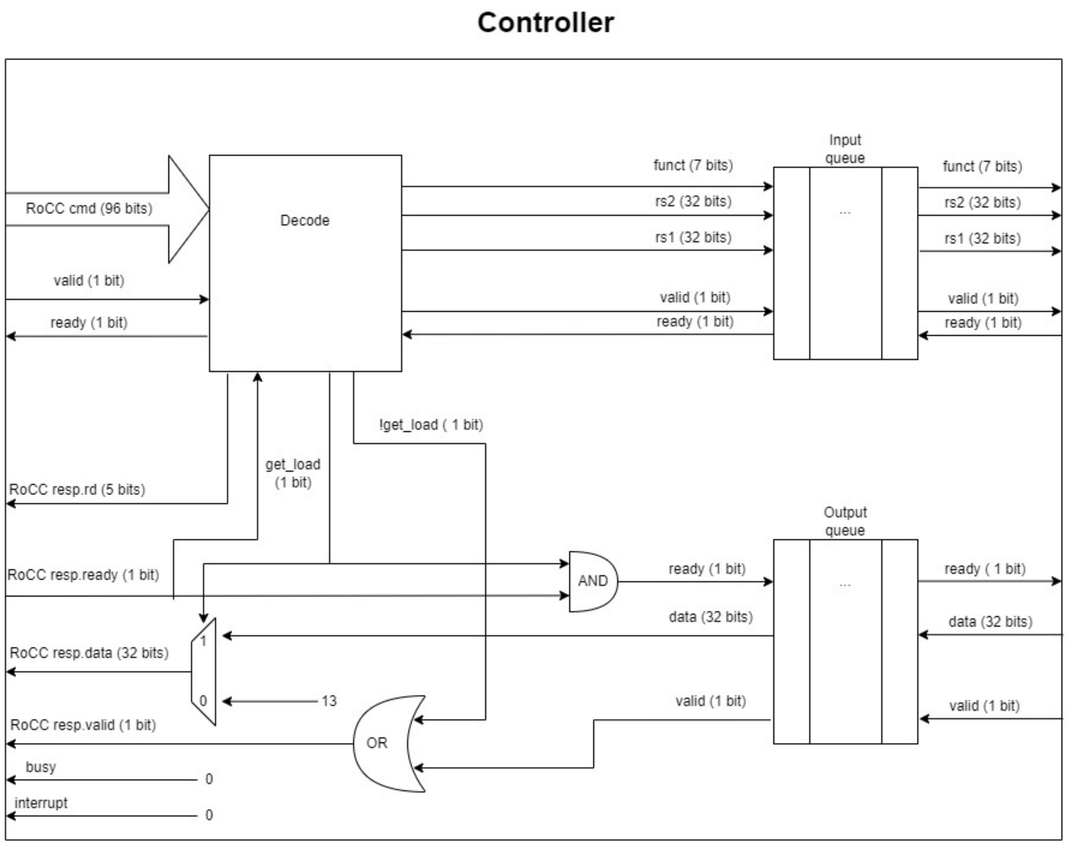

Figure 10: Controller high level representation

3.3.1 Interface

As it's possible to see in figure 10, the interface of the controller is composed as follows:
RoCC cmd and RoCC resp signals are already described in detail in the RoCC dedicated section.

Cmd is an output decoupled interface containing the decoded command that will be forwarded to each PE :

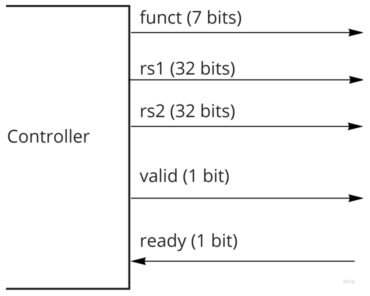

Figure 11: cmd interface

When both valid and ready signals are high, a new command is correctly issued to all the PEs. The type of command depends on the value of funct.

Resp is an input decoupled interface that contains the output of the internal module.

