Politecnico di Torino

Politecnico di Torino

ELECTRONIC ENGINEERING

A.a. 2024/2025

Sessione di laurea Dicembre 2025

An exploration on connectivity and efficiency in Coarse-Grain Reconfigurable Architectures

Relatori

Prof Maurizio Martina

Dott Luigi Giuffrida

Candidato

Mattia Cozzolino

Indice

Elenco delle figure

Capitolo 1

Introduction and Context

In recent years, reconfigurability and parallelism have taken a central role in the evolution of modern computing architectures. This trend is driven by the need to keep power consumption under control while still maintaining high computational performance, pushing both the scientific and industrial communities to seriously consider hardware solutions that are specialized yet flexible. Coarse-Grained Reconfigurable Architectures (CGRAs) emerge in this scenario as an increasingly relevant research area because they offer a very attractive compromise: they combine ASIC-like performance with the flexibility typically associated with FPGAs.

A CGRA can be described as a matrix of processing units, the Processing Elements (PEs). PEs are programmable units in terms of arithmetic operations and interconnect configuration, allowing communication with nearby elements through configurable links. Thanks to this feature, each CGRA can be programmed for different applications, exploiting data and operation-level parallelism as much as possible. The main difference that distinguishes CGRAs from FPGAs lies in the granularity of the operations processed by the two systems. CGRAs operate on larger "coarse-grained" blocks and work at a higher abstraction level compared to FPGAs. This means that they do not implement elementary logic gates, but arithmetic functions or mid-level logic blocks. This characteristic leads to significant benefits in terms of area, operating frequency, and power consumption.

Since high performance and reconfigurability have become equally important in many domains, the attention has shifted strongly toward CGRAs. Their flexibility allows them to address different requirements across different applications without needing to redesign the hardware each time. In practical fields such as machine learning, signal processing, artificial intelligence, and generally in systems requiring high computational capability, CGRAs now play a key role. Within these scenarios lies the present thesis work, whose main goal is the design of a fully parametric CGRA, capable of adapting to different configurations by allowing the user to modify both its topology and its size, while still enabling verification and synthesis.

1.1 Motivations and Objectives

The project originates from the idea of investigating the impact that different levels of interconnection among the PEs may have in terms of area and timing. The goal is to build a CGRA architecture reconfigurable at the RTL level, allowing the user to parameterize the size of the matrix ($N \times N$) and the interconnection topology among the Processing Elements during computation. Thanks to this implementation, it becomes possible to experimentally observe how the degree of connectivity affects the key design parameters.

In this project, unlike other simpler implementations, the aim is not only to simulate the behavior of the various processing units connected in a certain way, but to operate by creating a truly functioning system, capable of receiving input data and providing output data, so that it can be connected, if necessary, to other external modules without having repercussions on the overall operation.

The way in which data is transferred to the matrix that will then process them is through a DMA (Direct Memory Access) module that uses an AXI-Stream module. Through a double-buffer mechanism with a logic called ping-pong, this DMA is always in contact with another module called Frame Loader, which is responsible for managing the frames and sending them correctly to the matrix.

The system is indeed reconfigurable through a series of RTL parameters that, together with dedicated logic, determine which sources can be used as input by the various PEs.

The topologies implemented in this project are four:

- **FULL**: each PE can communicate with all the others in the matrix;
- **MESH4**: each PE communicates with the four neighbors (north, south, east, west);
- **D-MESH**: communication extends to the eight surrounding neighbors, including the diagonals;
- **D-TORUS**: similar to D-MESH but with wrap-around connections on the borders.

1.2 Technological Context

The limitations of traditional solutions such as ASICs and FPGAs have inevitably shifted the focus toward reconfigurable architectures. ASICs offer excellent performance and low power consumption, but they are completely static in terms of functionality: once a chip is manufactured, changing its behavior requires designing a new one. FPGAs, on the other hand, offer much greater reconfigurability but with limitations in area and operating frequency. CGRAs lie exactly in between these two extremes, offering a solid compromise. In CGRAs, each PE corresponds to a functional logic block rather than a LUT, which allows for higher operating frequencies and reduced routing complexity.

Another fundamental aspect of the entire project is certainly the comparison of the various topologies. It is important to underline how the different topologies are all generated from the same RTL with slight modifications, yet still within the same code. These are also synthesized following the exact same flow, so thanks to these small details it is possible to perform an effective comparative analysis and see what the real impact of the topology is on the design.

1.3 Design Approach

The entire project has been written in SystemVerilog, a language that ensures clarity in the descriptions and facilitates a modular and parametric structure. For testing and verification, Verilator has been used together with a C++ testbench. Inside this testbench, an automated flow was implemented, capable of generating stimuli, handling data loading through the DMA, running the required computations, and producing CSV files containing the information needed for final verification.

Entering the core of the project, the Processing Elements operate independently and execute 32-bit arithmetic operations such as addition, subtraction, and multiplication. Their execution is synchronized through validation signals. A separate logic block is responsible for constructing the matrix based on the selected topology and size, generating a real interconnected structure in which the links between PEs correspond exactly to the configuration chosen by the user.

A module of utmost importance is certainly the DMA (Direct Memory Access), which is responsible for retrieving data from a memory and "transporting" them —in this case toward another module—through the AXI-Stream protocol. What the DMA does, in short, is read data from memory and send them together with the control signals `tvalid`, `tready`, and `tlast`, which are signals that will then be interpreted by another module and help determine the validity of the data as well as their termination, without needing external triggers for this operation.

The Frame Loader is based on a ping-pong double buffer. It receives the packets sent by the DMA and places them into two memory banks, `NEXT` and `ACTIVE`. When the `NEXT` bank becomes full, a swap signal allows the data in `NEXT` to move into `ACTIVE`, enabling the DMA to immediately refill the `NEXT` bank. This mechanism maintains a constant throughput and prevents unnecessary stalls in the system.

The parametric PE matrix is the central part of the project. Each PE receives two inputs (A and B), which can be selected from a list that includes the two values provided by the DMA (`GEN0` and `GEN1`) as well as the outputs of other PEs. The number of input sources available to each PE depends on the chosen topology. The executed operation is defined by the opcode received in that cycle, and the result is produced within a single cycle and synchronized through validation signals. The chosen topology significantly affects the system, as it changes the number of available sources, and therefore the size of the multiplexer and its selection signal. This has a direct impact on both area and timing.

All the modules described above are connected together in the top-level file `cgra_single_top.sv`. The project is fully parametric, and the values `N` and `TOPOLOGY` can be modified through parameters passed either to the simulator or to the synthesis tool (`-GN` and `-GTOPOLOGY` for Verilator, or environment variables for Design Compiler).

1.4 Results

For synthesis, Synopsys Design Compiler was used. Through standard `gtech` libraries, the tool produced results consistent with professional synthesis flows. In order to carry out an objective analysis of the obtained results, each topology was synthesized in fully equivalent environments with identical timing constraints. Once the reports were generated, an analysis of the fundamental parameters was performed to evaluate how the different topologies influenced multiplexer complexity and the length of the critical paths.

A crucial point of the entire work is the comparison among the four topologies. This comparison makes it possible to understand how internal connections, and their increasing complexity, affect the overall design. The results clearly show that a topology with very dense connectivity, such as `FULL`, provides maximum flexibility in the distribution of information, but at the cost of a significant increase in area and delay. Conversely, topologies with fewer connections, such as `D-MESH` and `MESH4`, exhibit lower latency and smaller area usage, although they introduce limitations in terms of data distribution. Positioned between these extremes is the `D-TORUS` topology, which represents a true compromise, improving data distribution by periodically adding wrap-around connections at the edges while keeping area costs under control.

From this analysis emerges the main objective of the study: identifying a balance between physical cost and flexibility, highlighting the importance of topology selection when designing scalable CGRAs. The work presented in this thesis aims to build a connection between academic rigor and practical hardware design. The possibility to vary the interconnection topology dynamically offers a controlled way to study how the internal structure influences communication. The approach adopted in this project is fully based on RTL descriptions and real synthesis flows, enabling the extraction of reliable and consistent results and laying the groundwork for future developments aimed at automatic optimization of connectivity.

Capitolo 2

Background and State of the Art

2.1 Introduction to CGRA Architectures

Coarse-Grained Reconfigurable Arrays (CGRAs) have gradually consolidated their role as one of the most promising classes of programmable accelerators. Their appeal stems from the ability to combine flexibility and specialization in a balanced manner—a characteristic that becomes increasingly crucial in modern computing platforms. A detailed and influential overview of the field is provided by Podobas et al. [1], who trace the evolution of CGRAs from early DSP-oriented designs to the highly heterogenous and application-driven solutions seen today.

What distinguishes CGRAs from more traditional reconfigurable technologies, such as FPGAs, is the level of abstraction at which programmability is exposed. Instead of manipulating single bits or LUTs, CGRAs operate on coarse computational blocks, typically *Processing Elements* (PEs). Each PE includes an ALU, small registers, and lightweight interconnect interfaces that allow local communication with neighbouring nodes. This coarse granularity leads to dramatically lower configuration latencies and much more predictable execution patterns, both of which are essential in embedded and edge scenarios where energy and area budgets are tightly constrained.

From an architectural perspective, CGRAs were originally conceived to accelerate structured numerical kernels such as matrix multiplications, filtering operations, and finite impulse response pipelines. These workloads align well with the spatial execution model in which data dependences remain stable and regular. Over time, however, the interest in CGRAs expanded significantly, driven by three converging trends: the stagnation of general-purpose CPU scaling, the proliferation of data-intensive tasks at the edge (e.g., computer vision, tinyML, multi-sensor fusion),

and the desire to deploy accelerator-like performance in environments where FPGA devices may be too costly or power-hungry.

One of the central strengths of CGRAs lies in their communication locality. Many computations involve short-range data dependences that can be satisfied directly within the fabric, without relying on multi-level caches or complex coherence mechanisms. Architectures such as OpenEdgeCGRA [2] explicitly leverage this principle: adjacent PEs can exchange data with negligible overhead, reducing pressure on external memory and enabling sustained streaming execution. This locality-driven approach is particularly advantageous in embedded systems, where the gap between on-chip computation and memory access latency is often large.

In summary, CGRAs occupy a unique position in the design space of programmable accelerators: more flexible than ASICs, significantly more energy-efficient than generic processors, and far easier to reconfigure than fine-grained FPGA fabrics. This combination makes them a compelling candidate for next-generation edge-computing platforms, especially when predictability, efficiency, and scalability are equally important.

A comprehensive and widely cited overview of this architectural family is provided by Podobas et al. [1], who classify CGRAs based on granularity, reconfigurability model, and interconnect structure. Their analysis highlights how CGRAs evolved from early DSP-oriented accelerators toward more heterogeneous and compiler-driven systems, setting the conceptual foundations for much of the modern research discussed in this chapter.

2.2 Computation Models and CGRA Classification

The diversity of applications that rely on CGRAs has fostered a correspondingly diverse set of execution models. One well-established distinction is between *spatial* and *spatio–temporal* architectures.

Spatial CGRAs maintain a fixed mapping of operations onto PEs for the entire duration of a kernel. Each PE performs a single arithmetic or logical function, and the whole dataflow graph (DFG) is laid out statically across the array. This execution model greatly simplifies control, minimizes instruction overhead, and allows for highly predictable performance. However, it presupposes that the DFG fits onto the available hardware resources. For kernels with many operations, deep dependency chains, or frequent branching, such direct mapping may be impractical.

Spatio–temporal architectures introduce a temporal dimension: instead of assigning exactly one operation to each PE, they allow each element to execute a short sequence of instructions across multiple cycles. This approach extends the range of kernels that can be mapped on a fixed-size array and supports more complex

behaviours. The STRELA accelerator [3] embodies this philosophy through an *elastic* execution model, where ready/valid signalling and small internal buffers help PEs synchronize dynamically and tolerate variable latencies. Elastic designs thus offer more flexibility, though they increase control complexity and area footprint.

