POLITECNICO DI TORINO

Master thesis

Master's degree in Electronic Engineering

Interconnect analysis and system integration for a RISC-V based System on Chip

Advisor
Prof. Luciano Lavagno

Candidate
Ettore Antonino Giliberti

Supervisors research center BSC

Doc. Eng. Carles Hernandez

Doc. Eng. Jaume Abella

Prof. Ramon Canal

April 2019

Abstract

This thesis was carried out at the research center Barcelona Supercomputing Center (BSC) which is the national supercomputing center in Spain. It is specialized in High Performance Computing and its mission consists in investigating, developing and exploiting technologies in order to facilitate the scientific progress.

I had this opportunity through the research group CAOS (Computer Architecture - Operating System interface); it has a long experience in important projects with the industry, the European Space Agency (ESA) and with the European Union (EU), specially as regards the embedded systems.

Critical real-time embedded systems, such as automated driver assistance, surveillance cameras, and drones, require high-performance hardware to be exploited in new applications. In this scenario, the FPGAs provide a low development cost alternative with respect to the ASIC custom chips; they also have the advantage of reconfigurability which allows to reduce the lead time. Recently, research is moving towards open-source hardware modules that represent an opportunity; in parallel, during the last decades, Systems-on-Chip (SoCs) gained an important role in the electronic world, for this type of applications and generally in the embedded systems and Internet of Things (IoT) fields.

Performance improvements are continuously required; modern SoCs include multiple cores and a higher number of each type of IP blocks. The system bottleneck can become the interconnection because the traditional buses could not provide the required performance, e.g. in terms of scalability and data transfer speed.

This thesis fits in this framework. Currently, the CAOS group is working on adapting a


RISCV-based platform to the space domain. According to the requirements for this application, the objective of this thesis has consisted in analyzing different typologies of interconnection along with the related SoCs and cores (RISC-V ISA); finally trying to implement a new mixed SoC configuration. The experience, in broad terms, could be split in two phases: the first one concerns the analysis of a bus-based crossbar interconnect in the open source SoC *Untethered Rocket Chip*; the second part involves the study of a Network-on-Chip Tile, in order to replace the original MIPS core with the new RISC-V processor *Lagarto*.

Abstract

Questo lavoro di tesi é stato svolto presso il centro di ricerca Barcelona Supercomputing Center (BSC) che rappresenta il centro nazionale di supercomputazione in Spagna, specializzato in High Performance Computing (HPC). La sua missione consiste in investigare, sviluppare e gestire le tecnologie allo scopo di facilitare il progresso scentifico.
L'opportunitá di svolgere quì questo progetto mi è stata data dal gruppo di ricerca CAOS (Computer Architecture – Operating System interface); il quale ha una lunga esperienza in importanti progetti con l'industria, con la Agenzia Spaziale Europea (ESA) e con l'Unione Europea (EU), specialmente nel settore dei sistemi embedded.
L'industria spaziale richiede hardware ad alte prestazioni da sfruttare in nuove applicazioni. In questo scenario, le FPGA forniscono un'alternativa a basso costo rispetto agli ASIC custom chip; avendo inoltre il vantaggio della riconfigurabilità che permette di accorciare i tempi tecnici. Recentemente ci si sta muovendo verso moduli hardware open-source che rappresentano un'opportunitá; parallelamente, durante le ultime decadi, i System-on-Chip (SoC) hanno acquisito un ruolo importante nel mondo dell'elettronica, per questo tipo di applicazioni ed in generale nel settore dei sistemi embedded e del Internet of Things (IoT).
Miglioramenti prestazionali sono continuamente richiesti; i moderni SoC includono più cores e un maggiore numero di ogni tipo di IP. Il collo di bottiglia del sistema può diventare la interconnessione poichè i consueti bus potrebbero non essere in grado di fornire le performance richieste, scalabilità e velocità di trasferimento dati su tutte.
Il progetto realizzato si incastra in questa cornice. Attualmente il gruppo CAOS sta lavorando sull'adattamento di una piattaforma RISCV-based per il dominio spaziale. Sulla

base dei requisiti per le applicazioni spaziali, l'obiettivo di questa tesi consiste nella analisi dei diversi tipi di interconnessione insieme ai relativi SoC e processori (ISA RISC-V), infine provando a implementare una nuova configurazione di System-on-Chip. Il progetto può essere suddiviso in due fasi: la prima riguarda l'analisi di una crossbar/bus-based interconnessione nel SoC *open source* called *Untethered Rocket Chip*; la seconda parte prevede lo studio di un *Tile* che costituisce un nodo di una Network-on-Chip, allo scopo di rimpiazzare al suo interno l'originale MIPS core con il nuovo processore *Lagarto* che implementa il sopracitato ISA RISC-V.

Acknowledgements

This master thesis concludes a long universitary experience which allowed me to deeply improve my technical and human knowledges, travelling in different places and meeting a lot of people. My first thanks is addressed to my family, for their love and to have always believed in me, supporting in everything, giving to me the possibility of studying, by encouraging on my choices, always ready for advice and helps.

Secondly, I am very grateful to the CAOS group of BSC for giving me the opportunity of carrying out this thesis; especially to my supervisors Charles Hernandez Luz, Jaume Abella and Ramon Canal for their assistance, availability and kindness.

My sincere thanks also to Prof. Luciano Lavagno from Politecnico di Torino for being my advisor always nicely helpful.

Then, I would like to express my gratitude to my friends who play an important role in my life, for their time and for the strong awareness of being able to rely on them.

I am happy to share this achievement togheter with Paolo and Angelica that are differently but deeply living this experience in Spain with me.

Finally, many thanks to my colleagues at BSC, kind and helpful since the beginning of this work.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Overview

This thesis was carried out at the research center Barcelona Supercomputing Center (BSC) which represents the national supercomputing center in Spain. It is specialized in High Performance Computing and its mission consists in investigating, developing and managing the technologies in order to facilitate the scientific progress.
The opportunity was giving to me by the research group CAOS (Computer Architecture - Operating System interface); it has a long experience in important projects with the automotive, space and avionics industry, the European Space Agency (ESA) and with the European Union (EU), specially as regards the embedded systems.
Critical real-time embedded systems industry (e.g. space, avionics) requires high-performance hardware to be exploited in new applications. In this scenario, the FPGAs provide a low-cost alternative with respect to the ASIC custom chips; they also have the advantage of reconfigurability which allows to reduce the lead time. Recently, the research is moving towards to open-source hardware modules that represent an opportunity.

During the last decades, the Systems-on-Chip (SoCs) gained an important role in the electronic world, above all in embedded systems and Internet of Things.

Performance improvements are continuously required, so modern SoCs are used to incorporate multiple cores (MPSoCs) and, generally, a higher number of IPs: the system bottleneck can become the interconnection.
For systems with intensive parallel communication requirements buses may not provide the required bandwidth, latency, and power consumption. Thus, to achieve high performance and scalability, is more and more taking off the idea of scaling down the basic architecture of the large-scale networks, by carrying out an embedded switching network, called Network-on-Chip (NoC), to interconnect the IP modules in SoCs.

1.2 Objectives

The realized project fits in the described above framework. Currently, the CAOS group is working on adapting a RISCV-based platform to the critical real-time embedded systems domain, with emphasis on the space domain. According to the requirements for this application, the objective of this thesis has consisted in analyzing different typologies of interconnection along with the related SoCs and cores (RISC-V ISA); finally trying to implement a new mixed SoC configuration.

The experience, in broad terms, could be split in two phases:

1. The first one concerns the analysis of a bus-based crossbar interconnect in the open source SoC *Untethered Rocket Chip*.

2. The second part involves the study of a Network-on-Chip Tile, in order to replace the original MIPS core with the new RISC-V processor *Lagarto*.

The structure of this document is now outlined.
The following section deals with an overview of SoCs and of the types of interconnection up to the NoCs, in order to frame the project background.
In the second chapter, the systems assessed during the experience are presented.

Subsequently, there is a focus on the methodologies used to simulate and to analyze the systems.

The fourth chapter includes the results obtained from the first part of the experience, so the interconnect analysis and the deriving latency results of the SoC.

The subsequent chapter describes the steps carried out during the second phase of the work focusing on the NoC Tile, reporting the achieved outcome.

Finally, the conclusion and some possible developments in the future are listed.

Chapter 2

State of the art

2.1 SoCs

System-on-Chip (SoCs) are integrated circuits including various parts: processors, memories, interconnections and other elements depending on the specific chip. They are usually employed in embedded applications, since they consume lower power, they have lower costs and higher reliability than the multi-chip systems that they can replace. They are commonly used in handheld electronic devices, such as tablets and smartphones.

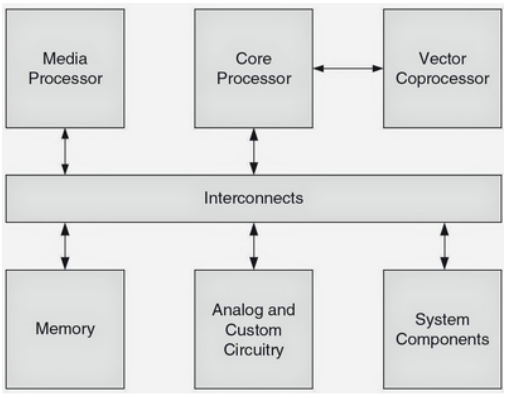

Figure 2.1: Basic SoC structure. [2]

A SoC can include digital, analog and radio-frequency circuits over the same chip. Overall, it can contain:

- one or more cores (μC, μP or DSP);
- a memory system eventually composed by different cache levels, one or more blocks of RAM, ROM, EEPROM or Flash;
- clock generator, PLL, counters;
- A/D and D/A converters;
- connectors for standard interfaces such as USB, USART, SPI and so on;
- voltage regulators and power supply management circuitry.

2.2 SoCs interconnects

2.2.1 Buses

SoCs can exploit different type of interconnects, the most commons is the bus-based solution, with a hierarchical topology for optimization issue (latency, bandwidth, power consumption, ...). Buses with different features and performance are connected; bridges to adapt the signals are necessary but their implementation can be complex. During the last years, several bus-based communication architecture standards have been designed in order to simplify that issue, to speed up SoC integration and enhance IP reuse over different designs; e.g. AMBA from ARM, IBM CoreConnect, Altera Avalon. AMBA 2.0 has initially defined three different buses: Advanced High-performance Bus (AHB), Advanced Peripheral Bus (APB), Advanced System Bus (ASB, not used anymore). AMBA 3.0 introduced the Advanced eXtensible Interface (AXI) bus to improve the AHB bus with advanced features to support the new high-performance MPSoC.

This second part of the section deals with the brief explanation of generic bus-based and crossbar structures, the reason why there are moves towards NoCs and their overview.

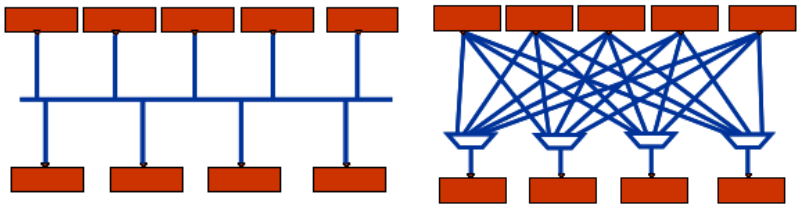

Figure 2.2: Traditional bus-bused (left) and crossbar (right) interconnects [3]

In the bus-based interconnect there are several wires connecting IP modules and an arbiter which manages the requests to use the bus. This initial and simple type of interconnection comes across some limitations; first of all latency and bandwidth, which depends on clock-frequency that is limited by the wires length. Increasing the IP modules in a SoC and the required performance, a first evolution of the simple bus-based interconnect consists in crossbar structures. They allow to have higher latency predictability and a significant increase of aggregate bandwidth; but, on the other hand, they require a very larger number of wires [3]. A first version of the interconnect structures described above can be defined as "coupled", there is an unique communication protocol among IP modules. A big drawback affects it: the difficulty of integration and IP re-use.

To overcome this problem and to handle more complex SoC, a second version was designed and defined as "decoupled" because the Transaction Layer is decoupled from the Transport Layer; it means that there is not anymore just one communication protocol. There is a protocol for the signals traveling within the interconnect (Transport Layer) and another one to communicate from the IP to the interconnect (Transaction Layer).

Even though, on the one hand, the decoupled solution leads to steps forward, on the other hand it also comes across further limitations of all the bus-based interconnect, such as data transfer bottleneck and the inherently *unscalability* (the higher the number of IP linked to a bus, the higher the congestion; the higher the load, the higher the power consumption).

These restrictions, together with the new high performance requirements (power management, multiple clock domains, error handling), leads to the development of a new type of interconnection called Network-on-Chip (NoC).

2.2.2 NoCs

A NoC is basically defined as an embedded switching network which uses packets to route data from the source PE to the destination PE.

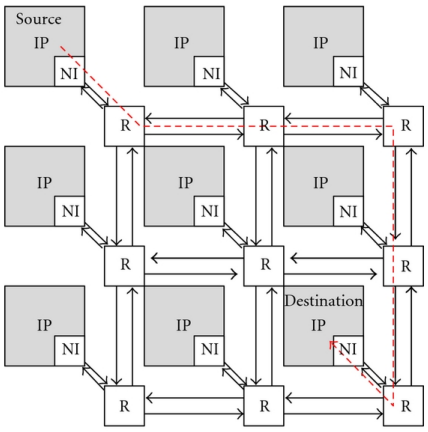

Figure 2.3: Typical NoC architecture (mesh topology). [6]

The NoC architecture is composed by three fundamental entities:

- Network Interface (NI): it converts the signal coming from the IP to the protocol used into the net based on packets.
- Router (R): it deals with the routing operations.

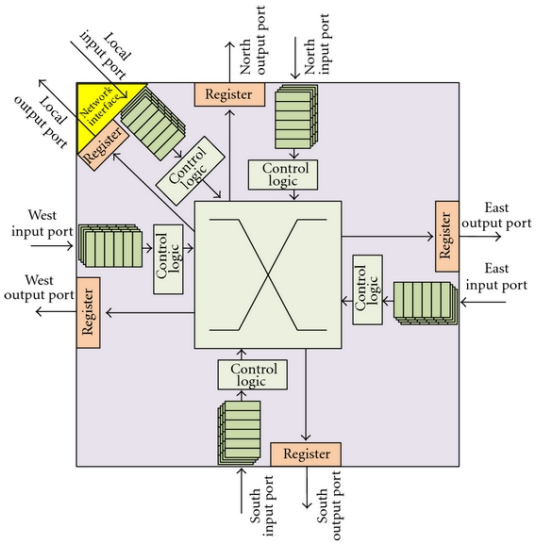

Figure 2.4: Typical NoC router architecture. [6]

Through a buffer, all the input lines are connected to a *crossbar switch*, which is used to select the proper output line; there is an *arbiter* determining which packets have the priority when two or more of them need the same channel to be transferred.

- Link: it is the physical connection between routers, it consists of wires and it can include also repeaters.

By considering as a starting and reference point the ISO/OSI network protocol stack model, NoCs use three different communication schemes, related to Transaction, Transport and Physical layers (fig. 2.5).

