Politecnico di Torino

Master of Science in Electronic Engineering

Developement of RISC-V based System Controller for Coarse Grain Reconfigurable Architecture

Supervisors:
Prof. Luciano Lavagno
Prof. Ahmed Hemani (KTH Stockholm)

Student:
Riccardo Cappai

Abstract

As of 2019, RISC-V Instruction Set is drawing more and more attention among companies and academia. Due to the fact of being easy to use and not proprietary, it allows developers to create cheaper designs without the limitation of proprietary Instruction Set Architecture (ISA), enabling even faster innovations. The Instruction Set is composed of a very basic one and a lot of independent addable extensions, allowing developers to customize the ISA in order to fit their needs.

At the same time, the ASIC industry is struggling to adapt its methods to always larger designs. The introduction of standard cells allowed a decrease in design complexity, but since then the designs grew from a complexity of $O(10K)$ gates to $O(10/100M)$ gates.

In this thesis work, a design of a RISC-V processor is made adaptable to the SiLago design methodology, developed by KTH university with the goal of adapting the present needs of the VLSI community. On top of that, a new algorithm for an adaptable Network on Chip is proposed. The RISC-V component and the Network on Chip router have been designed down to their GDSII files following the SiLago flow.

Contents

Chapter 1

Introduction

1.1 Objective and organization

The aim of this thesis is to describe and contextualize the work done on a RISC-V processor and the development of a Network on Chip algorithm and its implementation.

One of the most compelling problems of ASICs (Application Specific Integrated Systems) is the growing cost of designs [4] that limits their usage. With the introduction of standard cells, a similar problem was solved: in order to cut a design phase (e.g. achieve a cheaper design flow), a performance cost was paid. SiLago, a project developed by the Department of Electronics at KTH, tries to mimic that approach to solve today's problems. By using atomic blocks larger than standard cells, designs in the order of $O(10M)$ gates can become more affordable by cutting another phase of design, in order to let ASICs be used to reach the better performance that they bring compared to general purpouse processors.

One of the beliefs behind the SiLago project is that today's architectures need to be more hardware-centric (Hardware accelerators are supposed to

do most of the work), so that software is only responsible for control and communication between the accelerators and memory (or other accelerators).
In the software realm RISC-V is one of the newest Instruction Set developed (It got introduced in 2010 at the University of Berkeley). Due to the fact of being open source and modular, it fits the needs of a SiLago processor able to take care of the control and communication duties of the chip.
The communication duties are also fulfilled thanks to a Network on Chip that operates on two levels. One is primarily used to initialize, control and configure region instances, instantiate and terminate applications under the control of the RISC-V processor, the other is used to exchange data between regions, and whose connections can be changed at run time.

1.2 Organization

This work is divided in five chapters. In this first one the background behind the thesis is explained. Each section prepares one of the following chapters. "History of VLSI automation" 1.3.1 is related to the SiLago project, "Instruction Set Development" 1.3.2 explains the reasons why a push for open source ISA was necessary in the industry, and "Interconnections on System on Chips" 1.3.3 explains the history of interconnections within systems on chip, from shared buses to Networks on Chip.
The second chapter gives a overview on Silago, a project carried out by the Department of Electronics of KTH, and explains in which way it solves the problems of the standard cell based design flow for Very large Scale Integration (VLSI) designs.
The third chapter gives a short overview of the RISC-V project and describes the work done in the thesis on the processor side, from the high level C++ code, to the Network on Chip interface hardware that handles the communication within the chip.
The fourth chapter is dedicated to the work done on the SiLago Global Net-

work on Chip. There a description of the communication protocol is given, as well as a description of the new algorithm developed. At the end of it, a deadlock analysis of the algorithm is presented, in which the limitations of the new work are highlighted.
The fifth and last chapter covers the design stages followed during the thesis work, in particular simulation, logic synthesis, and physical synthesis. At the end of the chapter, future improvements of the work are suggested.

1.3 State of the art

1.3.1 History of VLSI automation

This section contestualizes standard cells and their historic role in VLSI automation. At the end an argument to raise the abstraction level from standard cells is presented.

In order to get an ASIC design, different steps are necessary. The next figures describes their relationship and how the space of solution varies with each step of abstraction.

In figure 1.1 it can be seen how a system comprises several application. Each application can be described by means of different algorithms. An example would be an application for sorting. One can use different algorithms to do so: quicksort, bubble sort, binary search, linear search, radix sort, and others. Each algorithm can then be realized with different RTL (Register Transfer Level) descriptions using different HDLs (Hardware Description Languages) or with the usage of High Level Synthesis tools, for more complicated designs. Examples would be commercial tools like Stratus by Cadence (C/C++/SystemC description to RTL) and VivadoHLS by Xilinx (C/C++/SystemC description to VHDL/Verilog/SystemC), or academic

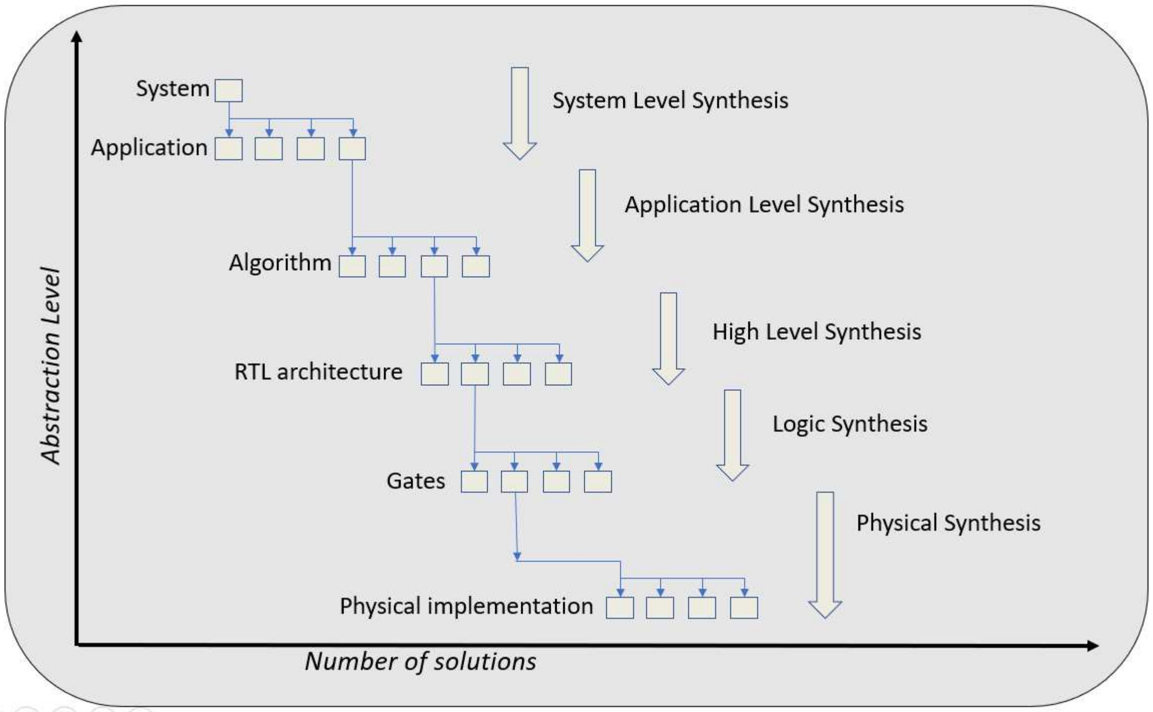


Figure 1.1: Abstraction level from a system to its physical implementation

tools like DWARV by TU.Delft.
Each RTL architecture can be mapped to several different gates structure. Each design differs by the type of target and by the tools and methods used. For FPGA (Field Programmable Gate Arrays) targets there is Quartus II by Altera, Vivado by Xilinx, and others. For ASICs some examples are Design Compiler by Synopsys and Genus Synthesis Solution by Cadence, which is the one used in this thesis.
At the end of the chain there is the actual physical implementation in which the actual geometries of the architecture are shaped out as well as the clock tree and metal layers. Tools for this synthesis phase are developed by Cadence, Synopsys, and others.

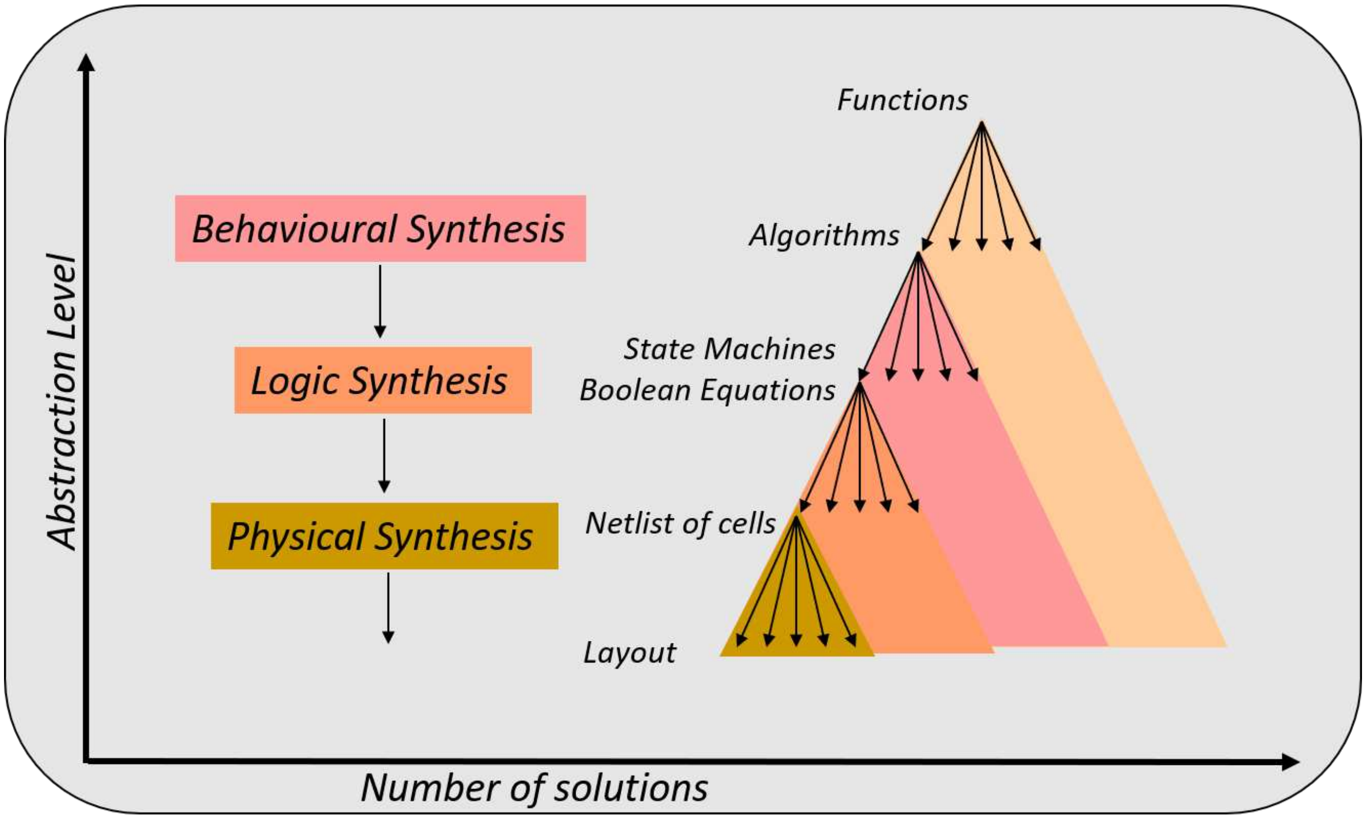


Figure 1.2: Limit of global solutions with each synthesis step

From figure 1.2 it can be seen how from each step of the synthesis, the number of possible solutions available goes down as the abstraction level becomes lower. This is because as the abstraction level goes down, less variables can be changed within the representation. For example, a netlist (Output of logic synthesis) will have a certain number of physical implementations depending on some variables in the physical synthesis. If the abstraction level is raised to the algorithmic one, the number of physical implementations will be considerably higher because now, other than the variables already mentioned, there will be also the ones from logic synthesis.

The method described in figure 1.1 used to be done all manually with the usage of Stick Diagrams and Silicon Compilers. The workflow was described in the book [1]. Eventually with the increase of complexity of designs it was necessary to automate some of the processes. That's why when the number of gates for designs became O(10k) the introduction of the so called standard cells was necessary, as explained in [2] and [3]. The introduction of standard cells cut down the design space by freezing the leaf nodes in figure 1.1 with a one-time engineering effort. The result of that was a method able to eliminate one stage of verification (e.g. reduce the engineering cost) and improve the engineering efficiency in spite of a loss in customization possibilities. Less customization means mainly a loss of optimization capabilities.

In short standard cells:

- Abstracted away the circuits and the physical design details enabling logic synthesis at gate level;
- Introduced a standard layout for placement enabling physical design automation (figure 1.3);
- Allowed a more reliable verification system at higher abstraction levels;
- Achieved an higher precision for power and timing analysis;
- Increased power consumption and area usage as a tradeoff.

Nowadays designs are in the order of O(10M) gates and the favorable improvements of the introduction of the standard cells are less visible.

Figure 1.4 shows the trend of the cost of ASIC designs for different transistors technology. The cost is increasing and how it is suggested in [4] one of the reason of that increase is that the granularity of standard cells is not enough to substain today's state of the art in VLSI design.

Different solutions have been proposed to raise the abstraction level from

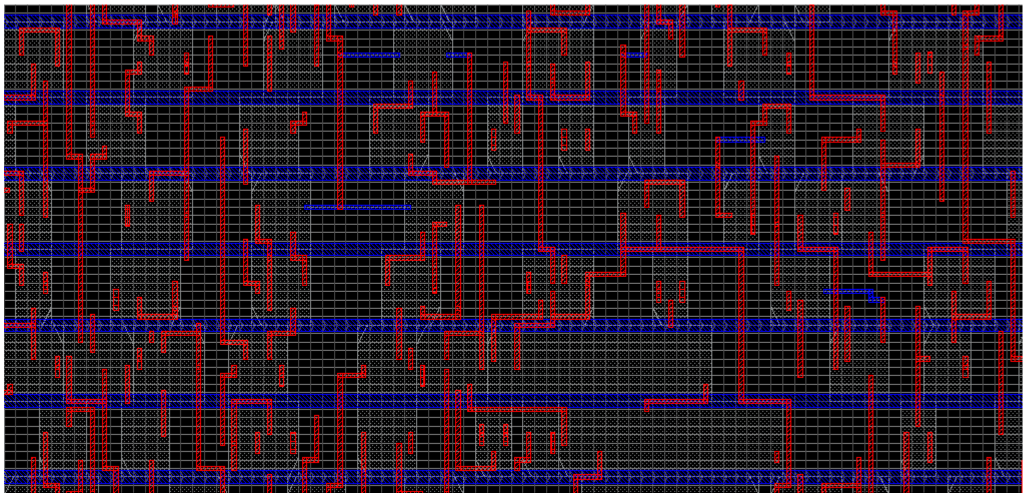


Figure 1.3: Example of standard layout automation. In blue power lines, in red vias of communication, in grey standard cells which all have the same height and differ by width

standard cells ([6],[7]). The solution presented in 2.1 is the one used in this thesis.

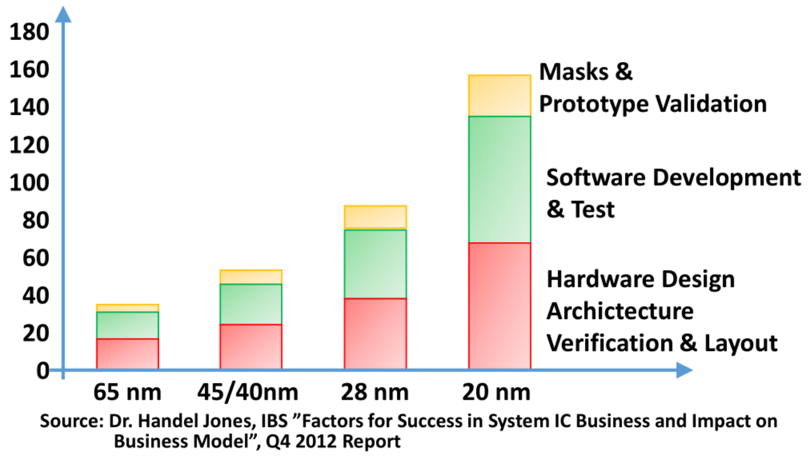


Figure 1.4: Cost of ASIC designs divided into different phases of realization, from [5]

1.3.2 Instruction Set Development

In this paragraph, an overview of Instruction Sets history is given, from CISC to RISC and the latest call for Open Source Instruction Set Architecture is reported.

An instruction set architecture is the set of instructions needed by a processor to operate. It defines addressing modes, data types, memory architecture, interrupts, and other characteristics proper to the processor. It's the interface between a machine's hardware and the software.

Historically machines have been of the CISC type. CISC stands for complex instruction set computer. Examples of CISC machines are the following [8]:

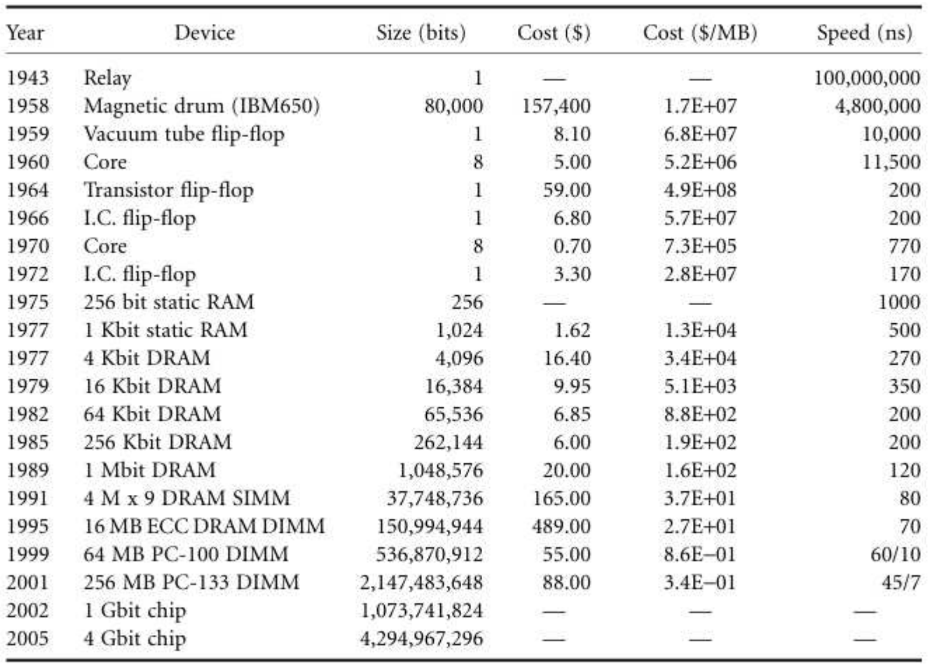
- VAX 11/780 [9] ,[10]
  - 303 instructions;
  - Instruction size not constant (from 2 to 57 bytes);
  - Instruction format not fixed;
  - 22 addressing modes;
  - 16 general purpose registers;
  - Developed in 1978.
- Intel 80486 [11]
  - 235 instructions;
  - Variable instruction size (1 to 11 bytes);
  - Not fixed instruction format;
  - 11 addressing modes;
  - 8 general purpose registers;

  - Developed in 1989.

- IBM 370/168 [8]
  - 208 instructions;
  - Not constant instruction size (From 2 to 6 bytes);
  - 4 addressing modes;
  - 16 general purpose registers;
  - Developed in 1973.

In the examples posted it can be seen how CISC machines have a large number of instructions and very limited general purpose registers. Also they have a variable instruction size and multiple addressing modes. The reason for these choices can be understood with the historical context in mind. Back when the computers listed above were developed, memory was a very critical component. Figure 1.5 shows the cost of memory devices throughout the years. The key column is the cost per MB that has been going down thanks to Moore's law. Designers' goal was then to minimize memory consumption (e.g. shorter code size) and maximize work per instruction [12].

Over the years, a new generation of processors has tried to take over the CISC machines. As written before memory cost has gone down thanks to Moore's law. Also, different studies showed how computers effectively used few instructions for most of their execution time. Patterson in [14] and Anderson in [15] noted how in a particular IBM 360 compiler, only ten instructions accounted for more than 80% of the execution time. Also, for a IBM 370 program (COBOL), Shustek [16] reports that it uses only 84 of the 183 instructions. Other than that, only 26 instructions account for 90.28% of the total execution time.



Year	Device	Size (bits)	Cost ($)	Cost ($/MB)	Speed (ns)
1943	Relay	1	—	—	100,000,000
1958	Magnetic drum (IBM650)	80,000	157,400	1.7E+07	4,800,000
1959	Vacuum tube flip-flop	1	8.10	6.8E+07	10,000
1960	Core	8	5.00	5.2E+06	11,500
1964	Transistor flip-flop	1	59.00	4.9E+08	200
1966	I.C. flip-flop	1	6.80	5.7E+07	200
1970	Core	8	0.70	7.3E+05	770
1972	I.C. flip-flop	1	3.30	2.8E+07	170
1975	256 bit static RAM	256	—	—	1000
1977	1 Kbit static RAM	1,024	1.62	1.3E+04	500
1977	4 Kbit DRAM	4,096	16.40	3.4E+04	270
1979	16 Kbit DRAM	16,384	9.95	5.1E+03	350
1982	64 Kbit DRAM	65,536	6.85	8.8E+02	200
1985	256 Kbit DRAM	262,144	6.00	1.9E+02	200
1989	1 Mbit DRAM	1,048,576	20.00	1.6E+02	120
1991	4 M x 9 DRAM SIMM	37,748,736	165.00	3.7E+01	80
1995	16 MB ECC DRAM DIMM	150,994,944	489.00	2.7E+01	70
1999	64 MB PC-100 DIMM	536,870,912	55.00	8.6E−01	60/10
2001	256 MB PC-133 DIMM	2,147,483,648	88.00	3.4E−01	45/7
2002	1 Gbit chip	1,073,741,824	—	—	—
2005	4 Gbit chip	4,294,967,296	—	—	—

Figure 1.5: Cost of selected memory devices, from Chapter 4 of [13]

The two evolutions (cheaper memory and realization of underused instructions) brought a new generation of processors. RISC (Reduced Instruction Set Computer) started to be developed and the core ideas, as highlighted in chapter 10 of [10] were:

- A small pool of instructions;
- A standardized instruction format;
- Direct hardware instruction execution (In contrast to CISC's microprogrammed instructions);
- A limited number of addressing modes;
- Load/store architecture with register to register arithmetic instructions;
- Small, fixed instruction cycle time.