Another major classification axis concerns the dataflow semantics. In **static dataflow**, all data dependences are determined during compilation, producing deterministic schedules. Although this simplifies analysis and optimization, it complicates the execution of kernels with data-dependent control flow or irregular memory accesses. **Dynamic dataflow** architectures, by contrast, allow tokens to carry control information at runtime. Systems such as RipTide [4] and SNAFU [5] demonstrate how dynamic dataflow can be realized even under ultra-low-power constraints. RipTide, in particular, takes advantage of the network-on-chip, delegating several control-flow primitives to routing components while keeping PEs lightweight.

A third dimension of classification arises from the role of compilation. Traditional CGRAs relied heavily on manual mapping, but recent proposals have placed much more emphasis on automated toolchains. Morpher [6] is emblematic of this shift, introducing an *Architectural Description Language* (ADL) that allows designers to specify PEs, interconnects, and scheduling policies with fine granularity. The compiler—rather than the architecture—drives exploration, enabling rapid prototyping of variants and facilitating reproducibility. This is a decisive step forward, reflecting the broader transition from monolithic CGRA designs to modular, reusable design ecosystems.

Finally, interconnect topology remains one of the most distinctive elements in CGRA classification. Meshes and tori represent the mainstream solutions due to their simplicity and scalability. However, more expressive fabrics exist: multi-hop connections in architectures related to HyCUBE, as described by Juneja et al. [6], reduce average communication distances at the cost of moderately more complex routing logic. The interconnect, more than any other component, profoundly influences the system's performance, energy consumption, and ability to sustain concurrency.

2.3 Modern Microarchitectures and Processing Element Features

The Processing Element (PE) constitutes the computational core of a CGRA, and its microarchitectural design largely determines the overall efficiency of the system. Different CGRA families emphasize different trade-offs, but many recent proposals agree on a minimalist PE structure: an ALU supporting basic arithmetic and logic

operations, a handful of registers to buffer intermediate values, multiplexed input paths, and—when temporal behaviour is allowed—a tiny instruction memory.

OpenEdgeCGRA [2] offers a clear example of this design philosophy. Each PE includes a 32-bit ALU capable of basic operations, a small local memory for storing up to a few dozen instructions, and simple routing logic that supports direct communication with neighbouring PEs in a toroidal configuration. This simplicity keeps area and power overhead minuscule, enabling the array to scale to multiple rows and columns even within modest silicon budgets.

More sophisticated PEs appear in designs that target near-sensor computing, where complex control-flow structures and nested loops are frequent. The Integrated Programmable Array (IPA) accelerator described by Das et al. [7] includes a richer set of local resources, complemented by a tightly coupled multi-bank TCDM memory subsystem. This allows the array to handle a broader class of kernels, including those featuring non-trivial conditional execution.

Elastic architectures introduce yet another form of PE complexity. In systems like STRELA [3], each PE contains ready/valid buffers that act as local synchronization points. While these additions increase hardware cost, they allow the system to tolerate latency variations caused by memory accesses or network congestion. They also relax the pressure on the global scheduler, which no longer needs to impose strict timing alignment across all nodes.

Communication among PEs, finally, plays a central role. Whereas some CGRAs employ direct neighbour-to-neighbour links, others adopt multi-hop routing or rely partly on buffers inside the interconnect. The HyCUBE architecture described in [6] explores a lightweight multi-hop model that achieves a balance between routing flexibility and hardware simplicity. As communication overhead increasingly dominates energy expenditure in modern workloads, the structure of the interconnect becomes a first-class architectural component.

2.4 Generation Frameworks and Compiler Toolchains

The growing architectural diversity of CGRAs has made the compiler a decisive part of the design process. Unlike conventional processors, where instruction scheduling and register allocation dominate the compilation flow, CGRA compilation must additionally handle spatial placement, routing, pipeline initiation intervals, memory bank selection, and buffer management. Each of these steps can profoundly influence the overall execution efficiency.

Historically, early CGRA compilers were tightly coupled to specific architectures, often relying on fixed templates and manual mapping. This approach limited scalability and prevented designers from exploring heterogeneous PEs or alternative

routing fabrics. As CGRAs became more expressive, such rigid toolchains proved increasingly inadequate.

A significant conceptual leap arrived with frameworks that embraced architectural parametrization and full-stack modularity. Morpher [6] is perhaps the most mature example of this new generation. The framework introduces an extensible *Architectural Description Language* (ADL) that allows architects to define the structure of PEs, functional units, register files, buffers, memory subsystems, and the topology of the interconnect. By separating architecture specification from compiler internals, Morpher enables rapid exploration of entire design families while preserving functional correctness and toolchain stability.

One of Morpher's strengths is its integration with LLVM, which automates the extraction of DFGs from C code. This high-level flow is particularly valuable in research contexts, where exploring multiple architectural variants is as important as pushing performance. The ADL also helps enforce consistency between hardware and software, reducing manual effort and minimizing the risk of mismatches between RTL implementation and compiler assumptions.

At the opposite end of the design spectrum lies SNAFU [5], a framework deliberately crafted for ultra-low-power environments. Instead of pursuing maximal flexibility, SNAFU is built around a simple, almost frugal philosophy: keep PEs lightweight, make network routers bufferless, and enforce strictly in-order dataflow. This approach pays off in scenarios where nanowatt-to-microwatt efficiency is the priority. Notably, SNAFU's mapping and scheduling algorithms explicitly avoid complex heuristics, ensuring deterministic compilation and minimal overhead.

Between these two extremes—full architectural agility and fully energy-optimized specialization—lies a broad middle ground occupied by a wide variety of academic and industrial frameworks. Although they differ in scope, most share three common objectives:

1. provide a structured way to describe CGRA components,
2. automate mapping and routing,
3. ensure correctness even under complex control-flow or memory behaviors.

An additional, increasingly relevant line of work concerns frameworks that integrate architecture generation with application-driven optimization. This is exemplified by systems such as DSAGEN (commonly referenced within the CGRA community), which start from a set of representative workloads and automatically synthesize a CGRA optimized for them. These designs highlight how workload-aware generation can yield significantly more efficient architectures.

All these trends underline an important fact: the rise of CGRAs has been fueled not only by advances in silicon architectures but also by the emergence of flexible, robust toolchains capable of managing the complexity of spatial computing.

Without such toolchains, even elegant CGRA architectures would struggle to find adoption beyond research prototypes.

2.5 Memory Systems, Data Movement, and DMA in Modern CGRAs

Data movement is often the hidden bottleneck in CGRA-based systems. As applications become more data-intensive and memory hierarchies grow more heterogeneous, sustaining a consistent throughput requires careful orchestration of memory access patterns, buffering, and communication paths. Thus, modern CGRAs tend to incorporate memory subsystems that minimize global memory traffic and exploit locality whenever possible.

One common strategy is the adoption of multi-bank scratchpads or TCDM (Tightly Coupled Data Memory) structures. In the IPA accelerator [7], a multi-bank TCDM is tightly integrated with the CGRA fabric, enabling several PEs to issue concurrent load/store operations while spreading requests across independent banks. This design significantly reduces contention and prevents memory hotspots, which are particularly harmful for dataflow execution where a single delayed operand can stall an entire pipeline.

Another strategy—increasingly popular in embedded and edge platforms—is the use of multiple parallel DMA engines. OpenEdgeCGRA [2] A recent evaluation by Carpentieri et al. [8] further demonstrates the practical relevance of OpenEdgeCGRA, showing that its distributed DMA structure and lightweight routing fabric enable efficient acceleration of convolutional layers. Their study confirms that even relatively small CGRA instances can sustain high throughput when the memory subsystem is carefully co-designed with the computational array.

Capitolo 3

Architecture of the Proposed CGRA

3.1 General Overview

In the proposed thesis, the core architecture is designed with a specific goal: creating a fully parametric platform in which both the size of the array and the interconnections can be modified while keeping the overall processing flow unchanged. Another central aspect of this work is not simply building an accelerator capable of performing its task, but providing a platform that allows the designer to investigate which configuration best suits specific needs. This is achieved by evaluating which interconnection topology between the Processing Elements is more appropriate for given requirements and by observing how different topologies influence the actual behavior of the digital design.

The base architecture consists of three main blocks responsible for organizing and processing the data flow. The blocks are:

- an AXI-Stream DMA that handles the propagation of input data and acts as a conveyor belt from memory to the PE matrix,
- a ping-pong frame loader that receives the data transported by the DMA and forwards them in the correct order to the matrix,
- an $N \times N$ matrix of Processing Elements with a selectable topology that processes the data and produces results.

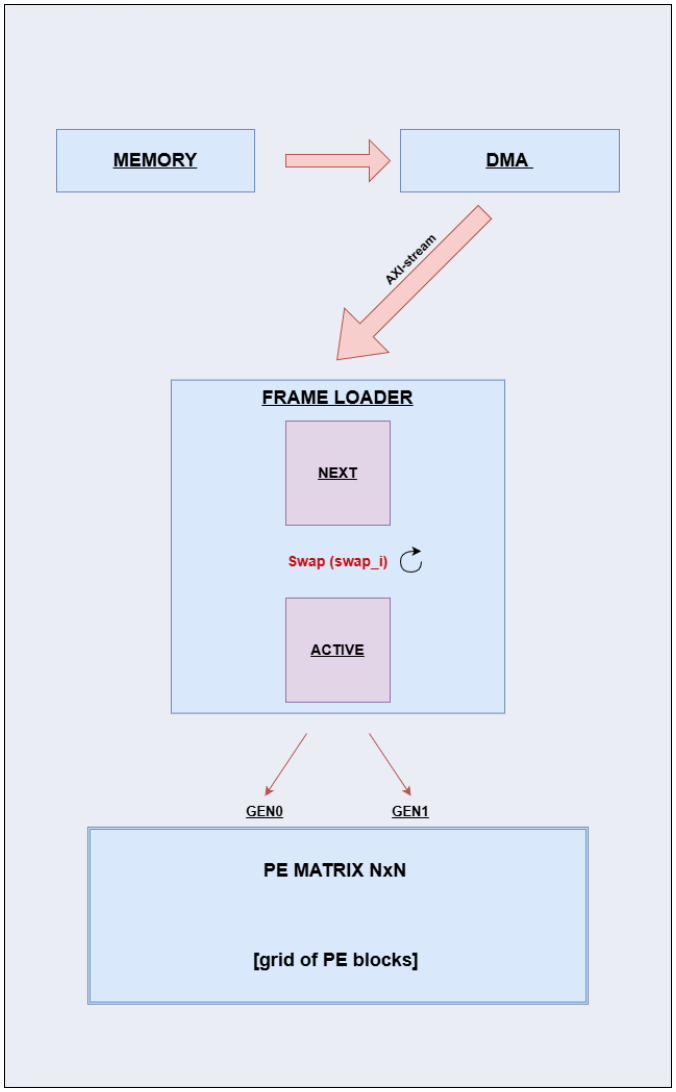

Figura 3.1: Three main components of the processing pipeline are shown in this figure. The AXI-Stream DMA, the Frame Loader and the parametric matrix of Processing Elements - PEs. These modules cooperate in receiving the input planes and reconstructing them inside the ping-pong buffers. and process them in parallel within the CGRA.

To keep the overall process as linear as possible, a pipeline has been implemented to maintain a continuous flow without interruptions. These three blocks have different roles, allowing a clear separation of responsibilities: the DMA transfers the stream, the frame loader normalizes it and transforms it into frames, and finally the PE matrix processes it according to the selected topology.