1. Transaction: it defines the communication protocol before the actual network, so right down the IP module and before its *Network Interface Unit*, outlining the primitives to interconnect IP blocks.

2. Transport: it establishes the rules for the packets that flow inside the net and so through the *routers*. The main goal is decomposing messages into packets at the source and then assemble them at the destination.

3. Physical: it defines in which way the packets physically travel though the interface. A limit of bus-based architectures is related to the maximum frequency. Buses exploit a synchronous and multipoint approach; on the other hand, NoCs are based on a point-to-point, Globally Asynchronous Locally Synchronous method. Thus, NoC-based system can operate at higher clock frequencies [8].

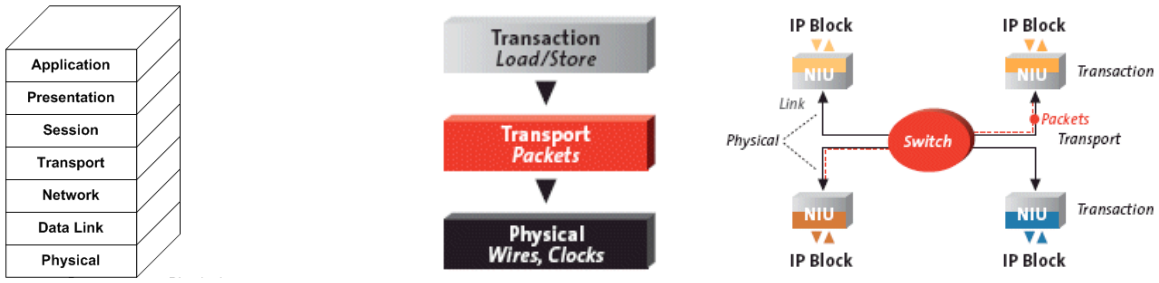

Figure 2.5: OSI stack model (left) and NoC communcation layers: Transaction, Transport, Physical (right). [4]

There are many ways to connect nodes, switches and links to each other. NoCs can have different topologies divided in three categories:

- Direct networks - This topology foresees for each node a direct point-to-point link to the *neighboring nodes*, a well-determined subset of nodes (e.g. Mesh, Octagon network).
- Indirect network - Each node is connected to only one switch and the switches have point-to-point links to other switches (e.g. Fat-tree network).
- Irregural network - it is based on a mix of shared bus, direct and indirect network topologies.

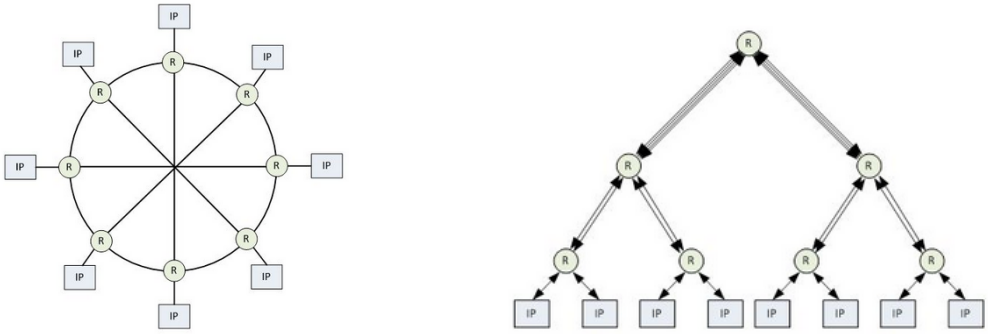

Figure 2.6: Octagonal network (left). Fat-tree network (right). [7]

As regards how data flow through the routers in the network, the idea consists in dividing the messages generated from the PEs in data *packets*; then, the packet is further divided into *flits*, elementary packets on which link flow control operation are performed. Finally, each flit is divided in more *phits*, a unit of data that is transferred over a link in a single cycle, whose size is the width of the communication link (fig. 2.7). The choice of the size of the phit has a high influence on the trade-off cost/performance of the NoC.

The two main modes to transport flits are the *circuit switching* and the *packet switching*.

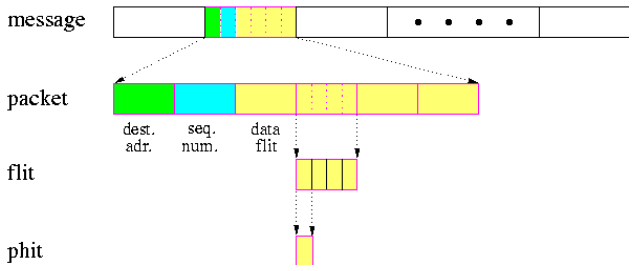

Figure 2.7: PEs message partitioning from http://pages.cs.wisc.edu.

A key-point in the NoC design is the routing algorithm: it determines how data are routed from the sender to the receiver. It is necessary to face several trade-offs, such as minimizing power and, on the other hand, reducing area and delay. They can be divided in oblivious (i.e static) and adaptive (i.e dynamic) algorithms. The first category routes packets without

information about traffic amounts and conditions of the network. The second one does it taking into account the current state of the network.

During the routing process, some problems have to be faced:

- Routing is in deadlock when two packets are waiting each other to be routed forward. Both of the packets reserve some resources and both are waiting each other to release the resources.

- Livelock occurs when a packet keeps spinning around its destination without ever reaching it. This problem exists in non-minimal routing algorithms. Livelock should be cut out to guarantee packet's throughput.

- Starvation: Assigned different priorities, the packets with higher priorities reserve the resources all the time and packets with lower priorities never reach their destinations.

To manage the transmission properly, eventually solving transmission errors and link congestion, flow control mechanisms are implemented.

Nowadays, the open source world offers different opportunities to implement a custom SoC, sometimes also exploiting the Network-on-Chip interconnect topology. Since this work is focuses on this background, some examples of this type of projects are now reported.

- *minsoc* is a minimal SoC with standards IPs which includes the synthesizable core OR1200 (exploting the OpenRISC[1] ISA). It mainly uses the Wishbone[2] bus for the communication of the blocks.

[1]OpenRISC: open-source project proposing SoC architectures, simulators and a RISC-based ISA.

[2]Wishbone: bus-based open-source standard using a single, high-speed synchronous bus specification in order to connect all the blocks within a SoC.

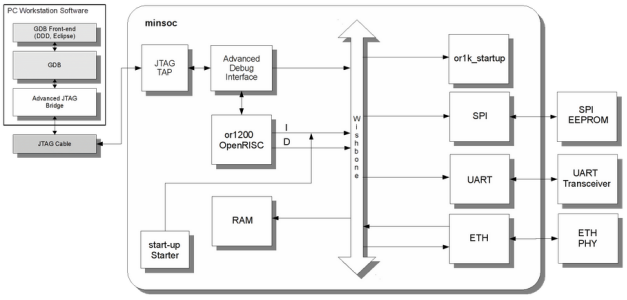

Figure 2.8: *minsoc* System-on-Chip. [10]

- OpTiMSoC is a flexible multicore SoC adopting OpenRISC processors (mor1kx) and based on a Network-on-Chip structure.
- *OpenPiton + Ariane* is a brand-new SoC consisting of a scalable, cache coherent system based on Network-on-Chip (OpenPiton) whose tiles incorporate the Ariane core. It is an open-source 64-bit Linux capable, RISC-V[3] core (supporting RV64-IMC [4]), 6-stages, in-order, described in System Verilog [11]. This processor belongs to the *PULP* family and it was designed at *ETH Zurich (Switzerland)* and *University of Bologna (Italy)*. It can be considered the next of kin for the cores evaluated in this thesis since they exploit the same ISA.

The *lowrisc* SoC is the platform used in the first part of this work and it fits in this category.

On the other hand, the second part of this experience focuses on a SoC that is not yet publicly released, it is being developed at the present time at the *Universitat Politècnica de Valencia* (UPV).

[3]RISC-V: free and open RISC-based ISA, defined by the creators as "clear, micro-architecture-agnostic and highly extensible"[14]. This Instruction Set Architecture, developed at UC Berkeley, aims at being an industry standard.

[4]RISC-V ISA base and extensions. I: Base Integer Instruction Set; M: Standard Extension for Integer Multiplication and Division; C: Standard Extension for Compressed Instructions.

Chapter 3

Baseline SoC

After having framed the background, this chapter concern the actual components and systems assessed in this work, starting from the processors. After that, there is an overview of the analyzed SoC, together with a focus on the communication standard used in its main interconnections.
At the end, the NoC Tile processed during the second phase of the work is outlined.

3.1 CPU cores

The evaluated two processors follow the open source trend and implement the RISC-V Instruction Set Architecture. They are known as *Rocket* and *Lagarto* cores.

3.1.1 Rocket Core

Rocket: The Rocket core is an in-order scalar processor that provides a 5-stages (or 6 with the addition of a *pcgen* stage) pipeline described in Chisel[1]. It implements the RV64G[2] variant

[1] "Chisel is an open-source hardware construction language developed at UC Berkeley supporting advanced hardware using highly parameterized generators and layered domain-specific hardware languages". [17]

[2] G extension provides a general-purpose, scalar instruction set. G includes the common extensions I, M, A (single-precision), F (double-precision), D (floating point).

of the RISC-V ISA It was developed by the *Berkeley Architecture group, United States*. The Rocket core provides a Branch Target Buffers (BTB) to reduce the branch penalty, a 64-bit Register File and a Floating Point Unit (FPU) can be set, alongside the usual integer ALU. In addition there is an accelerator or co-processor interface, called RoCC. This core is able to detect data hazards involving instructions with multi-cycle latencies thanks to a scoreboard.

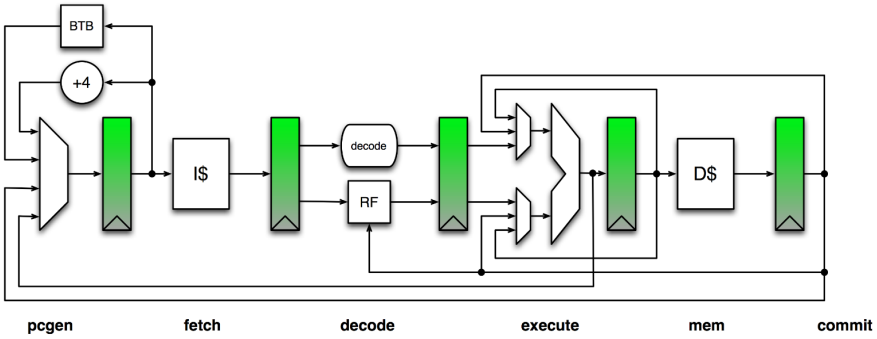

Figure 3.1: Rocket core, simplified pipeline scheme. [12]

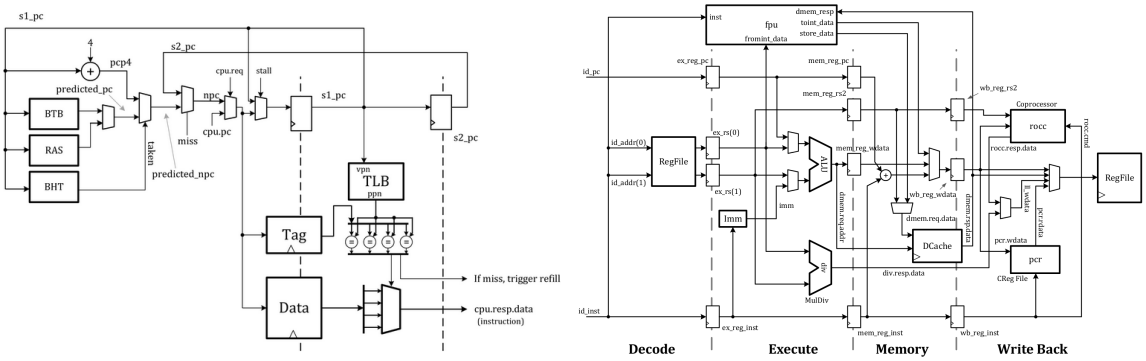

Figure 3.2: Rocket core focus. *Pcgen* and *fetch* stages on the left, the remaining four pipeline stages are shown on the right. [13]

3.1.2 Lagarto

Lagarto I: the first "mexican" processor, it is open-source and addressed to the research 3.3. Lagarto exploits a 32 bits RISC-V architecture and it is described in Verilog. Further improvements are being carried out. It was designed by the *Centro de Investigación en Computación, Instituto Politécnico Nacional, Mexico.*

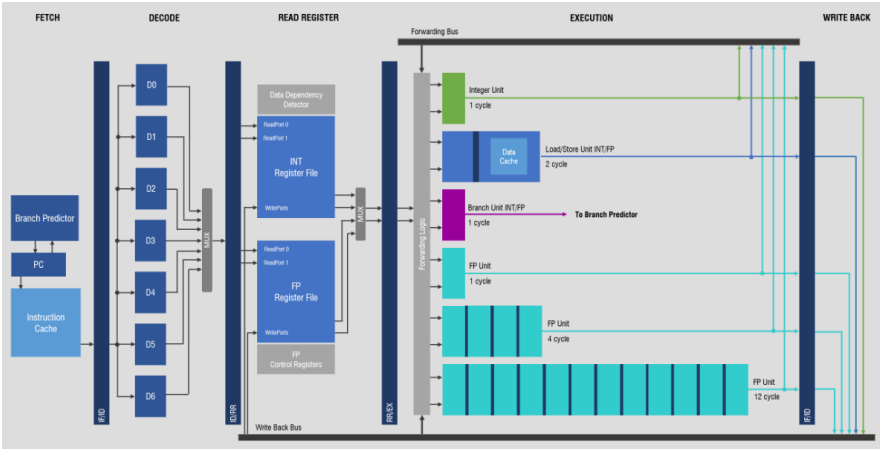

Figure 3.3: Lagarto I core pipeline. [15]

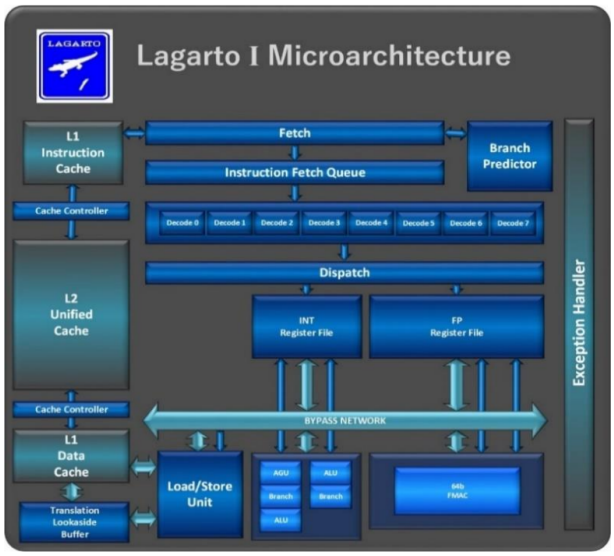

Figure 3.4: Lagarto I core microarchitecture. [16]

Lagarto I exploits a dynamic predictor and a complete bypass net to bring forward data among the functional units (e.g. integer unit, floating point unit, ...) allowing to improve the core performances.