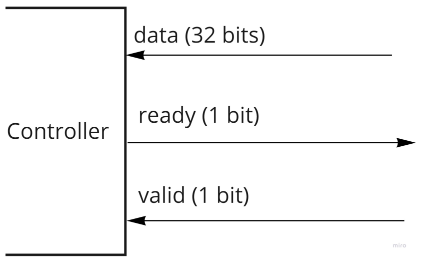

Figure 12: resp interface

The output is valid only when a load has been previously issued and executed, otherwise all the operations do not return anything.

3.3.2 Components

The main component is the Decoder. It takes care of correctly decoding the RoCC cmd, that: when is a valid load, store or exchange, it issues them to the input queue, returns a ready response to the RoCC cmd interface, and sets get_load signal as false (so !get_load as true); when instead the RoCC cmd is a valid get_load command, returns a ready respnse to RoCC cmd that is equal to the ready response send by the RoCC resp interface, sets the valid bit of the input queue as false, and sets properly the get_load and !get_load flags.

The input and ouptut queues are regular queues that take as input decoupled interfaces and returns as output the same interface flipped (inputs become outputs and viceversa), where: valid is true the queue is not empty (at least one element), ready is true the queue is not full (at least one free slot).

In order to properly work, the RoCC interface needs to recive as output both the data and the register in which storing it, that needs to be the same issued by the correspondent RoCC cmd; although it would require to buffer the rd in a buffer until the result is ready to be returned, in this implementation a useful data is returned only when a get_load is issued. For this reason, a convention has been implemented :

- If the command issued is not a get_load, the module always return a valid default data through the RoCC resp interface (the module returns 13, but it can possibly be whatever)
- If the command issued is a get_load, the data returned by the module is the one buffered in the output queue (and if it is not valid, the controller stalls until a valid data enters the queue). Here is assumed that the programmer **never** issues a get_load without having issued before a load; even if this might seem a strong assumption, the reason why it has been made is because the load / store infrastructure has been used only to test the exchange command, and not designed to be scalable.

In both cases, the data is formally returned in the same clock cycle of the command issued, thus leading to the rd register address to be simply forwarded from the RoCC cmd to the RoCC resp at eack clock cycle (RoCC resp.rd := RoCC cmd.rd). If the module is not ready to receive a command, the RoCC interface will stall and send it again until the module is.

3.4 PE

This section describes the submodule which embeds the highest amount of accelerator logic : the Processing Element (PE).