A major strength of the project lies in the ability to freely modify the topology

of the interconnections between the processing elements (PEs) without altering the behaviour of the remaining blocks. This characteristic enables reliable comparative analysis, allowing different architectural configurations to be evaluated without introducing undesired side effects.

From a functional point of view, the entire system can be modelled as a three-stage pipeline: *production*, *loading*, and *processing*. Each of these stages is characterised by distinct timing constraints and responsibilities.

The process starts at the DMA, which can be interpreted as a conveyor belt continuously sending two data packets, `GEN0` and `GEN1`, through an AXI-Stream interface. These packets are emitted at a constant rate, while the receiving module regulates the effective data transfer via the standard handshake signals `TVALID` and `TREADY`, ensuring that communication proceeds only when both ends are ready.

The receiver, namely the *frame loader*, decodes the packets transmitted by the DMA and stores them into two memory banks, `ACTIVE` and `NEXT`. The `ACTIVE` bank contains the data currently being consumed by the PE array, whereas the `NEXT` bank holds the data scheduled for processing in the following iteration.

Once the loading of `ACTIVE` into the matrix is complete, a swap mechanism moves the contents of `NEXT` into `ACTIVE` and triggers a new handshake cycle that allows the DMA to refill `NEXT`. Finally, the PE matrix processes the data stored in `ACTIVE` and generates outputs that, depending on the topology, can be used as new inputs for other PEs. The internal structure of each PE is intentionally simple, since the goal is to isolate the behavior produced by the topology itself: two input values are processed using basic arithmetic operations (addition, subtraction, multiplication) to generate an output.

The overall organization of the system is intentionally simple and highly predictable. While the processing matrix handles the current frame, the frame loader is already preparing the next one. Once the computation finishes, the *NEXT* buffer is cleared and promptly refilled as the DMA streams in new data. This continuous, stall-free flow ensures that the entire pipeline operates smoothly and, more importantly, that synthesis results remain reproducible and directly comparable across different system configurations.

3.1.1 Design Objectives

In this project, the main guiding principles were modularity, parametricity, and observability.

Modularity was achieved through the complete independence of each module. from the others. The three main modules - DMA, Frame Loader, and Matrix-do does not share internal signals that could compromise the operation of one module with respect to another. They are not generated using fixed assumptions about the array size, and such information is not required for their internal functionality. Communication happens through formal interfaces only. This permits each module that can be independently verified, analyzed, and replaced if needed, without in turn affect the whole system.

Parametricity is the guiding design principle. All modules adapt correctly to the N $\times$ N array size selected at configuration time. The matrix is generated using Produce constructs that would make it dependent on constant N set by the user. Signals such as multiplexer selectors, which determine for each PE its inputs, are will be created automatically, depending on the topology selected. The frame loader and DMA adapt dynamically: the DMA simply delivers information packets without caring about their content, while the frame loader adjusts the buffer sizes according to the selected configuration.

Finally, there is observability as another critical aim. The purpose is to assess how different topologies influence synthesis metrics. This is enabled by the low design complexity: a shallow pipeline, explicit interfaces, simple control logic and Isolated combinational paths. All these aspects make it possible to analyze results. and reach conclusions without vague or misleading behavior.

3.1.2 Hierarchical Structure of the Architecture

From an RTL point of view, the architecture is organized with a regular and fully synthesizable hierarchy. The main module, `cgra_top.sv`, is responsible for assembling and connecting the three functional blocks. Inside it we find:

- the instance of the AXI-Stream DMA, configurable according to the data width,
- the instance of the ping-pong frame loader, responsible for reconstructing the frames,
- the generation of the PE matrix through a double `genvar` loop,
- the generative module that builds the interconnections between the PEs based on the selected topology.

The top-level does not contain complex control logic: the pipeline is designed so that each module behaves as a closed box, with explicit synchronization and clear validity signals. The AXI-Stream interfaces ensure a strict separation between data transmission and data consumption, reducing the risk of unexpected stalls.

3.1.3 Data Flow and Synchronization

Data flow is deterministic in the platform, and there is a well-defined temporal sequence. The **DMA** will continue generating data as long as the frame loader asserts its `TREADY` signal. The loader writes into the **NEXT** bank until the frame is complete. At the swap signal, **ACTIVE** and **NEXT** swap their roles, and the **PE matrix** immediately starts consuming the data now stored in **ACTIVE**, propagating values according to the selected **topology**.

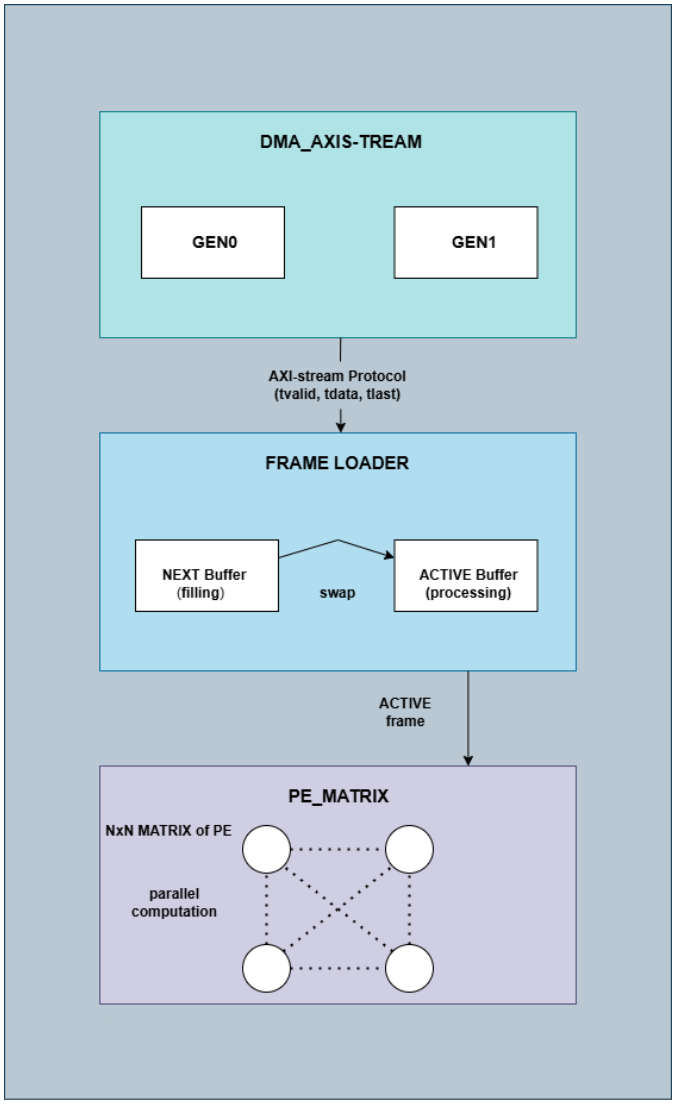

Figura 3.2: The DMA transfers the two input planes, GEN0 and GEN1, through AXI-Stream. The Frame Loader reconstructs the frame inside the NEXT buffer, while the PE Matrix processes the ACTIVE buffer. When NEXT is fully loaded and the matrix finishes the current frame, the swap signal synchronizes the ping-pong mechanism so that loading and computation keep executing continuously in parallel.

3.2 DMA AXI-Stream

Communication protocols in CGRA architectures represent a clear critical point, mainly because modern accelerators increasingly require linearity in data transmission, as the goal is to obtain a rhythmic and predictable processing flow. In the proposed thesis, the task of data transmission is assigned to a DMA, which is intended to act as a "conveyor belt" between the local memory and the logic responsible for loading the frames into the processing pipeline.

The protocol selected for the transmission of data is **AXI-Stream**. The main role of **DMA** is designed to fetch data blocks from the memory, divide them into AXI-Stream packets, which are **GEN0** and **GEN1** in this project. Whole design was conceived with the main purpose of providing for continuous data Transfer while fully conforming with **AXI-Stream** criteria both to meet modern system requirements and enable easier integration of this module with the frame loader.

Entering more into the details and trying to understand the intrinsic functioning of the DMA, one can see that it extracts "raw" data from memory and transforms them into AXI-Stream packets. For each iteration, the DMA provides two packets—`GEN0` and `GEN1`—which correspond to the two planes on which the system's start phase is based, since they contain the initial input values for each PE. The decision to adopt two physically separate packets was made to maintain simplicity and order in the design, a simplicity that allows the frame loader to associate each input with its specific value without ambiguity. Since the packets are separated, there can be no interleaving that would lead to incorrect computation due to mismatched inputs. The DMA therefore plays a crucial role in the project, as it must guarantee precise ordering and reproducibility with constant timing.

3.2.1 Internal Architecture and State Machine

The operation of the module is based on a state machine composed of four main states: `idle`, `send0`, `send1`, and `done`. In the first state, everything is deactivated—the AXI-Stream output port as well as the memory address pointer—while the module waits for a start command. Once the start command is received, it transitions to the next state, where the two base addresses for the planes to be transferred are stored, along with the number of words that will form the two planes.

During the `send0` state, the DMA reads the internal memory starting from the specified address and continues sequentially; once a word is extracted, it is placed on the `tdata` signal (AXI-Stream bus), and immediately after, the `tvalid` signal is asserted. The exchange also depends on the availability of the receiver: only when `tready` is active do the two protocols perform the handshake. Otherwise, the DMA blocks the entire flow, keeping the `tdata` value frozen to avoid unnecessary memory reads and preventing undesired interruptions.

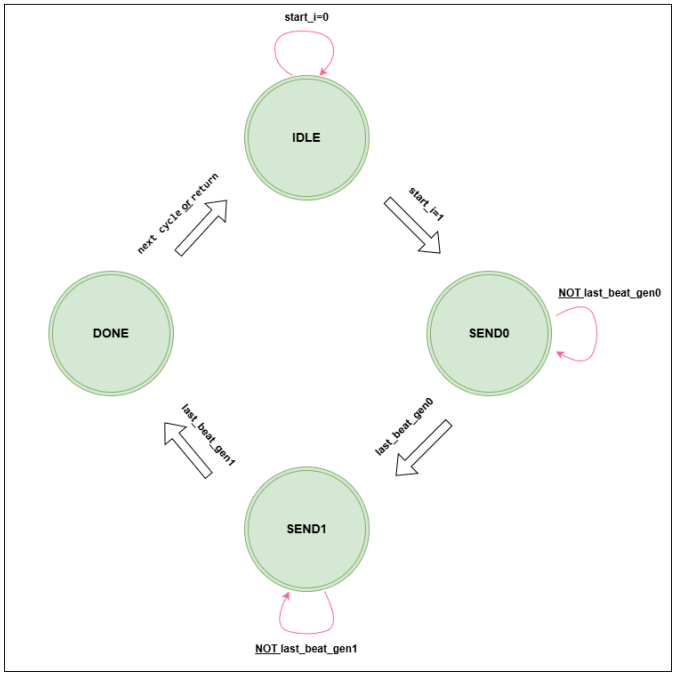

Figura 3.3: The DMA sequences the transmission of GEN0 and GEN1 over AXI-Stream through the IDLE, SEND0, SEND1, and DONE states..

3.2.2 AXI-Stream Protocol Handling

This state machine design was necessary for several reasons: primarily, to maintain coherence with the AXI-Stream protocol itself, but even more importantly, to allow the frame loader to manage its internal operation in a simple and rhythmic way, without having to deal with potential data loss or overruns.

A key element of AXI-Stream is the `tlast` signal, which indicates when the last word of the packet has been reached. In the DMA designed for this project, `tlast` is generated using a counter that counts the internal beats and marks the last position. This signal is extremely useful, especially for the receiver, which thanks to `tlast` can easily identify the end of the `GEN0` and `GEN1` packets.

The main reason for this mechanism lies in the frame loader buffer: without a signal like `tlast`, the buffer could be filled with values that do not belong to the same packet, creating overlaps and processing errors. Thanks to this protocol, this allows the frame loader to correctly fill its buffer and ensure that **GEN0** and **GEN1** packets without overlap.