Another version of this core is called Lagarto II and it has a different microarchitecture. In this thesis only Lagarto I is used and so, from now on, the name *Lagarto* will stands for *Lagarto I*.

3.2 *Lowrisc* project & *Untethered Rocket chip*

"LowRISC is a not-for-profit organisation working closely with the University of Cambridge and the open-source community" [13]. It created a fully open-sourced, Linux-capable, RISC-V based SoC; it can be directly used or exploited as basis for further processing and design. Thus, this system is prone to constant maintaining and varied developments.

During the last four years, six official releases have been published, improving some features and adding new functionalities from time to time (table 3.1).

Function	*Tagged-v0.1*	*Untethered-v0.2*	*Debug-v0.3*	*Minion-v0.4*	*Ethernet-v0.5*	*Refresh-v0.6*
Rocket Priv. Spec.	?	?	1.7	nearly 1.91	nearly 1.91	1.10
Tagged memory	*			*	*	
untethered operation		*	*	*	optional	*
SD card	tethered	SPI	SPI	SD	SD	SD
UART console	tethered	standard	standard/trace	standard/trace/VGA	standard/VGA	standard/VGA
PS/2 keyboard				*	*	*
Minion Core				*		
Kernel md5 boot check				*	*	*
PC-free operation				*	*	*
Remote booting					*	*
Multiuser operation					*	*
Compressed instructions						*
Debian binary compatible						*

Table 3.1: The six official *lowrisc* releases in comparison. [13]

This work provides the use of the lowrisc release Untethered-v0.2, as a starting point. The choice was made since BSC has an on-going project which exploits this release and it led to the following considerations:

1. focusing on the same system allowed to collaborate with the group working on that project;
2. this *under test* release could be less insidious with respect to an undiscovered version;
3. finally, merging the master thesis objective with the BSC on-going official *Lagarto* project could be an opportunity.

The release version 0.2 dates from December 2015 and it constitutes the *untethered Rocket chip*.

"Rocket Chip is an open-source System-on-Chip design generator that emits synthesizable RTL. It leverages the Chisel hardware construction language to compose a library of sophisticated generators for cores, caches, and interconnects into an integrated SoC". [14]

This is the definition given at the UC Berkeley, where it was developed for research and industrial purposes.

A high-level overview of the untethered Rocket chip is reported in the figure 3.5.

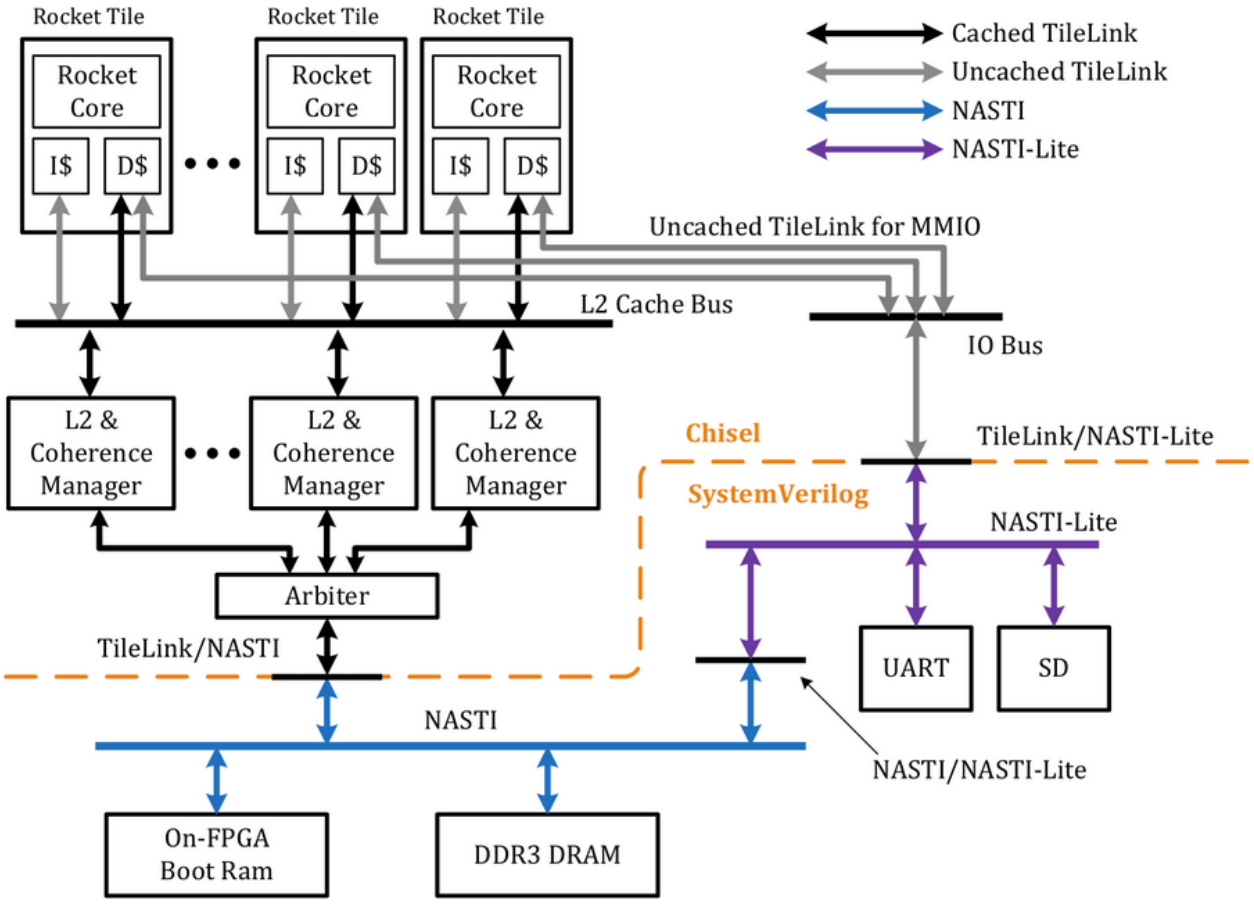

Figure 3.5: Untethered Rocket chip overview. [13]

The SoC is composed by a configurable number of Rocket Tile including a processor, an instruction cache iL1 and a data cache L1. The tiles share a banked L2 cache and an I/O bus.

As visible in the picture 3.5, the part at the bottom is described in SystemVerilog (known hardware description language based on verilog and extensions) and the one at the top is built in Chisel.

In the original Untethered Rocket chip, the processor included into the tile is the *Rocket core* presented in the section 3.1, exploiting the 32-bit instruction format of its RISC-V implemented version.

Recently, it was possible to replace the Rocket core with Lagarto in this SoC letting unaffected the rest of the tile and of the system. Thus, it is possible to talk about an alternative Rocket chip including the Lagarto core in the tiles.

Both the cases have been evaluated in this work but, since the whole system is the same except for the core, the steps performed, described in the following chapters, have been carried out by selecting a core and going on using it. It was chosen the Rocket core and so the original unthetered Rocket chip.

As regards the memory system, inside the tile, each core has its private L1 instruction and write-back data caches. It means that when a core changes the value of a variable, it is not upgraded also in the lower memories. If another core needs that updated value, the coherence mechanism is applied flowing through L2. This procedure is explained in the section 3.2.1. The implemented caches are non-blocking, so they allow the CPU to continue executing instructions while a miss is being handled; they exploit the miss status holding registers (MSHRs) that are hardware structure for tracking outstanding misses: one MSHR register for each miss to be handled concurrently.

The default values of some memory parameters are the following ones:

- Set associative mapped caches.

- Cache block size: 64 Bytes.
- Istruction cache L1 size: 16KB.
  Instructions cache L1 number of sets: 64.
  Instructions cache L1 number of ways per set: 4.
- Data cache L1 size: 16KB
  Data cache L1 number of sets: 64.
  Data cache L1 number of ways per set: 4.
  Data cache L1 number of MSHRs: 2
- Cache L2 (multi-bank) size: 128 KB. Number of L2 cache banks: NBanks = NTiles.
  Cache L2 number of sets: 256
  Cache L2 number of ways per set: 8.

The interconnection used in the "Chisel island" is bus-based exploiting a communication protocol called TileLink (it constitutes all the black and gray links in the figure 3.5). It is designed to implement a particular cache coherence policy within an on-chip hierarchy. TileLink will be deepened in the following dedicated section, focusing on its features and how it works in this system.

In the SoC there are two NASTI/NASTI-Lite interfaces towards to the FPGA peripherals. They implement a limited subset of the AXI/AXI-Lite functions. On the one hand, The NASTI interface is used by the L2 cache for memory reads and writes; on the other hand, the NASTI-Lite interface is used by the I/O bus for peripheral accesses.

The whole open project can be downloaded from the github. The main folder contains all the SoC sources, the RISC-V libraries and tools, the resources needed for the FPGA implementation and some files necessary for the configuration and for the simulation of the system.

The system is highly customizable, through a configuration file named *Configs.scala*. It is possible to choose a lot of parameters to customize the SoC, most of them are listed in the webpage lowrisc parameter, some main examples are listed below.

- The number of Tiles and so of the cores. The set number of L2 banks has to be equal to the chosen Tiles number.
- The number of ways and of sets of the L1 and L2 caches that are set-associative, but they can also behave also as direct mapped caches.
- The number of MSHRs in the data cache L1.
- The width of the data bus and size of a cache block.

Chosen the configuration for the SoC, it is possible to analyze it by carrying out several types of test, described in the first two sections of the chapter 4 dedicated to the methodologies.

3.2.1 TileLink

TileLink (TL) is a chip-scale interconnect standard able to manage multiple masters and slaves with coherent memory-mapped access to memory, implementing a particular cache coherence policy. [18][19]
The TL newest version 1.7.1 was released in December 2018 and it includes several upgrades with respect the older versions.
The TL release used in the unthetered Rocket chip is the 0.3.3 which involves the main basic features of the protocol.

It provides two types of *agents*:

- clients requesting access to cache blocks;
- managers supervising the propagation of cache block permissions and data.

Five independent transaction channels are defined with priority avoiding deadlock:

Finish >> Grant >> Release >> Probe >> Acquire.

- Acquire: the client starts a transaction signaling a block acquire request.
- Probe: the manager asks to the other clients if they have an upgrade version of that block.
- Release: the questioned clients answered to the probe of the manager. It is also used to voluntarily write back data when a cache client makes room for a new block.
- Grant: After that the manager communicates with backing memory if required, the needed data or permission travel to the original requestor (client).
- Finish: The manager receives the final acknowledgement indicating the end of that transaction.

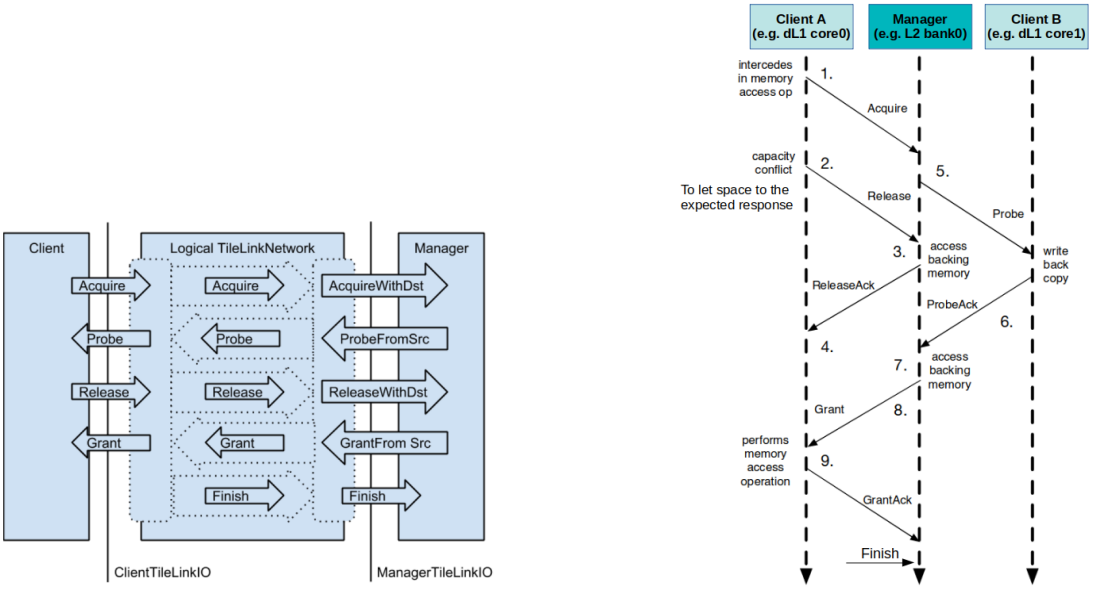

Figure 3.6: TL agents and channels (left) [18]. Overview of a TL complete transaction (right).

In each channel there are several signals involved: from the data and address signals up to all the necessary control (e.g type of operation, validity, ready).

In the analyzed system in fig. 3.5 TileLink is used for three different links:

1. L1 to L2: cached[3] Tilelink; L1 = client side; L2 = manager side.

2. L2 to NASTI interface: cached Tilelink; L2 = client side.

3. L2 to IO Bus interface: uncached[4] Tilelink; L1 = client side.

Focusing on the core/main memory path, it possible to get the simplified scheme shown in the figure 3.7.

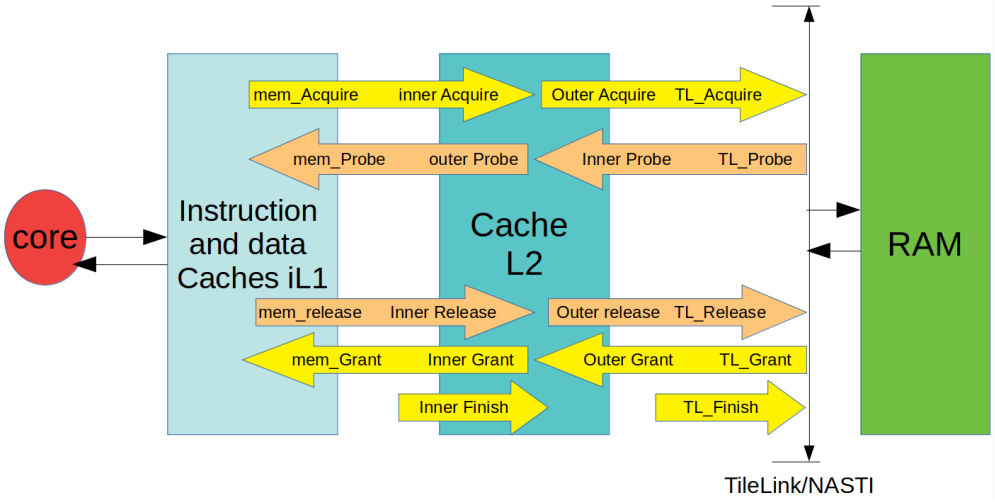

Figure 3.7: TL channels and links in untethered Rocket chip.

It is necessary to specify that data cache dL1 and instruction cache iL1 are separated and the *Probe* and *Release* channels are connected only between dL1 and cache L2. Indeed these two are the channels exploited only to manage the data coherence when it is needed and for voluntary cache block release in case of block replacement. The remaining channels (Acquire, Grant, Finish) are shared between iL1 and dL1 and used in each type of memory transaction.