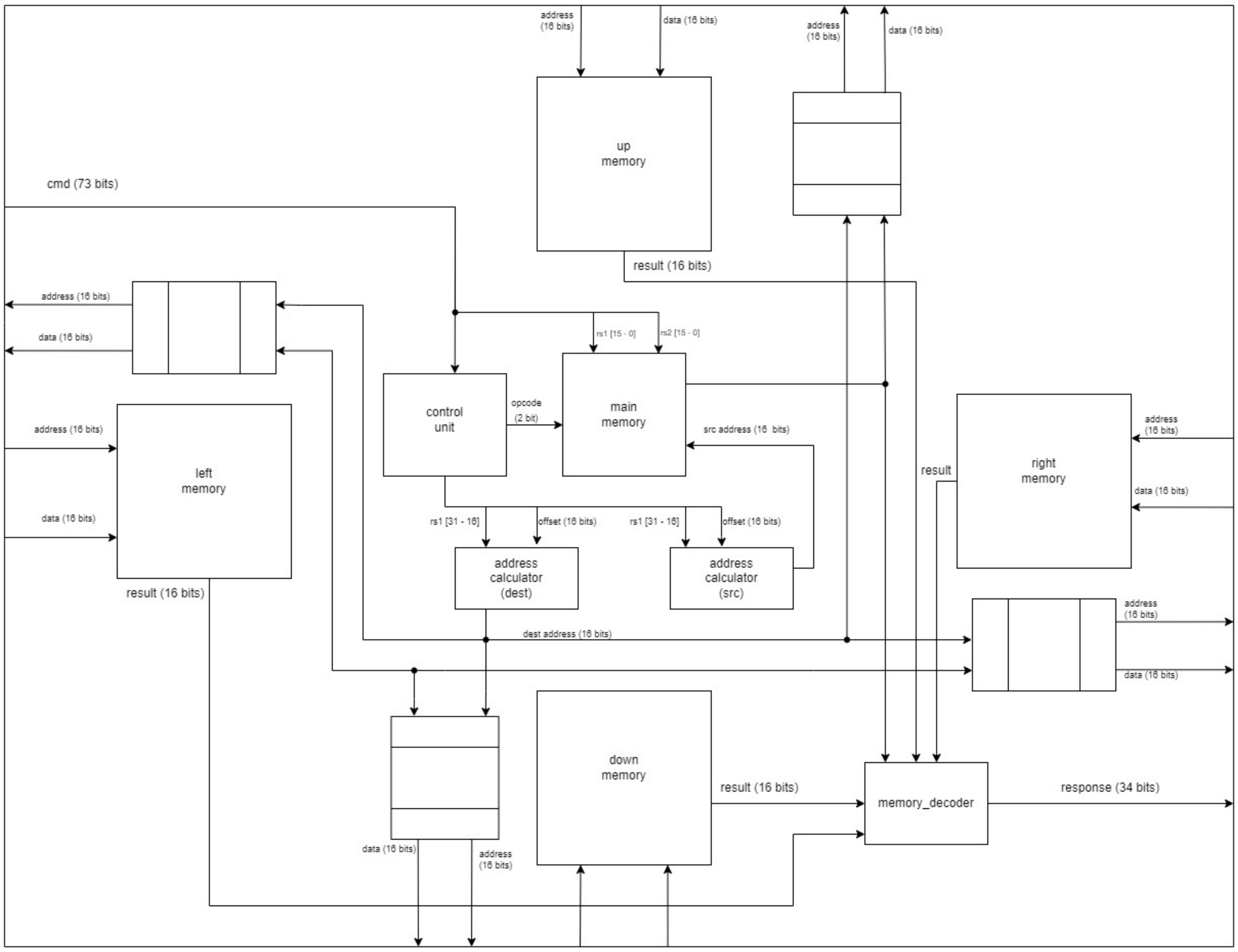

Figure 13: PE high level design

The main feature of the design is how memory is managed, in fact the design provides the usage of 5 different memories. In principle, a unique memory can be used, but this would lead to some complexity due to the data exchange command: assuming to work with a single memory, when a PE receives from 4 different sources (4 neighbours) data to store, it has to logically partition the memory, and based on the initial value of the exchange n, create a convention for which:

- if it is receiving the data from the above logical channel, store the data at : $address = destination + 0 * n + i$
- if it is receiving the data from the below logical channel, store the data at : $address = destination + 1 * n + i$
- if it is receiving the data from the left logical channel, store the data at : $address = destination + 2 * n + i$

- if it is receiving the data from the right logical channel, store the data at : *address* = *destination* + *3* * *n* + *i*

In such a way, all the memory locations are well defined (assuming that destination + 3 * n ¡ memory size holds always). The implementation however, in order to simplify the development process, consists on a set of five different physical register memories (one main memory plus four memories associated with each side), that are accessed only once per cycle. This behaviour, even if does not fully exploit the parallelism of a register memory, was used because it is possible to assume that each PE has been simulated as a memory with five banks having one input and one output port; each bank of memory that fulfils this behaviour thus can be used instead of a register memory. Moreover, simulating each memory with just one input and output is enough to implement the accelerator functionality correctly.

For this reason, instead of using one physical memory and divide it in 4 logical ones, the design proposed consists on 5 physical memories, 4 for each side in order to parallelize the memory accesses during an exchange command (each memory can be seen as an independent memory and it requires just 1 port), and 1 main memory, from which, when an exchange is issued, the source data comes from (more on this in section 3.4.2).

3.4.1 Interface

The interface of the PE is composed as follows:

Cmd is the interface though which the module receives commands from the controller, indeed it has already been described in section 3.3.1.

Resp is the interface through which the module send the command once it has been executed, and it forwards to the LoadArbiter component; this interface has also been described in section 3.3.1.

Neighbour is instead the interface through which the PE exchanges data with its neighbours.