Once the transmission of the first packet is completed, the DMA moves to the `send1` state, which replicates the logic already seen for sending `GEN0`, except for the memory pointer. Moreover, the data transmission occurs immediately and sequentially, following the same logical rules and guaranteeing ordered and coherent modularity.

When the packet transmission is finished, the module enters the `done` state, where it generates a pulse on the `done_o` signal, allowing the upper-level logic to understand that the transfer is complete. In this project, this signal is mainly used by the testbench, which can then send either the swap command to the frame loader or a new start for an additional data transfer.

3.2.3 Internal Memory

A brief analysis must also be made regarding the DMA's internal memory: this is not a real memory, but rather it is implemented in the simulation environment as a register vector. It was designed this way to verify correct DMA behavior in a realistic situation, but it is not a distinct module; instead, it is directly integrated within the DMA, which therefore does not include any communication protocol toward the outside—an element that could certainly be added in future work.

This memory is used solely to simulate the DMA's real behavior and is filled with random values by the testbench. The issue with this memory appears during synthesis, because register vectors are synthesized as millions of flip-flops, making the synthesis results unrealistic. For this reason, the memory structure is excluded during synthesis using conditional directives that separate the RTL code and prevent this memory from being synthesized.

3.2.4 Robustness

One of the strengths of this design lies in the robustness of the communication protocol. The state machine that handles data transmission was designed to prevent simultaneous activation of multiple start commands or unexpected interruptions during packet transfer.

The entire module relies on synchronous logic, in order to avoid undesirable situations such as data arriving out of order and to prevent malfunctions if the receiver is not yet ready. The internal counters are designed never to exceed valid limits, thus ensuring that no more data than necessary is read. Similarly, the mechanism that controls the memory read address prevents access outside the region dedicated to the current plane. In this way, any unexpected variations in AXI-Stream signals are handled without side effects, and potential anomalies are prevented from propagating toward the frame loader or compromising the execution of the entire matrix.

3.2.5 Conclusion and Considerations

The decision to adopt a very simple logic was driven by the idea that, if in the future the module needed to be expanded—for example, by adding more planes or modifying the frame size—the entire architecture would not need to be redesigned. Moreover, the AXI-Stream protocol is widely used in the vast majority of embedded systems, which means that this architecture can easily be extracted and integrated into a different context, whether another CGRA or a different type of project.

Everything therefore relies on the coherence of the design and the simplicity of the state machine, which together enable implementations suitable even for more advanced technologies.

3.3 Frame Loader

In the developed architecture, the frame loader, which is placed as a "connection" element between the DMA and the matrix of Processing Elements, has a crucial role in guaranteeing the specific features that have already been widely discussed. What the frame loader does is to take the AXI-Stream flow produced by the DMA and convert it into suitable two-dimensional structures that will then be processed by the PE matrix. Its task is also to decouple the serial nature of the DMA from the parallelism of the CGRA, so that each Processing Element immediately has the initial data available for computation, without having to deal with the data transfer logic itself.

To understand the usefulness of this module, it is useful to state a few basic requirements that the Processing Element matrix must satisfy in order to operate correctly:

- the data must be organized in the intended position (i, j);
- the two planes `gen0` and `gen1` must be available in a synchronous way, that is, without sudden variations or updates during normal execution;
- the loading of the following data must not interfere in any way with the computation that is currently in progress.

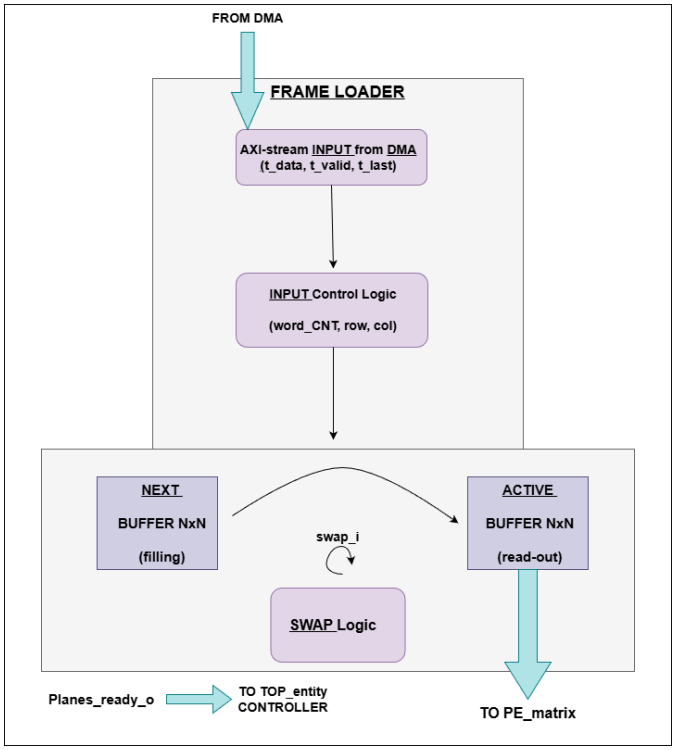

Figura 3.4: The module receives the AXI-Stream data from the DMA, reconstructs the incoming planes inside the NEXT buffer through the input control logic (row/col indexing), and exposes the ACTIVE buffer to the PE Matrix. Swap logic toggles the roles of NEXT and ACTIVE upon swap_i, implementing the ping-pong mechanism used for continuous frame processing.

All these requirements are fulfilled precisely thanks to the addition of the frame loader module, which stabilizes the system by taking a completely linear data stream, without any intrinsic subdivision into planes and possibly irregular because of AXI backpressure, and transforming it into something stable and ordered, ready to be used by the matrix.

Moreover, its functionality does not stop here: it also serves to keep the CGRA always operational. While the PE matrix is processing the current data, the frame loader is not idle, but it is already taking care of loading the next frame. All of this is implemented with a dedicated mechanism; without it, the CGRA would be forced to remain in a stall phase until the AXI loading was completed and only then could it resume computation. The presence of the frame loader therefore allows a better throughput, precisely thanks to this dedicated logic.

3.3.1 Ping-pong Buffer

In modern architectures that require continuity in the data flow, the use of ping-pong buffers is quite common. In the developed project this mechanism becomes even more important, because the frame loader has to cooperate with a fully parametric matrix, which is therefore independent from the loading mechanism.

The basic idea is simple: there are two buffers. One is called **ACTIVE**, and it contains the values used by the matrix for the current computation. The other one is called **NEXT**, and it contains the data coming from the DMA. When the transfer is completed, the system remains in that state until a swap signal arrives. Through this swap signal, the data that were in `NEXT` move to `ACTIVE`, and the `NEXT` buffer can be filled again with new values from the DMA.

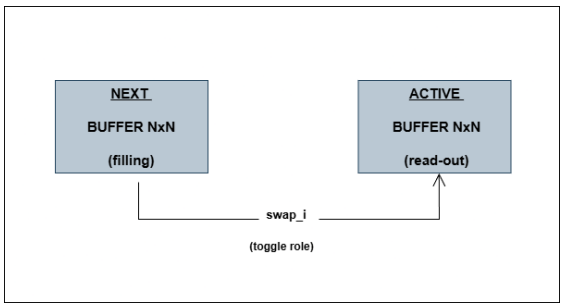

Figura 3.5: The Frame Loader maintains two memory banks: NEXT, which is being filled by the DMA stream, and ACTIVE, which is consumed by the PE Matrix. When both the frame loading and the current computation are done, the swap_i signal toggles the roles of the two banks, enabling continuous dataflow without stalling the matrix.

This has been done to guarantee a clear separation between:

- the **data in use**, that is, the data that the matrix is using at that moment to perform the requested computation;
- the **data being loaded**, that is, the data coming from the DMA following the rhythm of the AXI transfer.

In addition, adopting a double-buffer system avoids unpleasant issues such as unwanted overwrites while data are still being processed, or long stall situations where the PE matrix remains idle.

We can say that the double buffer introduces a temporal decoupling level: the matrix (through the frame loader) and the DMA become almost completely independent modules, connected only through the swap signal.

3.3.2 Internal Structure

Looking at the internal structure of the module, it contains several macro-components that communicate with each other. The whole block has been designed in a non-trivial way in order to correctly handle the AXI-Stream flow.

The module can be divided into three main blocks:

- **AXI-Stream front-end**, which manages the protocol signals (`tvalid`, `tready`, `tdata`, `tlast`) and also takes care of checking and regulating the input data, that is, whether they arrive in the correct format, at the right moment, and in compliance with the protocol constraints;
- **2D plane reconstruction**, where row and column counters are used to make the addressing of beats inside the buffer very simple. The beats are placed in their predetermined position, and this process is not influenced by the AXI transfer rate;
- **Ping-pong management**, which is the macro-block responsible for selecting the bank (`ACTIVE` or `NEXT`) and managing the storage of frames as well as the generation of the `planes_ready_o` signal and the swap signal.

The beats can be seen as flowing through a pipeline composed of these three macro-blocks. Everything is fully synchronous, in order to avoid critical combinational paths and to make the synthesis of the system easier.

If we want to analyze in more detail the flow of reconstruction of the bidimensional frame, we can start from what happens immediately after the frame loader receives the AXI flow from the DMA. The frame loader transforms the sequence of values into two bidimensional matrices of size $N \times N$. This process is not trivial, because the management of the row and column counters must be very precise: they must respect both the rules of the AXI-Stream protocol and the structure that each Processing Element expects to receive.

The two counters are managed as follows:

- the column counter is incremented every time the frame loader receives one beat from the AXI stream;
- the row counter is incremented each time the column counter reaches the value $N - 1$.

In this way, all beats are mapped to the correct position (i, j) in the `NEXT` buffer. Moreover, thanks to the `tlast` signal, which is a characteristic of the AXI protocol, misalignment errors in the frames can be avoided. This is possible because the frame loader constantly checks both the state of the counters and the expected

position of the last beat, and it verifies that `TLAST` arrives exactly where it is supposed to.

Thanks to this constant checking, several non-negligible issues are avoided, such as:

- **premature end**, where part of the matrix remains empty and therefore without valid values, which would lead to completely wrong computations by the PE matrix;
- **delayed end**, which causes overflow in the counters;
- **mismatch between `TLAST` and the internal progression**, which is one of the most common issues in streaming-based systems.

Once the `gen0` matrix has been completed, the state changes, the counters are reset, and exactly the same flow is repeated for `gen1`.

To make the internal flow of the frame loader clearer, it can be summarized in the following steps:

1. accept a beat from the DMA if both `tvalid` and `tready` are active;
2. interpret the data as belonging to `gen0` or `gen1` according to the current state;
3. compute the position (i, j) using the row and column counters;
4. write the value into the `NEXT` buffer at the corresponding position;
5. update the counters, checking whether a row increment is required;
6. handle `TLAST`, verifying that it occurs at the correct position.

The sequence may seem linear and simple, but a correct result is guaranteed only if the synchronization between all these steps is extremely strict.

3.3.3 State Machine and State Management

The state machine of the frame loader is divided into three main phases. It is designed to be very simple but at the same time robust. The process is handled from the first incoming beat up to the moment when the entire frame is fully available.

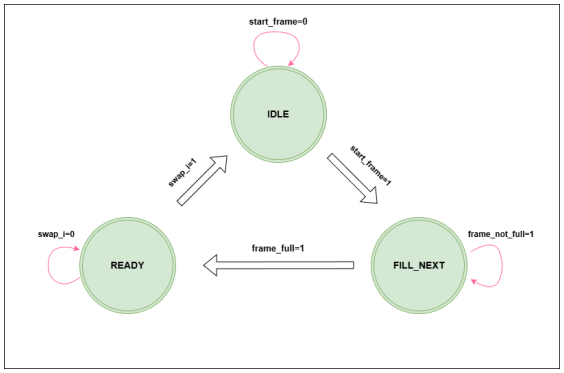

Figura 3.6: This state machine has three phases of operation: waiting for the first GEN0 beat to arrive, loading all the beats into the NEXT buffer while monitoring counters and TLAST, and checking the completeness of a frame before asserting planes_ready_o. This structure ensures synchronous, reliable, and protocol-compliant reconstruction of the two planes.