[3]TL configuration providing the cache coherency using all five TL channels.

[4]TL configuration which does not provide the coherency, it use only the TL channels: Acquire, Grant and Finish.

In case of multiple tiles and multiple L2 banks, the L2 Cache Bus of the Rocket SoC (figure 3.5) consists of a crossbar (figure 3.8) in which the clients are the double of the managers. It happens because the tiles number and the L2 banks number are the same for the given constraint: each L2 bank is a manager, each cache in the Tile is a client and every tile includes an instruction and a data L1 cache.

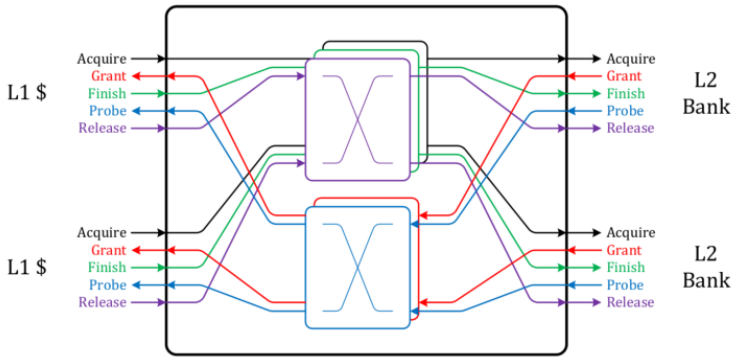

Figure 3.8: TileLink Crossbar. [20]

Similarly, the TL link between L2 and the TL/NASTI interface consists of a crossbar in case of multi-core system. It has only one manager and a number of clients equal to the number of L2 banks and so of the cores. As normal, an arbiter manages concurrent accesses before than the TL/NASTI converter.

3.3 UPV NoC

The *Universitat Politècnica de Valencia* (UPV) is developing a SoC based on Network-on-Chip. Each node of the network consists of a tile (in figure 3.9) including the following elements:

- A MIPS processor named *MIPSCORE* which includes the instruction cache, so it receives directly blocks of 16 instructions from NI.

- A data cache L1.
- A second level of cache L2.
- A bank of registers called *TILEREG*, with a configurable number of registers, currently it is equal to 32. This module is designed in such a way that the registers shared by all the tiles are at the beginning of the bank and specific tile register at the end.
- The Network Interface (NI) to convert the traditional signals in the packets format required for the data stream across the network and vice versa.
- Three routers to shunt the packets: one is addressed to data and memory traffic, and the other two are used for control and debugging support.
- A gather network (GN).

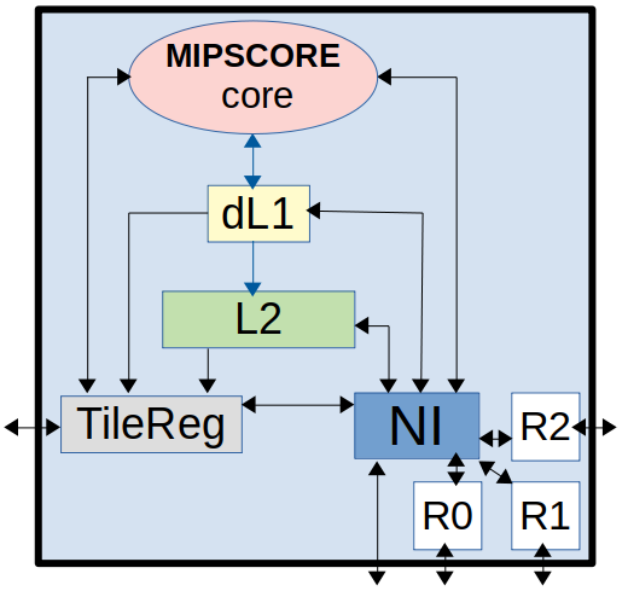

Figure 3.9: Simplified scheme of the UPV NoC Tile

This tile is used as a starting point in order to try to incorporate Lagarto in a NoC-based system, as described in the appropriate chapter 6.

Chapter 4

Methodology

This chapter concerns the methodologies exploited to simulate and analyze the systems involved in the whole experience.

4.1 RTL and Behavioral simulations of Rocket chip

The untethered Rocket chip was tested through software simulations and only later on the FPGA. Thus, from time to time, each step forward was processed with software tests (RTL and behavior simulations) and after verified on the programmable device.

The RTL simulation of Rocket chip is executed by using *verilator*, a free HDL simulator which compiles SystemVerilog and Verilog into single or multi-threaded C++ or SystemC code. [21]
A part of the system is already described in SystemVerilog (see figure 3.5); on the other hand, the parts in Chisel are converted into Verilog sources. After that, the simulation can be run and all the SystemVerilog/Verilog sources are processed in cpp files.
The benchmark which runs on the simulated SoC has to be designed in C or in assembly (or both) and its executable file has to be generated by considering the RISC-V ISA and so the corresponding libraries.
By exploiting the obtained files and the provided tool *Spike*, it is possible to have a behavioral

simulation and so to generate the waveforms related to the benchmark running on the SoC, saving the information in a *vcd* (Value/Variable Change Dump) file.
Finally, the waveforms are displayed by using an appropriate visualization tool named *gtkwave*.

These simulations were carried out by increasing the number of Tiles in the system, obtaining a dual-cores, four-cores system and an eight-cores system. In the two last configurations, the tests were made by exploiting the cluster *arvei* belonging to the *Universitat Politècnica de Catalunya* (UPC), because of the needed high computing power.

Verified the proper behavior of all the multi-cores configurations, it was decided to go on with the interconnect analysis by using the 4-cores system due to the following considerations:

- It allows to get a general idea about how the multi-cores system work. It can be widespread and extended to higher tiles number system.
- The FPGA used to make tests is able to contain up to the 4-tiles configuration. It covers almost all the space over the programmable device.

4.2 FPGA testing of Rocket Chip

A part from the software simulations, the system can also tested on the FPGA. It can be synthesized and implemented through *Vivado Design Suite* offered by *Xilinx*. The generated bitstream is run and the provided Kintex-7 KC705 Evaluation Platform (xc7k325tffg900-2) is programmed.

To verify and to analyze the functioning of the system implemented on the FPGA running the chosen benchmark, two procedures are exploited.

The first one was used only in case of single-core system, it deals with the design of a test exploiting the UART communication to print some real-time information about the built

SoC on the connected PC screen, proving the proper functioning of the *under-test* system.

This type of FPGA verification was not used for the multi-cores systems because of a problematic handling of the *UART* port in the event of concurrent accesses; in these configurations only the second FPGA testing procedure was exploited. It is the most useful for our goals since it allows to observe the internal signals of the system. Indeed, thanks to the Integrated Logic Analyzer (ILA) tool provided by *Vivado*, it is possible to instantiate some debug modules (even called *debug cores*). In this way, it is possible to select the signals of interest and to store a configurable number of their samples during the execution on the FPGA, likewise using some appropriate probes. Finally the corresponding waveforms are displayed, showing the correctness of the tested system.

This method is exploited during the analysis of the Rocket chip interconnection, the debug modules were employed to record the trends of the program counters of the cores and the TileLink signals involving the data transfer along the memory system.

4.3 Approach to analyze the interconnect

The analysis of the interconnection involves its stress, produced by generating the highest number of transactions. This scenario, which normally corresponds to the worst case in terms of performance, is particularly relevant for critical real-time embedded systems, since it allows estimating the Worst-Case Execution Time (WCET) of critical real-time tasks, as needed to guarantee that the allocated time budget suffices for their execution. In order to get it, the idea consists in designing the benchmarks so as to lead to as many caches misses as possible, by ensuring a constant and dense data stream through the different levels of the memory system and so along the TileLink channels.

The initial study involves the simplest system configuration with only one tile.
To achieve the predefined goal, a first faced issue consisted in how to create systematical

cache misses.
The SoC with its original configuration parameters provides set-associative caches L1 and L2 with random replacement policy which leads to an unpredictable estimate about the cache block that is replaced time by time. Thus, keeping this caches configuration, is not possible to get systematical caches misses.

To overcome this problem, the idea was to modify some parameters in the *Configs.scala* file in order to implement direct mapped caches in place of set associative ones; in this way the random replacement policy becomes invalid and the prevision on the replaced cache block can be done. Direct-mapped L1 and L2 are obtained by setting the number of ways per set equal to "1".

The size of the caches depends on the chosen parameters: cache block size, number of sets, number of ways per set. It has been chosen to keep fixed the cache block size (64 Bytes) and also the number of sets (dL1 sets = 64, L2 sets = 256). Anyway the benchmarks can be adapted to whatever combination of these parameters simply adjusting the count of the addresses bits involved (belonging to the fields: tag, index, block and byte offset).

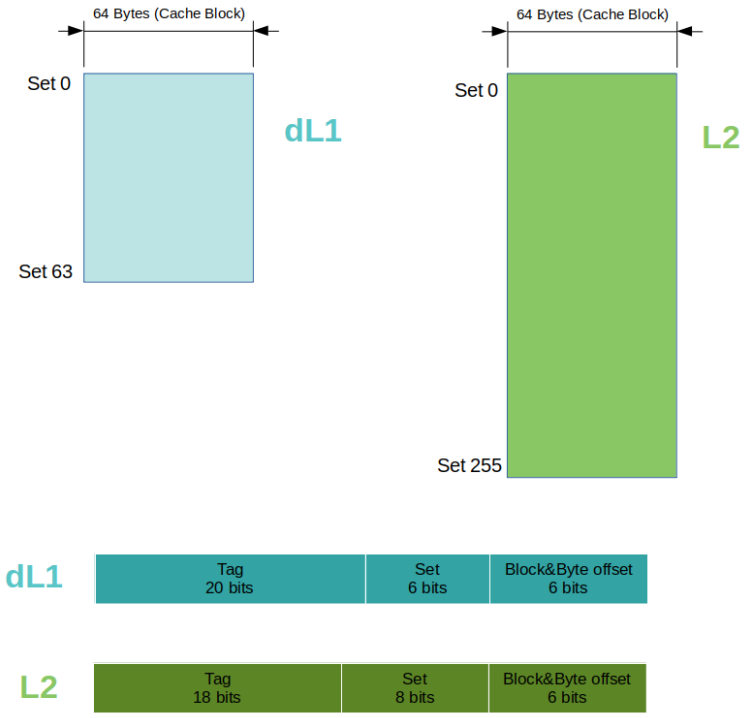

Figure 4.1: L1 and L2 caches and addresses composition

The designed benchmarks are divided in two groups depending on the type of generated misses:

- Systematical dL1 miss & L2 hit. It is obtained by loading data consecutively on a loop from equal memory addresses except for a bit of the two MSBs in the field **Set** of the L2 address (described in green in the figure 4.1). L2 is four times bigger than dL1, its addressing involves the above-mentioned two bits more in the Set field with respect to the dL1 address (cyan in the same figure). This type of cyclic load requests leads to systematical dL1 cache misses, but whose data instead fit in the L2.
  Considering two memory locations at the addresses A and B from where the data are read/loaded:

  A = 32b00....00100000000000000 (tag=0..01, set=00000000, b.offset=000000)

  B = 32b00....00101000000000000 (tag=0..01, set=**01**000000, b.offset=000000)

The basic benchmark structure is the following one. Below the corresponding waveforms are reported.

load A

load B

label_loop:

load A

load B

jal label_loop

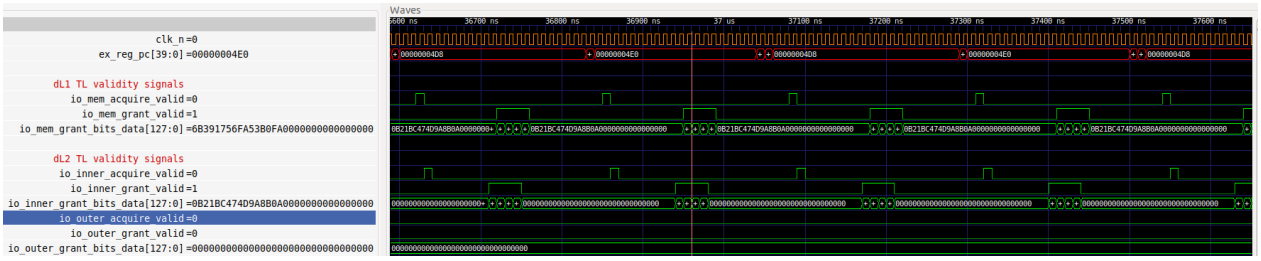

Figure 4.2: Single-core system, Systematical dL1 miss & L2 hit. dL1 ask for a cache block to L2 that owns it and so it does not need *outer acquire* to request these data to the main memory. The *outer acquire* validity signal (highlighted in blue) is fixed at *0* and no *outer grant* data flow from main memory to L2.

- Systematical dL1 miss & L2 miss. It is achieved by loading data consecutively on a loop from equal memory addresses except for a bit in the field **Tag** of the L2 address (figure 4.1). This operation guarantees that always the same location is replaced in L1 and in L2, this time data are unable to fit in the latter.

  The basic benchmark structure is still the one above described, in this case two possible memory addresses A and B are:

A = 32b00....00100000000000000 (tag=0..01, set=00000000, b.offset=000000)

B = 32b00....01000000000000000 (tag=0..**10**, set=00000000, b.offset=000000)

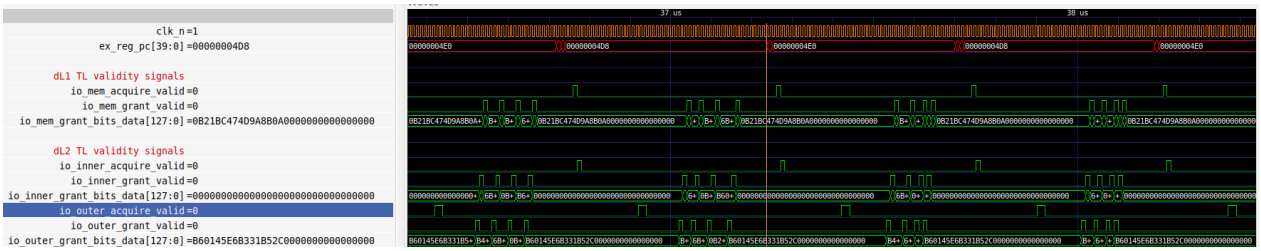

Figure 4.3: Single-core system, Systematical dL1 miss & L2 miss. dL1 ask for a cache block to L2 that it does not have, so it needs *outer acquire* to request these data to the main memory. In this case the outer acquire validity signal (highlighted in blue) goes to *1* and the outer grant data flow from main memory to L2.

The figures 4.2 and 4.3 are displayed on *GTKWave*, after RTL (verilator) and behavioral (spike) simulations. These tests, how even all the following ones, were carried out also on the FPGA by exploiting the Vivado ILA tool previously described to debug. In this way it was possible to verify the functioning of the SoC on the programmable device and also to observe eventual discrepancies between software and FPGA system tests.
In this context, in case of these two benchmarks running on single-core system, there are no differences in terms of trend and of latency as regards the TL signals involved.

Starting from the benchmarks previously described in single-core configuration, the same test concept is appropriately extended to multi-cores system.