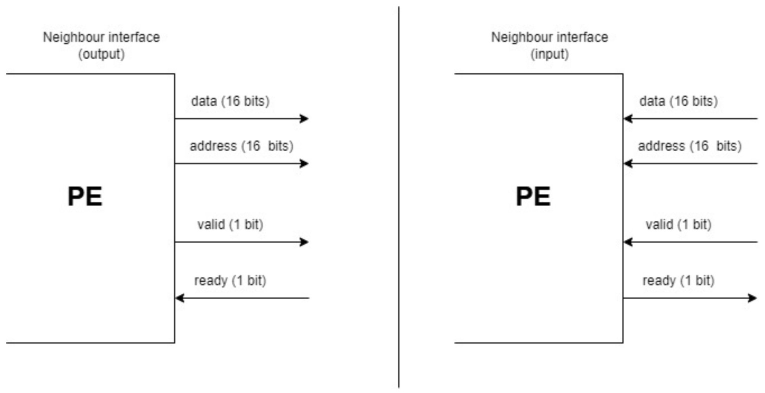

Figure 14: PE neighbour interface

In figure 17 both the interfaces relative to input and output are represented.
Since the torus is implemented in 2 dimensions, each PE has to handle 4 connections with other PEs, thus presenting this kind of interface in each of them.
In particular, when a PE receives a data exchange command with parameter n, for n cycles it will send through the neighbour input interface (that is handled internally as a queue) a valid data, and the address in which the neighbour has to store it in its internal memory. When instead a PE receives a valid data from a neighbour input interface, no matter from which interface neither how many, it does not accept new commands from the controller (sets the cmd.ready := false) and stores that data inside the memory dedicated to that specific interface.

3.4.2 Components

To accomplish its tasks, the PE has a variety of components that work together.

- Memories : as already mentioned in section 3.4, each PE has 5 internal register memories (main, upper, lower, left, right) used to handle the data flow, that are composed by $2^{16} - 1$ registers of 16 bits. In particular, when a load / store command is received, the signal that contains the logical address of the operation (cmd.rs2) that is 32 bits wide is split in 2 parts:
  - rs2[18 - 16] : bits between 18 and 16 (3 bits wide) that represent the logical memory in which the operation has to be done (000 = main memory, 001 = upper memory, 010 = lower memory, 011 = left memory, 100 = right memory)
  - rs2[15 - 0] : last 16 significant bits are used to actually address the physical memory chosen in the [18 - 16] bits

  When instead an exchange commands from *src* to *dest* of size *n* is issued by the controller, the signals are handled as follows :
  - rs1[15 - 0] : last 16 significant bits representing the source address of the **main memory** from which the data are loaded at each cycle (from 0 to n-1) and then forwarded to the queues and then exchanged with neighbours. Notice that the data to be exchanged **must** be in the main memory, because the command loads it from there (that's the reason why an additional memory has been designed).
  - rs1[31 - 16] : first 16 significant bits representing the value n (= amount of data block to exchange).
  - rs2[15 - 0] : last 16 significant bits representing the destination address in which data at each cycle (from 0 to n-1) have to be stored; in the design proposed also the correct destination address is forwarded with the data to store to the neighbours. Notice that the destination address is of 16 bits wide because each PE will store the data coming from neighbours in their dedicated memory that has $2^{16} - 1$ blocks (registers).
- control unit : component that decodes the instruction and :
  - if the instruction decoded is a load / store, it simply sends to the right memory the command
  - if the instruction is an exchange, it resets a counter and sets its treshold to n where n is the number of data to exchange, and at the first cycle it sends the source and destination address respectively to the source address calculator and destination address calculator components, and for each cycle from 0 up to n-1, it outputs the value of the counter updated (0, 1, 2, ... , n-2, n-1) that, at each cycle, will be added to the source and destination addresses (operation executed by source and destination address calculator), then the PE will load from the main memory at the **computed** source address the data and send in broadcast to all the neighbours with the **computed** destination address. Notice that if any of the neighbors is not ready to receive a data (and the queue used to send data fills up) the counter will stop incrementing its value, so that the same value will be sent (and discarded by the full queue) until the neighbor PE is ready again; this however implies that the potential bottleneck of the operation is represented by the **slowest** PE, because the counter is unique, and all the other PEs will store the same value in the same memory location over and over again.
- address calculators : these two components have the same structure and functionality and the only difference between the two is that they take a different input. First if all, their usage is meaningful only when an exchange command (exchange(src, n, dest)) is issued, and :
  - In the same clock cycle in which the command is issued, they will receive respectively the source and destination addresses that will be buffered inside the component for the subsequent n-1 cycles, and for each iteration of the cycle i, the counter will give them as input the right value of the iteration, and the components will return as output respectively $exc\_src = source + i$ and $exc\_dest = destination + i$. Then, the module will load the data at address exc_src and send it in broadcast with the exc_dest to the neighbors.