Jamil in [12] also highlighted the presence of more registers in RISC architectures and their philosophy to "Move all functions to software". Down below a list of RISC architecture taken from [8] will be presented. The main differences to be noted from CISC to RISC is the instruction number, the limited addressing modes, the number of registers, and a constant instruction format.

- Sun SPARC
  - 52 instructions;
  - Constant instruction size of 4 bytes;
  - Two addressing modes;
  - Up to 520 general purpose registers;
  - Developed in 1987.

- PowerPC
  - 225 instructions;
  - Constant instruction size of 4 bytes;
  - Two addressing modes;
  - 32 general purpose registers;
  - Developed in 1993.
- ARM
  - 122 instructions (Some versions provide subsets);
  - Instruction size of 4 bytes (standard) or 2 bytes (Thumb instruction set);
  - Fixed instruction format (But different between standard and Thumb);
  - Three possible addressing modes;
  - 31 general purpose registers.

Although RISC architectures can be seen as an evolution of CISC, the evolution has been slow because of the retro-compatibility problem: RISC machines can't run CISC programs. As noted by Asanović (2014) in [17], modern CISC machines have solved the compatibility problem designing CISC that have an outer shell that translates CISC instructions into easier to execute ones (micro-ops or μops).

Asanović, in the same paper, explains the state of the art of ISAs and advocates for the introduction of Open Source ISAs, in particular RISC-V. The main point raised are the long negotiations and high prices of licenses (That are not compatible with the always shorter time-to-market), the obsolescence of proprietary ISA like ARM and 80x86, and the risk of the company owning an ISA shutting down.

1.3.3 Interconnections on System on Chips

The aim of this section is to list the evolution of interconnections on SoCs and to explain why Network on Chips have become the preferred way of communication in the realm of VLSI design.

Historically SoCs' interconnects have been a mix of point-to-point connections and shared bus, as explained by Bjerregaard and Mahadevan in [18]. The two kinds of connection are illustrated in figure 1.6. In the figure a System on Chip for a simple traffic light controller is presented. On the left a shared bus is used to interconnect all the components, on the right a point-to-point communication is used. Each of the two methods was valid at the beginning of SoCs because interconnections were not a critical component in neither timing nor power consumption.
Nowadays connections between parts brings the majority of energy dissipation, per [19]. Figure 1.7 from [20] describes the difference on power dissipation between technology nodes 10 years apart (11 nm from 2018 and 45 nm from 2008). It can be seen how, throughout the years, the employment of smaller transistors brought a drop in computational power (From about 100 pJ to less than 10 pJ), but it didn't change the power cost of interconnects. The table in the same figure shows that the delay numbers got worse when a smaller transistor width was used (From 130 nm to 55 nm).

Figure 1.8 shows a more complete point of view of communication power costs and how it fails to scale down with the transistor width. As of 2008 a floating point operation (DP flop) was as costly as moving 64 bit through 15 mm of chip. The same operation scaled to 2018 technology has similar power requirement as moving 64 bit through 5 mm of a chip.

It can be understood then how the effects of technology scaling are very successfull on the computation realm but fail to maintain the same scalabil-

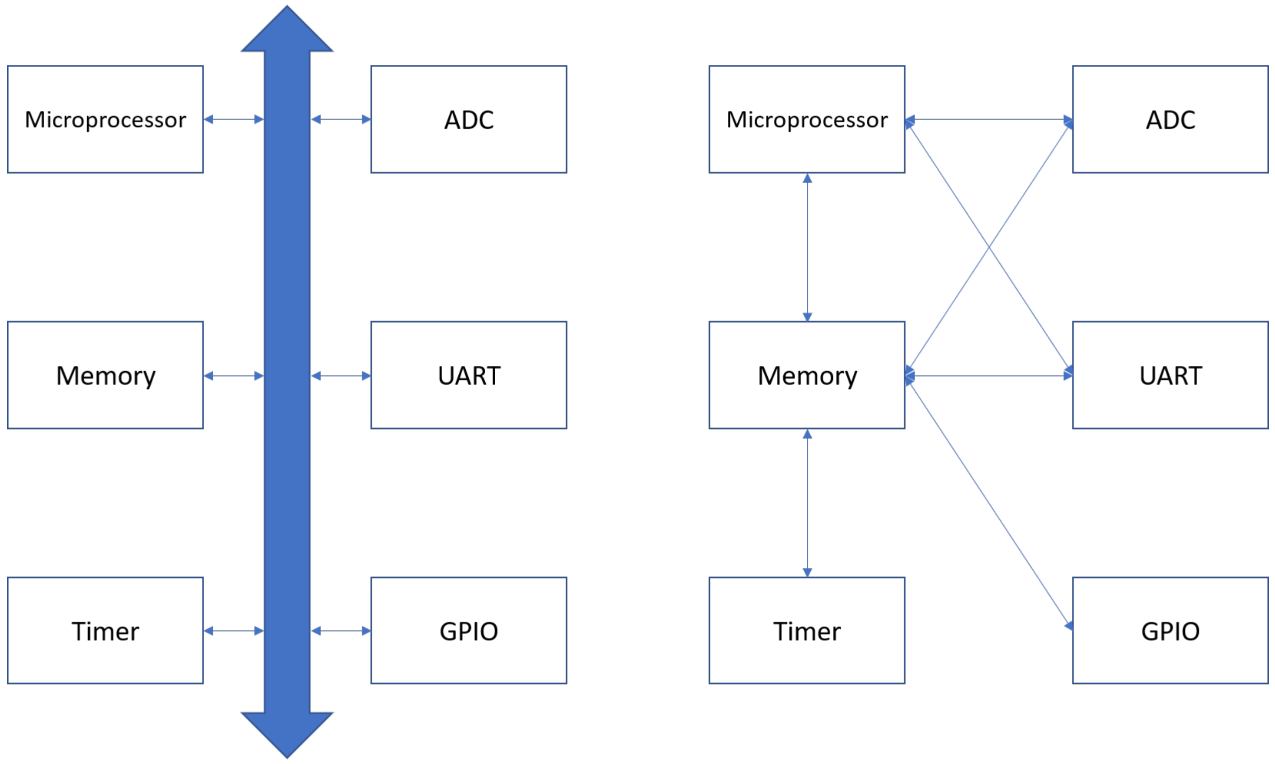
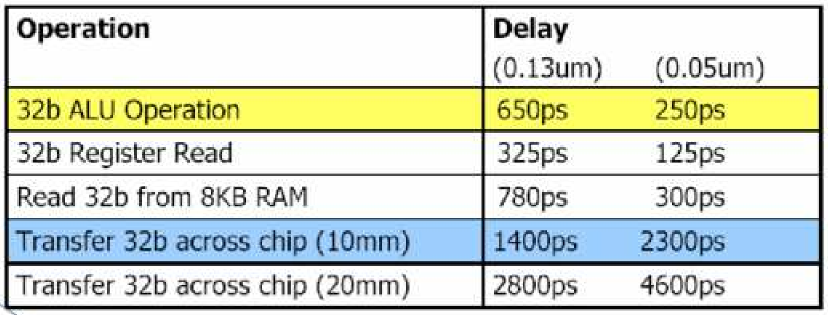


Figure 1.6: Example of point-to-point only communication (right) and shared bus based communication (left), typical of Systems on Chip in the 90's



Operation	Delay (0.13um)	(0.05um)
32b ALU Operation	650ps	250ps
32b Register Read	325ps	125ps
Read 32b from 8KB RAM	780ps	300ps
Transfer 32b across chip (10mm)	1400ps	2300ps
Transfer 32b across chip (20mm)	2800ps	4600ps

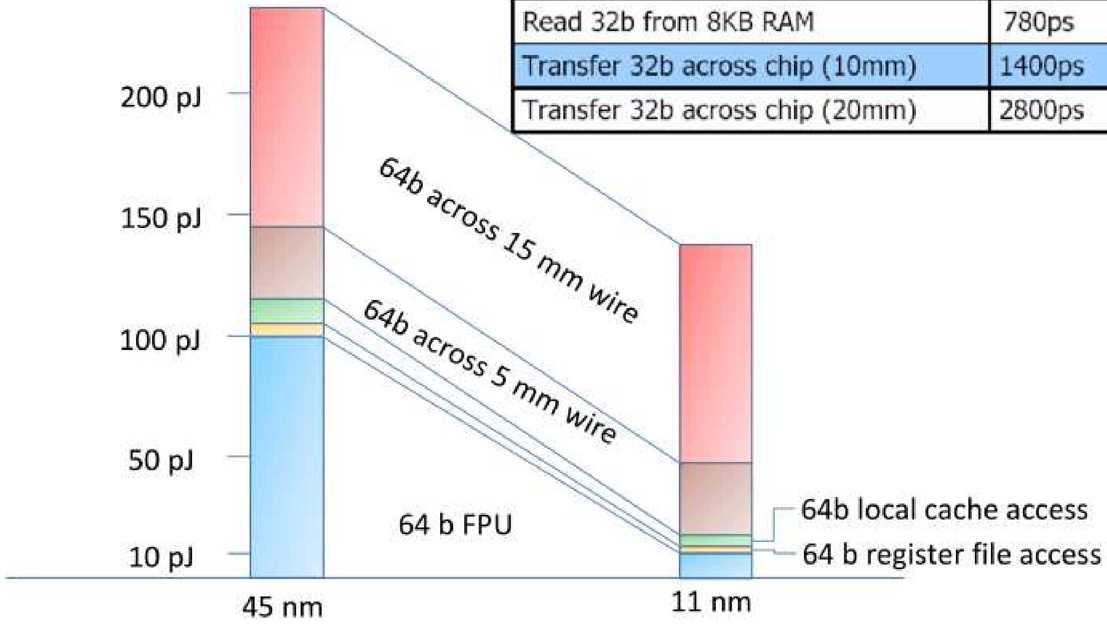


Figure 1.7: Trend of power dissipation between the 45 nm (2008) and the 11 nm (2018) node and delay estimation between the 130 nm and the 55 nm node, per [20]

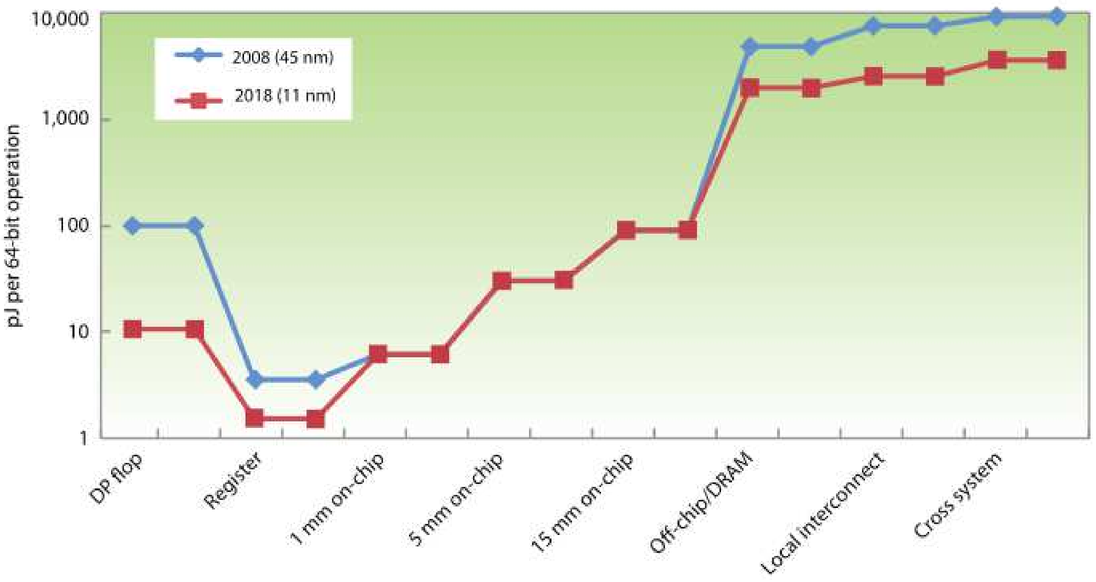


Figure 1.8: Trend of power dissipation for different communication operations between the 45 nm (2008) and the 11 nm (2018) node, per [21].

ity when it comes to interconnects. Architectural improvements of interconnects have been employed. From point-to-point communication, shared bus became the standard way to globally connect different parts of SoCs. Many companies employed their own Intellectual Properties on bus arbitration and methods of bus communication. One of the most famous one was the ARM's Advanced Microcontroller Bus Architecture (AMBA) presented in 1996 [22].

When multicore and multiprocessor architectures began to spread, the bus type of communication stopped to be the most preferred method of global interconnects in SoC design, as Aaron Boxer explains in [23].
With multiple processors, the arbitration time and latency figures were too high for buses so crossbars became the standard ways of global interconnects on Systems on Chips. An example of crossbar is in the UltraSPARC T2 [24] produced in 2007 by Sun Microsystem. In the figure 1.9 it is possible to see

the structure and the benefits of the crossbar scheme. Each of the 8 cores can be connected to each of the level 2 cache bank allowing a maximum of 8 load/store request per clock cycle from the cores and 8 returns from the cache banks.
A crossbar can be idealized as a series of multiplexer, each set of multiplexers decides what core to connect to a specific cache block (and viceversa, what bank to connect to a specific core). The scheme of the crossbar, taken from [25], is presented in figure 1.10.

Crossbars are a very powerful solution for interconnects but they have several problems. Those limitations, explained below, are worsened by the advancement of technology.

- Crossbars require a lot of wires to function, and they don't scale well when new blocks are added;
- They need to be designed specifically for the network and they don't have any re-usability (a crossbar between a microprocessor and memory can not be used for the connection between the microprocessor and a timer);
- Cascaded crossbars are able to reduce wires but they are likely to limit the maximum frequency of the system;
- Crossbars between N elements require N multiplexer sets, and each set needs to be connected with wires to both N components and a logic block to decide the direction. On top of that most wires in today's technology require repeaters (As represented in 1.11)

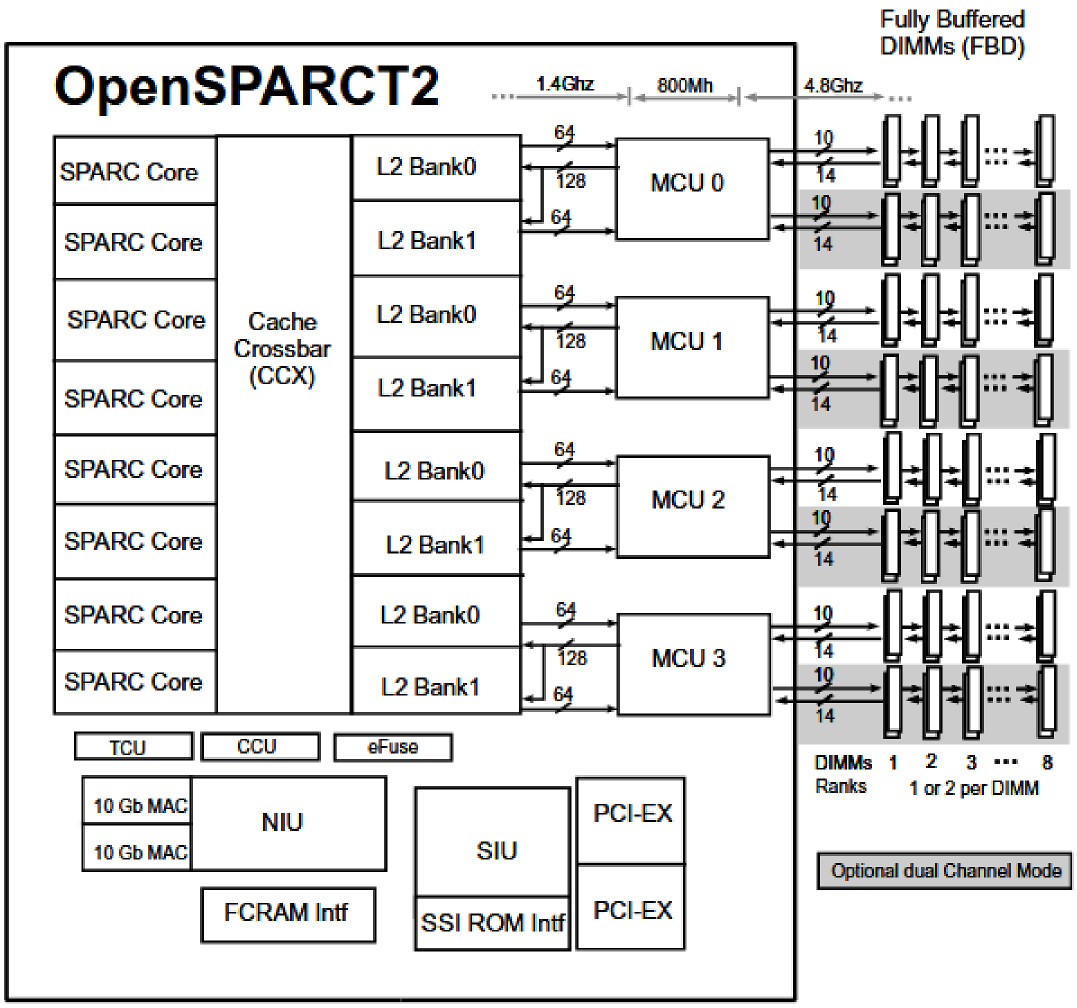


Figure 1.9: Scheme of the UltraSPARC T2 from Sun Microsystem, from [24].

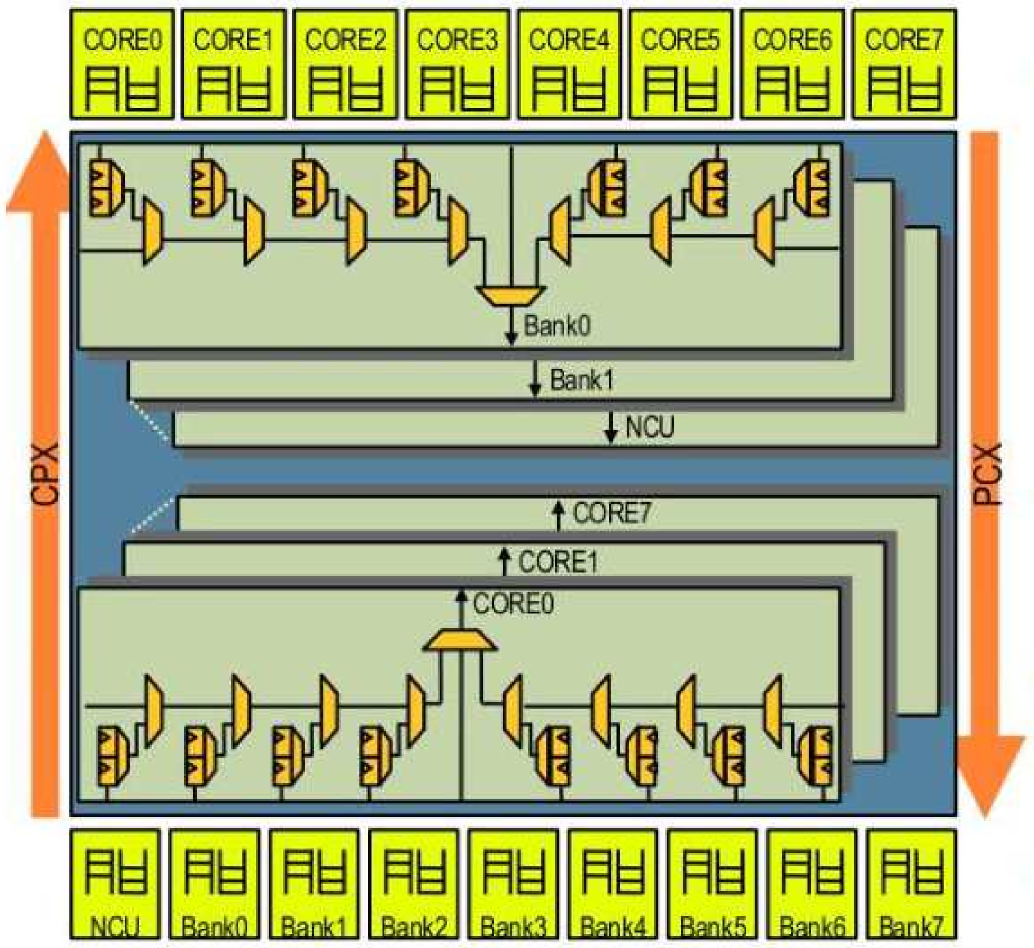


Figure 1.10: Scheme of the UltraSPARC T2 crossbar, from [25]. The crossbar works is bidirectional from the cache banks to the cores and viceversa

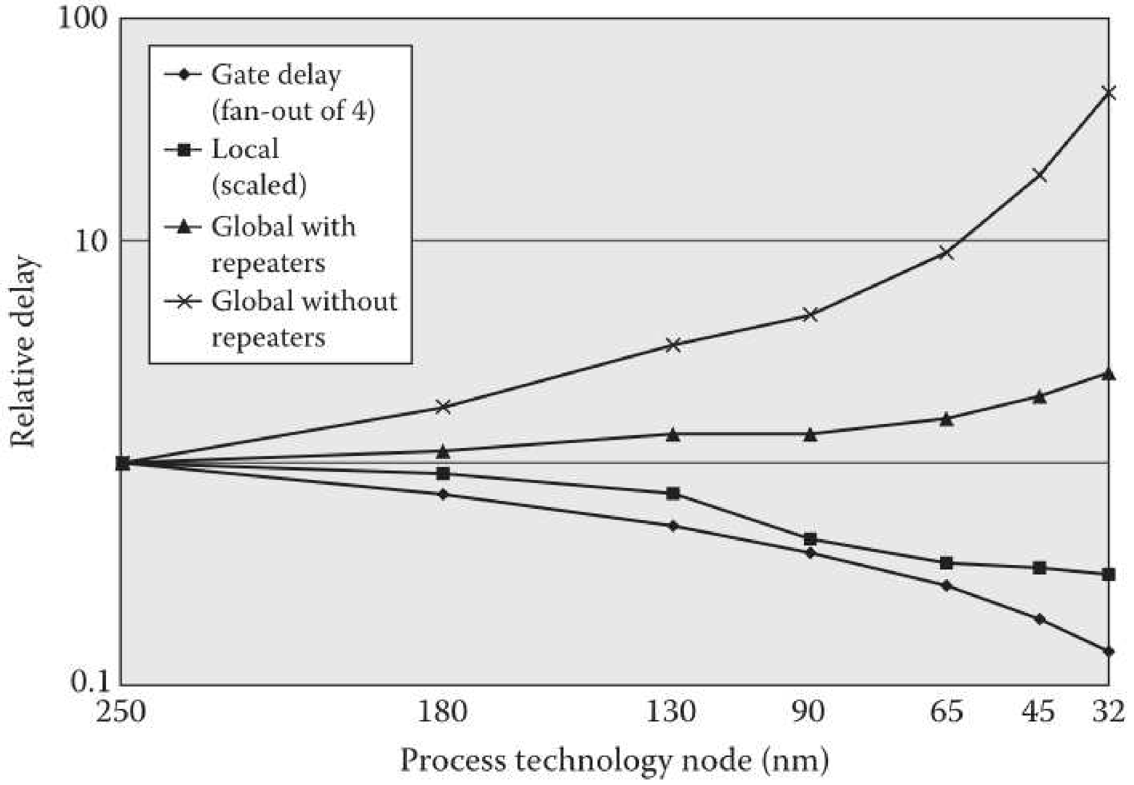


Figure 1.11: Projected relative delay for local and global wires with and without repeaters at different technologies, per [26].