In this case it is considered on the one hand core 0 as *main core* and, on the other hand, the other cores that can be defined as *contenders* (at the same level among them); so this approach basically consists in observing the influence of other cores on the main one during concurrent memory requests under different scenarios.

Whereas each tile includes a core and that the number of L2 banks has to be equal to the tile number, first of all, it was necessary to understand in which way the main memory maps the data in L2 when there is more than one bank.
As simply shown below in the figure 4.4, the RAM maps the memory locations alternately among the several L2 banks. In the following graphical explanation let us suppose that there are 2 cores and so 2 banks of L2, but anyway the concept can be extended to higher numbers of cores/L2banks. The number of ways and sets of L1 and L2 are the same chosen in the one-core simulations of the previous section and so the parameters are:

- Ncores = NL2banks = 2.
- Cache block size = 64 Bytes.
- dL1: ways/set=1; sets=64.
- L2: ways/set=1; sets=256.

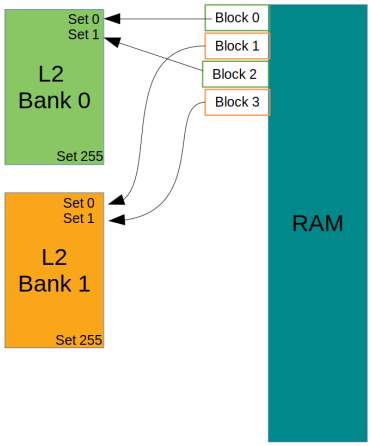

Figure 4.4: How the Main Memory is mapped in L2 multi-bank.

As visible, in the case of two cores system, simply even blocks of the RAM are mapped in L2 bank0 and odd blocks into L2 bank1.

The main parameter evaluated in these simulations is a time interval that it is going to be called $\delta\tau$ and it represents the difference on time between the *mem_acq_valid* signal of the data cache L1 of the core 0 and the *inner_finish_valid* signal of the used L2 bank; thus it is the time interval from when the dL1 core 0 starts its request for a data block to L2 up to this transaction is completed properly and the requestor receives the data.

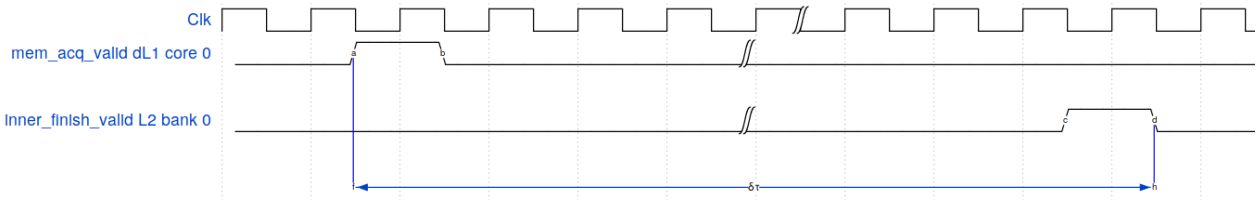

Figure 4.5: $\delta\tau$: parameter evaluated to estimate the latency transaction.

By observing the final tables reporting the values of the latency $\delta\tau_o$ for each analyzed case (chapter 5), it is possible to have an idea about the system interconnect performances and about how the other cores influence the timing execution of core0 memory requests.

4.4 Simulation of the UUT NoC Tile

The simulations of the NoC tile have been carried out through the *Vivado simulator*. It was designed a testbench in *SystemVerilog* instantiating the involved blocks, from time to time, observing the waveforms of the probed signals.

Firstly, to collocate Lagarto in place of the original MIPSCORE inside the UPV NoC tile, it was created a fake instruction memory addressed to send the instruction data to the core, generating appropriately the signals involved. Lagarto works with this fake memory as it was its external instruction cache.
Adjusted and verified the functioning of the core, the following step was to append the data cache L1 of the original Tile. The interfaces were not compatible, so some modules were purposely designed in *verilog*.

In accordance with this operation, step by step is added a new block of the UPV NoC tile in the testbench and it is connected to Lagarto, changing some modules and adapting the interfaces when necessary.

Chapter 5

Analysis of the interconnections

5.1 System configuration

The first operations carried out on the system consisted in setting and simulating the system in its single-core and multi-core configurations.

The default value of Tiles is equal to 1. This parameter is set in the previously described *Configs.scala* file, in addition the initial status of the other configurations files allows only the single-core system works.

The waveforms obtained from the simulations have been converted into simplified *chronograms* in order to clarify the dynamic. Here, the names used for the signals match with the ones in the figure 3.7.

A first timing diagram in figure 5.1 is obtained by running a simple benchmark on the simulated system, a matrix multiplication is executed and particular attention was given to the data stream trough the memory system.

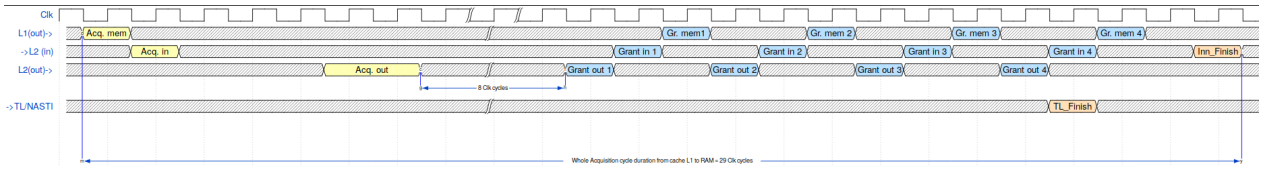

Figure 5.1: Single-core simulation. Timing diagram of a data request flowing through the entire memory system.

As visible, for a cache block acquire request of the core, four beats of data are received. It happens because a cache block consist of 64 Bytes that it the size of information that can be transferred for each acquire/grant iteration. On the other hand, the data bus consists of 128 bits, so four pulses are needed to transfer the block. Since each data (intended also as instruction) is composed by four Bytes, every transfer involves 16 data of 32 bits.

The successive step consisted in configuring the multiple-cores system. A first change is done in the *Configs.scala*, by setting the number of Tiles and of L2 banks equal to two. Nevertheless, even if the two cores are instantiated, the execution of the second one was stuck. To solve this issue and to manage properly the two cores, it was necessary to act on some configuration and initialization files (crt.S and syscalls.c) in two ways:

1. Deleting some lines of code which allowed only the first core goes on executing.

2. Adding an instruction to store the core id in a core temporary register which could be read directly in the benchmark, allowing to split the workload among cores depending on its identifier value, by giving them different functions. It was needed because the original initialization file only saved the id core in a reserved register which is inaccessible in user mode.

After having verified the correct execution of the two cores in the simulations, it was possible to observe how the coherence is managed among cores and their private data caches L1. A simple chronogram was extracted and reported in the figure 5.2.

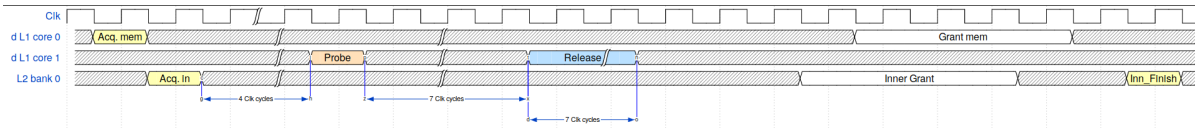

Figure 5.2: Two-cores system simulation. Chronogram describing the coherence management through L2 by using the Probe and Release channels.

Data cache core 0 (client A) asks for acquiring a data block to cache L2 bank 0 (manager) that sends a necessary *Probe* to the other client Data cache core 1 (client B). Basically, the procedure takes place as described in the figure 3.6. L2 bank 1 waits for receiving a *Release* for the Probe that was sent and it communicates with backing memory if required (not in this case). Having obtained the required cache block, the manager L2 responds to the original requestor with the Grant data and the transaction is completed with the *Finish* signal.

Overall, it was observed that the communication works properly. Nevertheless, in presence of multiple masters/cores, it was noted that, even if the coherency policy basically works, this TileLink version is prone to bugs and a perfect functioning is not guaranteed.

5.2 Examined scenarios and latency results

The basic procedure used to carry out the interconnect analysis is explained in the section 4.3. This paragraph described the system configuration set during this study, the wide range of designed benchmarks, the latency results and the related comments.

The system parameters set in the configuration files have been kept unaffected (expect for the Tiles and L2 banks numbers) during the simulations of all benchmarks to allow a coherent evaluation and comparison of the several scenarios.

System configuration file (Configs.scala)	
Parameter name	**Assigned value**
N Tiles	Variable (1, 2, 4)
N L2 banks	N Tiles
L1D_MSHRS	2
L1D_SETS	64
L1D_WAYS	1
L1I_SETS	64
L1I_WAYS	1
L2_XACTORS	2
L2_SETS	256
L2_WAYS	1
TC[1]_XACTORS	1
TC_SETS	256
TC_WAYS	1
N IO Sections	4
N Mem Sections	4

Table 5.1: System configuration parameters used for the interconnect analysis.

The system has been simulated in single core, dual core and quad core configurations for all the scenarios described in the table 5.2. On each of these three different sets of benchmark were designed, by changing their structure to lead to higher contentions, in theory and as proven in the tables 5.3 and 5.4, pound for benchmarks:

contention set 1 < contention set 2 < contention set 3.

1. Set 1: loops composed by only two load instructions (4.3).

2. Set 2: unrolled loops consisting of 32 load instructions with the purpose of increasing

[1]TC stands for Tag Cache, it is used in the TL/NASTI conversion stage.

the relative number of loads by decreasing the relative impact of control loop instructions.

load A
load B
label_loop:
load A
load B
... 32 times overall ...
load A
load B
jal label_loop

3. Set 3: unrolled loops of 32 *atomic* load instructions. In the RISC-V ISA complex atomic memory operations on a single memory word are performed with the load-reserved (LR) and store-conditional (SC) instructions. LR loads a word from the address in a source register, places the sign-extended value in a destination register, and records a reservation on the memory address. In this case it is used RISC-V 32 and so the suitable load instruction is "lr.w"; it has not bits dedicated to the offset so each address has to be explicitly written. In general, atomic loads either cause the same contention as regular loads, or cause higher contention if they lock some specific resources (e.g. the interconnect).

Atomic load A
Atomic load B
label_loop:
Atomic load A
Atomic load B

... 32 times overall ...

Atomic load A

Atomic load B

jal label_loop

System Conf.	Scenario label	Scenario description
Single core	**1A**	L1 miss & L2 hit.
Single core	**1B**	L1 miss & L2 miss.
Dual Core	**2A**	For both tiles L1miss&L2hit: Core0 and Core1 access to the same L2bank but replacing blocks at different addresses.
Dual Core	**2B**	For both tiles L1miss&L2hit: Core0 replaces blocks in L2bank0; Core1 replaces blocks in L2bank1.
Dual Core	**2C**	For both tiles L1miss&L2miss: Core0 and Core1 access tothe same L2bank but replacing blocks at different addresses.
Dual Core	**2D**	For both tiles L1miss&L2miss: Core0 replaces blocks in L2bank0; Core1 replaces blocks in L2bank1.
Dual Core	**2E**	Core0 L1 miss & L2 hit, Core1 L1&L2 miss, both cores work with the same L2 bank but different couple of addresses are replaced.
Dual Core	**2F**	Core0 L1&L2 miss, Core1 L1 miss & L2 hit, both cores work with the same L2 bank but different couple of addresses are replaced.
Dual Core	**2G**	Core0 L1 miss & L2 hit, Core1 L1&L2 miss, each core work with a different L2 bank.
Dual core	**2H**	Core0 L1&L2 miss, Core1 L1 miss & L2 hit, each core work with a different L2 bank.

Quad core	**4A**	For all tiles L1miss&L2hit: Core0 and Cores 1,2,3 access to the same L2bank but replacing blocks at different addresses.
Quad core	**4B**	For all tiles L1miss&L2hit: Core0 replaces blocks in L2bank0; Core1 replaces blocks in L2bank1; Core2 replaces blocks in L2bank2; Core3 replaces blocks in L2bank3.
Quad core	**4C**	For all tiles L1miss&L2miss: Core0 and Cores 1,2,3 access to the same L2bank but replacing blocks at different addresses.
Quad core	**4D**	For all tiles L1miss&L2miss: Core0 replaces blocks in L2bank0; Core1 replaces blocks in L2bank1; Core2 replaces blocks in L2bank2; Core3 replaces blocks in L2bank3.
Quad core	**4E**	Core0 works replacing different addresses in L2 bank 0 (L1miss&L2hit); Core1, Core2 and Core 3 also work with L2bank0 but having systematically L1&L2 miss.
Quad core	**4F**	Core0 works replacing different addresses in L2 bank 0 (L1&L2miss); Core1, Core2 and Core 3 also work with L2bank0 but having systematically L1miss & L2 hit.
Quad core	**4G**	Core0 L1 miss & L2 hit; Core1, Core2 and Core3 L1&L2 miss, each core work with a different L2 bank.
Quad core	**4H**	Core0 L1&L2 miss; Core1, Core2 and Core3 L1 miss & L2 hit, each core work with a different L2 bank.

Table 5.2: Listing of the examined scenarios for the interconnect analysis.

Contenders	*TUA core 0 (loops composed by only 2 load instructions)*			
	L1miss&L2hit		L1miss&L2miss	
	Same L2 bank	Different L2 banks	Same L2 bank	Different L2 banks
-	1A = 16 clk	-	1B = 40-42 clk	-
1xL1missL2hit	2A = 15 clk	2B = 15 clk	2F = 40-43 clk	2H = 40-42 clk
1xL1missL2miss	2E = 15-20 clk	2G = 15 clk	2C = 40-42 clk	2D = 40-42 clk
3xL1missL2hit	4A = 25 clk	4B = 15 clk	4F = 41-56 clk	4H = 40-42 clk
3xL1missL2miss	4E = 61-70 clk	4G = 15 clk	4C = 77-79 clk	4D = 42 clk before all cores get stuck
Contenders	*TUA core 0 (unrolled loops, 32 load instructions)*			
	L1miss&L2hit		L1miss&L2miss	
	Same L2 bank	Different L2 banks	Same L2 bank	Different L2 banks
-	1A = 16 clk	-	1B = 40-42 clk	-
1xL1missL2hit	2A = 15 clk	2B = 15 clk	2F = 40-43 clk	2H = 40-42 clk
1xL1missL2miss	2E = 15-20 clk	2G = 15 clk	2C = 41-43 clk	2D = 40-43 clk
3xL1missL2hit	4A = **21/29 clk**	4B = 15 clk	4F = 41-56 clk	4H = 40-42 clk
3xL1missL2miss	4E = 61-70 clk	4G = 15 clk	4C = 77-79 clk	4D = All cores get stuck immediately
Contenders	*TUA core 0 (unrolled loops 32 instr, atomic instr for the contenders)*			
	L1miss&L2hit		L1miss&L2miss	
	Same L2 bank	Different L2 banks	Same L2 bank	Different L2 banks
-	1A = 16 clk	-	1B = 40-42 clk	-
1xL1missL2hit	2A = **17 clk**	2B = 15 clk	2F = 40-42 clk	2H = 40-43 clk
1xL1missL2miss	2E = **15-19 clk**	2G = 15 clk	2C = **58-64 clk**	2D = 40-44 clk before all cores get stuck*
3xL1missL2hit	4A = **28 clk***	4B = 15 clk*	4F = **43-58 clk***	4H = 40-42 clk*
3xL1missL2miss	4E = 61-70 clk*	4G = 15 clk*	4C = 77-79 clk*	4D = 42-44 clk before all cores get stuck*

Table 5.3: Latency results obtained from software (RTL and Behavioral) simulations.