- queues : the final components that characterize the PE are the queues associated with each side. Their role is not so complicated, indeed they have to behave like a buffer for the neighbor interface, queuing the address and the data that will be forwarded to the neighbor when it's ready. The implementation proposed has a default queue size of 3, but it can be changed at design time.

3.5 Load Arbiter

This section describes the Load Arbiter component, that is used mainly to arbiter n 16 bits wide incoming data from the various PEs in input to 1 32 bits wide data for cycle in output. The operation takes n clock cycles to complete.

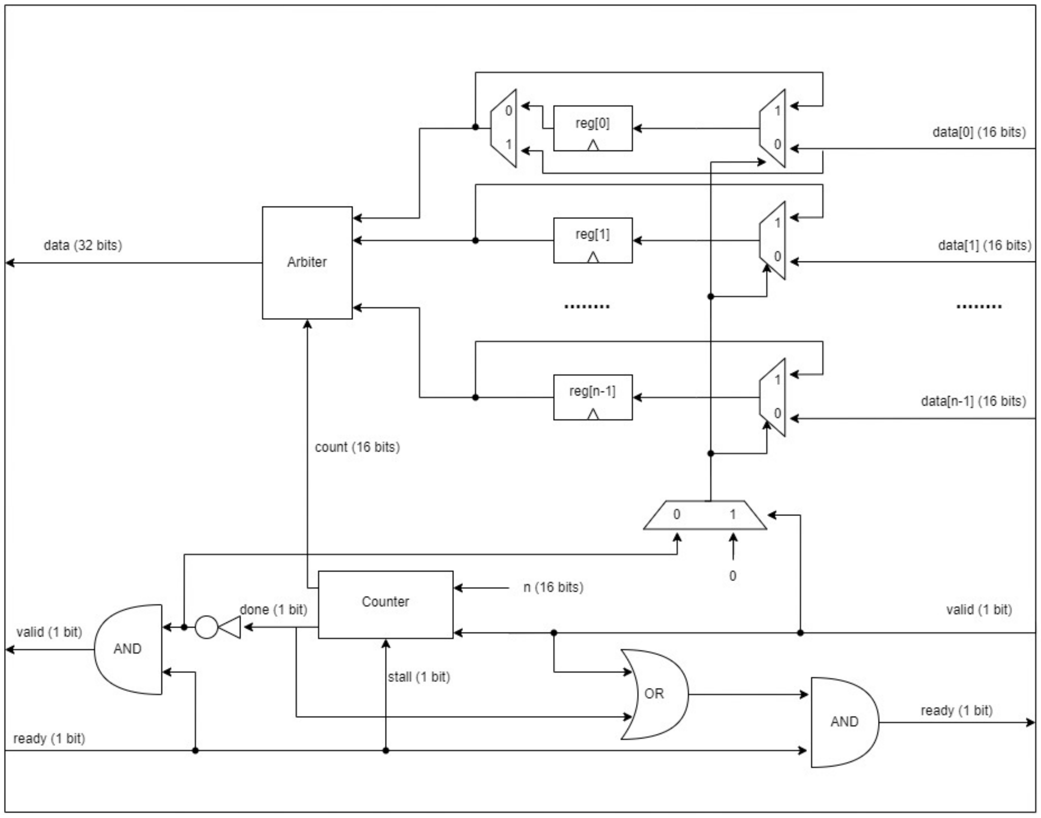

Figure 15: Load Arbiter

3.5.1 Interface

The component exposes 2 interfaces : one "input" interface with the set of PEs and one "output" interface with the controller.