To solve the problems highlighted above, network-on-chips were developed.
An example of network on chip is showed in figure 1.12. Each block of the system has an extra component called network interface that takes care of the communication between a block and the network on chip. The network on chip itself is an array of routers.
The routers are less in number than possible crossbars and wires are less than in a crossbar system. The routers are consisted of logic to route the information towards their destination and they can also have a small memory to hold values in case of congestion. Each router can be used by possibly all the blocks of the system (solving the re-usability problem and enabling a resource sharing system). The big limitations of the network on chip is the possibility of deadlocks (Avoidable most of the time if a smart routing algorithm is chosen) and a variable latency.

In spite of their problems, network on chips have become the standard in chip communications, especially in big systems in which different blocks are physically far apart. In chapter 4 different routing algorithms and NoC topologies will be presented and the architecture of the Silago Global Network on Chip will be explained.

In this first chapter, the background of the thesis was explained, as well as the context behind the choice of technology used. The next chapter instead will focus on the SiLago project, developed by the Electronics Department of KTH university.

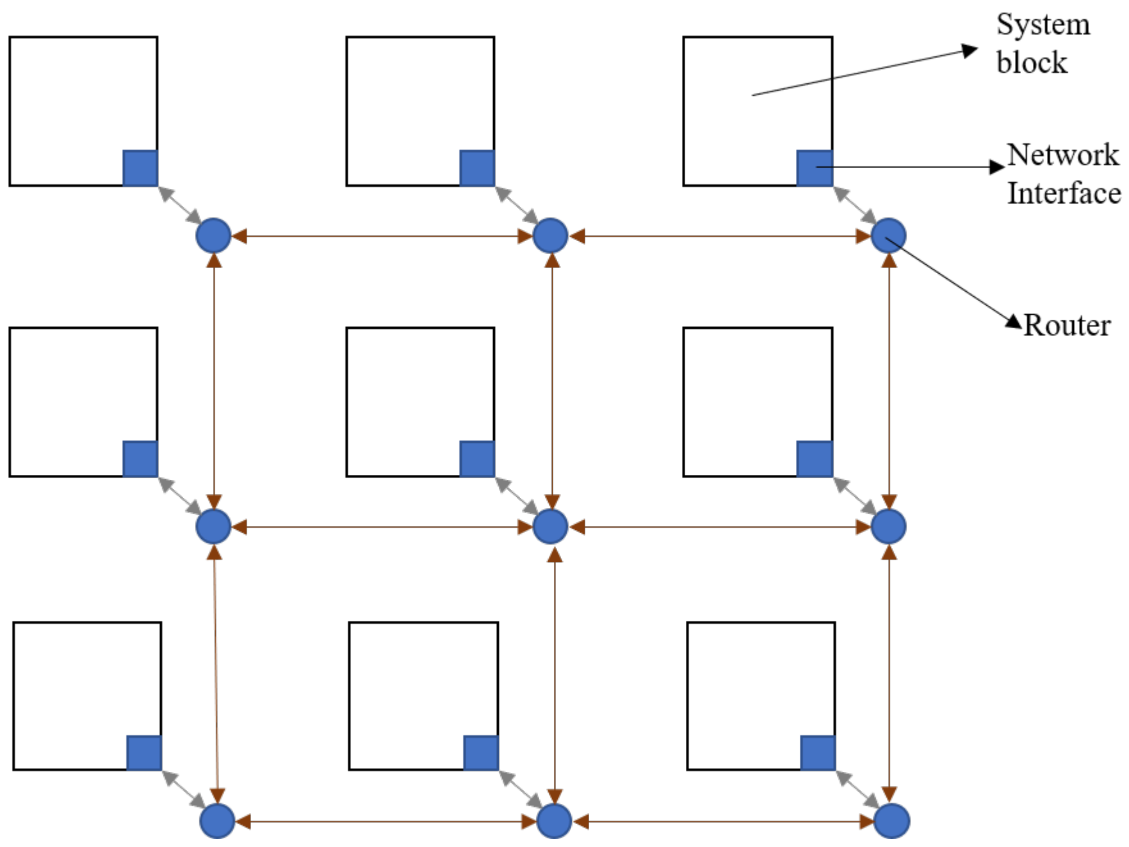


Figure 1.12: An example of a 3x3 regular Network on Chip

Chapter 2

SiLago framework

This chapter is threefold. At the beginning, a description of the SiLago framework is given, as well as a list of motives behind the project.
The second section tries to state all the steps necessary to create a SiLago instance, and in what way it solves the problems stated in section 1.3.1 with standard cells.
The third and last section is dedicated to the two tools of SiLago (VESYLA and Sylva), and how they are used in order to create a SiLago instance.

2.1 Description of SiLago framework

SiLago is a project carried out by the Department of Electronics, school of ICT, from KTH university. Its objective is to develop a new method for designing VLSI architectures in a cheaper way than the existing ones. This is possible thanks to a structured grid based physical design scheme (Linked to the synchroricity property) and a higher abstraction level for Physical Design.

The need of a change in the way VLSI design is done was explained in 1.3.1. Silago tries to introduce, as highlighted in [4]:

- An approach in VLSI design hardware-centric where functionalities are mainly mapped to coarse grain reconfigurable fabrics and a small processor is used mainly for control. This is in contrast with the current approaches that use multi-processors, in which most of the functionalities are implemented, and specific hardware accelerators used for power and performance critical applications.

- A fully customizable architecture for computation, control, storage, interconnect, and address generation. This is in contrast with the current customization centered only on computation optimization.
- A runtime customization for what concerns parallelism and the voltage-frequency operating point.
- A design methodology which is orders of magnitude more efficient than the standard cell based EDA flow, achieved by raising the level of abstraction from standard cells to micro-architecture level SiLago blocks.

In this chapter the SiLago design flow and its environment will be presented. A more complete description can be found in [4], [27], [28], and [29].

2.2 Silago Design Flow

In section 1.3.1, it was explained how standard cells became the preferred way to do VLSI design and how their benefits are becoming less as designs get bigger. Silago tries to replicate those benefits and apply them to today's designs scale.
Figure 2.1 visualizes the differences between the Silago method and the current Standard Cells based flow. On the left it can be seen that the standard cells novelty was the freezing of the lowest granularity possible (Physical level) and the creation of libraries that couldn't be changed by EDA tools. What Silago aims to do is the creation of libraries for larger objects (4-5 orders of magnitude larger per [28]) and make them the atomic building blocks

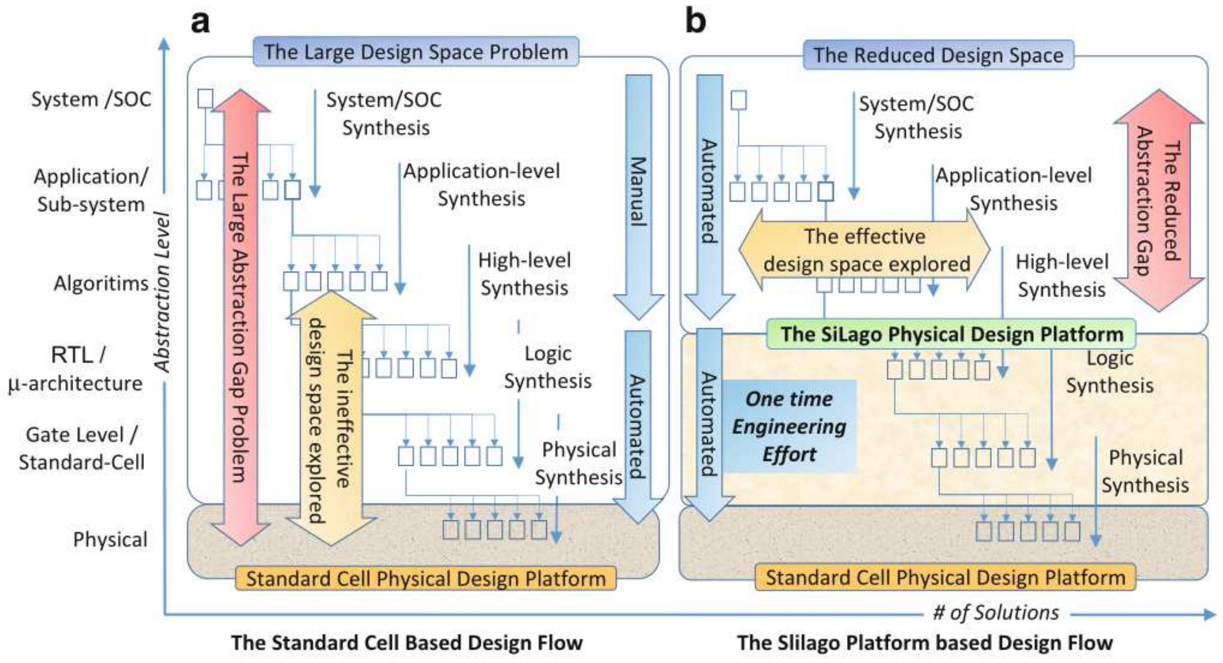

a
b
The Large Design Space Problem
The Reduced Design Space
System /SOC
Application/ Sub-system
Algoritims
RTL / µ-architecture
Gate Level / Standard-Cell
Physical
Abstraction Level
System/SOC Synthesis
Application-level Synthesis
High-level Synthesis
Logic Synthesis
Physical Synthesis
The Large Abstraction Gap Problem
The ineffective design space explored
Manual
Automated
The effective design space explored
The Reduced Abstraction Gap
The SiLago Physical Design Platform
One time Engineering Effort
Standard Cell Physical Design Platform
# of Solutions
The Standard Cell Based Design Flow
The Slilago Platform based Design Flow

Figure 2.1: Comparison between Standard Cells design flow (a) and Silago design flow (b) from [4]

for VLSI designs. On the right it can be seen that the Silago method aims to raise the abstraction level to RTL level. This choice causes a series of effects:

- Bigger atomic building blocks for VLSI designs;
- Interconnections between standard cells and clock tree synthesis are designed and verified a priori;
- EDA tools have a lower design space to explore;
- The difference between the highest abstraction level (System) and the lowest (In this case RTL) in the new flow is greatly reduced.

With bigger atomic building blocks, each Silago instance defines a microarchitecture operation rather than the boolean ones of standard cells.

And inside a block the clock-tree and interconnections are already designed and most importantly verified.
In addition to that, input and output pins are placed in such a way (on the right metal layer and on the right position) that two Silago blocks, if necessary, can be placed one next to the other without the need of other logic or physical synthesis. This ensures a correct by construction design that doesn't need an extra round of verification that standard cell designs need (The post physical verification, a costly task as highlighted in 1.4).

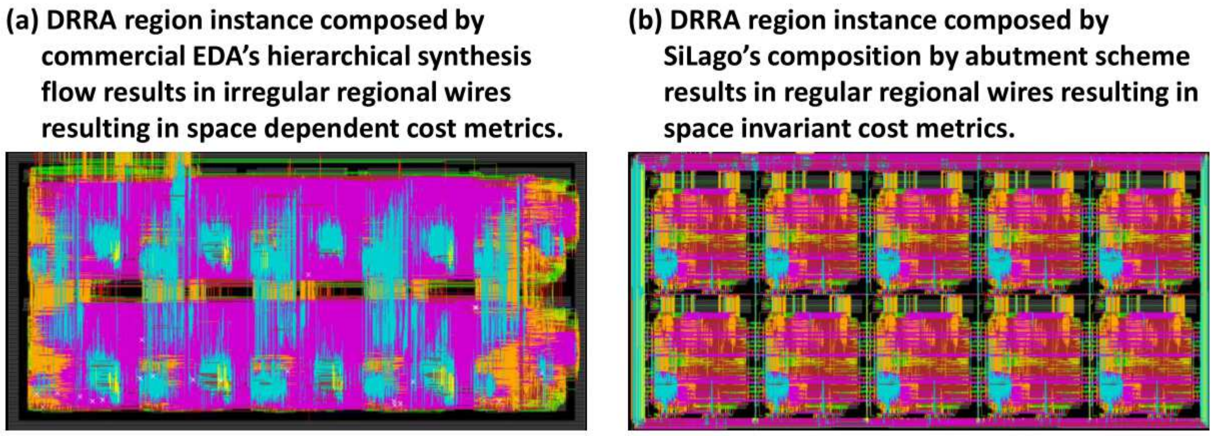

(a) DRRA region instance composed by commercial EDA's hierarchical synthesis flow results in irregular regional wires resulting in space dependent cost metrics.
(b) DRRA region instance composed by SiLago's composition by abutment scheme results in regular regional wires resulting in space invariant cost metrics.

Figure 2.2: A comparison between standard cell based designs (a) and Silago design with physical regularity in interconnections, from [28]

The concept of physical regularity in Silago is described in [28] where also the syncroricity property is defined. The effect of that is that at a global level, point-to-point connections are forbidden and regular interconnection patterns are defined, as it can be seen in 2.2.
The physical discipline ensures that a Silago cell can be fully characterized and its metrics correctly estimated. In [28], Hemani et al. argue that by raising the abstraction level to RTL and using Silago blocks instead of standard cell designs, the estimation of energy can be up to two orders of magnitude

better than standard cell based high level synthesis tools.

This last result is very interesting because it allows high level synthesis tools to have a more realistic idea of the characteristics of a design at a very high abstraction level. The fact that power estimation is more reliable at the RTL level means that more savings can be done if correct design choices are taken.
To do a computer science comparison, the choice of an algorithm to do a certain action (A merge sort instead of a bubble sort) can bring more energy saving than the choice of the variable length within the code.

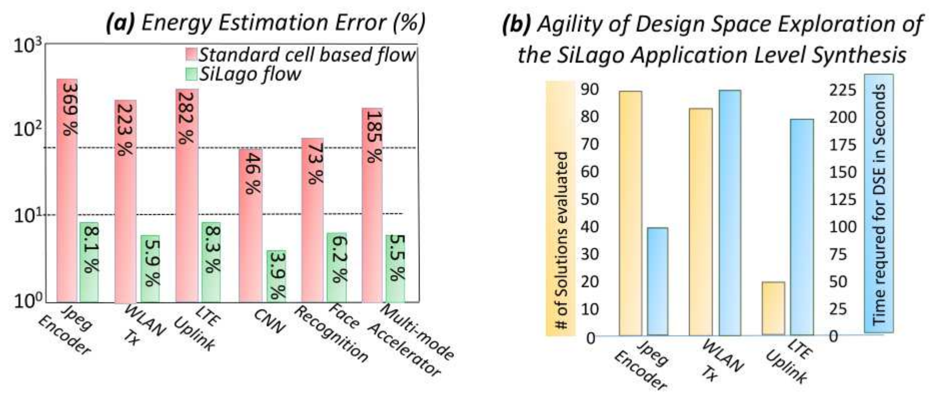


Figure 2.3: Findigs of Hemani et al. in [28] for what concerns the error in energy estimation and synthesis speed

On top of having a more reliable power estimation, the SiLago framework of high level synthesis allows solutions to be evaluated in a shorter time. In the same paper, Hemani et al. state that the SiLago method brings a three orders of magnitude improvement in synthesis speed. Figure 2.3 shows their findings in both energy estimation (a) and synthesis speed (b).

The benefits of the SiLago solution come with some drawbacks, that will be stated here and then explained:

- Area and power consumption overhead with respect to standard cells ASIC flows;

- Lower design space explorable;

The area and power consumption overhead comes from the syncroricity property and for the library of blocks created with a one-time-engineering effort (e.g. non modifiable).
In [28] it is stated that the area underutilization is on average of 35% on the applications explored. In [27] an area overhead average of 54% is reported, while Jafri, Farahini, and Hemani [29] registered an average of 50% in the same metric.
Hemani et al. in [28] note that similar drawbacks were also present when standard cells were first introduced [30], but despite that not only standard cells were accepted in the ASIC community, but they became the benchmark for ASIC applications.

The other drawback is related to the lower design space explorable by the high level synthesis tool. This comes from the fact that designs are frozen at a RTL level rather than at the standard cells level of today's designs. In fact the number of solutions possible for a design is in the order of

$$\mathcal{O}((((P^S)^M)^L)^A)$$

where A is the total number of applications in a system, L is the total number of possible algorithms, M is the micro-architecture level operations, and P is the physical design options of the standard cells.
The Silago solution reduces the design space to

$$\mathcal{O}(M^L)$$

where M is the number of implementations of the same function present in the library of Silago blocks and L the number of application needed by the system. The number of possible solutions is significantly lower but by having a more precise estimation of the energy consumption of each Silago block the high level synthesis tool is able to make better choices to meet the constraints.
Other than that, the amount of saving possible at RTL level is significantly more than the one achievable at physical level [27].
At last it should be noted how the Silago method enables a faster synthesis (2.3), therefore more solutions can be evaluated in a shorter time than commercial HLS tools. The shorter time allows the Silago HLS tool to also explore global optimization at algorithmic level, something not always present in commercial HLS tools [27].

2.3 Silago Environment

The Silago method utilizes two main tools: VESYLA [31] [32] for the arithmetic level synthesis (High Level Synthesis) and Sylva [33] [34] for application level synthesis.

Figure 2.4 describes the structure of the flow. As a foundation, the Silago Physical Design Platform has to be defined. In that part of the flow, Silago blocks like DiMarch (Distributed Memory Architecture), DRRA (Dynamic Reconfigurable Resource Array), NOC, and flexilators (Small processors used for support and control operations) need to be defined down to their physical description.They need to be compliant to the SiLago methodology and fully characterized.
Each block is then used by the High Level Synthesis Tool Vesyla in order to create the FIMP (Function Implementation) library. What the tool does is an arithmetic logic synthesis of algorithms defined in MATLAB. It then writes the Configware necessary to run the blocks and does the binding and

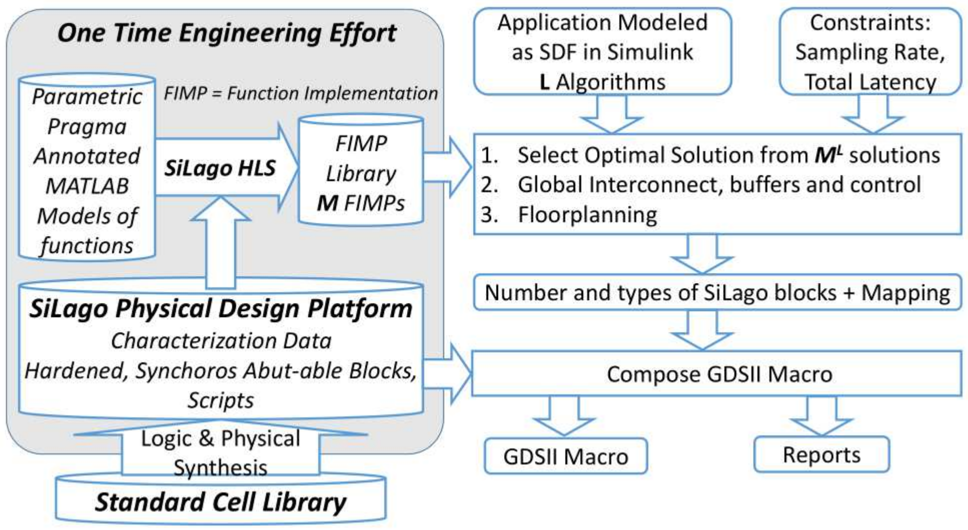

One Time Engineering Effort
Parametric Pragma Annotated MATLAB Models of functions
FIMP = Function Implementation
SiLago HLS
FIMP Library M FIMPs
SiLago Physical Design Platform
Characterization Data
Hardened, Synchoros Abut-able Blocks, Scripts
Logic & Physical Synthesis
Standard Cell Library
Application Modeled as SDF in Simulink L Algorithms
Constraints: Sampling Rate, Total Latency
1. Select Optimal Solution from M^L solutions
2. Global Interconnect, buffers and control
3. Floorplanning
Number and types of SiLago blocks + Mapping
Compose GDSII Macro
GDSII Macro
Reports

Figure 2.4: The Silago flow from standard cells description to final GDSII and reports

hardware allocation. The output of VESYLA is a certain amount M of FIMP, each of them differing in terms of parallelism and architecture, therefore carrying different cost metric. The entirety of the flow described above is part of the One Time Engineering effort required by the Silago flow in order to create the FIMPs library. After that, the application level synthesis can take place.