Contenders	*TUA core 0 (loops composed by only 2 load instructions) on FPGA*			
	L1miss&L2hit		L1miss&L2miss	
	Same L2 bank	Different L2 banks	Same L2 bank	Different L2 banks
-	1A = 15 clk	-	1B = 35 clk	-
1xL1missL2hit	2A = 15 clk	2B = 15 clk	2F = 37 clk	2H = 35 clk
1xL1missL2miss	2E = 15 clk	2G = 15 clk	2C = 35 clk	2D = 35 clk
3xL1missL2hit	4A = 25 clk	4B = 15 clk	4F = 39/49 clk	4H = 35 clk
3xL1missL2miss	4E = 56 clk	4G = 15 clk	4C = 65 clk	4D = 35 clk
Contenders	*TUA core 0 (unrolled loops, 32 load instructions) on FPGA*			
	L1miss&L2hit		L1miss&L2miss	
	Same L2 bank	Different L2 banks	Same L2 bank	Different L2 banks
-	1A = 15 clk	-	1B = 35 clk	-
1xL1missL2hit	2A = 15 clk	2B = 15 clk	2F = 37 clk	2H = 35 clk
1xL1missL2miss	2E = 15 clk	2G = 15 clk	2C = 35 clk	2D = 35 clk
3xL1missL2hit	4A = 25 clk**	4B = 15 clk**	4F = 39/49 clk	4H = 35 clk
3xL1missL2miss	4E = 56 clk	4G = 15 clk**	4C = 65 clk**	4D = 35 clk**
Contenders	*TUA core 0 (unrolled loops 32 instr., atomic instr for the contenders) on FPGA*			
	L1miss&L2hit		L1miss&L2miss	
	Same L2 bank	Different L2 banks	Same L2 bank	Different L2 banks
-	1A = 15 clk	-	1B = 35 clk	-
1xL1missL2hit	2A = **17 clk**	2B = 15 clk	2F = 35-38 clk	2H = 35 clk
1xL1missL2miss	2E = 15 clk	2G = 15 clk	2C = **55 clk**	2D = 35 clk
3xL1missL2hit	4A = **28 clk***	4B = 15 clk*	4F = **38/39 clk***	4H = 35 clk
3xL1missL2miss	4E = 56 clk	4G = 15 clk*	4C = 65 clk*	4D = 35 clk*

Table 5.4: Latency results obtained from FPGA tests.

Latency tables legend

x-y clk means values within the interval [x;y].
x/y clk means that the latency values are these ones alternately (i.e. x,y,x,y,..).

* These cases are related to the verilator and FPGA simulations with atomic instructions. To make the benchmark was executed properly, the number of load instructions inside the loops was shortened to less than 16 instructions (assuming because of the atomic instructions limitations, described below).

** These cases are related to the fpga simulations without atomic instructions. To make the benchmark was executed properly, the number of load instructions inside the loops was reduced (supposing a different reason with respect to the case above, blaming the FPGA shortcomings).

N.B. Proper execution of the benchmark stands for a program execution in which the simulation goes on carrying out the designed systematical functioning during all the simulation time. When it does not take place, in this context it means that the program counter(s) of the core(s) gets stuck on an instruction without proceeding its designed operation.

Latency tables comments

- *Latency values meaning.* As mentioned in the section 4.3, the analyzed latency $\delta\tau$, whose values fill the tables above, consists in the time interval from when the dL1 core 0 starts its request for a data block to L2 up to this transaction is completed properly and the requestor receives the data.

- *Clock frequencies.* The tables report $\delta\tau$ in terms of clock cycles. To have an idea of the latency in terms of time is sufficient to know the clock period which in the *verilator* simulations is equal to 10 ns. As regards the FPGA tests, the main clock frequency of the system on the programmable device is 200 MHz, so the clock period is 5 ns (i.e half of the clock period of the software simulations).

- *Overall results observation.* As expected the system comes across higher contention in the third set of benchmarks consisting of atomic loads. On the other hand, the second set presents an unique different case with respect to the first one. In general, When the cores work with different L2 banks, no contention is observed, so the crossbar between L1 and L2 manages the concurrent requests properly without delaying them. Instead, in the case in which the cores needs data from the same L2 bank, higher latency values and so higher contentions show up with the increase of the contenders number. As regards single core and dual core systems, the latency results almost coincide, it occurs because of the set *L2_Xactor* value [2]; a detailed explanation is outlined in the following section 5.3.

- *Discrepancy results between FPGA and software tests.* In case of L1 and L2 misses of core 0 or of the contenders there is a considerable difference between the latency values obtained in the verilator simulations and on the FPGA. This discrepancy goes from 7 up to 14 clock cycles depending on the number and on the systematical behaviors of the cores. This means that the FPGA implementation execute faster (in terms of clock cycles) the benchmarks which need to take data from the main memory with respect to the software simulation. Thus the part of the system between L2 and the RAM is built with some small differences during the two types of test.

- *Atomic instructions restriction.* The main disadvantage of the atomic instructions LR/SC is livelock; indeed LR/SC atomic sequence could livelock indefinitely on some systems. This phenomenon can be avoided with an architectural guarantee of eventual forward progress. So certain constrained LR/SC sequences are guaranteed to succeed eventually. Among the limitations which lead to ensure the proper functioning of the system in presence of LR/SC sequences there is one which justified the obtained results. The length of LR/SC sequences has to fit within 64 contiguous instruction bytes in the base ISA to avoid undue restrictions on instruction cache and TLB size and

[2]It was set *L2_Xactor* = 2. It was not used *L2_Xactor* = 1 since with this value the quad core system is not able to execute the designed benchmarks, indeed all the cores get stuck in an initialization instruction.

associativity. Considering that each atomic load is composed by 32 bits (4 bytes), the maximum number of instructions in a LR sequence who should allow the system works is $\frac{64bytes}{4bytes} = 16$ instructions placed sequentially in memory. LR/SC sequences that do not meet these constraints might complete on some attempts on some implementations, but there is no guarantee of eventual success. [22] In these simulations, for all the system configurations, the sequences of LR into the loops had to be of 32 instructions, but the quad core system did not work, no core was able to execute the program; it makes sense since the requirement described above was not respected and so the proper functioning could not be ensured. That is why, in order to get meaningful latency results, the number of atomic loads inside the loops was reduced just lower than 16 in the benchmarks running on the four cores system.

5.3 Focus on L1/L2 interface

The final part of this analysis focuses on the L1/L2 interface, outlining the different stages of a memory request with its timing constraints and evaluating the influence of the parameter *L2_Xactor*. It is briefly defined as the number of trackers in L2: trackers are provided to allow multiple memory requests to be served in parallel.

The latency results, reported in the previous section, show that there is a high contention when the cores do concurrent accesses in the same L2 bank. This scenario is deepened in case of L1 miss & L2 hit for double-cores and four-cores systems.

To better explain the analysis outcome, some chronograms have been extracted and drawn in the cases of set 1 (rolled loops, two instructions), set 2 (unrolled loops of 32 instructions) and set 3 (unrolled loops, atomic instructions for the contenders). Subsequently the ones related to first set are going to be appended; anyway the features of the transaction in the system are always the same; simply, different structures of the benchmark can lead to some slight differences in the waveforms trends.

First of all, the complete chronograms related to the case 2A e 4A are reported, with the appropriate description. Secondly, there will be a focus on the parameter L2_Xactor, analyzing its effects on the transactions series.

Benchmark 2A: 2-cores system, core 0 has L1 miss and L2 hit, the contender core 1 has also systematical L1 miss and L2 hit too in the same L2 bank 0 but replacing different couples of addresses.

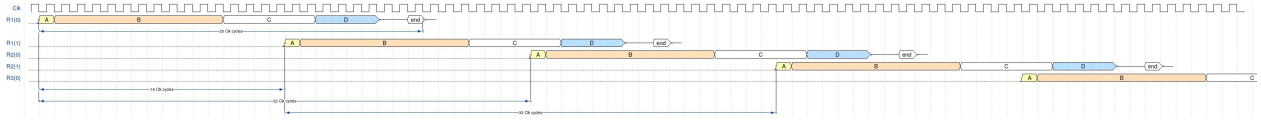

Figure 5.3: Chronogram related to the benchmark 2A by setting L2_Xactor=1

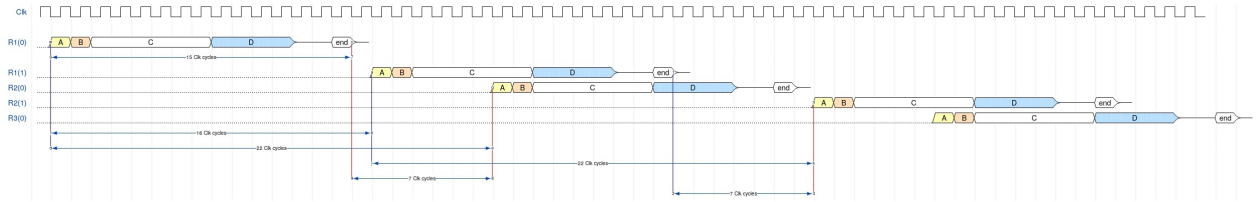

Figure 5.4: Chronogram related to the benchmark 2A by setting L2_Xactor=2

Benchmark 4A: 4-cores system, core 0 has L1 miss and L2 hit, the contenders core 1, core 2, core 3 have also systematical L1 miss and L2 hit in the same L2 bank 0 but replacing different couples of addresses.

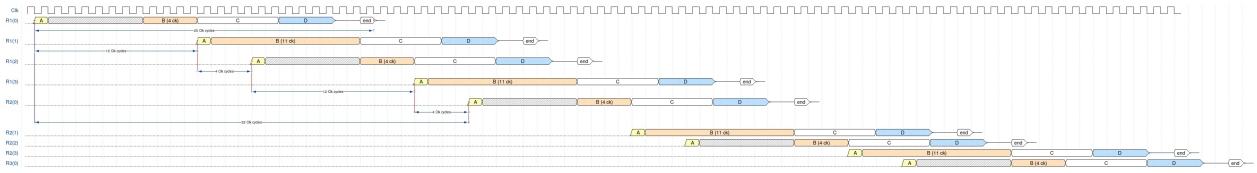

Figure 5.5: Chronogram related to the benchmark 4A (L2_Xactor=2)

In these chronograms some letters are used to refer to particular transaction phases. The legend, explaining the different highlighted phases of a single transaction, is now reported.

1. A : the data cache L1 sends an acquire request to L2 bank 0, [1 ck cycle].
2. B : the L2 bank 0 validate the request [variable ck cycles (1/4/11 in our cases)].
3. C : time interval from when L2 bank 0 has validated the acquire request up to it starts to send the required cache block to the dL1 requestor, [6 ck cycles].
4. D : the L2 sends out the required data to the dL1 requestor, [4 ck cycle].
5. *end* : TL finish signal which indicates the end of the transaction dL1/dL2. So it indicates that the dL1 receives the data and not that the data has been stored in the register of the core (not yet). Indeed, the core receives that data at the third clock cycle after the *end* signal.

As regards the requests, the labels on the left identifying them, have the following meaning: R1(0) stands for 1 st request of core 0; R1(1) stands for 1 st request of core 1; R1(2) stands for 1 st request of core 2; R1(3) stands for 1 st request of core 3; R2(0) stands for 2 nd request of core 0; R2(1) stands for 2 nd request of core 1 and so on...

It is possible to notice that in every case the interval between the "end" of a transaction (dL1/dL2) and a new acquire request of the same tile is 7 clock cycles; so there are 4 clock cycles between when the core effectively receives the data related to its previous request and when it starts the new acquire request.

As regards the 4-cores case, looking at chronogram 5.5 the following considerations have been done:

- A never overlaps with A (even if generally it could happen), B with B (it can not occur, unique channel for acquire request shared among tiles), and D with D (it can not take place, once again there is only an unique channel for data grant shared among the tiles).

Only C can overlap with C (specifically, what it is possible to see is that the second part of C of the request i is overlapped with the first part of C of the request $i+1$).

- There are several critical paths: Due to B and given that there are 7 cycles among requests of the same core, it cannot send again after at least 30 cycles.
- Due to the critical paths (29-30 cycles) and particular alignments of events, the repetition interval reaches 32 cycles.

To understand the difference of the "B" stage in the chronograms 5.3 and 5.4 and so the task of *L2_Xactor*, let us observe the following chronograms that have a different structures with respect to the previous ones where each line corresponded to a request in its different stages. In these diagrams, the lines correspond to specific signals of the tiles and of L2, so the stages of a same transaction (A,B,C,D,end) are distributed over the lines in iterative way (request by request).

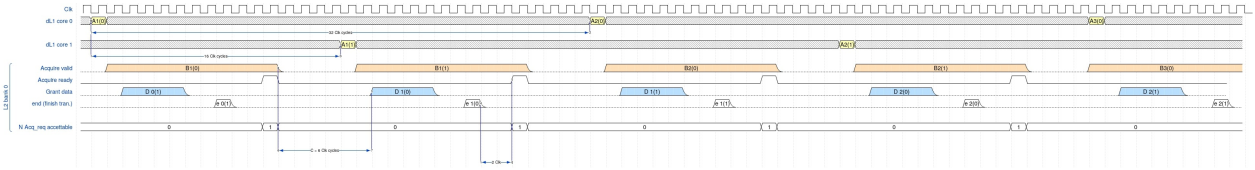

Figure 5.6: Alternative chronogram 2A with L2_Xactor=1

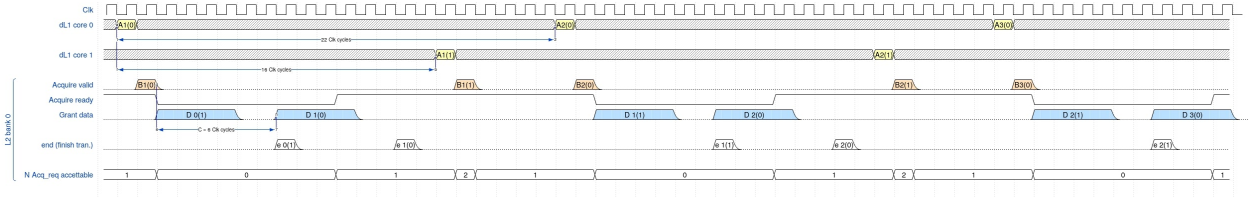

Figure 5.7: Alternative chronogram 2A with L2_Xactor=2

Similarly to the labels previously assigned to the requests, here each stage contains information about the id of the core which is executing it and its request number, e.g. A1(0)

stands for stage "A" of the first request of core 0.