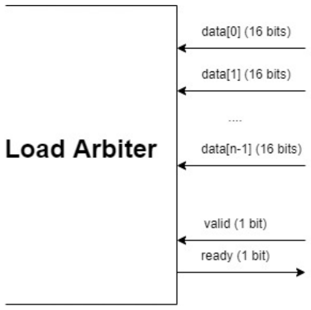

Figure 16: ”input” interface

The first interface is characterized by all the data coming from each PE handled as a decoupled interface; in fact, the *valid* signal is the logical *AND* of all the valid output signal of each PE. Since the interface exposed is uni-dimensional but the topology of the PEs is bi-dimensional, the signals have been routed in this way :

(i, j) = position
n = matrix dimesion
vector index = n * i + j

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \Rightarrow [0, 1, 2, 3, 4, 5, 6, 7, 8]$$

so the interconnection is linearized from two dimensions to one dimension.
The ready signal instead is used to communicate if the component is ready to receive data (it is not if it is busy because of a previous load, or because the controller is not ready).

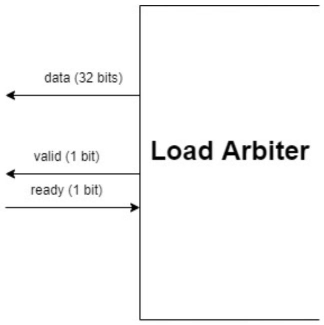

Figure 17: ”input” interface

The second interface is exposed between the Load Arbiter and the controller, and returns only one data for each clock cycle. The ready signal is directly connected with the output queue, and so is true when the output queue is not full. The valid signal instead is true when a valid data is returned (this happens for at least the next $n * n$ cycles after a load has been issued).

3.5.2 Components

The module can be logically divided in 3 main sections :

- Registers : if the valid signal coming from the PEs is high, the module stores ALL the data in a set of registers, which keep the value stored until all of it have been correctly sent to the output interface (controller). Notice that in order to gain one clock cycle, the fist register *reg[0]* is slightly different because it allows to return the data in the same clock cycle in which it has been issued, and it is not constrained to wait 1 cycle due to the register; however, it is still buffered because the controller may not be ready to receive it.

- Control unit : the control unit is represented by a counter and all the logic that handles the ready/valid flags.
  The counter gets initialized when a set of valid data are coming from the input interface, and it counts for $n * n$, that is the number of PEs of the accelerator. It outputs then two signals : count, that is the iteration number, and done, that when false indicates that the iteration process is not over yet. If the controller is not ready to receive outputs, the counter stops counting until the controller returns to be ready again.
  The module returns a positive ready signal to all the PEs if : the controller is ready to receive data (otherwise the PE would keep issuing data that would be lost) *AND* the data issued are valid *OR* the module is not iterating.

  The valid signal is also used to optimize the return of the first value, because when it is high (and only in that case) we can let the first data bypass the register and be directly sent to the controller.

  The module sends a positive valid signal to the controller if the controller is ready to receive it and if the counter is not over.
- Arbiter : this component is used to map $n * n$ incoming data in one data based on the counter output; since the counter counts from 0 to $(n * n) -1$, the arbiter will route all the data from 0 to $(n * n) -1$.

4 Results

This section aims at analyzing the performances of the exchange command, focusing mainly on the aggregated bandwidth of data exchanged
Performances of other operations (load, store, get_load) are not analyzed because not relevant for the analysis (they are implemented for testing purposes only).

n	n^2	**Channels**	**Aggregated Banwidth**	**Aggregated Bandwidth Used**	**Throughput**
2	4	16	64 Bytes	32 Bytes	26,66 $\frac{Bytes}{clock}$
3	9	36	144 Bytes	72 Bytes	59,99 $\frac{Bytes}{clock}$
4	16	64	256 Bytes	128 Bytes	106,66 $\frac{Bytes}{clock}$
5	25	100	400 Bytes	200 Bytes	166,66 $\frac{Bytes}{clock}$
6	36	144	576 Bytes	288 Bytes	239,99 $\frac{Bytes}{clock}$