Sylva is the application level synthesis tool used by Silago. It performs the mapping of the FIMP library depending on the constraints and algorithms dictated by the system model. It does so by evaluating a number of combination of FIMPs (M^L if M is the number of FIMPs per algorithm and L the number of algorithms needed by the application) and selecting the optimal one based on their metrics (energy consumption, latency, and area). Sylva also takes care of the global interconnections (The Global NOC which will be explained in chapter 4) and the floorplanning. At the end of the flow a GDSII Macro is produced along with reports on power, latency, and area.

The objective of this thesis is to create SiLago blocks for the system controller and the global NOC, from RTL down to physical synthesis. In the following chapters, the two type of blocks will be presented, and the methodologies used to design them explained. In particular the next chapter will take care of the system controller, which will use the RISC-V Instruction Set.

Chapter 3

SiLago System Controller

In this chapter the system controller made for Silago will be presented and the methodologies used to make it usable by Silago explained.
The first section will give an overview of the RISC-V project and the very processor used for the thesis.
Then two sections are dedicated to the architecture of the processor and the software characteristics of it (What happens from the C++ code to the binary file that is used at run-time by the processor).
The last section takes care of the network interface from the RISC-V processor to the Network on Chip architecture.

3.1 Processor decision

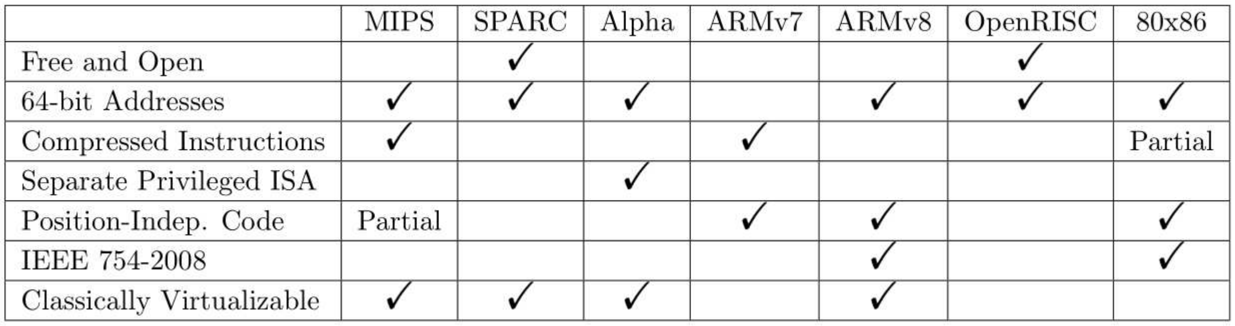
In section 1.3.2 it was explained how the industry moved to RISC architectures and why a call for open source ISAs was made by Asanović [17]. One of the objective for this thesis was to get over the LEON3 implementation for a system controller already present and introduce a RISC-V based controller.
In the next section, an overview of the RISC-V project will be given.

3.1.1 RISC-V project

RISC-V is a project carried out by the University of California, Berkeley. It started in 2010 and it was originally designed to support education and research. Their core motives and goals were described in [35], [17], and [36]:

- Create an ISA free and open source;
- Support both 32 and 64 bits address space;
- Provide a small but complete ISA which allows very different implementations (ASIC, FPGA, or full-custom);
- Support the IEEE 754-2008 floating point standard;
- Create an ISA fully virtualizable and with position-independent code;
- Support compressed instructions;
- Orthogonalize (e.g. separate) the user ISA and privileged architecture.

The following table, taken from [35] lists commercial and open source ISAs based on the list given above.



	MIPS	SPARC	Alpha	ARMv7	ARMv8	OpenRISC	80x86
Free and Open		✓				✓	
64-bit Addresses	✓	✓	✓		✓	✓	✓
Compressed Instructions	✓			✓			Partial
Separate Privileged ISA			✓				
Position-Indep. Code	Partial			✓	✓		✓
IEEE 754-2008					✓		✓
Classically Virtualizable	✓	✓	✓		✓		

Figure 3.1: Comparison of several ISAs based on RISC-V design goals, from [35]

From the analysis of the UC Berkeley group, no ISA followed their beliefs, and the best one was ARMv8, which is not suitable for academic purposes (Not free to use nor open source). From that, they decided to develop their own open source and free to use ISA. In 2013 the RISC-V Instruction Set was presented [37] and since then it has been taken care of by the RISC-V foundation.

3.1.2 RISC-V Insruction Set

The RISC-V Instruction Set is modular. At the very base there is the extension RV32I that consists of 45 instruction or RV64I which has 12 extra instructions.
On top of that different extensions can be added. Extensions after a while are frozen so that no further change can be added. Down below is a table taken from the RISC-V foundation website [38] that certifies the state of different extensions as of 17th of July 2019. Some extensions are also being ratified.

As mentioned above RISC-V has extensions that can build on the basic Instruction Set. A description of the main ones will be here presented:

- I - Basic Instruction Set
  - 32 or 64 bits registers;
  - Integer computational instructions;
  - Integer load and store instructions;
  - Control Flow instructions;
  - Mandatory code: each processor must be at least an I;
  - 32 registers $(x_0; x_{31})$.

Base	Draft Frozen?
RV32I	Yes
RV32E	No
RV64I	Yes
RV128I	No
Extension	**Draft Frozen?**
M	Yes
A	Yes
F	Yes
D	Yes
Q	Yes
L	No
C	Yes
B	No
J	No
T	No
P	No
V	No
N	No

Table 3.1: State of Base and Extension for RISCV, per [38] as of 17th of July 2019

- M - Mul/div extension
  - Multiply and divide integers operations.
- A - Atomic extension
  - Atomic read/write from/to memory;
  - Useful for inter-processor synchronization;
  - Load reserved and store conditional instructions.

- F - Single precision floating extension
  - Floating point registers added;
  - Single precision FP arithmetic;
  - Single precision FP load and store.
- D - Double precision floating extension
  - Expansion of floating point registers to 64 bits;
  - Double precision FP arithmetic;
  - Double precision FP load and store.
- G - General purpouse scalar instruction set
  - I+M+A+F+D;
  - 32 or 64 bits architecture.
- Q - Quad precision floating extension
  - 128 bits floating point registers;
  - Requires a base of RV64IFD (64-bit architecture with codes I+F+D).
- C - Compressed instructions
  - 16 bit instruction encoding for common operations;
  - RV32, RV64, RV128 compatible;
  - $\sim$ 30% code reduction.
- E - Embedded (Not a real code)
  - Special type of RV32I instruction set (RISCV with 32 bit instruction set with the I code) which has only 16 registers (Normal I has 32 registers);

  - It can be extended with codes M, A, or C;
  - It can't support floating point operations.

- Other codes (Most of these do not exist yet, they were only proposed)
  - L - Decimal floating point extension;
  - B - Bit manipulation extension;
  - J - Dinamically translated languages extension (Support for Java);
  - T - Transactional memory extension (Support for atomic operations involving multiple addresses);
  - P - Packed-SIMD extension;
  - V - Vector operations extension;
  - N - User-level interrupt extension.

This thesis work aimed at introduce the RISC-V Instruction Set to the SiLago framework. Because of that, a decision on the base and extensions of RISC-V used needed to be made. Because of the design goals of SiLago explained in 2.1, the targets have to be very basic and used for simple operations and mainly control of the Global Network on Chip.

With the above premises in mind, the simplest architecture would be a base I core. Useful extensions would be M, A, T, C, and E. M to be able to do multiplication or division (although not necessary at the current stage of development), A and T to allow better control operations (At the time of this writing, the T extension doesn't exist yet), and C and E to have the possibility to have 16 bit instructions or a more efficient area usage with the reduced number of registers. As for the type of address space 32, 64, and 128 bits are all valid but because of the limited use of the processor in SiLago phylosophy, a 32 bit architecture would be the most suitable.

3.1.3 Core decision

Since the objective of the thesis was more centered on the integration of RISC-V to SiLago rather than the creation of a processor, it was chosen to use a open source core to start the Silagofication of the RISC-V Instruction Set. Between all available cores at the RISC-V foundation website [38], only the ones that targeted the extension spectrum highlighted in section 3.1.2 were taken into consideration. The list below presents the pool of candidates for the first SiLago RISC-V processor core:

- Rocket Chip
  - A SoC design generator that emits synthesizable RTL;
  - Written in Chisel;
  - Extension for custom accelerator is possible;
  - 5 stage in order "G" architecture.
- VexRiscV
  - 5 stage RV32I with possible "M", "A", and "C" extensions;
  - Written in Scala;
  - Optimized for FPGA;
  - Instruction and data caches possible, as well as MMU and debug by Eclipse;
  - Optional interrupt and exception handling with machine and user modes;
  - Plug-in can be used to add custom instructions;
  - Tested on Cyclone IV with different configurations (different plugins added)
  - Minimum area $\rightarrow$ 732 LUT 494 FF with frequency of 177 Mhz. Maximum area $\rightarrow$ 3,324 LUT 2,010 FF with frequency of 116 Mhz.

- RV12
  - Highly configurable, single issue, single core RV32I or RV64I architecture;
  - Harvard architecture with simultaneous data/memory accesses;
  - Written in System Verilog;
  - 6 stage pipeline;
  - Optional modules: branch prediction, instruction cache, data cache, debug unit;
  - Configurable modules: instruction and data interfaces, BPU, cache size, associativity, and replacement algorithm;
  - Fast and precise interrupts.
- SCR1
  - It can be adapted to RV32E (Originally it's a RV32I);
  - Machine privilege mode;
  - Written in System Verilog;
  - 2 to 4 stage pipeline;
  - AX14/AHB-lite interfaces;
  - IRQ controller and advanced debug;
  - Features a number of configurable parameters.
- PicoRV32
  - The one chosen to replace the LEON3 in SiLago;
  - RV32I, RV32E, RV32IC, RV32IM, or RV32IMC;
  - Written in Verilog;
  - ASIC compatible;

  - Only one bus for both data and instructions;
  - IRQ features;
  - Co-processor and look-ahead memory implemented;
  - AXI compatible;
  - Clock frequency between 416 and 714 Mhz, area between 750 and 2000 LUTs with different implementations.

- MR1
  - As minimal as possible;
  - No interrupts, halt, traps;
  - Written in Scala;
  - 3 stage pipeline;
  - ∼1300 LUT and ∼84 Mhz clock.

- Ibex
  - Supported codes are ”I”, ”M”, ”C”, and ”E”;
  - 18.9 kGE with RV32IMC and 11.6 kGE when RV32EC is used;
  - ASIC and FPGA compatible;
  - 2-stage in order;
  - Written in System Verilog.

As it can be seen from the list above, PicoRV32 was the core used to introduce RISC-V into the SiLago framework. The choice was made mainly because its compatibility with ASIC, its semplicity, and the presence of the look-ahead memory interface.

3.1.4 RV32I Instruction Set

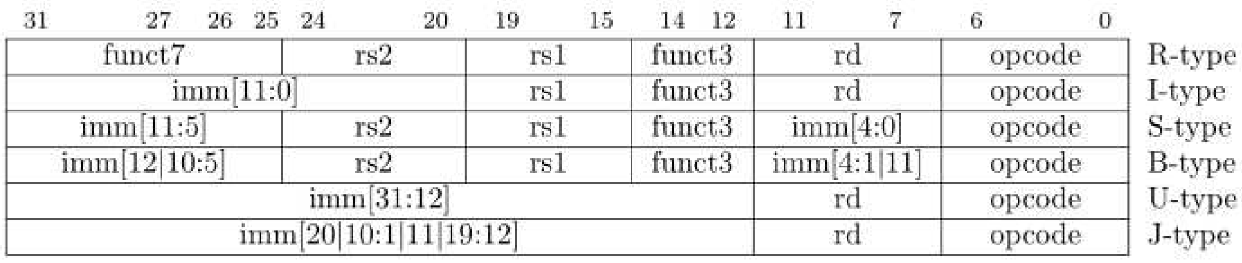
This subsection will be used to present the range of RISC-V instructions that are available with the choice of the PicoRV32 core, with the exclusion of the C extension. The figures reported, taken from the manual available in the RISC-V foundation website [38] (version dated 2019-06-08 yyyy-mm-dd) present:

3.2 The Instruction types in RV32I;

3.3 The RV32I base Instruction Set;

3.4 The RV32M extension;

3.5 The general purpose register listing.



31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

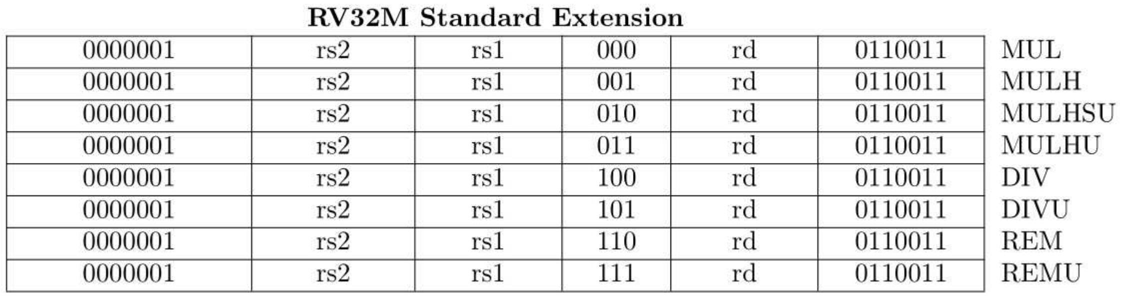
Figure 3.2: The bit division for the RV32I Instruction Set

In the next section the design of the SiLago global controller will be presented, and the design choices explained.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm	pred succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

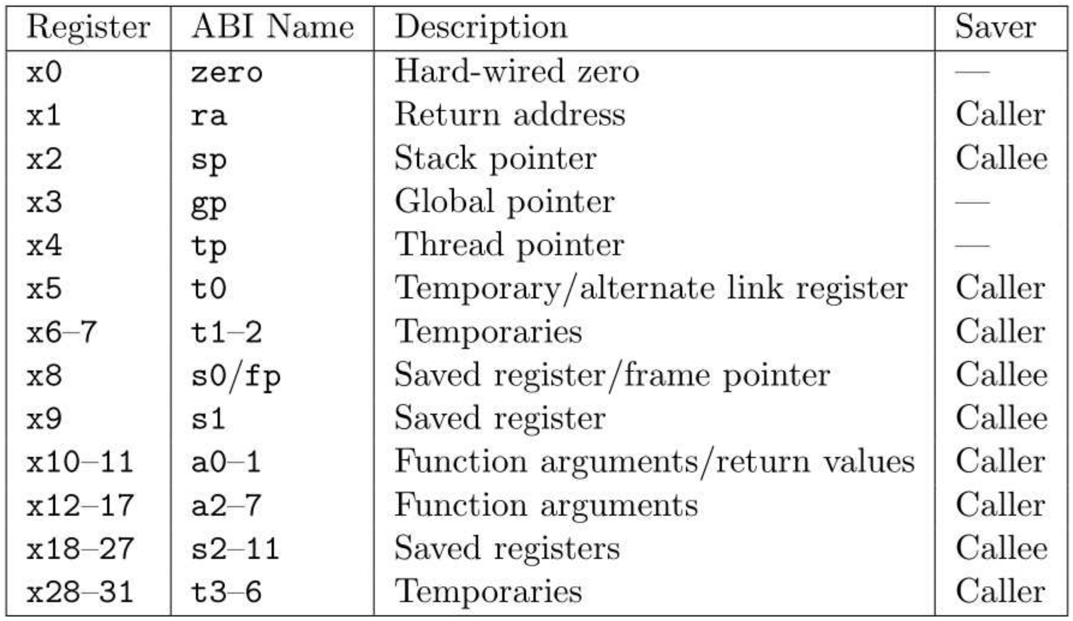
Figure 3.3: RV32I Base Instruction Set



RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 3.4: RV32M Extension Instruction Set



Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5`	`t0`	Temporary/alternate link register	Caller
`x6–7`	`t1–2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10–11`	`a0–1`	Function arguments/return values	Caller
`x12–17`	`a2–7`	Function arguments	Caller
`x18–27`	`s2–11`	Saved registers	Callee
`x28–31`	`t3–6`	Temporaries	Caller

Figure 3.5: The General Purpose Register listing for the RV32I Instruction Set

3.2 Architecture of SiLago processor

One of the main and peculiar characteristics of the PicoRV32 processor is the presence of only one bus for both instructions and data. The main goal of the processor is to control the Network On Chip and send data to it. In order to do it, the architecture showed below 3.6 has been designed.

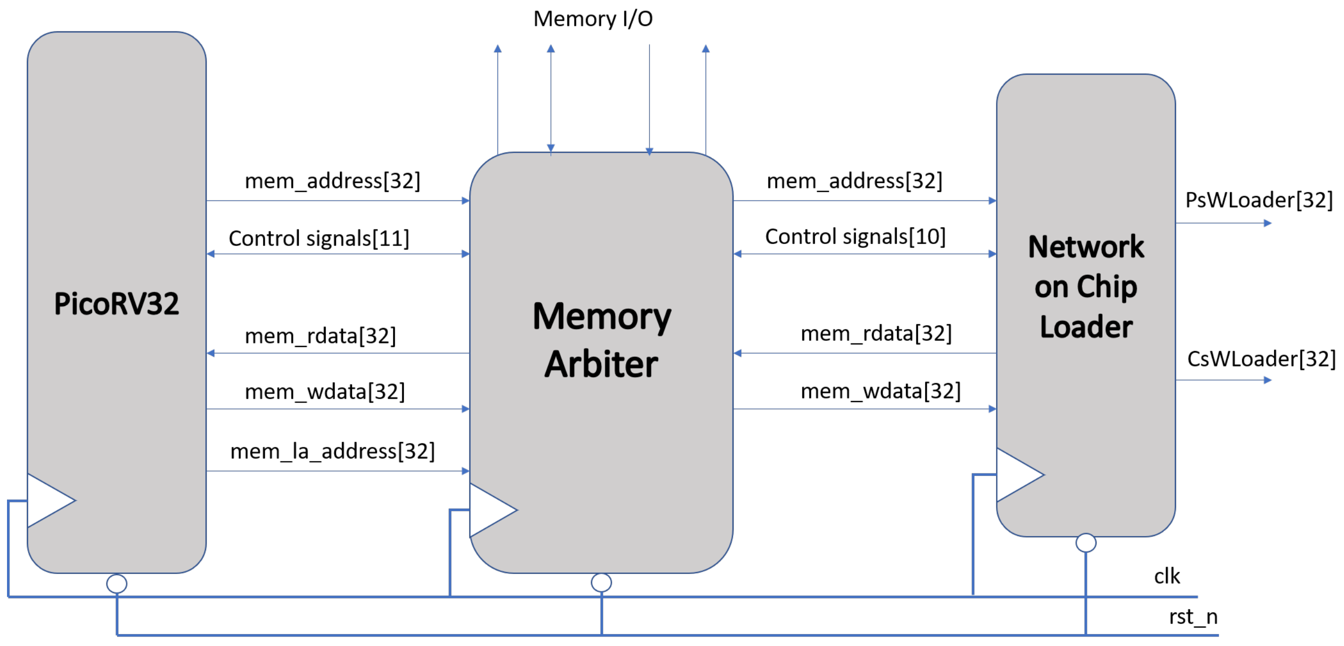


Figure 3.6: A system view of the Silago Processor. PicoRV32 is the processor itself, memory arbiter decides whether the memory request has to be handled by the external memory of the Network on Chip, loader is the network interface between the processor and the Network on Chip

The component on the left is the PicoRV32 processor described in 3.1.3. It interfaces only with the memory arbiter to which it exchanges addresses, data, and control signals. The control signals are: mem_valid, mem_instr, mem_ready, mem_wstrb[4], and mem_la_wstrb[4] A memory read works as follows:

- The processor raises mem_valid;
- mem_strb gets a value of ”0000”;
- the memory reads the value of mem_address and writes the correct value to mem_rdata in the same clock cycle as it writes ’1’ to mem_ready;
- The processor deasserts mem_valid.

A memory write instead needs these steps:

- The processor raises mem_valid;
- mem_wstrb gets one of these values:
  - ”0001” if the bits 0 to 7 need to be written;
  - ”0010” if the bits 8 to 15 need to be written;
  - ”0100” if the bits 16 to 23 need to be written;
  - ”1000” if the bits 24 to 31 need to be written;
  - ”0011” if the bits 0 to 15 need to be written;
  - ”1100” if the bits 16 to 31 need to be written;
  - ”1111” if the bits 0 to 31 need to be written;
- The memory writes the right range of data at mem_wdata to the mem_address address;
- The memory sends confirmation by raising mem_ready;
- The processor deasserts mem_valid.

The other two signals (mem_la_address and mem_la_wstrb) are part of the look-ahead interface. They anticipate the main signals by one clock cycle.

The component in the middle of figure 3.6 is the memory arbiter. Its role is to catch every memory transfer and vehicolate it to the right component: either the main memory or the Network On Chip. This can be made possible if the network on chip is mapped to the main memory. More details about it will be explained when the API are presented.
The component is structured as a simple Finite State Machine that rotates between two main states: NOC and RAM. In the first one, it opens the communication with the Network on Chip, in the second one the memory request is forwarded to the memory where the code is placed.