- **The first phase is a waiting phase**: the module is idle, waiting for the first beat of the `gen0` packet. The counters are in reset state, as are the internal structures. The receiver keeps `tready` high to inform the DMA that it is ready to receive data.

- **The second phase is the frame loading phase**: as soon as the AXI-Stream flow starts, the frame loader accepts beats one by one, placing them in the `NEXT` buffer according to the current state of the counters. It monitors the transition from the `gen0` plane to the `gen1` plane and verifies the `TLAST` signal. In this phase there is still no communication between the matrix and the frame; the objective is to reconstruct the frame as precisely as possible.

- **The third phase is the validation phase**: when the last beat of `gen1` has been received, a coherence check of the transfer is performed. After that, the loader stops the counters and sets a flag indicating that a complete frame is available. Finally, it asserts the `planes_ready_o` signal toward the top level.

The activation of `planes_ready_o` does not automatically trigger the swap signal. This is an intentional design choice, so that this mechanism can be controlled externally, for example through another state machine or, as in this case, simply by the testbench. In this way, it is possible to decide exactly when the new frame should be passed to the matrix and to avoid building a system that iterates indefinitely.

This mechanism makes the entire CGRA more flexible, since it can also accept computations triggered by other peripherals or external controllers.

3.3.4 Swap Handling and Stability of the ACTIVE Planes

The moment dedicated to the swap between the two buffers is one of the most delicate steps of the entire component. The frame loader must ensure that the replacement of the planes does not interfere with the computation already in progress, guaranteeing that the PE matrix always operates on a stable and unchanged set of values for the entire duration of the computation.

To obtain this result, the system performs the swap only when it receives an explicit command from the upper level, represented by the `swap_i` signal.

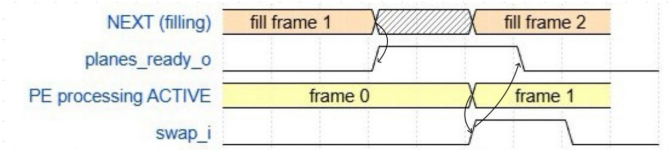

Figura 3.7: The signal planes_ready_o is asserted at the end of the NEXT filling phase. The swap only occurs when both conditions are satisfied: NEXT is fully loaded, and the PE Matrix is done processing the ACTIVE frame.The swap_i pulse toggles the buffer roles and initiates the next pipeline iteration.

When the swap request arrives, the module performs three fundamental operations:

1. it deactivates the buffer currently marked as `ACTIVE`;
2. it promotes the `NEXT` buffer to the role of new `ACTIVE`;
3. it prepares the old `ACTIVE` buffer to contain the next frame.

It is essential to note that the frame loader never performs the swap autonomously: this prevents unwanted or premature updates and ensures that the entire matrix always sees consistent data.

The toggle between the two planes is done at RTL level merely by toggling the bit `active_bank_q`, which determines which bank must be presented to the PEs. This design choice greatly reduces hardware cost and avoids the need to physically copies the data, making this solution very efficient.

3.3.5 Robustness, Error Handling and Prevention of Anomalous Conditions

Beyond correct data routing, a crucial part of the frame loader concerns its ability to react to irregularities in the AXI-Stream flow. Even if the protocol is quite flexible, it may produce non-ideal conditions such as unexpected delays, extended backpressure or beats that are not perfectly aligned in time. The module must therefore guarantee the correct reconstruction of the frame even in the absence of a constant rhythm.

To deal with these situations, the design implements two complementary forms of protection:

- **temporal protection**: each beat is accepted only if `tvalid` and `tready` are both active at the same time, preventing premature reads;
- **structural protection**: the system prevents counter advancement or state transitions if the flow does not respect the expected order.

In other words, every beat is treated as an independent event and becomes valid only if all the expected conditions are met. If something appears inconsistent, the module automatically suspends its progression until the input returns to a correct state. This avoids overwriting data in the `NEXT` buffer and prevents the generation of corrupted frames.

At the same time, the loader verifies that:

- the total number of beats for each plane is the expected one;

- the `TLAST` signal appears only on the last beat of the plane;
- both planes — `gen0` and `gen1` — are complete before `planes_ready_o` is asserted.

If any of these constraints is violated, for example because of a misplaced `TLAST`, the system asserts the `err_protocol_o` signal, informing the upper level that an anomaly has occurred. This approach drastically reduces debugging time and prevents the error from propagating into the matrix.

3.3.6 Integration with the Top-Level and with the PE Matrix

The true importance of the frame loader becomes clear when it is observed in the wider context of the architecture. Its logic has been developed to integrate naturally both with the DMA, which is responsible for the AXI-Stream flow, and with the PE matrix, which instead requires parallel data that are perfectly stable.

On the input side, the frame loader communicates with the DMA through the main AXI-Stream signals:

- `tdata`, which carries the data content;
- `tvalid`, which indicates when the data are available;
- `tready`, which tells the DMA that the frame loader is ready to receive;
- `tlast`, which marks the end of a plane.

This full adherence to AXI-Stream makes the module highly reusable and compatible with any AXI-Stream source, increasing the modularity of the entire project.

On the output side, the interface toward the PEs is intentionally kept simple: `gen0_active_o` and `gen1_active_o` are presented as static arrays, already ready to be used by the matrix, without any handshake or validity logic. In this way:

- the data path toward the PEs remains extremely linear;
- the matrix is completely isolated from the complexity of the AXI protocol;
- the planes remain stable for the entire duration of the computation.

Even while the DMA is already sending the next frame, the matrix continues to work without interruptions on the `ACTIVE` plane, maintaining deterministic and reliable behavior.

3.3.7 Timing, Performance Aspects and Relation to Global Throughput

One of the key goals of the architecture is to avoid situations where the CGRA has to wait for data loading, so that it can quickly move from one frame to the next. The frame loader contributes directly to this goal through two fundamental mechanisms:

- the ability to fill the `NEXT` buffer while the matrix is working on the `ACTIVE` buffer;
- the fact that the swap is performed only when requested by the controller.

In this way, the time required by the DMA to send the data does not directly affect the overall throughput of the architecture. Loading becomes a "shadow" activity that proceeds in parallel with the actual computation.

For example, if the matrix needs 2000 cycles to complete a batch, and the DMA needs 1400 cycles to transmit the next frame, the architecture can operate without any pause: by the time the matrix finishes, the next frame is already ready in the `NEXT` storage.

From a performance point of view, this means that the maximum speed is determined almost exclusively by the computation time, while the AXI-Stream transmission is hidden behind the processing. This makes the architecture particularly effective in streaming or iterative scenarios.

3.3.8 Scalability and Adaptation to Different Matrix Sizes

One of the most interesting features of the **frame loader** is its ability to scale together with the size of the **matrix**. Thanks to the automatic generation of the internal structure, the module can be easily adapted for matrices of any size without modification of the core logic.

As the parameter N increases, we mainly observe:

- a proportional increase in the memory required for the `ACTIVE` and `NEXT` buffers;
- a higher number of beats needed to complete each plane;
- longer cycles to fill the `NEXT` buffer;
- slightly longer stream-to-matrix reconstruction times.

Despite this, the internal organization — `TLAST` management, counters, FSM and ping-pong mechanism — remains unchanged. This allows the frame loader to be used both in compact configurations and in large designs, making it extremely flexible.

In an ASIC context, this property is particularly useful, since the only resource that really grows is the internal memory, which can be efficiently implemented with dedicated SRAM blocks.

3.3.9 Final Design Considerations

The double-buffer frame loader is one of the cornerstones of the entire architecture. Its ability to translate an AXI-Stream flow into two coherent bidimensional planes, while keeping loading and computation separated, makes it an essential element from both an architectural and a performance perspective.

The combination of:

- structured frame reconstruction,
- double `ACTIVE`/`NEXT` buffer,
- full adherence to AXI-Stream specifications,
- swap controlled by the upper level,
- complete isolation between the serial data flow and the parallel domain of the matrix,

allows the CGRA to reach performance levels typical of much more complex systems, without sacrificing modularity, readability of the RTL code, or scalability.

3.4 Processing Element

The Processing Element represents the basic functional unit at the core of the entire project. This unit has been designed with the idea of keeping the structural complexity as low as possible, in order to avoid, in realistic scenarios where many units are needed to build the final matrix, an excessively large area that would no longer be compatible with a real system.

The design of the single Processing Element starts from the assumption that it can receive two inputs and then perform basic arithmetic operations such as addition, subtraction, and multiplication between the two operands. This basic instruction set is certainly less rich than the operational set of a standard processor, but it still allows a good compromise between simplicity and the possibility of supporting real applications.

Moreover, the choice of a simple architecture is also motivated by the initial idea of building different interconnection topologies among the individual PEs. If a more complex architecture had been chosen, the more articulated topologies would have introduced several problems in synthesis, especially for topologies with very dense connections. Each PE therefore implements a reduced but effective ALU model, with low latency and easily integrable in a system like the one proposed, where the matrix size is parametric.

3.4.1 Structure of the PE Module

The general structure of the module is organized into three main macro-blocks:

- **input front-end**, the initial part that collects the two operands that will be processed according to the instruction indicated by the opcode;
- a small **accumulation and synchronization unit**, essentially registers that hold the operands until both of them are available;
- the **ALU**, which executes the operation specified by the opcode.

The module follows a very simple and predictable logic: a new instruction is accepted only if the module is completely free. This means that if a previous instruction is still being processed, it is not possible to start a new one, and the instruction that has been taken in charge is executed only when both inputs are available. Everything has been designed in this way to adapt to a very simple state machine and to avoid internal conflicts.

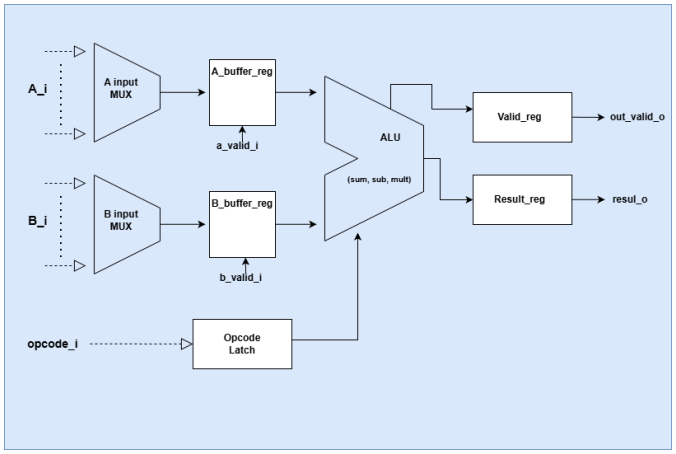

Figura 3.8: The operands A and B are selected via input multiplexers, are stored in local registers, combined through the ALU according to the latched opcode, and finally exported via registered outputs with an associated validity flag.

The logic followed by the module to perform the complete elaboration can be summarized as follows:

1. it receives the command through the activation of the `in_valid_i` signal;
2. it captures the opcode at the first available clock edge and sets the internal state to "operation in progress";
3. it stores the two operands A and B, which, it is important to note, can arrive at different moments;
4. it executes the operation (ADD, SUB, or MULT);
5. it emits the result by asserting the `out_valid_o` signal.

In this way, each Processing Element is equally effective even if data and their latencies are different and arrive in a non-ordered way.