As regards the dynamic, first of all, a new acquire request can not start before than the L2 is "ready" to accept another request.
The value assigned to L2_Xactor decides in which cases the L2 can be "ready".

1. L2_Xactor = 1 : L2 is ready only when there are no transactions pending. The state "B" takes 11 clock cycles because the *ready* signal is being waited. During that cycles L2 is not ready to accept that request because it is serving the previous one and it can accept only one request per time because L2_Xactor=1. A new request can be accepted after 2 clk cycles that there is the "end" state of the previous transaction.

2. L2_Xactor = 2 : L2 is ready when there are no transactions pending or also when there is only one pending. Thus L2 does not need to wait in the "B" state because the "ready" signal remains at "1" even if there is another acquire request pending.

5.4 Summary of the analysis

A brief summary with the main considerations resulted from this analysis of the system interconnect in multi-core configurations is now reported.

- The interconnect creates no contention or low contention when the cores require data from different L2 banks, so the crossbar between tiles and L2 almost does not influences the execution since no significant delay is observed.

- Contention occurs in the L2 banks: when the cores asks for data to the same L2 bank the latency considerably increases.

- The atomic instructions must respect the restrictions declared in the RISC-V manual in case of *critical* system (quad-core), only in this way its execution can go on properly.

- A key-parameter is L2_Xactors. The higher its set value, the higher the number of concurrent memory operations that L2 can manage in parallel. Stressing the quad-core

configuration with the minimum L2_Xactors leads to a non-working system. On the other hand, increasing too much its value leads to other *violations* by preventing the proper cores execution. That is why L2_Xactors = 2 has been a reasonable choice in this context.

- The main constraints of a memory request are two. The first one deals with its validation by L2 depending on the L2_Xactors value and so also on the number of memory instructions that are being handled. The second constraint concerns two consecutive memory operations executed by the same core which has to wait 7 clock cycles from when its DL1 receives the requested data (i operation) to start a new acquire request ($i+1$ operation).

Chapter 6

System Re-design: NoC-based

The UPV NoC tile described in the section 3.3 is now prone to an evolution. The main idea consists in replacing the MIPSCORE with the LAGARTO in this tile, so the outwards interfaces of the core should be the same in order to hold the connections unaffected inside the Tile. In addition, during this process, further developments are made from time to time as regards the whole system itself.

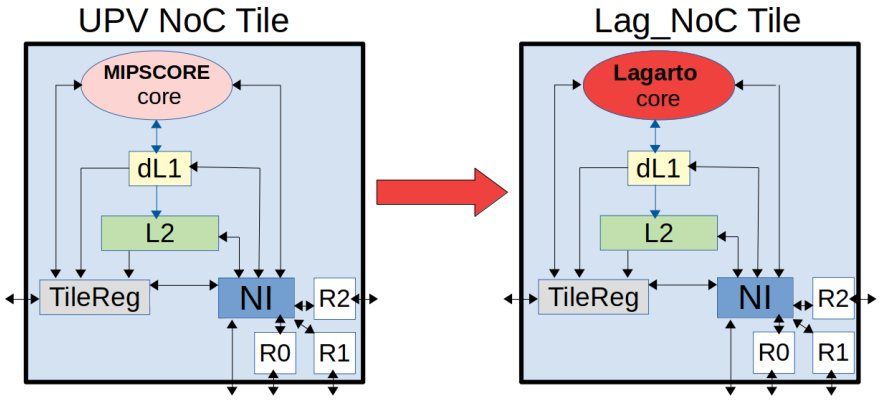

Figure 6.1: Lagarto placement inside the UPV NoC Tile

Thus, recently Lagarto was placed in the *Rocket chip* in place of the Rocket core, as mentioned in the section 3.2. The same core is now going to face a new challenge: the insertion in a Tile of a NoC-based system.

6.1 Lagarto placement

Initially, only Lagarto was instantiated in the testbench designed in the Vivado environment. The first task consisted in making this core execute the chosen instructions. In the original tile the processor MIPSCORE includes the instruction cache (it receives blocks of 16 instructions directly from NI), Lagarto does not include it. Thus, it is created a fake instruction memory that is seen by Lagarto as it was its instruction cache.
In simple terms: Lagarto was simulated in the *Rocket* Chip with verilator and Spike, finally observing the waveform on gtkwave, focusing on the way in which Lagarto receives the instructions from the *Rocket* instruction cache. After that, in the designed testbench, the appropriate signals at the Lagarto/iL1 interface are managed so as to copy the observed protocol, simulating an instruction cache and choosing the instructions from time to time.

The initial Lagarto interface was suitable to connect the core in the *Rocket* chip; since it is going to be insert in another system some signals at its interface became unconnected. To overcome the obstacles generated by this issue, some internal signals of Lagarto were driven, allowing its proper execution of the instructions.

Ensured the proper functioning of the core, the second goal consisted in appending the data cache L1 of the UPV NoC Tile. To do it, the Lagarto/*Rocket*_dL1 and MIPSCORE/UPV_dL1 interfaces were analyzed in order to accomplish the desired connection Lagarto/UPV_dL1.

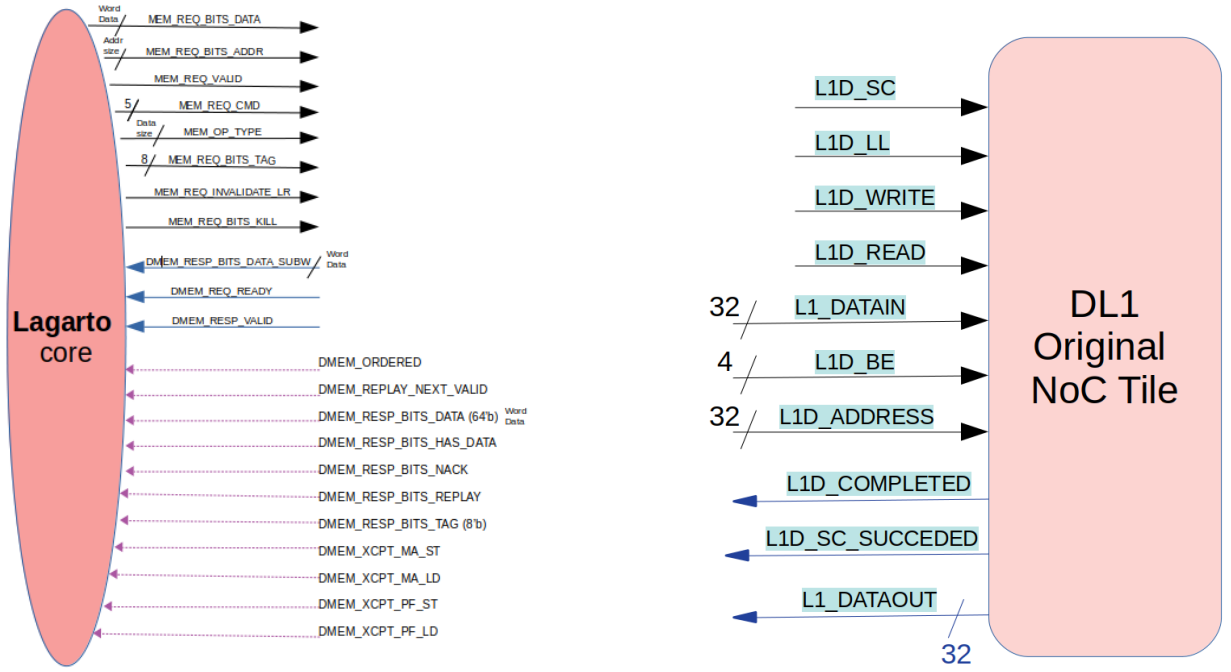

Figure 6.2: Lagarto in its interface with the *Rocket*_dL1 (left). UPV_dL1 in its interface with the MIPSCORE (right).

At this point, it was possible to simplify the two interfaces in two ways:

1. Considering that the purple signals in the picture 6.2 are used to connect Lagarto in the *low risc* platform; so placing the core in a new system, they are not needed anymore.

2. Presuming that the atomic instructions are not used currently. This hypothesis allow to deleted the blue highlighted signals *L1D_SC*, *L1D_LL*, *L1_SC_SUCCEEDED* at the UPV_dL1 interface that are only addressed for that type of instructions. In addition it allows to go on using the initial Lagarto version which is only able to decode the atomic load instruction, but not all the other ones.[1];

The resultant interfaces that have to be matched are reported in the picture 6.3.

[1]In the Rocket chip, all the atomic operations are implemented in the *Rocket* cache, Lagarto (or the Rocket core) only decodes that instructions and send the necessary information to the cache.
Thanks to the last updates, the newest Lagarto version is able to handle all the atomic instructions.

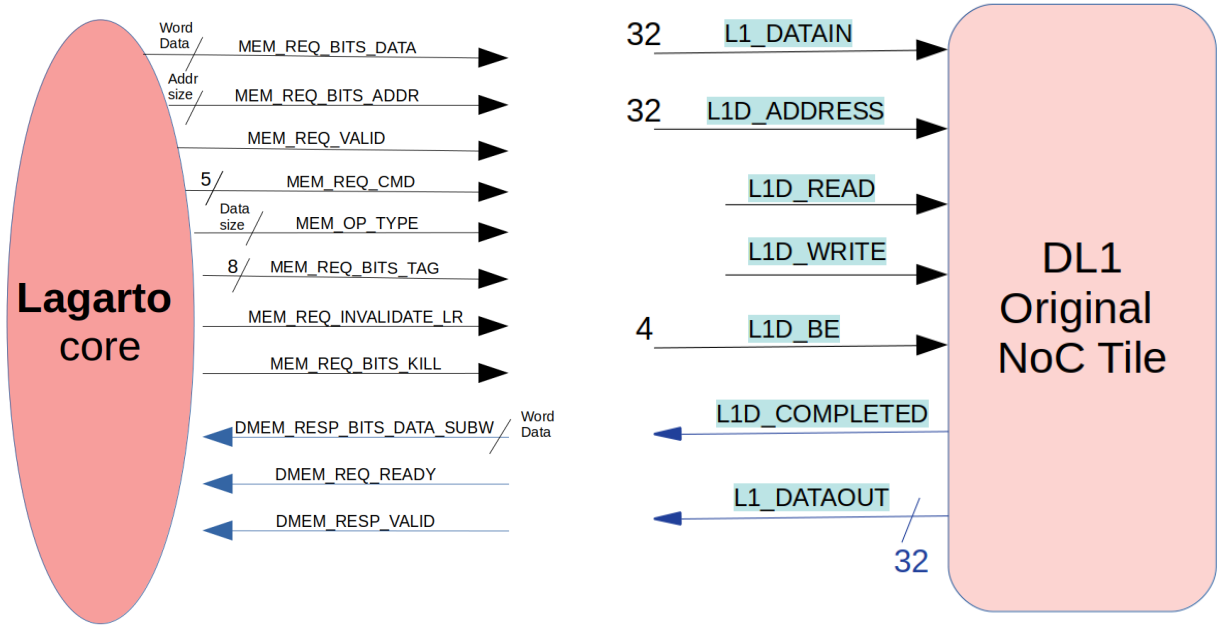

Figure 6.3: Final interfaces that have to be connected.

After a complete analysis of this partial system, consisting of the simulations of the two interfaces separately and the debugging of the code describing these modules, it was designed the Lagarto/UPV_dL1 interfacing (fig. 6.4).

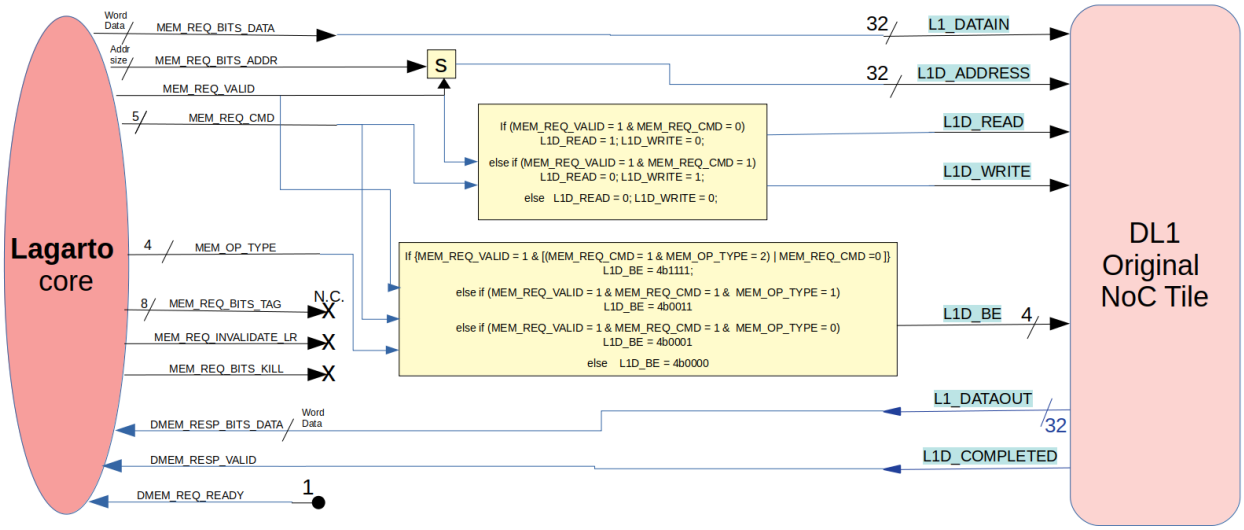

Figure 6.4: Lagarto/UPV_dL1 interfacing.

The *interfacing yellow blocks* in the picture 6.4 were purposely designed. In addition, for the proper DL1 functioning, some modifications were needed in the DL1 verilog codes.

Given these adjustments, it was proven that the data cache L1 receives properly the signals and so it is able to propagate the data and control signals towards the rest of the Tile modules that are appended (in figure 6.5).

Thus, at this point, the current system instantiated in the testbench provides:

- The core Lagarto that has been placed inside the NoC Tile and it was interfaced with the data cache through some interfacing modules.
- A fake instruction memory from which Lagarto read the instructions.
- The data caches (L1 and L2) and the Network Interface of the original UPV NoC Tile.

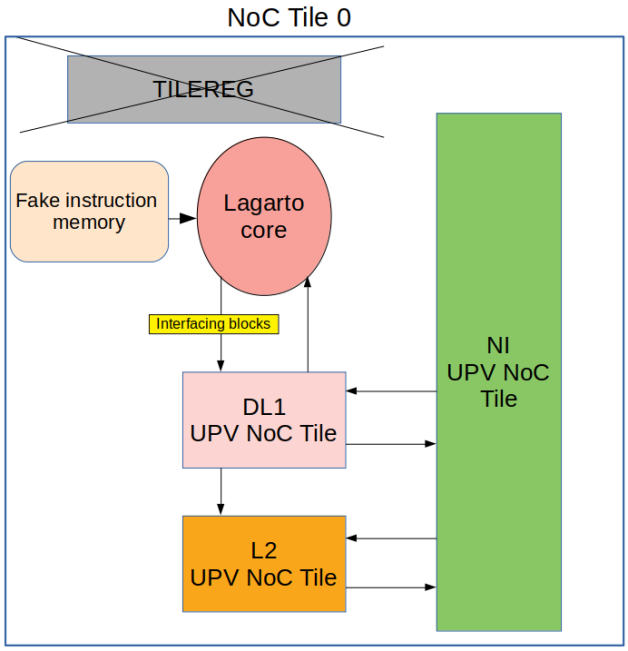

Figure 6.5: Temporary system composition after the Lagarto placement.