The component on the right of figure 3.6 is a Network Interface between the processor and the Network on Chip. It translates the information sent by the processor to Packet Switch words or Circuit Switch words, understandable by the Network on Chip. It is structured as a Finite State Machine.
When the loader senses a valid strobe coming from the arbiter it takes the mem_wdata and in case the address refers to a write in the CsWLoader, it writes the data directly in that output (State W1111 in 3.7), otherwise it uses the address and the data to create the right word understandable by the network on chip.

The bit manipulation used to create the right PsWLoader is explained in figure 3.8. The signal mem_address carries the information about the destination inside the network (Row R3 $\div$ R0 and Column C3 $\div$ C0), the type of message (T1 and T0) and Private DRRA signals (P4 $\div$ P0). The bits not displayed are just used as an offset (The Network on Chip is mapped to memory). The signal mem_wdata carries the proper data. Each write in the network on chip can be done 16 bits at the time (Except for circuit switch data). More about the network on chip will be explained in the next chapter.

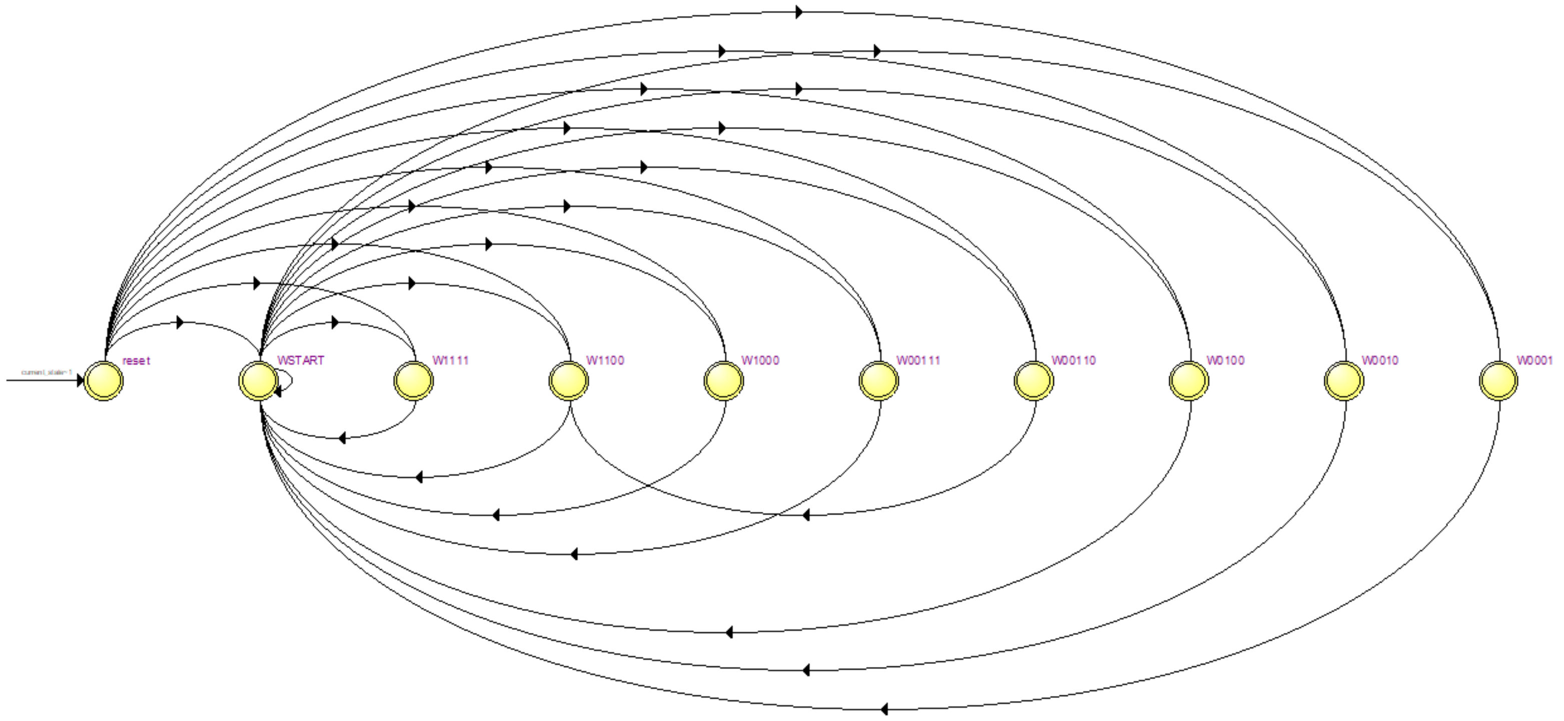


Figure 3.7: State machine table of the Network on Chip Loader. Picture taken from the software Quartus II

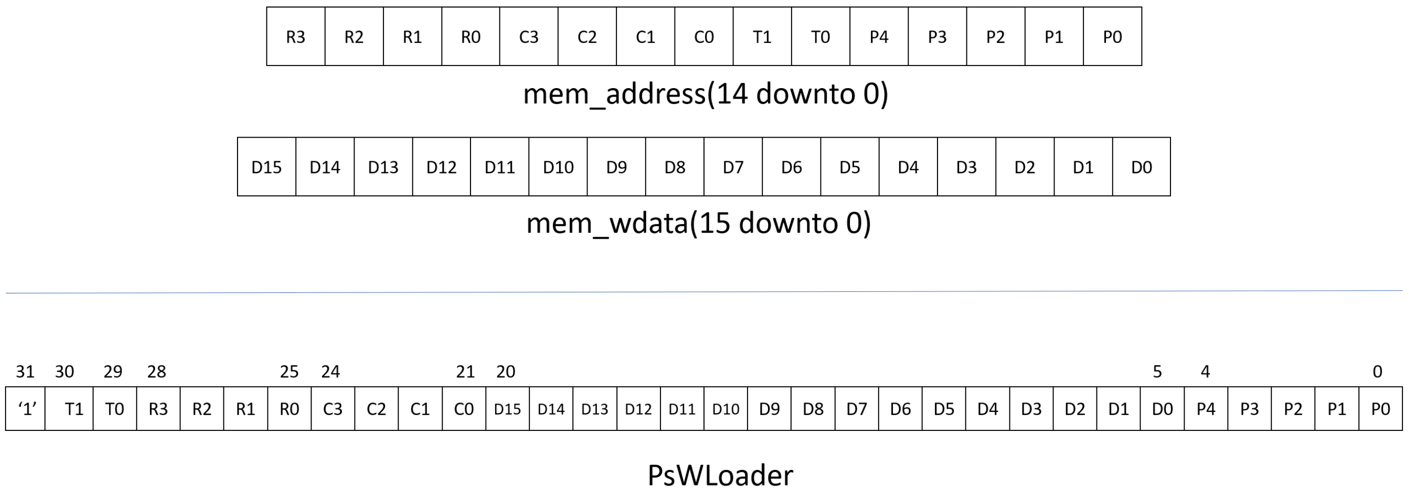


Figure 3.8: Visualization of the function used by the SiLago Network Interface between the processor and the Network on Chip

3.3 Software side of SiLago Processor

In the previous section (3.2) the hardware description of the processor side has been outlined. In this section the aim is to explain the chain of needed actions in order to go from a program written in a high level language like C/C++ to a binary file understandable by the processor. The process is divided into four parts:

1. Pre-Processing, the initial file (.c, .cc, .cpp, .cxx or similar) turns into a Preprocessor output file (extension .i or .ii). In turns:

    - Macros are substituted with actual values;
    - Comments are taken away;
    - Header files (i.e. stdio.h and stdlib.h) are substituted with the actual files.

2. Compiling, the pre-processed file is compiled and the output (extension .s) is an assembly file;

3. Assembling, the assembly instructions are translated into machine-level instructions (binary code, extension .o), divided in: text segment (the actual code), data segment (with the global variables), relocation information, and symbol table (these last two are used by the linker file);

4. Linking, the object files created in the assembling stage are linked together and library functions are put in a single executable file (executable and linkable format file, extension .elf). A custom linker file has been used which has its entry point at 0x10000.

On top of those actions, extra ones are needed to run a program in the SiLago processor.

The memory of the system control is not in the SiLago processor itself. In the design stage, a virtual memory has been created by means of a vhdl shared variable in the testbench simulation. In order to load the machine code into the memory some modifications of the file have been done by means of a python script. The output file of the python script is the final .hex file which is used in the actual ModelSim simulation. Figure 3.9 tries to visualize the entire process. It is possible to write different C/C++ files, everything will be linked together in a unique .hex file.

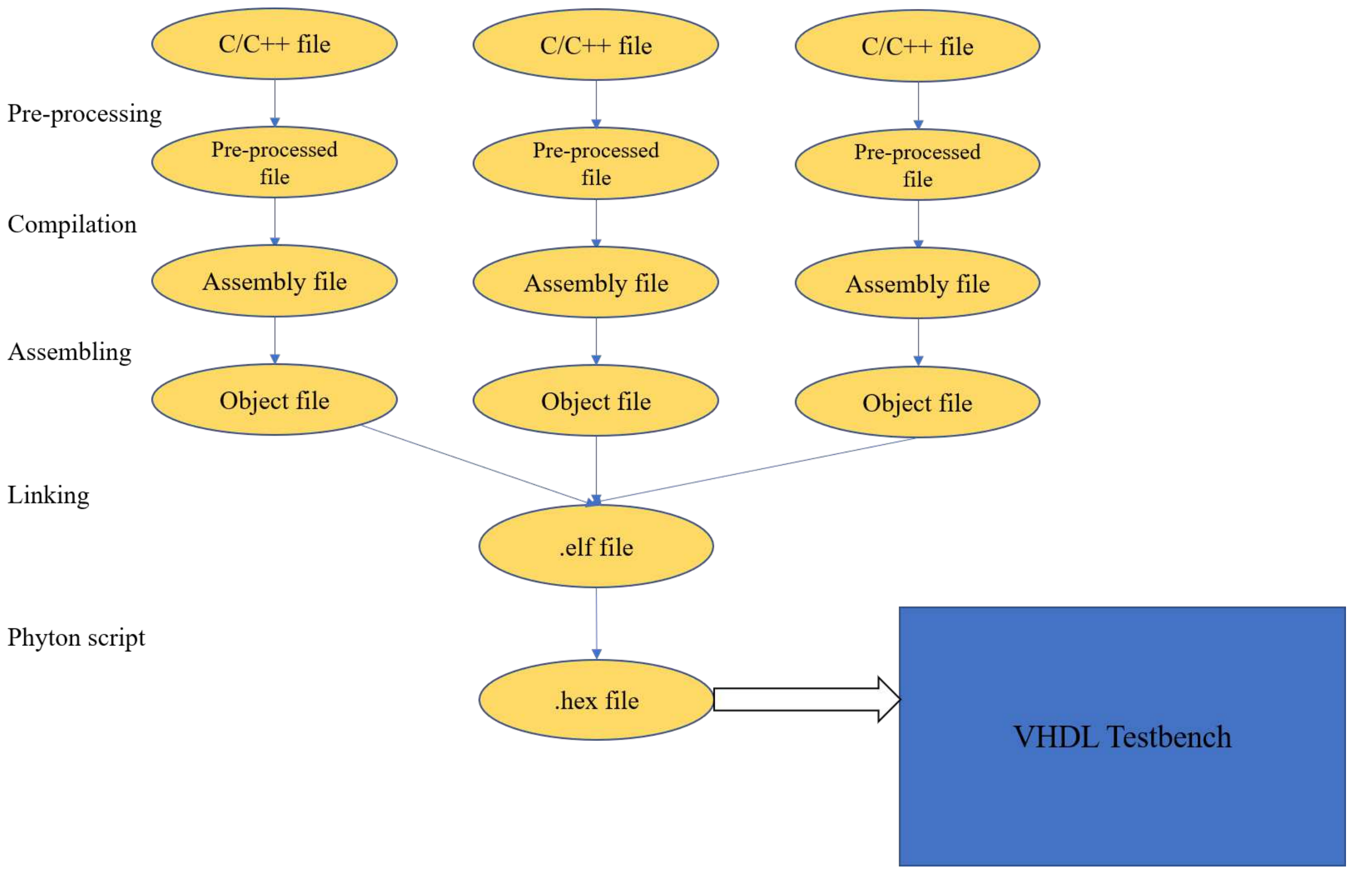


Figure 3.9: Visualization of the process from a high level language file to bare metal simulation

The standard memory size used is 8MB but programs tested with -Os compiling option (Optimization for small code) don't go over 1MB.

3.4 Network on Chip communication

In this section the decision of a memory mapped network on chip will be discussed. The design decision on the processor side of the network on chip was to map part of the addressing space of the processor to the global interconnects of SiLago. An offset value of the memory has been chosen for both the Packet Switch and Circuit Switch outputs of 3.6, and the component Memory Arbiter decides whether the memory operation has to be performed by the memory itself or the network on chip.

The decision on memory-mapped IO was made after the compilation of several programs. The total addressable space of the Instruction set is 32 bits. Every cell of memory addressed is 8 bits, so the total memory addressable for the RV32I Instruction Set is:

$$TOT_{memory} = \frac{8\ bit}{memory\ cell} * 2^{32} memory\ cell = 34,359,738,368\ bits = 4GB$$

Compilation of the simple programs who ran on the previous versions of the LEON3 processor never went over the 18th bit of addressing (256KB of addressable space), so the 32 bits addressable space was enough to fit also the network on chip. The NoC addressing takes 15 bits, so a total of 64KB has been reserved (32KB for the Packet Switch NoC, 32KB for the Circuit Switch NoC). The exact structure of the network on chip will be discussed in the next chapter.

Since the communication processor-NoC is memory-mapped based, the way to send information to it is using the store half word assembly instruction (sh, an S-type instruction as of 3.2), which writes 16 bits to the memory.

Figure 3.10 visualizes the store instruction. The opcode is 0x23, funct3 is 001, rs2 is the register that holds the value to be sent to memory (only the 16 lower bits are taken), and the destination in memory is found by adding the sign-extended 12-bit offset to the value of register rs1. An assembly store instruction then would look like this:

```
sh rs2,offset(rs1)
```

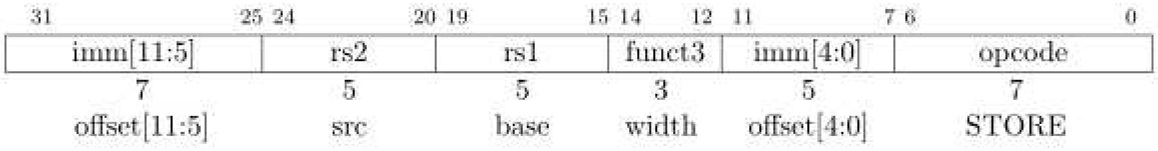


Figure 3.10: Format of the store instruction for the RISCV 32I instruction set, from [38]

Chapter 4

SiLago Global Network on Chip

This chapter's goal is to describe the existing SiLago Global Network communication protocol and to illustrate what algorithm has been designed for this thesis.
The first section a visual example of the Global NoC within a chip is presented. Then an explanation of the communication protocol is given.
The last two sections are dedicated to the new algorithm. Firstly there is a general presentation of Network on Chips as a whole, where different characteristics of them are presented (Mainly different topologies and routing techniques). Here the algorithm is presented along with some examples of its functioning. The last section instead focuses on deadlock analysis and what are the limitations of the algorithm as of the end of the thesis work.

4.1 Network on Chip for VLSI design

In section 1.3.3 it was explained the reasons why NoCs became the standard way of communications at the global level of VLSI systems.
The area overhead of network interfaces and the variable latency are tradeoffs that have been accepted in the VLSI community thanks to the sharability of routers and the reduction of wires (especially the longer ones).

For the SiLago project a global network on chip was needed in order to connect different blocks. In picture 4.1 a possible implementation of a Silago system is shown. The Network on Chip is pictured in orange and each router (dark orange) is connected to a network interface block (darker grey in the DiMarch, darker blue in the DRRA, dark green in the Protocol Processing Region).

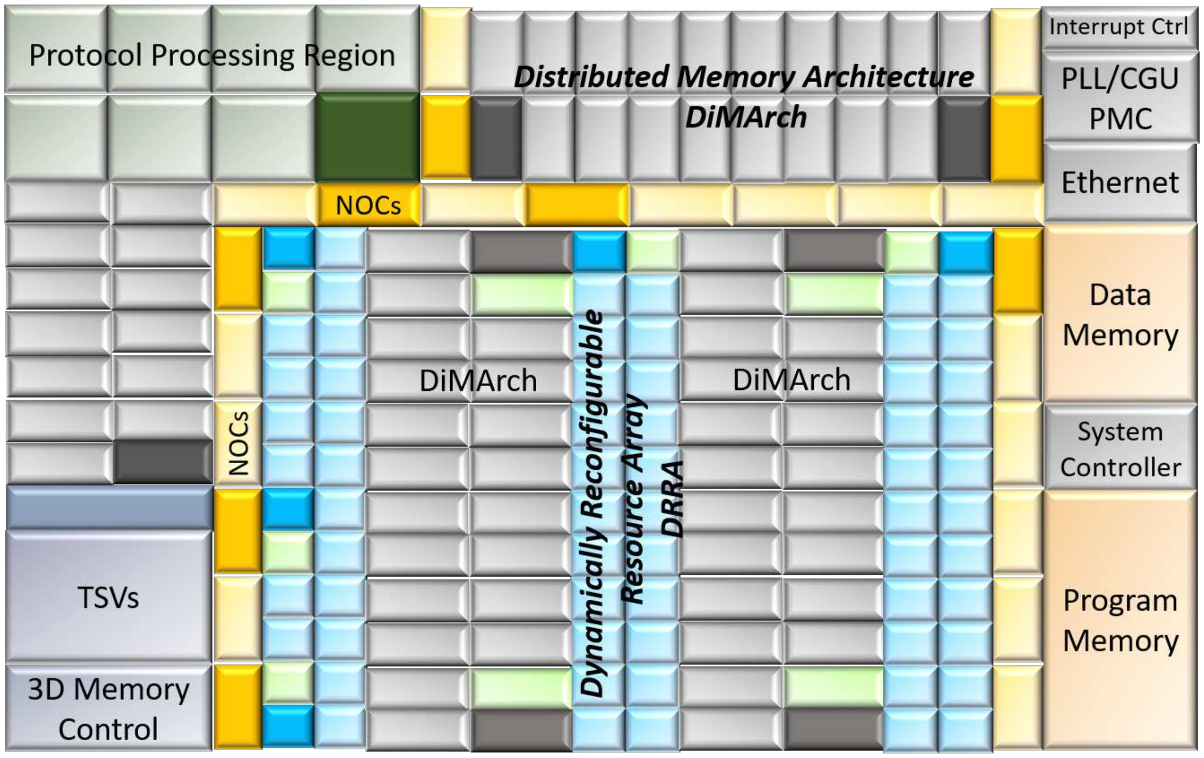


Figure 4.1: Example of a Silago system on chip. The Network on Chip puts into communication the different parts of the system

4.2 SiLago Global Network on Chip

The main goal of the network is to allow exchange of information along the chip. The main area are: control, configuration, and creating system paths. Hemani et al. in [28] explain the Network on Chip protocol and its structure. In this section the packet switch and the circuit switch will be presented.

The main structure is composed of three elements:

- Buffered or pipelined wires, passive elements that provide connectivity between the routers;
- Packet Switch, active element used for control and configuration;
- Circuit Switch, active element used for creating system paths.

4.2.1 Packet Switch

In figure 4.2 the architecture of the packet switch is explained. The input word is sent to both the destination selector, to compute the output direction, and the demultiplexers, to actually route the word. In case the word has found its destination (self), the packet type is used to understand the operation needed.
The Packet Switch (PsW) has then 5 full-duplex channels (North, South, East, West, Resource) and a Circuit Switch configuration channel (half-duplex). The data word is made of 32 bits organized as follows 4.3:

- Valid (1 bit);
- Packet Type (2 bits);
- Destination (8 bits, first 4 for row, second 4 for column);
- Data (16 bits);

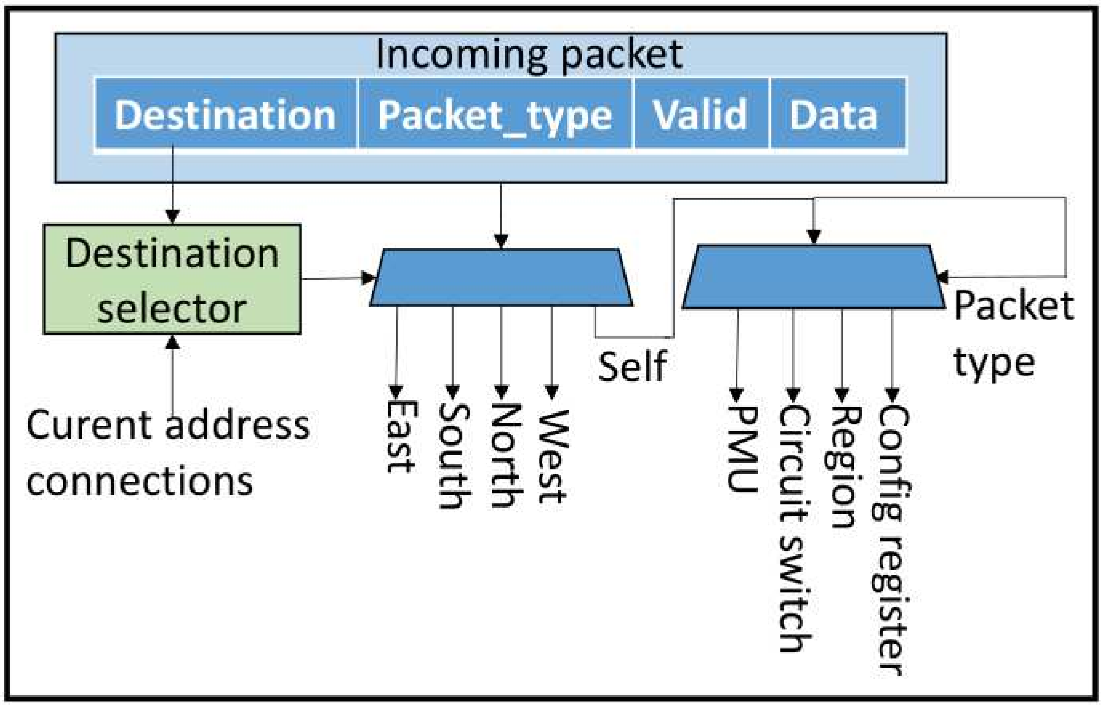


Figure 4.2: Packet switch architecture. The incoming packet goes into the destination selector and the 32-bit data word is then sent to the right output

- Private (5 bits).