One of the key characteristics of the Processing Element module is precisely the fact that the two operands A and B can arrive in separate cycles. In a structure of this kind, this is a fundamental requirement, because if one of the two operands has to pass through a larger number of PEs than the other input, the system must still be able to work correctly. All of this is made possible by the use of two internal buffers:

- an **A buffer**, implemented as a register with an associated validity flag;
- a **B buffer**, implemented in exactly the same way.

Only if both flags are valid can the operation start. This makes the system data-driven, meaning that the activation of the module depends exclusively on the arrival of data and not on internal clocks or external impulses.

Looking briefly at the synchronization mechanism: when the first operand arrives, the module stores it and activates its validity flag. If the second operand has not yet arrived, the PE remains in a waiting state. As soon as both flags are active, in the next cycle the computation starts. Once the operation is completed, the buffers are cleared and become available again to accept new operands.

The presence of these flags is an extra guarantee that avoids overwriting the Processing Element with a new instruction when the current one has not yet been completed.

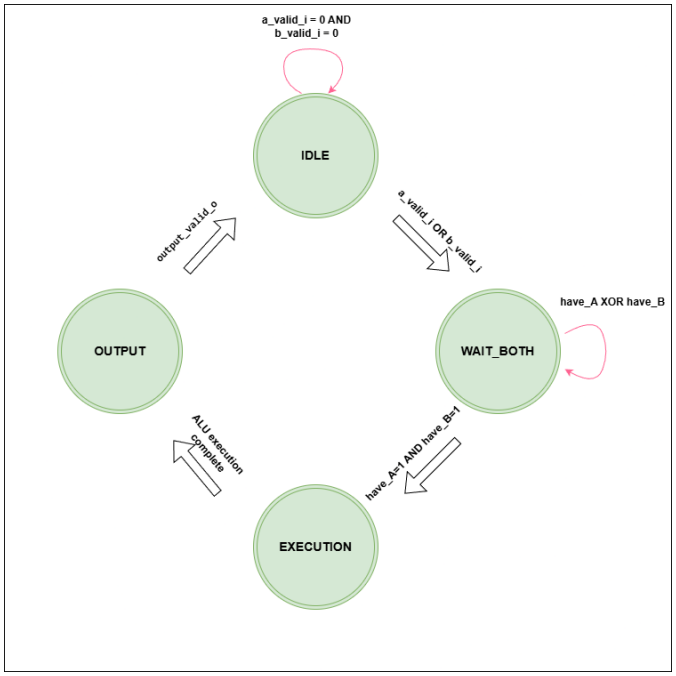

Figura 3.9: It first evaluates the availability of operands A and B, stores them in local registers, and then proceeds to the execution phase when both operands and the opcode are valid. The result and its validity bit from the ALU are registered, after which the PE transitions back to the idle state for the next operation.

3.4.2 Internal ALU

The ALU inside the module is very small and compact. In fact, it supports only three fundamental operations:

- **ADD**: addition between operands A and B;
- **SUB**: subtraction between operands A and B;
- **MUL**: multiplication between operands A and B.

The choice of such a limited set of operations is not casual but is based on a precise logic:

- ADD and SUB are two operations that are practically necessary in all computation flows;
- MUL is certainly the most used operation in numeric and matrix contexts;

- a more complex ALU would have provided very limited benefits with respect to the thesis objectives, while bringing a significant increase in complexity and also in area.

This implementation is perfectly consistent with classical CGRAs, respecting the principle of short pipelines where each PE is simple, fast, and easy to replicate.

3.4.3 Validation and Output Signals

Once the operation is completed, the PE has to communicate the result of the recently executed operation to the other modules. This is done in a very simple way: through the `out_valid_o` signal, the module indicates that the value present on `result_o` is effectively the result of the operation that has just been completed.

In addition, other ports such as `used_a` and `used_b` have been introduced. These signals were mainly used in simulation to test the correct behavior of the entire operational block. They are therefore not necessary for the normal functioning of the system but are useful for the testbench to assess the correctness of the results and are also very helpful during debugging, especially when everything becomes more complex as in the FULL topology.

This final cycle can be described with a few steps:

1. both operands are valid;
2. in the immediately following clock cycle, the operation is performed by the ALU;
3. the result is placed on the `result_o` port and the `out_valid_o` signal generates a pulse that lasts one clock cycle;
4. the PE buffers are reset and become available for a new operation.

3.4.4 Interaction with the Matrix Topology

Even though the Processing Element is an isolated module, it does not operate alone in the proposed structure. It can receive operands from different sources, and these sources vary depending on which interconnection topology is chosen for the `PE_matrix` module.

The possible data sources for a single PE can be:

- the **GEN0** and **GEN1** planes provided by the frame loader; these are fixed and independent from the topology, and whatever the chosen topology, the two planes are always available to the PE;

- the results of neighboring PEs in the four cardinal directions (N, S, E, W) in the **MESH-4** topology;
- the results of neighboring PEs in the four cardinal directions plus the four diagonals in the **D-MESH** topology;
- the results coming from a toroidal neighborhood in the **D-TORUS** topology;
- the results of all the other PEs in the matrix in the **FULL** topology.

Precisely because of this wide range of possible configurations, the A and B buffers have been implemented to handle situations where some results arrive earlier than others, either due to longer combinational paths or because of different positions of the sources within the matrix.

In practice, from the point of view of the Processing Element, each operand arrives as a single signal (data accompanied by its valid), without the PE needing to know from which source it originally comes. All the complexity of selecting the data path is completely absorbed by the interconnection logic of the matrix.

This choice introduces two key benefits:

- the PE remains extremely lightweight, because it does not need any information about the topology or about the position of its neighbors;
- the entire CGRA can change configuration or connection type without needing to modify the ALU or the internal micro-logic of the single elements.

Keeping communication and computation clearly separated is one of the principles of modern CGRAs, providing for a clean, modular design that can be easily reused and scaled up when larger configurations are needed

3.4.5 Robustness of Synchronization and Prevention of Critical Conditions

The PE has been designed to intrinsically avoid all those problematic conditions that are typical of systems processing data in parallel. In particular, its logic guarantees that:

- it cannot start a new operation before having completed the previous one;
- it cannot ignore a valid operand;
- it cannot generate two consecutive results without first returning to the idle state;

- it cannot execute a computation if it does not have both required operands at the same time.

These constraints are enforced through a mixture of internal status signals (such as `have_op_q`) and a set of rules that regulate the PE progression step by step. The result is a fully deterministic behavior: with the same inputs and under the same timing conditions, the module will always produce the same result in the same cycle.

This level of regularity is fundamental because in a **CGRA** even small anomalies can easily snowball when hundreds of PEs operate in parallel. A single-point failure in element could easily propagate, blocking dependencies or interrupting the whole Computational process. The PE design aims to minimize this possibility by providing stable and reliable behavior even when the matrix is very large.

3.4.6 Final Considerations on the Processing Element Design

The proposed Processing Element represents a well-balanced compromise between structural simplicity, essential functionality, and the ability to be replicated on a large scale. The main characteristics that distinguish it are:

- a dedicated buffering system for the two operands;
- a compact ALU, with constant and unambiguous behavior;
- a very linear internal state management;
- a fully synchronous architecture, designed to avoid complex combinational paths.

These features make the module perfectly suitable to be replicated in large arrays without significant impact on area, power consumption, or routing complexity.

From a more general point of view, the PE fully embodies the philosophy underlying CGRAs: simple units that can reach high performance thanks to massive parallelization and a configurable interconnection network. Furthermore, the RTL structure is clear and modular, and easy to modify if one wants to add new operations or introduce a deeper pipeline.

In conclusion, the PE is not just the elementary "brick" of the matrix: it is a key component that directly affects the regularity, scalability, and overall reliability of the architecture. Its implementation is one of the strengths of the entire project, both from a technical point of view and as a foundation for the global organization of the CGRA.

3.5 Parametric $N \times N$ Matrix: Structure, Scalability and Topology Management

3.5.1 Introduction and Role of the Matrix in the CGRA

The matrix of Processing Elements can be considered as the operational core of the entire CGRA. Here the workload is distributed across multiple elementary units, allowing simultaneous computation. In this kind of structure, unlike linear pipelines, the idea is not simply to duplicate a single computing unit, but to build a reconfigurable network that allows data to flow freely through it, according to the specific application that must be executed.

In the proposed project, the matrix is completely parametric with respect to the dimension N. This means that, by changing a single RTL parameter, it is possible to obtain matrices of different sizes: 2×2, 3×3, 4×4, and so on. This approach is perfectly in line with coarse-grained architectural philosophies, where the goal is to make the computing block easily replicable and scalable, so that several experiments can be carried out with different sizes without having to rewrite the code or modify the structure every time.

All of this is made possible in the `pe_matrix` module through the use of `for` loops and `generate` constructs, which allow a very compact and simple organization of the code, creating a 2D structure of rows and columns that is perfectly aligned with the design goals.

3.5.2 General Structure of the Module and Parametricity Criteria

What is really fundamental in this part is the way the inputs of the PEs are selected. To explain this aspect clearly, it is useful to describe in a simple way the information flow inside the `pe_matrix` module.

The `pe_matrix` receives:

- the two planes `GEN0` and `GEN1` sent by the frame loader,
- the selection signals for the two operands, `sel_a` and `sel_b`,
- the opcode to be executed,
- the validation signal `in_valid`,
- the global clock and reset.

The selectors `sel_a` and `sel_b`, and the way they are configured, determine the overall behavior of the network, because they define which source will be used by each PE. The concept of parametricity is therefore based on two main ideas:

- the **dynamic number of PEs**, determined by the size $N \times N$;
- the **number of sources** available to each PE, which depends on the chosen topology.

The sources are determined by an RTL parameter (`Kmax`), computed inside the module. It takes into account the two sources `GEN0` and `GEN1`, which are always present independently of the selected topology, plus a number of additional sources that depend on the topology itself. With this approach, it is possible to support any topology without having to change the internal logic of the PEs.

One of the main characteristics of the module is precisely the way in which these various sources are managed. For each position (i, j), the module builds a vector that contains:

- `data_wires[k]`: all values that can feed the operand;
- `data_valids[k]`: the corresponding validity signals.

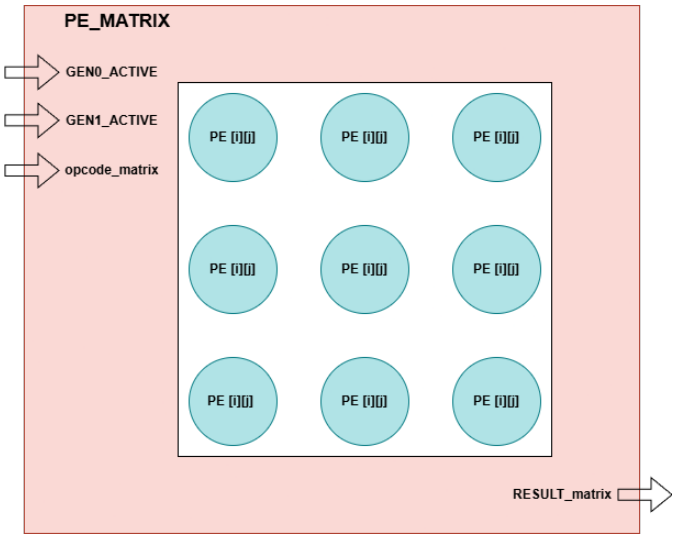

Figura 3.10: Each PE operates independently and in parallel, receiving data from the ACTIVE buffer and producing results through a parametrically generated grid structure.

These vectors have a size equal to `Kmax`, which varies according to the chosen topology. For example, in the **FULL** topology, a PE can receive data from all other PEs in the matrix, except itself, for a total of $(N^2 - 1)$ possible sources. In the **MESH-4** topology, the sources are 4, and so on for all other topologies. In every case, the `GEN0` and `GEN1` planes must always be taken into account.