- The *TILEREG* module is not instantiated currently. By considering only a tile, it would be necessary only to upload the communication protocol inside the caches and to manage the core program counter. The first issue was overcome by loading directly the protocols into the caches. As regards the pc issue: the original processor MIPSCORE manages its program counter through TILEREG; on the other hand, Lagarto handle its pc on its own, thus also this issue is bypassed.

6.2 System processing

When Lagarto executes a memory instruction (e.g. a *load*), the examined procedure, carried out along the system, to serve this request is the following one:

1. Lagarto asks for reading a block to dL1.
2. If this block is in dL1, the data cache gives it to the core. If dL1 does not contain this block, dL1 sends this request to NI.
3. Firstly, NI asks to L2 if it has the required data.
4. If L2 has got the block, it is sent to NI which gives it to dL1. If L2 responds with a negative outcome to NI, the Network Interface sends the request to the memory controller MC which takes this data block from the main memory.

   N.B. The MC and the RAM are outside the tile and they are shared among the eventual several tiles of the NoC.

Thus, the following step consists in appending a memory controller and a main memory.

The original UPV NoC exploits a Double Data Rate (DDR) RAM and a sophisticated memory controller (figure 6.6). The necessary related modules can be hardly handled, leading

to the main drawback of this NoC structure: a very complex specific and difficult to reuse design.

In order to get through this obstacle, the idea consisted in simplifying this part of the system (figure 6.7):

- Using a basic RAM in place of the DDR RAM.
- Designing a new MC, simpler than the original one and able to be interfaced with the original Network Interface on the one hand and with the new basic RAM on the other hand.

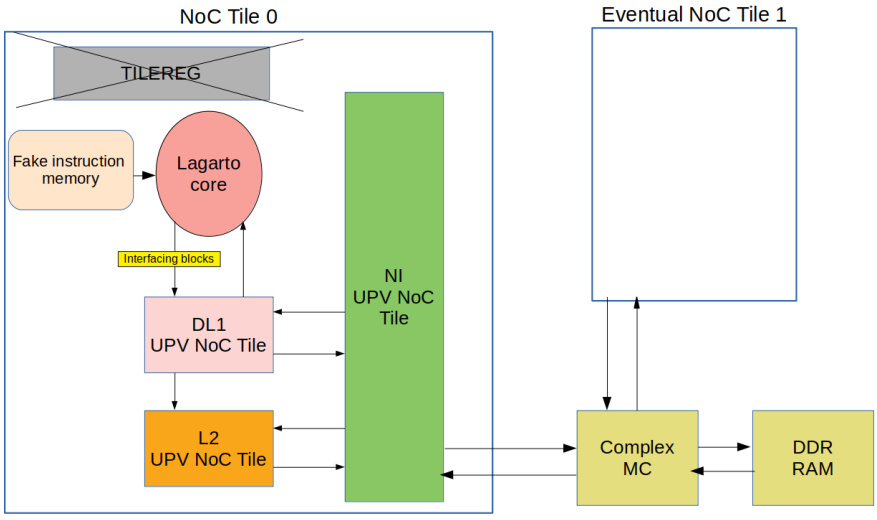

Figure 6.6: Original system with complex MC and DDR RAM.

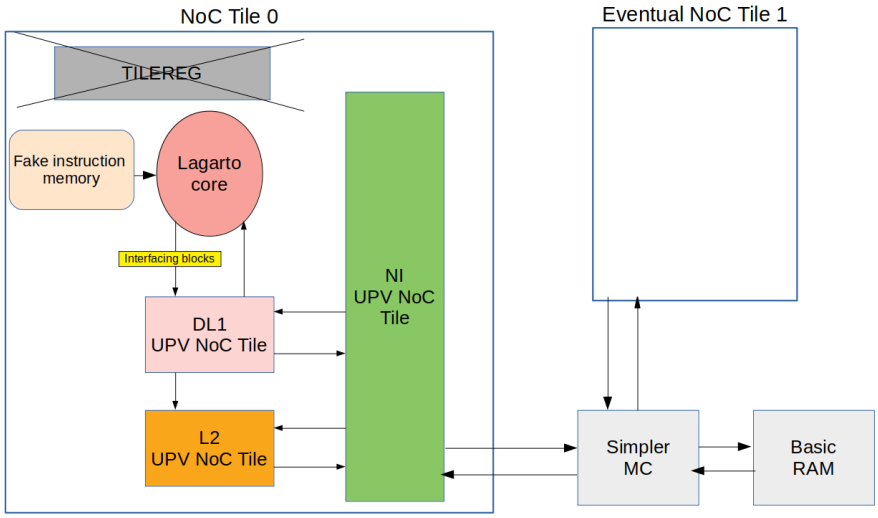

Figure 6.7: Adjusted system with simpler MC and basic RAM

The used synchronous basic RAM (figure 6.8) exploits a *txt* file to read and save information. This memory has only the following signals at the interface: address, data_in, data_out, read/write.

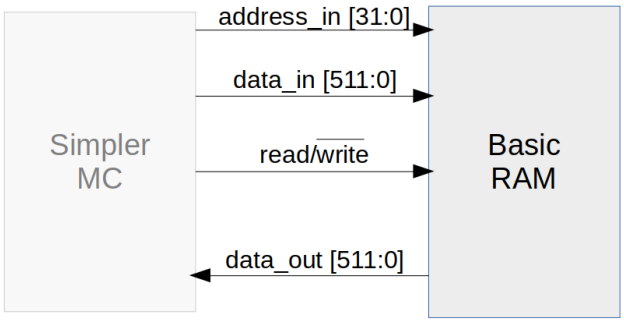

Figure 6.8: Basic RAM interface with the new MC.

The new designed MC 6.9 deals with the following tasks:

- it manages the NI inputs and sends the address and the command (read/write) to the RAM; writing or reading data depending on the required operation.
- it tracks essential information of the request in order to deliver the proper memory response to the original requestor (which tile and which module inside the tile).

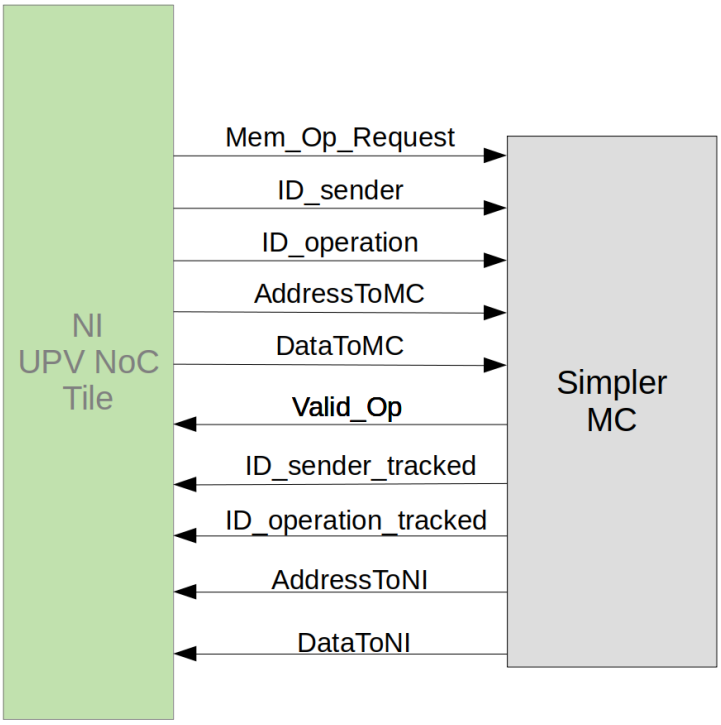

Figure 6.9: New MC interface with the NI.

Currently, this system is being processed in order to carry out steps forward to achieve the final goal of a complete incorporation of Lagarto in this NoC structure.

At present there is one instantiated tile and the rest of the system is the one reported in the picture 6.7. It needs to be adapted by taking into account the new inserted modules, thus several adjustments are being studied and carried out.

Today's system state allow to execute properly the load operation from main memory, following mostly the original communication protocol. The required data are taken from the RAM, stored in L2 and finally they can arrive up to Lagarto by forcing a dL1 signal or sending directly that data from the network interface, bypassing the first level of cache. As regards the store operation, it is possible to write data in the cache dL1; nevertheless this cache and L2 are both write back and currently their proper behavior is not guaranteed since it is not possible to release data from cache to the memory.

The troubleshooting of this caches issues is on-going, in the meantime it was enabled an alternative communication path between Lagarto and the main memory, it provides Non-Cacheable Access (NCA), storing and loading properly from memory in a specific addresses range bypassing the caches and following the path Lagarto ⇒ NI ⇒ MC ⇒ RAM and vice versa.

6.3 Summary of the core placement

The Lagarto placement process is proceeding. It is possible to store and load from memory properly with a NCA.
Anyway, on the one hand the data caches issue has to be handled, getting their proper functionalities; on the other hand Lagarto needs an actual instruction cache in order to replace the fake instruction memory that is using temporarily. In order to achieve this task, the idea consists in extracting the IL1 incorporated inside the MIPSCORE, placing it next to Lagarto with the needed interface adaptation and connecting it to NI in order to receive the block instructions from the main memory flowing through NI like in the original UPV project.

This experience shows the procedure involving a core replacement. The first step consists in the analysis of the initial system, focusing on the interfaces of the original core. The second step provides the placement of the new core with the designing of a fake instruction memory where the instructions are fetched. The third step consists in the core/memory connection, adapting the interfaces and leading to proper store and read operations. This stage involves above all the interfacing between core and first level of cache in case of usual cacheable access. Subsequently, the further modules of the system can be appropriately connected step by step, finally testing the whole tile.

Chapter 7

Conclusion

This thesis focused on two Systems-on-Chip exploiting different types of interconnection: crossbar and bus-based in the Rocket chip, NoC in the UPV tile.
The first one was analyzed, figuring out the latency values in several stress conditions and extracting the timing constraints which a memory request comes across.
The second structure, after an initial study phase, has been processed by placing the Lagarto core and some new modules, proceeding with the needed system adaptation.

This RISC-V processor played a central role in this context, it was recently incorporated in the *low risc/Berkeley* SoC and, for the first time, in this work it is placed in a NoC structure, as a starting point.

In order to achieve a complete working Network-on-Chip using Lagarto, many further improvements are required. First of all, the current limitations related to the memory operations, described in the previous section, have to be solved. Once the core is perfectly able to execute load and store instructions, the following step could consist in including another tile, building a first 2D-mesh net. Connecting the routers of the two tiles, the whole system can be globally tested, carrying out the necessary adjustments, finally analyzing the system performance.


The long-term objective consists in implementing a highly-capable RISC-V heterogeneous platform including multi-core, NoC and accelerators.

Bibliography

[1] Cota, E'., de Morais Amory A., Soares Lubaszewski M., 2012, *Reliability, Availability and Serviceability of Network-on-chip*, Chap. 2, Springer. [ISBN: 978-1-4614-0790-4]

[2] *Architecture of a Smartphone and SOC(system on chip)*, Top 5 stuffs. http://top5stuffs.blogspot.com/

[3] *From "Bus" and "Crossbar" to "Network-On-Chip"*, 2009, Arteris S.A.

[4] A comparison of Network-on-Chip and Busses, Arteris. https://www.design-reuse.com/

[5] S.Pasricha, N.Dutt, 2008, *On Chip Communication Architectures*, Chap. 12, Morgan Kauffman. [ISBN: 978-0-12-373892-9]

[6] Wen-Chung Tsai, Ying-Cherng Lan, Yu-Hen Hu, Sao-Jie Chen, *Networks on Chips: Structure and Design Methodologies*, Journal of Electrical and Computer Engineering, Volume 2012, Hindawi. https://www.hindawi.com/

[7] Dr. Tatas K. *Network-on-chip(NOC): a New SoC Paradigm.*

[8] NoC and bus architecture: a comparison, Rajeev Kamal, Neeraj Yadav, International Journal of Engineering Science and Technology (IJEST), April 2012.

[9] OpenRISC https://openrisc.io/

[10] minsoc: Overview https://opencores.org/projects/minsoc

[11] Florian Zaruba, Prof. Luca Benini, May 2018, *Ariane: An open-source 64-bit RISC-V Application-Class Processor and latest Improvements*, ETH Zürich.

[12] *RISC-V, Spike, and the Rocket Core*, Ben Keller, CS250 Laboratory 2 (Version 091713).

[13] LowRISC. A fully open-sourced, Linux-capable, System-on-a-Chip.
https://www.lowrisc.org/

[14] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo and Andrew Waterman, *The Rocket Chip Generator*, EECS Department, University of California, Berkeley. Technical Report No. UCB/EECS-2016-17. April 15, 2016.

[15] *Lagarto. Un procesador RISC de código abierto para la academia y la investigación.* instituto Politécnico Nacional, Centro de Investigación en Computación, Mexico.
Source available at www.bsc.es

[16] Cristóbal Ramírez, César Hernández, Carlos RojasMorales, Gustavo Mondragón García,Luis A. Villa,Marco A. Ramírez, *Lagarto I - Una plataforma hardware/software de arquitectura de computadoras para la academia e investigación*, Instituto Politécnico Nacional, Centro de investigación en Computación, México.

[17] *Chisel, Constructing Hardware in a Scala Embedded Language*. UC Berkeley, University of California. https://chisel.eecs.berkeley.edu/

[18] TileLink 0.3.3 Specification Tilelink 0.3.3 Specification

[19] *SiFive TileLink Specification*, SiFive, Inc. December 3, 2018.

[20] Wei Song, *Untethered lowRISC, Memory Mapped IO and TileLink/AXI*. July 27, 2015.

[21] Introduction to Verilator, https://www.veripool.org/wiki/verilator

[22] *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. Document Version 2.2. Andrew Waterman, Krste Asanović, SiFive Inc., CS Division, EECS Department, University of California, Berkeley, May 7, 2017.

Glossary

AMBA Advanced Microcontroller Bus Architecture

BSC Barcelona Supercomputing Center

DDR Double Data Rate

DL1 first Level Data cache

FPGA Field-Programmable Gate Array

ILA Integrated Logic Analyzer

IL1 first Level Instruction cache

IP Intellectual Property

ISA Instruction Set Architecture

LR Load-Reserved (RISC-V atomic instruction)

L2 Unified second Level of cache

MC Memory Controller

MPSoCs Multi-Processor System-on-Chip

MSHR Miss Status Holding Register

NCA Non-Cacheable Access

NI Network Interface

NoC Network-on-Chip

PE Processing Element

RTL Register Transfer Level

SC Store-Conditional (RISC-V atomic instruction)

SoC System-on-Chip

TL TileLink

TLB Translation Lookaside Buffer

UART Universal Asynchronous Receiver-Transmitter

UPC Universitat Politécnica de Catalunya

UPV Universitat Politècnica de Valencia

UUT Unit Under Test