Valid	Packet type	Row	Column	Data	Private

Figure 4.3: Packet switch word structure

The packet type indicates the purpouse of the PsW data. In particular:

”00” used to configure the packet switch registers. The data in this case is 4 bits long and indicates whether the router is connected to its neighbour to the South (bit 3), to the North (bit 2), to the East (bit 1), or to the West (bit 0). ;

"01" used to configure the region connected to the router. Here the data is a 16-bit word which is translated by the Network Interface of the destination region;

"10" used to configure the Circuit Switch at the same position of the Packet Switch. Here the data is 15 bits long (Pictured in 4.4). Each connection uses 3 bits and the possibilities are:

0x0 No change

0x1 This channel is the output of the West input;

0x2 This channel is the output of the East input;

0x3 This channel is the output of the North input;

0x4 This channel is the output of the South input;

0x5 This channel is the output of the Resource input.

"11" used for PMU (Power Management Unit), not yet defined.

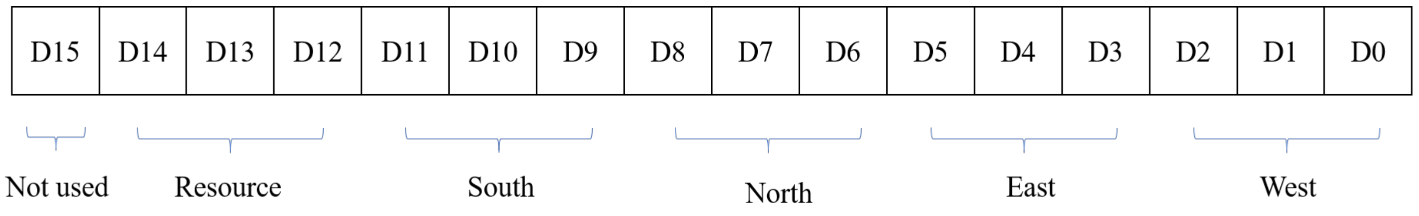


Figure 4.4: Data bits of a Circuit Switch configuration packet

4.2.2 Circuit Switch

The circuit switch is a parallel network to the packet switch and it's used for fast, low power communications. The circuit switch can be programmed by the packet switch to create dynamic system paths (unlike the packet switch, which relies on an algorithm to route the data). A picture of the circuit switch is presented in 4.5

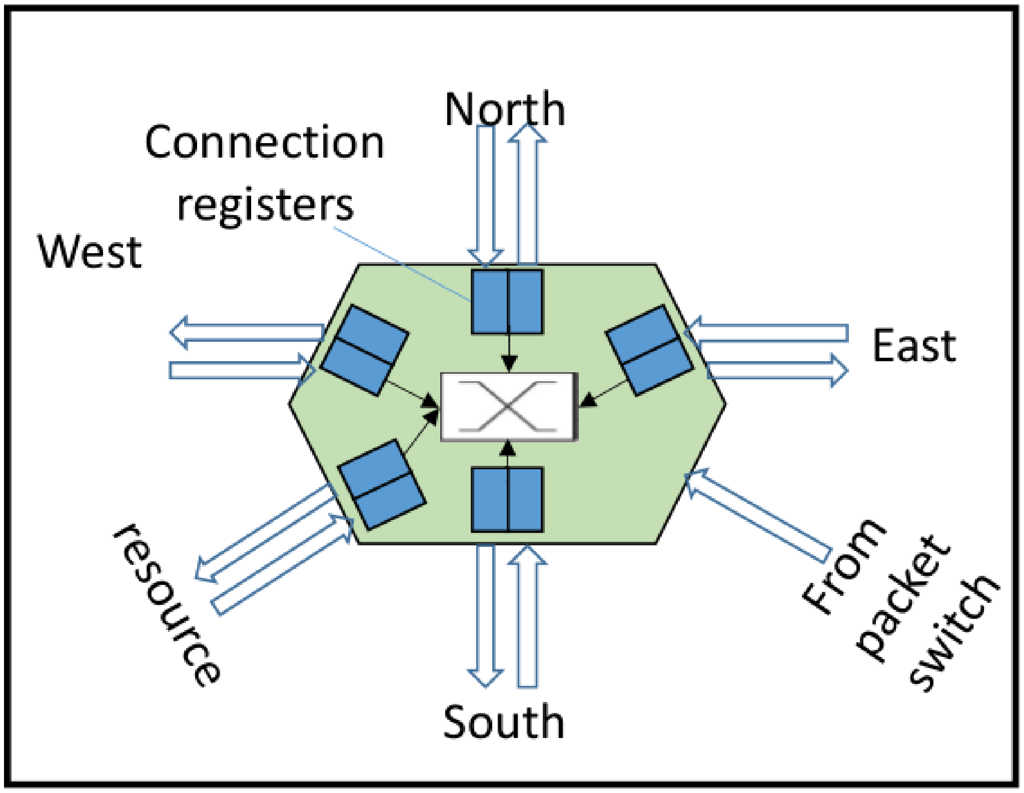


Figure 4.5: Circuit switch architecture. The incoming data gets routed following the indication on the connection registers, initialized by the packet switch

4.3 Global NoC Routing Algorithm

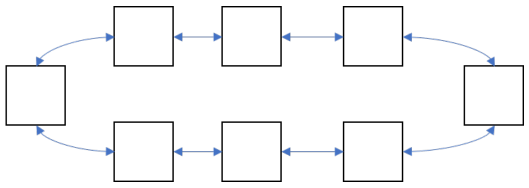
In this section some routing algorithms will be presented and the one used for the SiLago Global Network on Chip will be explained.

4.3.1 Topology

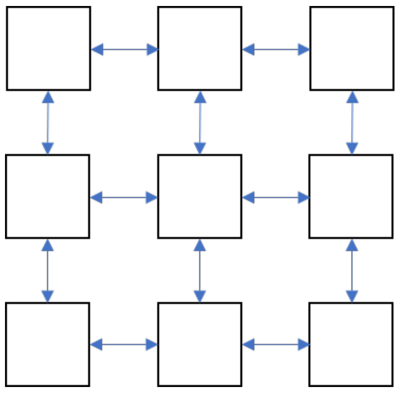
Network on Chips can have different topologies, the most common are:

- Rings, which are cheap and also the preferred disposition in multicores systems [39];
- Meshes, which have diversity of paths but have different performances depending on the routers disposition (A router on the edge has a worse performance than a router on the middle of the network);
- Torus, which are meshes that eliminate the performance diversity employing more links. The link length can be unequal (Normal torus) or equal (Folded torus);
- 3-D placing, used in supercomputers or data centers.

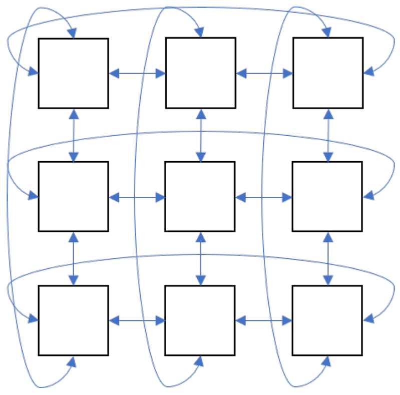
The SiLago Global Network on Chip employs an irregular mesh. The structure is fixed on a maximum of a 16x16 grid (Addresses of the NoC are enclosed in 8 bit, 4 for the row, 4 for the column, as explained in 4.2.1) but inside different configurations are possible. Two examples of topologies are pictured in figure 4.7.



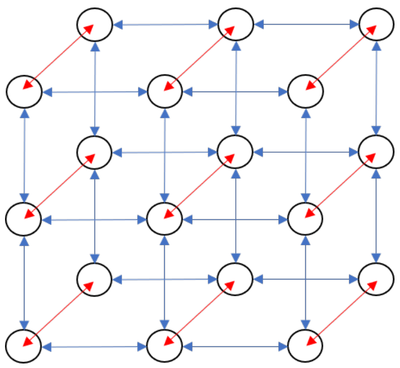
(a) Ring



(b) Mesh



(c) Taurus



(d) 3-D

Figure 4.6: Different Network on Chip topologies, grey boxes are the routers in the network, white boxes are empty

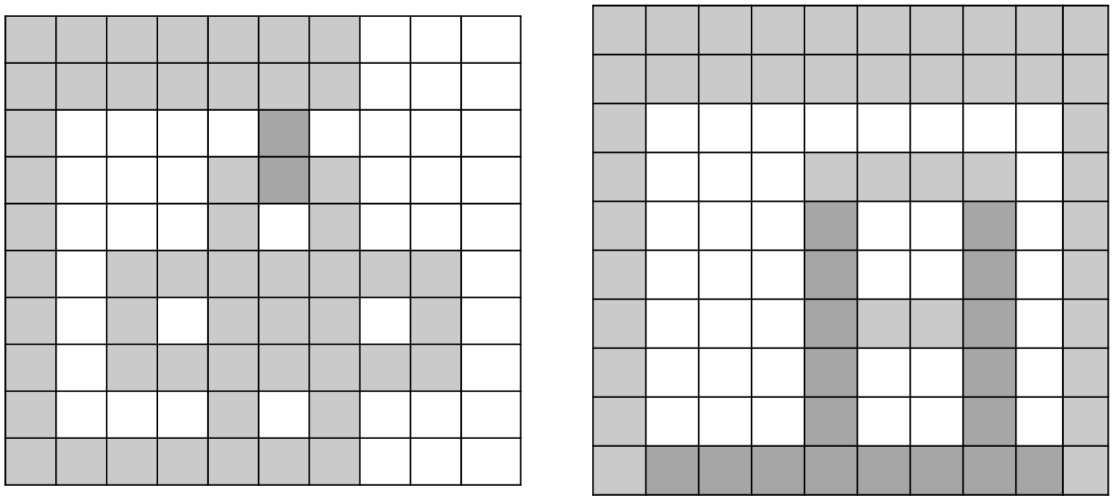


Figure 4.7: Two examples of Global NoC implementation inside a 10x10 grid. Maximum grid size is 16x16

4.3.2 Routing

Routing in a Network on Chip is closely related to topologies. Different topologies can employ the same routing algorithms, but the performance can vary a lot.

Routing tecnhiques can be observed on three different levels:

- Path length;
- Path diversity;
- Hardware implementation.

The path length can be of two types: minimal, if the shortest path is always chosen, or non-minimal, if non-shortest paths can be chosen. Since the Global NoC doesn't have a fixed structure, the routing style will be non-minimal.

The path diversity indicates how to select different paths between the set of paths $P_{a,b}$ from router a to router b. Path diversity can be:

- Deterministic if between router a and b always the same route is chosen. It's easy to implement and analyze but it is quite restrictive;
- Oblivious if a route is chosen without considering any information about the router and the network state;
- Adaptive if the route is chosen with a consideration on the current state of the network;
- Profitable if the packet always moves toward the destination (Minimal routing is profitable by definition);
- Non-profitable if routes moving away from the destination can be chosen;

For what concerns the hardware implementation choices can be:

- Source routing, in which the entire route is embedded in the packet. This moves the routing algorithm from the routers to the packets moving through the network so it will results in longer packets and smaller routers;

- Node-Table routing, in which the router itself has a look-up table with a number of rows equal to the number of routers in the network and as data the best output in order to reach the given router. This is the opposite of source routing and will give shorter packets but bigger routers in terms of area. This solution doesn't scale well because look-up tables can end up being very big depending on the number of routers;

- Combinational circuits, in which the packet carries only the destination coordinates and the router computes the output port based on the network state.

Each of the three metrics (Path length, path diversity, and hardware implementation) possibilities can be combined in order to fit the specific of the given network.

In the next subsection a couple of examples of routing techniques will be presented and in the next section, the one chosen for the Global NoC will be explained.

Routing examples

The DOR algorithm is the basic algorithm for mesh or torus type Network on Chip. DOR stands for Dimension Order Routing and the packet is first routed to the correct X coordinate, and then sent to the correct Y coordinate. That is called XY routing (x first, y second), the opposite is called YX routing. So if a packet need to go to position $P_{a,b}$ from position $P_{0,0}$, it will firstly go to $P_{a,0}$, and then finally to $P_{a,b}$. The example is pictured in figure 4.8.

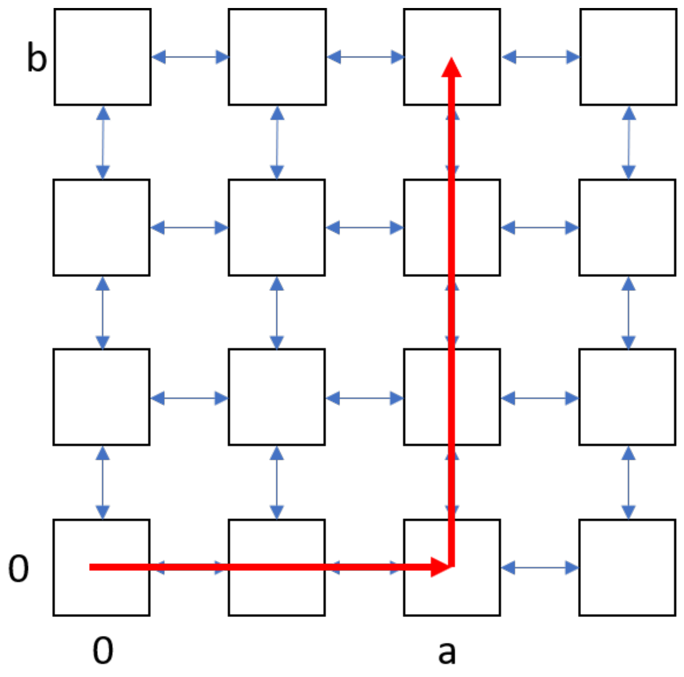


Figure 4.8: Example of a DOR x first algorithm, the packet gets routed to $P_{a,0}$ from $P_{0,0}$, and then to $P_{a,b}$

The basic DOR algorithm is minimal because the shortest route is always chosen and deterministic because the path can be predicted a priori. It then doesn't give a good load balancing between routers.
A modification of the DOR algorithm is the O1TURN [40] in which at the beginning of a path, the first router decides whether to do XY-DOR or YX-DOR. This algorithm adds some complexity to the routers but it's still minimal and not deterministic (it's oblivious because the XY or YX decision doesn't take into account the state of the network).

Another variation of the DOR algorithm is the Valiant's algorithm [41],

[42]. In this version of the DOR a random router D_{part} is chosen and the packet is firstly routed from source S to D_{part}, and then from D_{part} to D, as pictured in figure 4.9. This approach is not minimal but balances the loads in a better way than the O1TURN. The algorithm can be oblivious if the intermediate node is choosen randomly, or adaptive if the network state is taken into consideration in the choice.

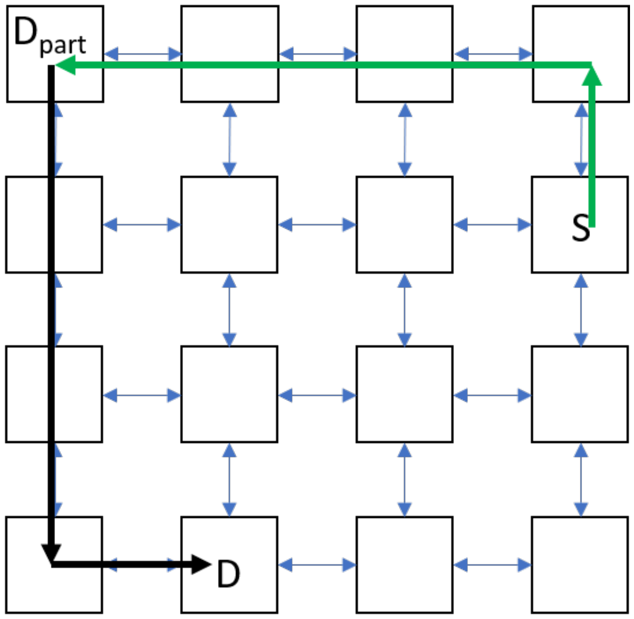


Figure 4.9: Example of a Valiant algorithm, the packet gets routed to D_{part} from S with YX-DOR, and then to D with again YX-DOR

4.3.3 SiLago NoC Algorithm

The NoC algorithm used in Silago is a XY-DOR with both oblivious and adaptive behavior in order to solve the randomness in the network construction.
The router uses a combinational circuit approach in order to compute the correct output, and the algorithm is non-profitable and not minimal.

The only NoC that needs an algorithm is the Packet Switch, since the Circuit Switch is totally passive to the Packet Switch. In the PsW router the registers are:

- 'configured', indicates whether the router is initialized or not, and it can only change once;
- 'row_reg', indicates the Y-position of the router, it can be initialized only once;
- 'col_reg', indicates the Y-position of the router, it can be initialized only once;
- 'config_reg', can be modified and indicates what kind of connection the router has. It is a 4 bit vector and the bits indicate:
  - config_reg(3), if '1' the South direction exists;
  - config_reg(2), if '1' the North direction exists;
  - config_reg(1), if '1' the East direction exists;
  - config_reg(0), if '1' the West direction exists.
- 'howmanyoutputs', indicates how many outputs the router has;
- 'circuitSW_config', used to send the data to the Circuit Switch router.

The main ideas of the algorithm are:

- The reading order is always Resource first, then West, South, East, and lastly the North input;
- The bits 5, 4, and 3 of the incoming data are called private_code and carry the history of the packet and information in case of re-routing (Adaptive part of the algorithm);
- The first valid word received by the router after power on has to be the initialization word;
- Each router has at least two outputs between North, South, East, and West directions.

The router's algorithm consists in three checks. At the start of the cycle the router checks if it's initialized (i.e. checks its "configured" register). If the initialization hasn't been made it checks every input in the read order and if it finds a valid word with a packet type (bit 30 and 29 of the incoming data) of "00" it changes the configured register to '1', writes the row and column bits in the row_reg and col_reg registers, sets the correct value of the register config_reg, and writes in the register "howmanyoutputs" the correct number of outputs (i.e. how many bits have been set in the config_reg register). Refer to figure 4.10 for a visualization of a configuration packet for the PsW router.

31	30	29	28			25	24			21	20												8			5	4				0
'1'	'0'	'0'	R3	R2	R1	R0	C3	C2	C1	C0	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	S	N	E	W	P4	P3	P2	P1	P0

Figure 4.10: Configuration of an initialization packet for the Packet Switch router, the bit 31 is a valid bit, the bits 30 and 29 are "00" to indicate the packet type, bits 28-25 will become the row_reg, bits 24-21 will become the col_reg, and bits 8-5 are copied into register config_reg

The second check is done to understand if a value has been sent back the previous clock cycle.

A packet is sent back if and only if in the previous clock cycle no other output direction other than the input one were available (The routing algorithm tries to never send back any packet, so if a data comes in from the South port, the South output becomes the least preferred output). A packet carries this information in the private_code (bits 4, 3, and 2 of the packet), which is in this case set to "X11" in the previous clock cycle, and reset to "X00" by the router before sending the word back.
Due to the function of the Packet Switch, it's high unlikely that a value is sent back: the Packet Switch main task is to set the Circuit Switch, where the bulky transfer of data takes place.

The third check is the main one and involves the routing algorithm.
If the packet has reached its destination, the data is routed to the resource output. Otherwise the routing algorithm takes action.
The algorithm behaves in two different ways, whether the router itself has two or more directional output (North, South, East, West, not considering the Resource). If the router has only two outputs the data is sent to the opposite output with respect to the input. If the router is of the type of figure 4.11 for example, if the data comes from the West input, it will be re-routed to the North output.

If the router instead of two has three or four outputs, it follows the classic XY-DOR algorithm: the router tries to send the data to the right X position first (right column), then to the correct Y position (correct row). If the desired position is not available, the router has different behaviors depending on the network state(adaptive behaviour).