Everything has been organized to keep the behavior extremely ordered: the PE simply has to select an index in the vector to get its input data.

The assignment of sources in the vector follows this scheme:

- `GEN0` → `data_wires[0]`,
- `GEN1` → `data_wires[1]`,
- all neighbors → `data_wires[2 .. Kmax-1]`.

This organization is one of the key points that enables the scalability of the project: as N grows, the matrix increases in size in a very regular way, and the routing of the signals is handled automatically by the RTL code, with no need for manual changes.

3.5.3 Integration Between Selectors, Topology and PE Behavior

The role of the `pe_matrix` module is crucial when it comes to topologies in the system. The individual PEs are not aware of which topology is being used; they simply see two inputs and execute the requested operation. It is exactly in this scenario that `pe_matrix` takes on its key role: building the correct context in which each PE will operate.

This is done through the selectors `sel_a`(i,j), which indicate which source is the correct one to feed the PE in position (i, j); similarly, `sel_b`(i,j) selects the B operand source. The width of these selectors is calculated internally in the module through a small function, such that `selW = clog2(Kmax)`.

Thanks to this mechanism, it is possible to modify the global behavior of the entire matrix without changing the PE module. At this level, the system can be seen as a set of independent nodes, each one with its own dedicated ALU, which from independent elements become interdependent through the external programming of the selectors.

The main point of flexibility of the module lies exactly in the fact that it can change the interconnection topology among the PEs. Once a topology is chosen, it defines, for each PE, which sources can be used as operands A and B. The topologies implemented in this thesis are four:

- **FULL**: in this topology, each PE can use as sources the results produced by all the other PEs, except its own result;
- **MESH-4**: in this topology, the neighbors are only those in the four cardinal directions (north, south, east, west);

- **D-MESH**: starting from a MESH-4 base, the neighbors on the diagonals are also included;
- **D-TORUS**: similar to D-MESH, but with coordinate wrapping, meaning that each border is connected to the opposite side, as in a toroidal surface.

The topology choice does not affect the internal behavior of the single PE, but rather the way in which PEs are connected to each other. In practice, it changes the size of the `data_wires` and `data_valids` vectors, and the logic that assembles the sources.

From a hardware point of view, what changes is the number of connections between PEs and the way these connections are mapped, which in turn produces an increase in area and routing complexity that grows directly with N.

In the **FULL** topology, for example, where each PE can receive (N^2-1) inputs in addition to the two coming from the frame loader, the costs grow significantly, even though the connectivity is almost maximal and it becomes possible to implement very irregular algorithms.

3.5.4 Data Synchronization and Propagation of Validity Signals

The management of the data validity signal is one of the most delicate aspects of the entire matrix. Each source connected to `data_wires[k]` has its own `data_valids[k]`, which indicates whether that data is actually usable. This mechanism allows the PE to activate only when it has correct information, avoiding partial or incoherent computations.

The `pe_matrix` module does not simply forward the valid signals from neighboring PEs: it regenerates them in a coherent way, based on the chosen topology and on the position of the node. In configurations without wrapping (**FULL**, **MESH-4** and **D-MESH**), the PEs located at the borders receive "dummy" values from directions where there is no neighbor: data equal to zero together with `valid = 0`. In this way, no PE risks using undefined operands.

The situation is the opposite in the **D-TORUS** topology: since there are no borders, the propagation of valid signals is continuous and cyclic, and a PE can receive data from a position on the opposite side of the matrix. This characteristic completely removes "dead" points and allows the creation of computational patterns that exploit the symmetries and periodicity of toroidal structures.

A punctual validation system is necessary and should be initiated to maintain the stability of the entire **CGRA**. Each **PE** is designed to activate only when both operands are declared valid. If even one of them is not available, the PE remains idle, preventing the risk of carrying out incomplete operations and thereby drastically reducing the risk of error propagation into succeeding stages.

3.5.5 Architectural Scalability and Implementation Implications

A central objective of the `pe_matrix` module is to ensure that the architecture behaves consistently even when the dimension N increases. The fully parametric nature of the design means that the matrix can be enlarged without any manual modifications to the RTL code: it is enough to regenerate the project with a different value for the parameter.

When N grows, several effects occur:

- the total number of PEs increases, and consequently ALUs, buffers and control signals multiply;
- routing complexity grows, especially in the **FULL** and toroidal topologies, where each node has a higher number of connections;
- the overall latency increases, not so much in the single PE, but in the propagation of valid signals along longer paths;
- area usage increases, particularly when the number of sources per node is large.

The decision to design PEs that are completely independent from the topology proves to be extremely effective for scaling the design. In an ASIC environment, replicating identical and regular blocks makes synthesis and physical layout more efficient. The matrix can grow in size without compromising order, regularity or temporal predictability.

Another important advantage comes from the clear separation of responsibilities: routing and valid signal generation are handled by the `pe_matrix` module, while each PE is responsible only for local computation. This division makes it much easier to verify the behavior of the whole system, because it is possible to change the topology without affecting the internal logic of the single PE.

3.5.6 Final Considerations

The $N \times N$ matrix presented in this project is a concrete example of a modular and reconfigurable architecture, designed to adapt to very different requirements. The ability to switch between different topologies, combined with the orderly management of validity signals and full parametrization, gives the module a high degree of flexibility. From a physical point of view, the regular structure of the matrix simplifies both synthesis and placement/optimization steps.

Overall, the use of a uniform matrix composed of identical PEs connected through dynamically generated logic makes it possible to experiment with various

computational models and to evaluate the impact that different topologies have on area, timing and throughput. This module is one of the most important blocks in the entire project and represents a solid foundation for any future extension or further research on CGRA architectures.

3.6 Implemented Topologies in the Matrix

The interconnection topology is one of the most delicate aspects in the design of a CGRA. Even when individual PEs are simple and regular, the way they are wired together determines the overall "character" of the architecture: which data patterns can be mapped, how far dependencies must travel, and how much flexibility is available for complex algorithms. For this reason, the proposed project does not rely on a single topology only, but introduces a parametric mechanism that can generate four distinct configurations by simply changing one RTL parameter.

The choice derives from two rather different motivations. On one hand, there's the practical need for a truly flexible architecture: some applications work well with a small, local neighborhood, while others need to reach more distant nodes or handle a larger number of dependencies. On the other hand, there's an experimental Motivation: Having multiple topologies allows to directly observe how **performance**, **area**, and **routing complexity** vary during synthesis while maintaining leaving everything else about the architecture identical. All four topologies have the same general structure, but their connectivity model is different.

3.6.1 FULL Topology

The FULL topology is the densest configuration: each PE can see all other PEs in the matrix, with no restriction on neighbourhood. Logically, every PE behaves like a fully connected node that can select any other unit as data source.

This offers the highest possible flexibility:

- any communication pattern can be represented;
- algorithms with highly irregular dependencies face no structural limitation;
- the logical distance between any two nodes is ideally reduced to a single hop.

However, this freedom has a substantial cost. The number of interconnections grows quadratically with the matrix size, and this quickly becomes hard to sustain:

- in a 4×4 matrix, each PE must consider 15 possible neighbours;
- in an 8×8 matrix, the sources become 63;

- in a 16×16 matrix, this number rises to 255.

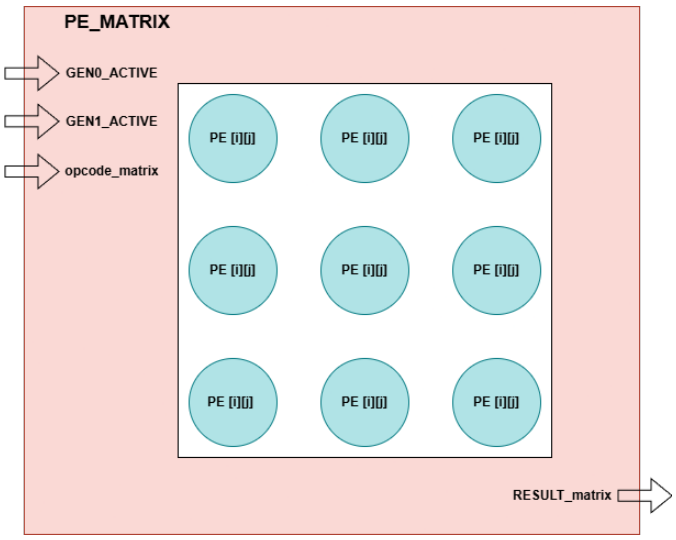

Figura 3.11: Each PE operates independently and in parallel, receiving data from the ACTIVE buffer and producing results through a parametrically generated grid structure.

In this kind of configuration, the real issue does not lie in the control logic, but in something far more challenging: the physical routing. As the value of N increases and the matrix becomes larger, routing quickly turns into something practically unmanageable. This setup is perfectly suitable when performing theoretical analyses and comparisons, but it is certainly not the most appropriate choice for a real-world system.

3.6.2 MESH4 Topology

The MESH4 topology is the most classical interconnection pattern in CGRAs: each PE is connected to its four cardinal neighbours (North, South, East, and West). The structure is reminiscent of regular systolic arrays or stencil-based grids, widely used in numerical computation and 2D filtering.

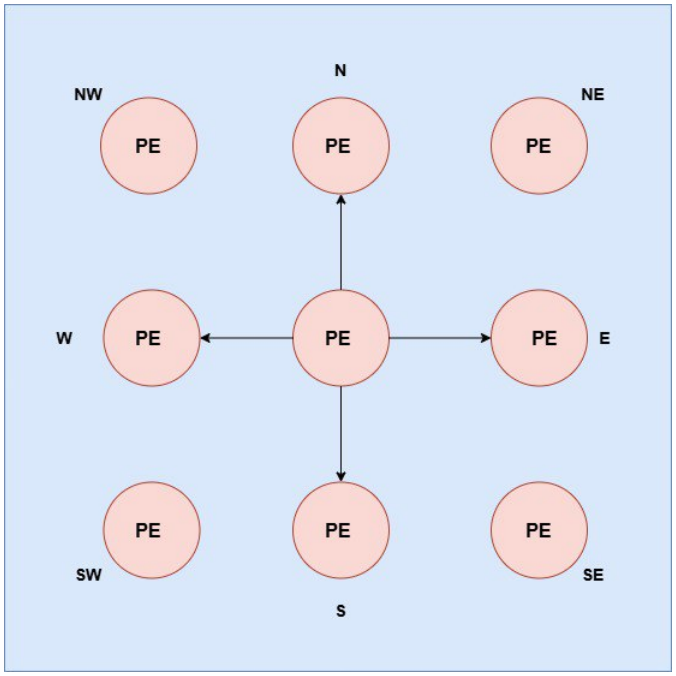

Figura 3.12: Each Processing Element is connected only to its four orthogonal neighbours: North, South, East, and West. This structure reduces the complexity in routing while still preserving the locality and regularity, hence is considered one of the most common topologies for CGRA scalable architectures.

Its main characteristics are:

- constant connectivity degree (each node has at most four neighbours);
- very limited routing complexity;
- strongly local and highly predictable behaviour;
- excellent scalability with increasing N.

From a computational perspective, MESH4 is particularly well suited to:

- regular operations such as convolutions and filtering;
- local propagation schemes;
- algorithms with clearly defined geometric dependencies.

Its simplicity is one of the reasons why it is widely adopted in industrial CGRA designs.

3.6.3 D-MESH Topology

The D-MESH topology extends MESH4 by also including diagonal neighbours. This increases the maximum connectivity degree from four to eight. The main advantage is a reduction of the logical distance between different points in the matrix, which can directly reduce the number of cycles needed to propagate a value along complex paths.

so:

- MESH4 focuses on purely orthogonal communication;
- D-MESH adds “radial” communication in all directions.