The table above summarises the results obtained, where n is the uni dimensional side of the torus. Although many parameters can influence the result, they are not so relevant for performance evaluation; for this reason, a default configuration is proposed where the memories of the PEs are all register memories with 16 bits wide registers, and to access them a 16 bits address is enough. In this way, each channel between neighbour PEs has 34 bits, 2 bits for control (ready/valid), 16 bits of valid data and 16 bits of address.
To compute the throughput of the exchange, a fixed number $n_{data} = 5$ has been used, so the total latency of the operation is $n_{data} + 1 = 6$.
The data are computed as follows:

- **Channels** : as it's possible to see, the number of channels in the structure is given by the formula

$$Ch(n) = n^2 * (2 + dimension)$$

  this holds because for each PE in the structure has 2 channels for each side (one channel in which receives, one in which sends data) and then the total number of bidirectional channels is the number of dimensions of the torus (in the bi-dimensional case it's twice the number of PEs).

- **Aggregated Bandwidth** : the aggregated bandwidth is computed as

$$A.B.(n) = Ch(n) * (address_w + data_w)bits$$

  where $address_w$ is the address width and $data_w$ is the data width (in the specific case, we have $address_w = 16, data_w = 16$).

- **Aggregated Bandwidth Used** : the bandwidth used to send actual valid data over the channel instead is computed as :

$$A.B.U(n) = Ch(n) * (data_w)$$

  (in the specific case, $data_w = 16$)

- **Throughput** : the throughput of the module is measured as number of data exchanged per clock cycle, that is computed as

$$Throughput(n, n_{data}) = \frac{A.B.U(n) * n_{data}}{n_{data} + 1}$$

  this because the exchange of n_{data} requires n_{data} CK + 1 (in the specific case, $n_{data} = 5$, so Throughput(n) = $A.B.U(n) * \frac{5}{6}$)

5 Future Work

In this final section are described possible extensions and improvements of the project made.

5.1 Three-Dimensions

The current implementation supports a two-dimensional torus, but it's possible to extend the topology up to three (or more) dimensions. This can be done by :

1. Adding two new neighbour interfaces (3.4.1) to the module, and replicating two new memories to handle the new dimension (PE module).
2. Increasing the Load Arbiter input interface (3.5.1) from accepting n^2 inputs and counting up to n^2, to accepting n^3 inputs and counting up to n^3 (Load Arbiter module).
3. Handling at Accelerator level the new set of interconnection introduced by the additional dimension (Torus module).

5.2 Bandwidth Burst

In order to increase the aggregated bandwidth of the exchange command, it's possible to implement a Bandwidth Burst.
In particular, each PE instead of sending data and address to its neighbour at each cycle, it sends :

- first cycle : n_{data} , the number of data blocks to be exchanged
- second cycle : address , the address from which it has to start storing data
- for the next $\frac{n_{data}}{2}$ cycles, 2 data block at a time instead of 1 (the channel that was previously used by the address now is used to send only valid data)

In this way, the aggregated bandwidth used becomes equal to the aggregated bandwidth, and the throughput (for n_{data} large) is almost double.

References

[1] David E. Shaw, J.P. Grossman, Joseph A. Bank, Brannon Batson, J. Adam Butts, Jack C. Chao, Martin M. Deneroff, Ron O. Dror, Amos Even, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Brian Greskamp, C. Richard Ho, Douglas J. Ierardi, Lev Iserovich, Jeffrey S. Kuskin, Richard H. Larson, Timothy Layman, Li-Siang Lee, Adam K. Lerer, Chester Li, Daniel Killebrew, Kenneth M. Mackenzie, Shark Yeuk-Hai Mok, Mark A. Moraes, Rolf Mueller, Lawrence J. Nociolo, Jon L. Peticolas, Terry Quan, Daniel Ramot, John K. Salmon, Daniele P. Scarpazza, U. Ben Schafer, Naseer Siddique, Christopher W. Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley C. Wang, and Cliff Young. Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 41–53, 2014.