If the router is in the wrong row, the private code (bits 5, 4, and 3 of the data) is checked. The private code, as mentioned before, carries information about the past state of the network, and adds both the adaptive and the oblivious behaviour. The possible configurations are:

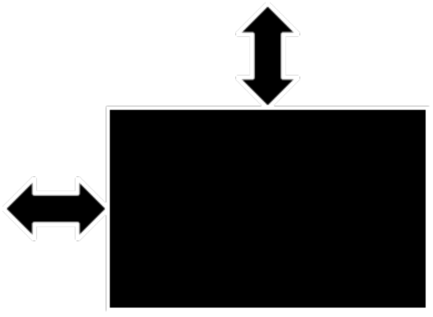


Figure 4.11: A representation of a router with two directional output, in this case North and West. If a data comes from the North input, the algorithm tries to re-route it to the West output and vice-versa

- "X00" means that the normal DOR algorithm can be performed. If the XY-DOR algorithm gives an output which is not available, or it's the input direction, the router tries to re-route the incoming word in this order (the first available output in the following list is used):

    1. Correct output of YX-DOR;
    2. Opposite output of the XY-DOR;
    3. Opposite output of the YX-DOR.

  ;

- "001" means that the word needs to be sent to the first available output of this list (if the output direction would not be the same as the input one):

    1. North;
    2. West;
    3. East;
    4. South.

- "010" is the same as "001" but the order is:

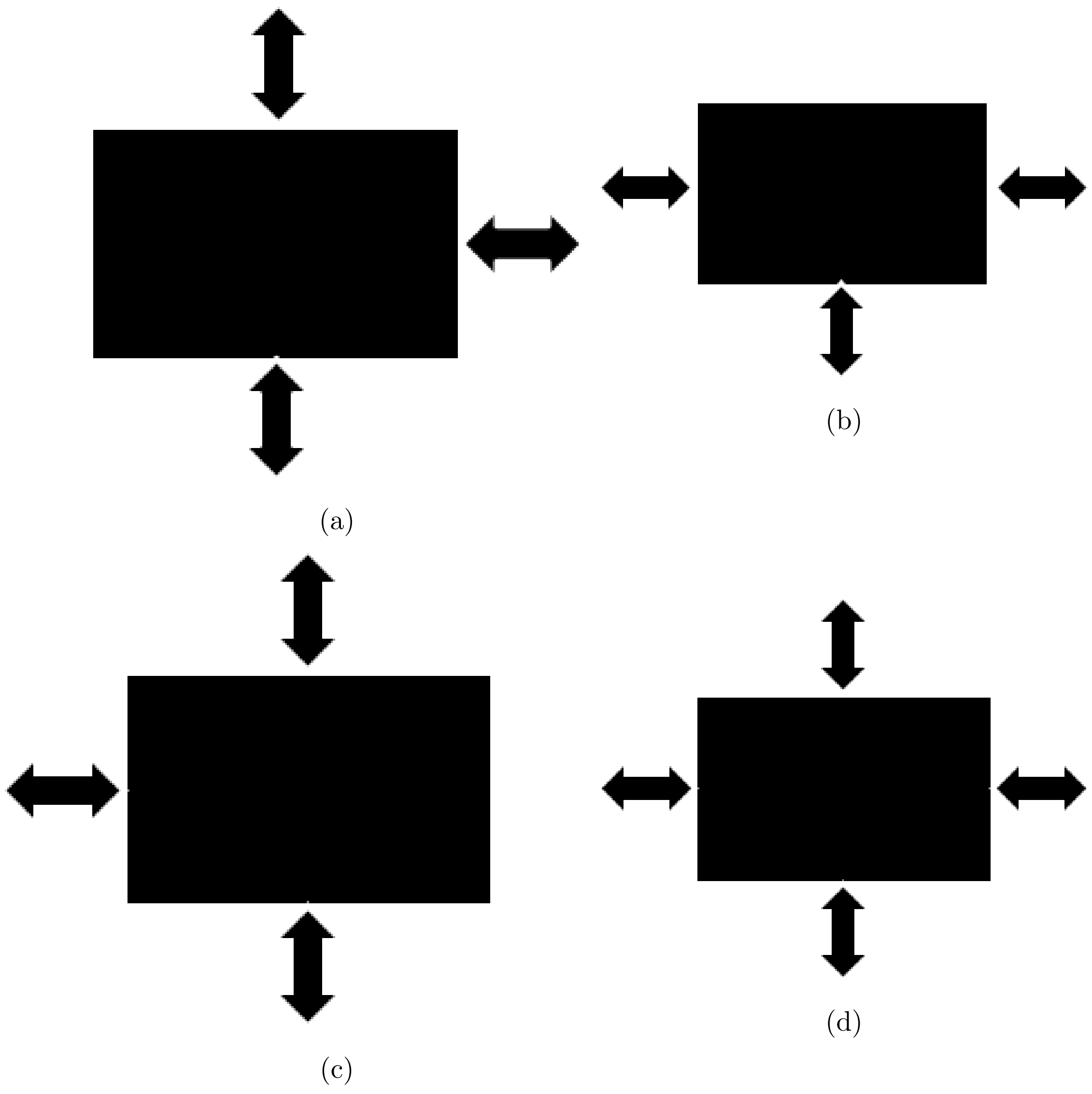


Figure 4.12: Fourrepresentations of a router with three or four directional outputs

   1. South;
   2. West;
   3. East;
   4. North.

- "101" is the same as "001" but the order is:
   1. North;
   2. East;
   3. West;
   4. South.

- "110" is the same as "001" but the order is:
   1. North;
   2. West;
   3. East;
   4. South.

- "X11" means that the word needs to be sent back to the input direction.

If the packet is in the correct row, and the bits 2 and 3 the incoming data are both '0', the router tries to send the packet North or South, depending on the XY-DOR correct direction. If that direction is not available, the oblivious part of the agorithm takes place:

- If the packet needs to go South but the direction is not available, the bit number 3 of the incoming word is raised, so that the word will try to go South as soon as possible. The packet is then routed following this order (The first output available which is not the same as the input is chosen):

1. East if bit 4 of the private_code is '1', West if '0'. That bit is then negated;
2. East if bit 4 of the private_code is '0', West if '1'. That bit is then negated;
3. North.

- If the packet needs to go North but the direction is not available, the bit number 2 of the incoming word is raised, so that the word will try to go North as soon as possible. The packet is then routed following this order (The first output available which is not the same as the input is chosen):

  1. East if bit 4 of the private_code is '1', West if '0'. That bit is then negated;
  2. East if bit 4 of the private_code is '0', West if '1'. That bit is then negated;
  3. South.

In figure 4.13 and 4.14 two simple examples of the routing algorithm are explained. In figure 4.14 in particular it's possible to understand the basic idea of the algorithm: the word tries first to find the right column, then goes around horizontally trying to move North/South as soon as it can.

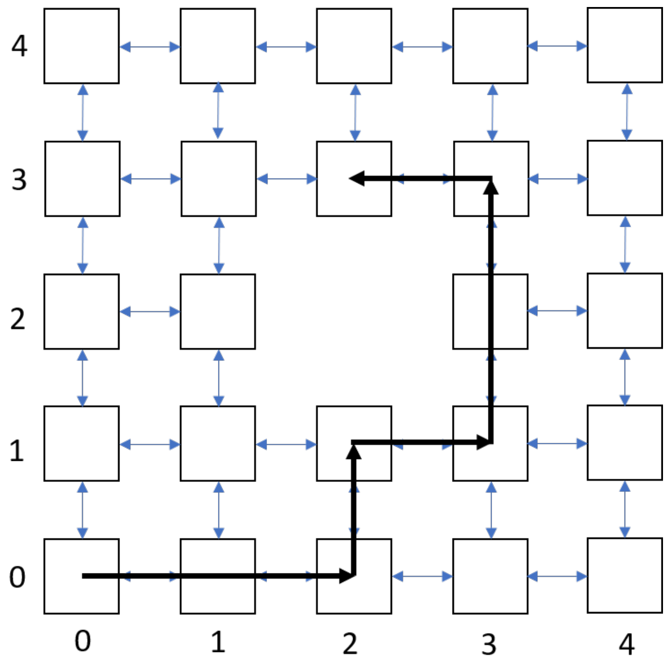


Figure 4.13: In this example the information travels from router (0,0) to router (2,3). As soon as the word reaches the right column it tries to go up and when in position (2,1) it goes east (supposing that the private_bit in that case was '1' to start off)

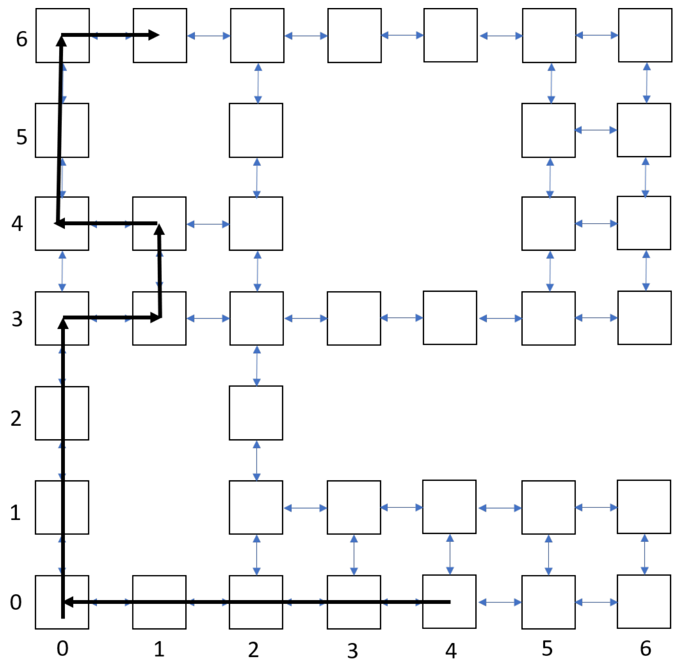


Figure 4.14: In this example a word needs to go from position (4,0) to (6,1). It can be seen how, the data tries to always go back to the right destination column.

4.4 Limitations and deadlock analysis

Deadlock in a Network on Chip happens whenever a packet is stuck in the same router (static deadlock), or it follows a circular trail without never reaching its destination (dynamic deadlock).

In the SiLago Global NoC static deadlock is virtually impossible for two main reasons:

- The majority of information for the Silago global NoC is supposed to be handled by the Circuit Switch, so the Packet Switch is supposed to have few words going around at all times;
- The algorithm is neither deterministic nor purely oblivious, so a packet is always re-routed in case the computed output direction is occupied.

On the other hand, some NoC configurations can lead to dynamic deadlocks.

Currently there are deadlock-free algorithms for irregular meshes but they mostly cover holes in a mesh structure, or add stringent limitations on the possible architectures [43], [44], and [45].

Dynamic deadlocks can come from two different reasons:

- Network on Chip structures that are not suitable for the algorithm;
- Some initialization words can not reach all the routers in order.

For what concerns the first cause of deadlock, the following example can explain the situation. In picture 4.15, a deadlock-prone structure is presented. There two parts of the structure are not very well connected so some packets can't find their way from a region of the NoC to the other.
The algorithm is developed so that the words move west and east until they find a path north/south. In this case some routers are impossible to connect,

like in the example of figure 4.16.
On the other hand the algorithm allows for some configurations to be viable even if the west/east research doesn't produce a viable output (figure 4.17).

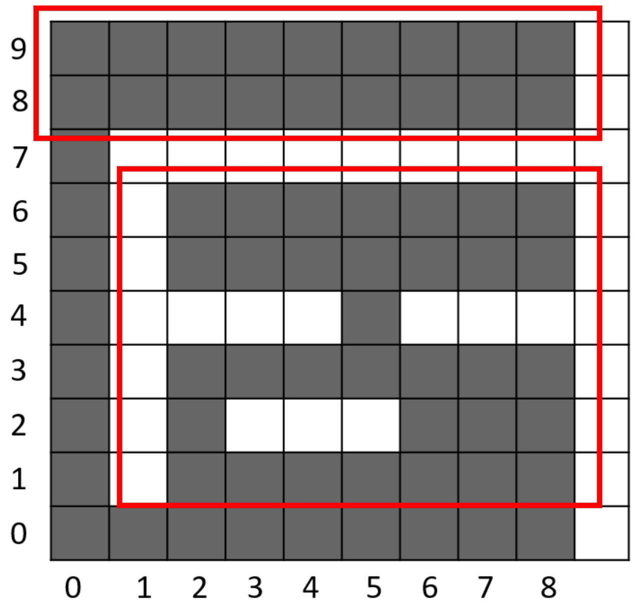


Figure 4.15: This configuration leads to deadlock because the structure is divided into two macroareas that don't have a good north/south connection

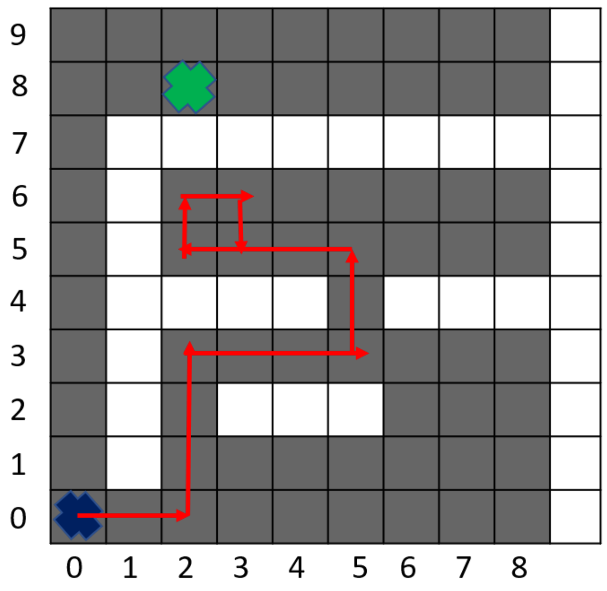


Figure 4.16: Going from router (0,0) to router (8,2) creates a static deadlock because row 7 is not available when moving east/west in row 6

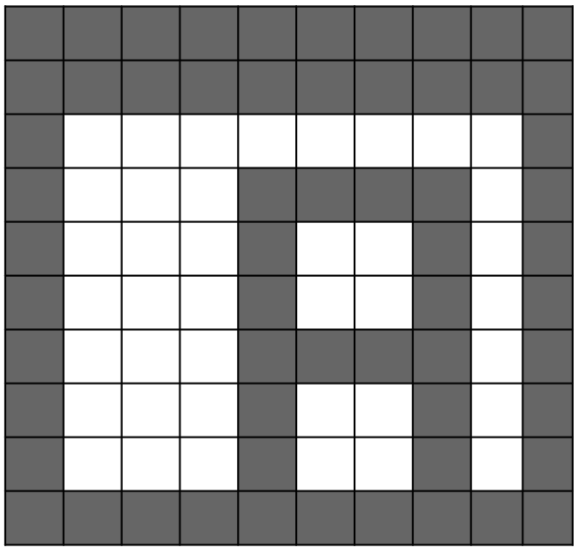


Figure 4.17: This configuration is allowed thanks to the oblivious/adaptive part of the algorithm

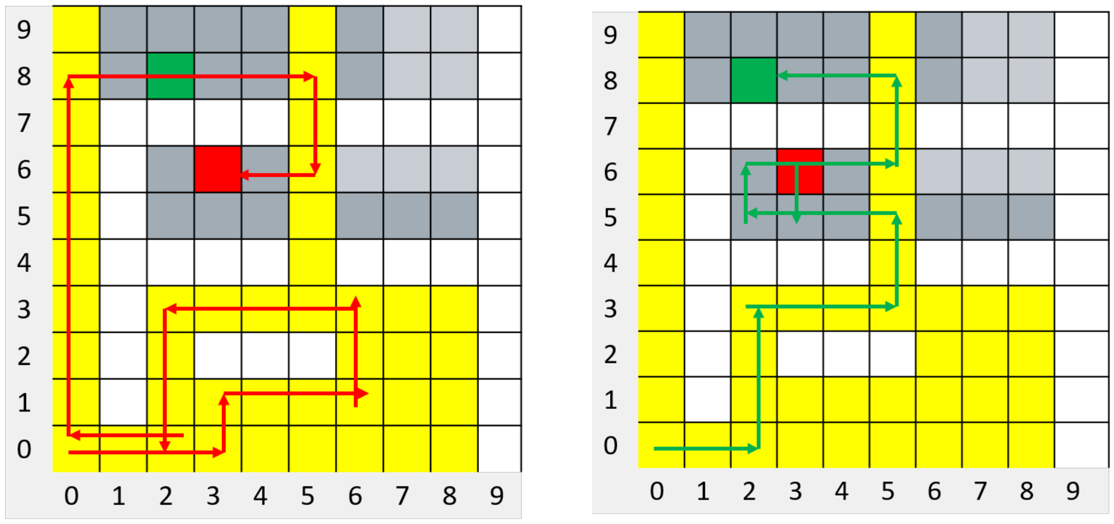


Figure 4.18: This configuration solves one of the problems of dynamic deadlock but initialization words can not travel correctly at start-up. In both configurations it is supposed that the yellow squares are the routers already initialized, while the grey ones are the routers left to initialize (as well as the red and green one). In 4.18a the route to initialize the red router (6,3) is drawn, and it goes over the green router (8,2). In 4.18b the opposite situation is pictured, and the router in (6,3) is in the way.

The second source of deadlock is given by the initialization words.
In order for the NoC to function correctly, each router needs to receive an initialization word before forwarding other messages. In the configuration highlighted in 4.18 it can be seen how different routers take different routes to arrive to their destinations. The potential problem highlighted there is that a router may need another one initialized in order to close the communication link and vice-versa, creating an impasse.

Another problem related to initialization words is the timing of them.

The developed algorithm is thought so that if two words are sent from router A to router B, they arrive in the order they have been sent. The same does not apply if two words are sent from router A to two different routers. Two things can be done in order to preserve timing (Both ways can be viable and automated):

- Creation of a model of the NoC to understand what is the critical path in term of clock cycles (e.g. the longest latency that exists in the structure);
- Simulate an initialization of the NoC in order to fix possible impasses;

Usually in order to avoid deadlock in mesh-based NoCs, a kind of turn is forbidden (For example it's never allowed for an input coming from the North to be directed to the West).
This approach is not viable in non-regular meshes like the ones in the SiLago Global NoC because sometimes a kind of turn is the only one available for the router, like in figure 4.11.

4.4.1 Deadlock solutions

Taking into account the randomness of the SiLago Global NoC structure, a deadlock-free algorithm was not implementable. What the current iteration of the SiLago Global NoC algorithm, it is allowed to create a wide variety of irregular mesh structures. The deadlock component could be eliminated by means of a check in the SiLago Physical Design Platform described in 2. In this way it would be possible to select an acceptable solution depending on the chip that is being developed.

As a final note, current iterations of SiLago structures do not utilize very complicated paths. A picture of a possible instance was presented at the beginning of the chapter 4.1, and that does not present any deadlock concern.

This work on the SiLago Global NoC tried to adapt the structure to possible future developments, so that the NoC algorithm could sustain future improvements of the structures size.
The next chapter will be the last of this thesis and will concentrate on the synthesis results.

Chapter 5

Conclusion

In this chapter information about simulation and synthesis will be given as well as their results. The last section will finally suggest some future improvements for the entire structure.

5.1 Simulation

The simulation tool used was QuestaSim by MentorGraphics. The simulation structure is described in figure 5.1. The whole simulation is done using a .do script able to run in QuestaSim. The C++ program is used to initialize the PacketSwitch routers and test different configurations of CircuitSwitch communications. Most of the simulation time is used by the processor to initialize. For all the simulation done the actual running time of the code took about 3% of the simulation time.

5.2 Logic Synthesis

Logic synthesis was done using the tool Genus by Cadence and scripts written in TCL language. The logic synthesis done on the RISCV processor and on the Global NoC router was done using very similar processes:

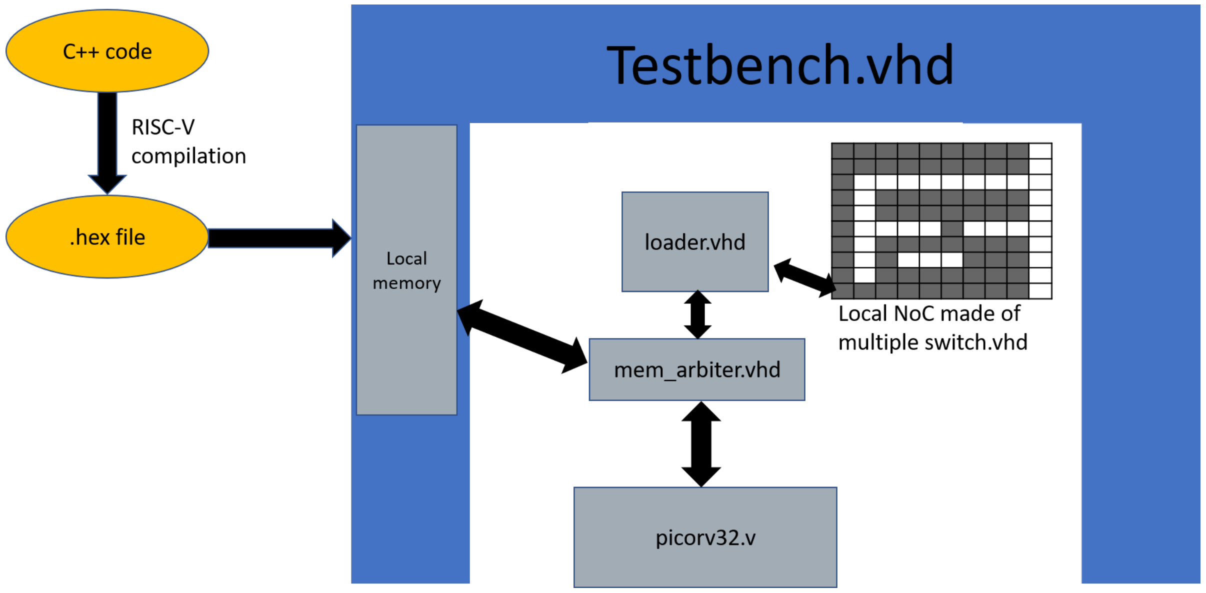


Figure 5.1: Simulation environment for the NoC+RISCV system. On the left the program to run is turned into an .hex file. It is then fed to the local memory which is initialized as a variable inside a testbench. Inside the testbench the processor and the NoC is initialized.

- Use of the TSMC 28 nm library;

- Clock gating with latch option on ”clock_gating_style”;

5.2.1 RISCV processor

The synthesis done on the PicoRV32 processor and the network interface produced a minimal clock period of 1.962 ns with a cell count of 7430. The table 5.1 gives some combination of clock period and area as reported by the tool used.

clock period (ns)	cell count	area
1.962	7430	7047.180
3.5	6338	5976.810
5	6325	5943.924
10	5799	5810.994

Table 5.1: Results on area given by Cadence Genus with different clock periods

5.2.2 SiLago Global NoC router

The synthesis was done on a single router which in itself has both a Packet Switch and a Circuit Switch.
Because of the 2-D nature of the structure, and the structure itself being a mesh, it was necessary to synthesize five different instances of the router. Each has a missing link because in a 2-D mesh each router can only have four in/out channels.
In the table 5.2 a report on the five different instances with a fixed clock contraint of 5 ns is given. Each instance is labeled as the direction it doesn't present in its structure. So for example the component noSouth has the North, East, West directions, as well as the Resource one (That substitutes the southern input/output), as pictured in 5.2.

On table 5.3 the result on synthesis with the highest achievable frequency is given for all the instances.

router	clock period (ns)	cell count	area
noNorth	3.619	2014	1513.764
noEast	3.627	2022	1529.640
noSouth	3.618	1986	1494.486
noWest	3.630	2027	1513.134
noResource	3.629	2041	1517.418

Table 5.2: Results on area and clock cycle on five different instances of the SiLago Global NoC router given by Cadence Genus with a fixed constraint on clock cycle (5 ns)

router	clock period (ns)	frequency (MHz)	cell count	area
noNorth	1.521	657.5	3503	2930.130
noEast	1.645	607.9	3702	3049.074
noSouth	1.623	616.1	3503	3120.264
noWest	1.603	623.8	3507	2985.444
noResource	1.546	646.8	3614	3196.998

Table 5.3: Results on area and clock cycle on five different instances of the SiLago Global NoC router given by Cadence Genus for highest achievable frequency

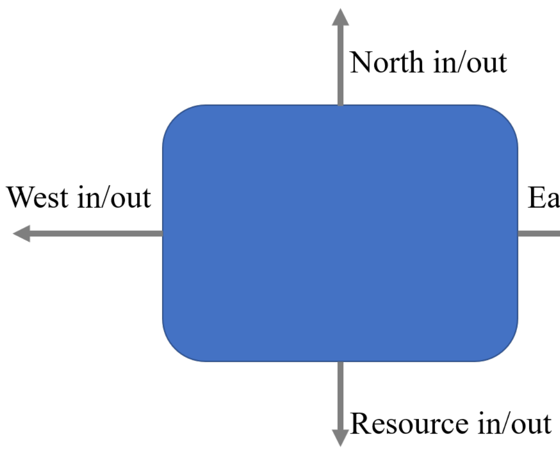


Figure 5.2: Representation of the noSouth instance of the Global Switch for the SiLago NoC. Instead of the South channel, the Resource one is inserted, in order to abide with the 2-D mesh structure

5.3 Physical synthesis

The physical synthesis was done with the tool Innovus by Cadence. The physical synthesis was done in this order:

1. Floorplan with an utilization of the area of around 70%;
2. Power and ground routing to add the rings of power supply and horizontal stripes;
3. Special route with vertical power supply lines;
4. First placement of standard cells;
5. Pin placement on the border of the chip;
6. Optimization of placement;
7. Clock tree-synthesis with optimization;
8. Routing between standard cells and optimization;

The floorplan, because of the TSMC 28 nm library size, is a square with a side of a multiple of 6.3 µm.
This is due to the horizontal and vertical pitches of the transistors, respectively 0.14 and 0.9 micron. Since the two values are different, the least common multiple has to be taken to ensure integer ratios for every square (and thus synchroricity). The power ring width is of 2 micron, with a spacing between two rings of 1 micron, and 0.5 micron offset from cell border. An overview of the cell floorplan, taken from Innovus, is presented in 5.3. For the pins, the ones put on the lateral sides are on metals M3 and M5, while the top and bottom sides use metals M2, M4, and M6.

The netlist used for all the designs are the ones synthesized with a clock period of 5 ns.

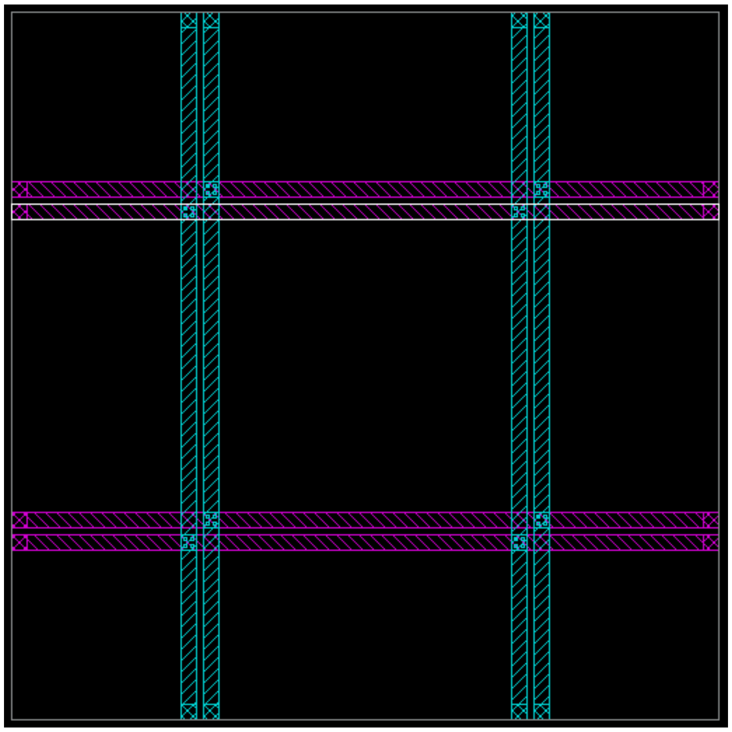


Figure 5.3: Floorplan for the RISCV processor. The structure is a square with a side of $K \cdot 6.3\,\mu\text{m}$. The structure can be divided in four sections with the same area.

5.3.1 RISCV processor

For the RISCV processor the utilization is at 70.4% and the square side is

$$94.5\,\mu\text{m} = 6.3\,\mu\text{m} \cdot 15$$

A screen dump taken from Innovus is presented in 5.4. On the right side of the chip, the interface with the router is put, while on the top the RISCV connects with the memory. The clock and the reset pins are put on the top left. The pins position are described on table 5.4

pins	metal layer	width(μm)/depth(μm)	side	spacing (μm)	spread type
Memory input	M4	0.05/0.34	top	0.60	from starting point
Memory Output	M2	0.05/0.34	top	0.14	from starting point
Packet Switch	M3	0.05/0.34	right	0.75	from starting point
Circuit Switch	M5	0.05/0.34	right	0.75	from starting point

Table 5.4: Pins position within the RISCV chip

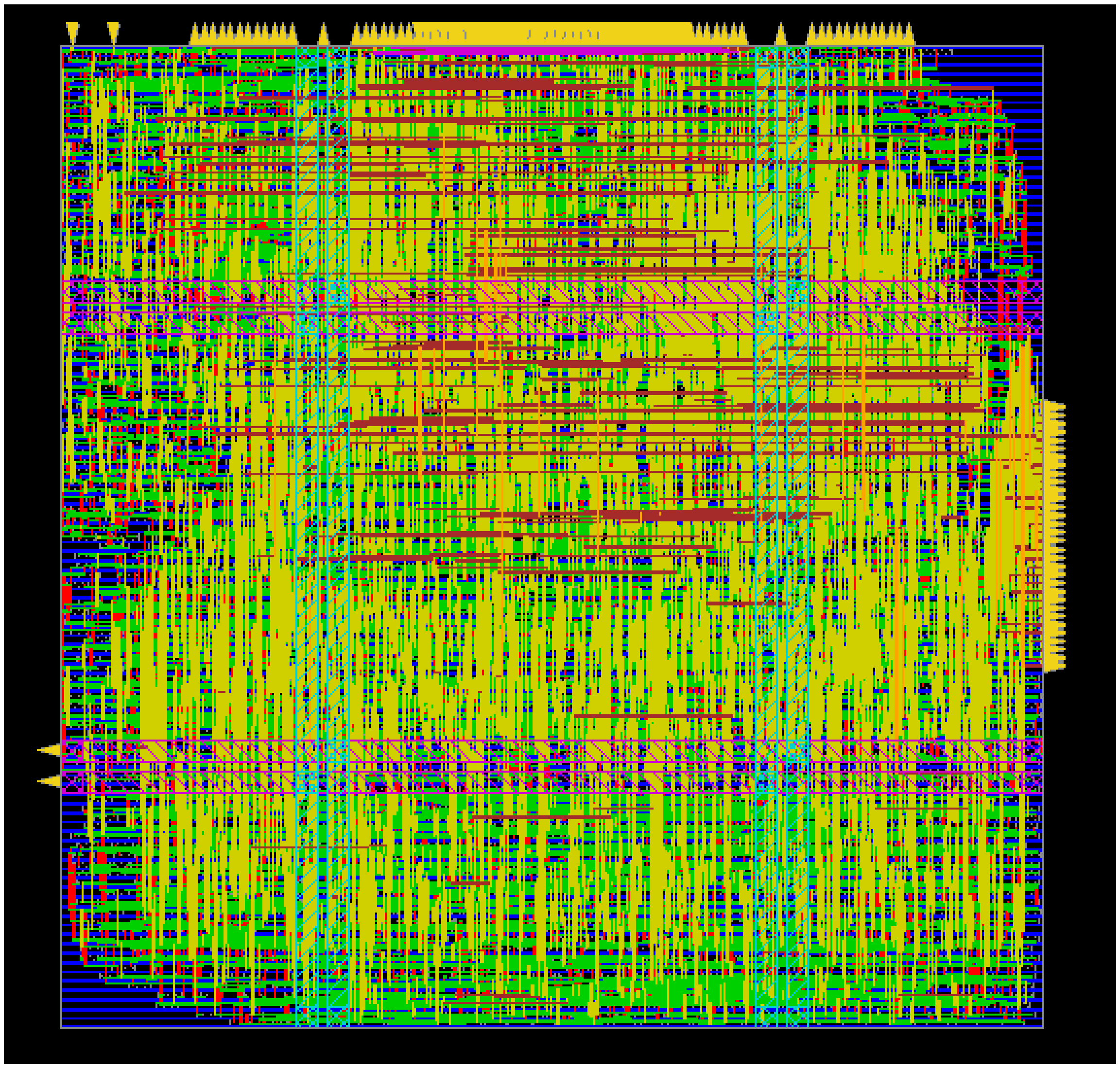


Figure 5.4: Screen dump at the end of physical synthesis on the RISCV component. The interface with the memory as well as the clock and reset inputs are on the top, on the right side the chip connects with the network on chip

5.3.2 SiLago Global NoC router

The five instances of SiLago Global NoC routers all have an utilization of under 65%. The square side is the same for all of them and it is

$$50.4\,\mu\text{m} = 6.3\,\mu\text{m} \cdot 8$$

The overview of the chip can be seen in figure 5.5, while the pins information can be found in table 5.5

pins	metal layer	width(μm)/depth(μm)	spacing (μm)	spread type
Packet Switch horizontal	M3	0.05/0.34	0.75	from starting point
Circuit Switch horizontal	M5	0.05/0.34	0.75	from starting point
Packet Switch vertical	M2	0.05/0.34	0.75	from starting point
Circuit Switch vertical	M4	0.05/0.34	0.75	from starting point

Table 5.5: Pins position within the instances of a SiLago Global NoC router. The pins are indicated as vertical (top or bottom side) or horizontal (left or right side) because each instance has a different placement of the Resource in/out pins

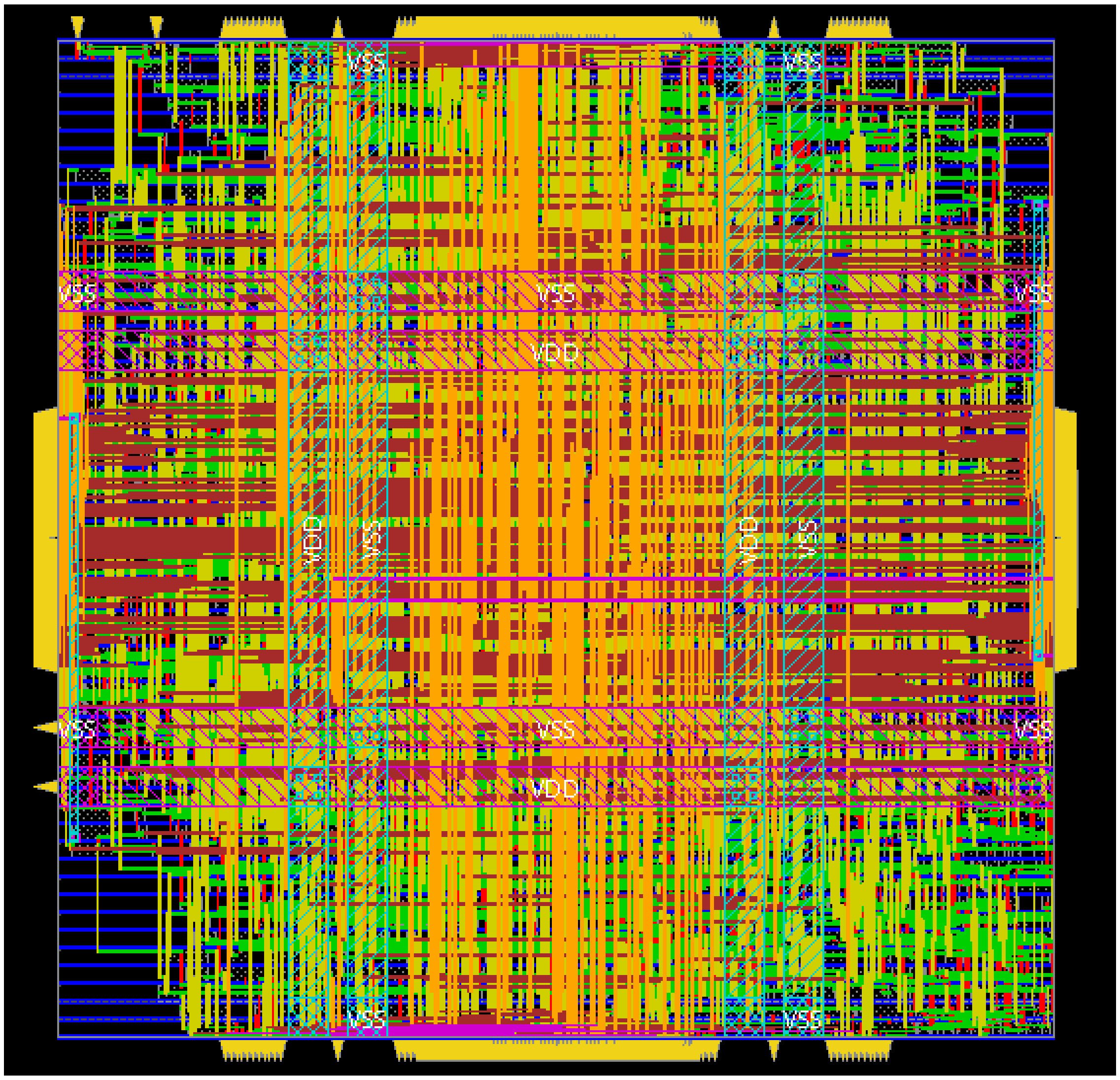


Figure 5.5: Screen dump at the end of physical synthesis of an instance of the SiLago Global NoC router. Each side has pins inserted with the same configuration.

5.4 Future improvements

The objective of the thesis was to use a RISCV controller in order to manage the SiLago Coarse Grain Reconfiguarable Architecture (CGRS) and the connection between the different parts of the chip on a global scale.

As of the end of this thesis work, a connection between a RISCV processor and the SiLago Global NoC exists. On top of that a more powerful algorithm of the Global NoC has been developed without changing the already existing protocol, so that it can be used also on future improvements of SiLago size-wise.
The new architectures have been tested twice (post behavioural design and post logic synthesis) and the design has been synthesized down to physical synthesis, so that a .gds file is available for all the instances described in this chapter.

On the software side, programs can be run using a VHDL/Verilog simulator and a RISCV-32I (32 bits on the basic instruction set) compiler and are able to send commands in order to: initialize the NoC routers and create dynamic links between routers using the packet switch, and send data through the circuit switches.

Among the possible improvements for the structure, a high level language model for the algorithm described in 4 would be necessary in order to identify possible NoC structures that are not deadlock free. On the hardware side a characterization of the designed components is missing as well as a verification post physical layout.

Bibliography

[1] L. C. Carver Mead, *Introduction to Vlsi Systems.* Addison-Wesley Pub (Sd), 1979.

[2] C. Sechen and A. Sangiovanni-Vincentelli, "Timberwolf3. 2: a new standard cell placement and global routing package," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 432–439, IEEE Press, 1986.

[3] A. Kessler and A. Ganesan, "Standard cell vlsi design: A tutorial," *IEEE Circuits and Devices Magazine*, vol. 1, no. 1, pp. 17–34, 1985.

[4] A. H. A. J. H. T. e. Amir M. Rahmani, Pasi Liljeberg, *The Dark Side of Silicon: Energy Efficient Computing in the Dark Silicon Era.* Springer International Publishing, 2017. Chapter 3.

[5] R. Merritt, "FPGAs add comms cores amid asic debate." EETimes, Accessed: 27-06-2019.

[6] S. Borkar, "Design perspectives on 22nm cmos and beyond," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 93–94, IEEE, 2009.

[7] W. J. Dally, C. Malachowsky, and S. W. Keckler, "21st century digital design tools," in *Proceedings of the 50th Annual Design Automation Conference*, p. 94, ACM, 2013.

[8] W. Stallings, *Computer Organization and Architecture.* Prentice Hall, 6 ed., 2002.

[9] Maynard, *Vax-11-780 Architecture Handbook.* MA: Digital Equipment Corp, 1977. Print., 1977.

[10] H. Levy and R. E. (Auth.), *Computer Programming and Architecture. The VAX.* Elsevier Inc, Digital Press, 2 ed., 1988.

[11] B. B. Brey, *The Intel Microprocessors: 8086 8088, 80186 80188, 80286, 80386, 80486 - Pentium and Pentium Processor - Architecture, Programming and Interfacing (Prentice Hall International Editions).*

[12] T. Jamil, "Risc versus cisc," *Ieee Potentials*, vol. 14, no. 3, pp. 13–16, 1995.

[13] V. G. O. et al., *The Computer Engineering Handbook.* CRC, 2002.

[14] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *SIGARCH Comput. Archit. News*, vol. 8, pp. 25–33, Oct. 1980.

[15] W. G. Alexander and D. B. Wortman, "Static and dynamic characteristics of xpl programs," *Computer*, no. 11, pp. 41–46, 1975.

[16] L. J. Shustek, *Analysis and Performance of Computer Instruction Sets.* PhD thesis, Stanford, CA, USA, 1978. AAI7814208.

[17] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[18] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surv.*, vol. 38, June 2006.

[19] "International technology roadmap for semiconductors 2.0," 2015.

[20] C. Batten, "An exa-op data center at ¡10MW by 2020? too many ops and not enough energy," *IAP Workshop*, 2013.

[21] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the" new normal"'for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[22] G. Budd and G. Milne, "Arm7100-a high-integration, low-power microcontroller for pda applications," in *COMPCON'96. Technologies for the Information Superhighway Digest of Papers*, pp. 182–187, IEEE, 1996.

[23] A. Boxer, "Where buses cannot go," *IEEE Spectrum*, vol. 32, no. 2, pp. 41–45, 1995.

[24] *OpenSPARC T2 CoreMicroarchitecture Specification.* Sun Microsystems, 1 ed., 2007.

[25] U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill, "Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 6–20, 2008.

[26] A. Allan, D. Edenfeld, W. H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, "2001 technology roadmap for semiconductors," *Computer*, vol. 35, no. 1, pp. 42–53, 2002.

[27] N. Farahini, A. Hemani, H. Sohofi, and S. Li, "Physical design aware system level synthesis of hardware," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 141–148, IEEE, 2015.

[28] A. Hemani, S. M. A. H. Jafri, and S. Masoumian, "Synchoricity and nocs could make billion gate custom hardware centric socs affordable,"

in *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, p. 8, ACM, 2017.

[29] S. M. A. H. Jafri, N. Farahini, and A. Hemani, "Silago-cog: Coarse-grained grid-based design for near tape-out power estimation accuracy at high level," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 25–31, IEEE, 2017.

[30] D. G. Chinnery and K. Keutzer, "Closing the gap between asic and custom: An asic perspective," in *dac*, pp. 637–642, Citeseer, 2000.

[31] M. A. Tajammul, M. A. Shami, and A. Hemani, "Segmented bus based path setup scheme for a distributed memory architecture," in *2012 IEEE 6th International Symposium on Embedded Multicore SoCs*, pp. 67–74, IEEE, 2012.

[32] O. Malik, A. Hemani, and M. A. Shami, "A library development framework for a coarse grain reconfigurable architecture," in *2011 24th Internatioal Conference on VLSI Design*, pp. 153–158, IEEE, 2011.

[33] S. Li, N. Farahini, and A. Hemani, "Global control and storage synthesis for a system level synthesis approach," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 239–239, IEEE, 2013.

[34] S. Li, N. Farahini, A. Hemani, K. Rosvall, and I. Sander, "System level synthesis of hardware for dsp applications using pre-characterized function implementations," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, p. 16, IEEE Press, 2013.

[35] A. S. Waterman, *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.

[36] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.

[37] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. A. Patterson, and K. Asanovic, "The risc-v instruction set.," in *Hot Chips Symposium*, p. 1, 2013.

[38] "Risc-v foundation website https://riscv.org/." Accessed: 16-07-2019.

[39] N. E. Jerger, T. Krishna, and L.-S. Peh, *On-chip networks*, vol. 12. Morgan & Claypool Publishers, 2017.

[40] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, "Near-optimal worst-case throughput routing for two-dimensional mesh networks," in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 432–443, IEEE Computer Society, 2005.

[41] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001. section 3.4.

[42] S. Haas, *The IEEE 1355 Standard. Developments, performance and application in high energy physics.* PhD thesis, Liverpool U., 1998. page 15.

[43] Z. Shi, Y. Yang, X. Zeng, and Z. Yu, "A reconfigurable and deadlock-free routing algorithm for 2d mesh network-on-chip," in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, pp. 2934–2937, IEEE, 2011.

[44] M. K. Schafer, T. Hollstein, H. Zimmer, and M. Glesner, "Deadlock-free routing and component placement for irregular mesh-based networks-on-chip," in *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pp. 238–245, IEEE Computer Society, 2005.

[45] V. Janfaza and E. Baharlouei, “A new fault-tolerant deadlock-free fully adaptive routing in noc,” in *2017 IEEE East-West Design & Test Symposium (EWDTS)*, pp. 1–6, IEEE, 2017.