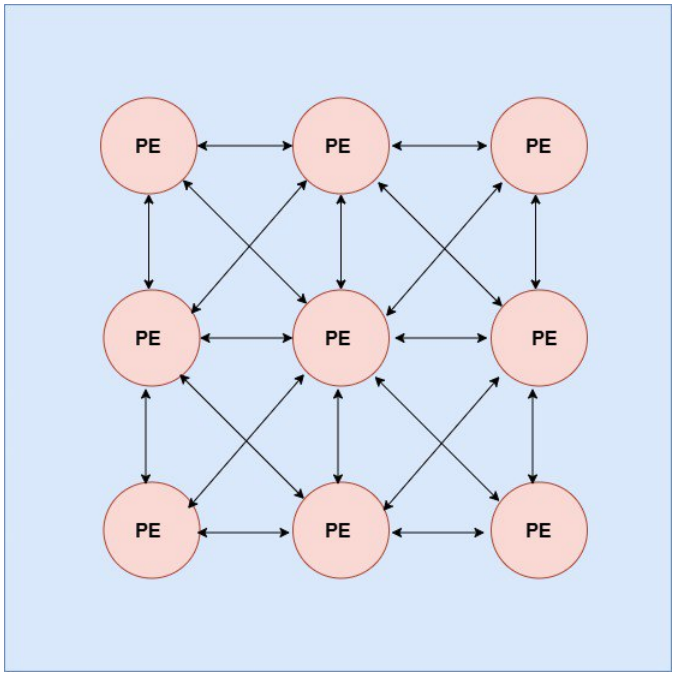

Figura 3.13: Processing Elements are connected to their eight immediate neighbours around themselves, including the diagonal links; that is, NW, NE, SW, and SE. Compared to the MESH4 topology, the D-MESH increases the routing flexibility and reduces communication distance, still avoiding global interconnections.

Numerical applications based on 8-neighbour stencils, complex cellular automata, or diffusion-like patterns can directly benefit from this richer connectivity.

In this topology, unlike the FULL one, the sources are completely independent of N; in fact, each PE will have at most 8 sources.

3.6.4 D-TORUS Topology

The D-TORUS topology is the most sophisticated option among the four. It retains the eight neighbours of D-MESH but removes the matrix borders by applying coordinate wrapping: a node at the right edge can access the corresponding node on the left edge, and the same holds vertically.

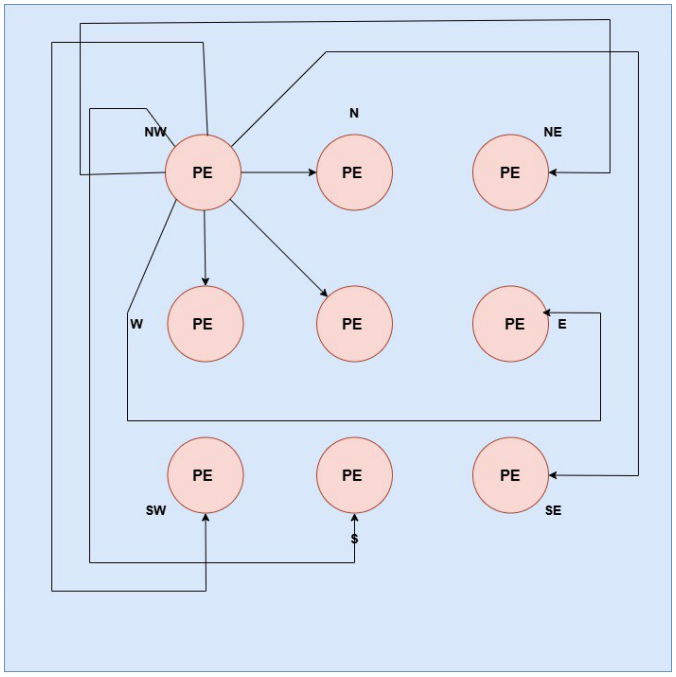

Figura 3.14: Each PE operates independently and in parallel, receiving data from the ACTIVE buffer and producing results through a parametrically generated grid structure.

The resulting consequences are quite interesting:

- the matrix becomes topologically equivalent to a torus;
- no edge cases or special conditions exist at the borders;
- each PE has exactly the same neighbourhood;
- the behaviour is perfectly symmetric along both dimensions.

This uniformity greatly simplifies some classes of periodic algorithms or computations that take advantage of closed topologies, such as iterative relaxation schemes or physical simulations with cyclic boundary conditions.

3.7 Scalability of the Matrix and Architectural Implications

Scalability is one of the key points of the entire project. Since the matrix is built through automatic generation (using `generate` blocks and `for` loops), its size can be increased without introducing irregularities or behavioural deviations.

As the matrix grows, three fundamental aspects change:

1) Number of PEs

The total number of computational units grows with N^2:

- a 10×10 matrix contains 100 PEs;
- a 16×16 matrix contains 256 PEs.

This impacts:

- overall chip or FPGA area;
- dynamic power consumption;
- complexity of validity synchronization;
- length of data paths.

2) Number of Interconnections

This aspect depends heavily on the chosen topology:

- FULL $\rightarrow$ quadratic growth, very heavy for routing;
- MESH4/D-MESH $\rightarrow$ constant per-node connectivity, highly predictable;
- D-TORUS $\rightarrow$ constant per-node connectivity with circular distribution.

This is one of the main reasons why CGRA designers often favour local topologies: they allow much more controlled growth.

3) Logical Distance Between PEs

Among the iterative algorithms, the number of cycles required for data propagation is essential. While dense topologies allow faster propagation, they are more demanding in synthesis;

3.7.1 Functional Complexity of the Matrix as N Grows

One of the most interesting effects of increasing N is the growth of the system's functional complexity. A larger matrix does not simply perform "more operations in parallel"; the way data propagates, the number of possible paths, and the interactions among PEs change in a qualitative way. This enables the CGRA to handle more complex algorithms or larger data structures while maintaining a homogeneous computational model.

The rigidity of the data paths, together with their limited length, forces many algorithms to be adapted or simplified in a small matrix. When the computational depth increases, however, and as Ngrows, it becomes possible to fully take advantage from :

- deeper spatial pipelines;
- partitioning of the computation across different matrix regions;
- more complex communication patterns (especially in D-MESH and D-TORUS);
- massive parallelism for grid-based repetitive operations.

Local topologies are leaner yet they take more cycles to shift values across the matrix. A typical example is that of numerical **stencil computations**, where each cell depends on local values, but several steps are required in the global evolution. In a small matrix a large number of cycles is required to propagate the data along diagonals or over long distances, and with a larger matrix the propagation time can be significantly reduced.

3.7.2 Effects of the Topology on Internal Latency

The intrinsic latency of the matrix depends not only on its size but also on the chosen topology. In a FULL topology, a PE can bypass any intermediate path, reducing logical distance to a single hop. However, this immediate connectivity translates into significant routing challenges as soon as N exceeds modest values. In **MESH4** and **D-MESH**, latency is distributed over several cycles, since values must traverse intermediate nodes. Curiously, this behaviour is often desiderated:

- timing closure is usually easier;
- pressure on interconnections is reduced;

The architecture becomes more predictable, in practice.

For example, in a MESH4 topology, the distance from one corner of the matrix to the opposite one is $2(N-1)$ orthogonal steps. In a **D-MESH** this distance is effectively reduced since the diagonal moves cover more ground with a single hop.

The **D-TORUS**, adding wrapping, removes the concept of strict maximum distance: any route can be taken along the shortest side of the torus, thus creating shorter and more uniform paths.

3.7.3 Interaction Between Selectors, Validity, and Synchronization

Correct management of validity signals is one of the pillars of the entire architecture. Each operation in the matrix is enabled only when both operands are truly available. This implies that validity logic must be consistent and locally reliable, regardless of N.

The **pe_matrix** module ensures that each node gets:

- a valid value if the source actually exists;
- a value with `valid = 0` if the source lies outside the matrix (for non-toroidal topologies);
- a real value coming from the "wrapped" node in **D-TORUS**.

This is essential in order to avoid unexpected behaviour. Without such mechanisms, a **PE** may perform instructions with a single valid operand, resulting in nonsensical results. The structure of the module is such that every node operates only on complete information, avoiding hazardous intermediate states.

The computation of `data_valids` is done each clock cycle and is fully synchronized. This means that, even in the presence of internal backpressure (due to delays somewhere else in the matrix), validity information remains constant and does not introduce spurious glitches.

3.8 Implementation Considerations: Regularity, Area, and Timing

The matrix has been designed from the beginning with regularity as a core requirement. In hardware architectures, regularity is one of the most important properties because it enables:

- more stable synthesis results;
- more uniform placement;
- better prediction of critical paths;
- improved portability across different technologies.

The matrix is characterized by an extremely regular structure, since every PE is identical and the interconnections follow a repeating pattern. These features make it well suited for ASIC implementations, because physical design tools can reproduce placement and routing in a very simple and intuitive way.

Lighter topologies also help to keep the overall area under control. The **FULL** Topology yields a very large number of interconnections also requiring much input. Multiplexers with heavy fan-in. In both **D-MESH** and **MESH4**, the number of inputs per PE is fixed and small, offering a very manageable fan-in. **D-TORUS** introduces a few extra links but remains considerably more tractable than FULL.

3.8.1 3.8.1 Combinational Logic and Critical Depth

The **pe_matrix** module is carefully designed to avoid deep combinational paths. The following tools implement selector decisions, validity propagation and source identification through relatively simple assignments. There is no internal pipeline, since the PEs themselves form a natural pipeline: each node waits for its two operands to become valid and then generates a stable outcome.

This significantly reduces the risk of timing issues and allows the matrix to run comfortably at relatively high frequencies. Moreover, the combinational path between `sel_a`/`sel_b` and the values delivered to each PE is so short that reconfiguration happens almost instantly, without adding unnecessary latency or undesired side effects.

3.8.2 Real-Time Behaviour and the "Border Effect"

An often overlooked aspect in CGRAs is the effect of matrix borders: PEs at the edges have fewer neighbours, which can lead to non-uniform behaviour. In the proposed design, this problem is completely eliminated in FULL and D-TORUS topologies because:

- in FULL, each PE sees all other nodes → no borders exist;
- in D-TORUS, borders are wrapped → the matrix behaves as a closed torus.

In both **MESH4** and **D-MESH**, the problem is minimized by setting null values with `valid = 0` for any missing neighbors. This keeps the behavior of the matrix regular and predictable, preventing the PEs from performing computations on spurious or undefined data.

This is essential for the architecture to be deterministic, and it allows PEs to remain perfectly homogeneous: every node follows identical local rules, irrespective of its position.

3.8.3 Separation Between Configuration and Computation

Another important design decision is the rigid division between configuration (selectors, opcodes, validity) and computation (PE operations). This separation provides two main benefits:

- PEs are simplified, since they only focus on performing operations, not on deciding where data comes from;
- The matrix becomes more reusable because data routing can be changed without changing the computation logic whatsoever.

In practice, the `pe_matrix` module behaves like a large parametric multiplexer network that decides which values are presented to each PE, while the PEs themselves remain focused solely on arithmetic and logical operations.

3.8.3 Practical Implications for Matrix Usage

Beyond purely structural aspects, it is useful to consider how the matrix behaves in real workloads. Since each PE operates autonomously and reacts only to the availability of its operands, the matrix can assume very different dynamic configurations depending on the application.

When the work contains a very regular structure—say, filters, convolutions, or iterative update patterns—the network moves in a nearly rhythmic way, validity signals flowing smoothly and PEs executing comparable operations step by step.

In more irregular situations, like sparse dependencies or graph-shaped computation patterns, FULL and D-MESH help data spread more quickly across the matrix. MESH4, instead, imposes a more controlled behaviour, almost "guiding" the flow so it stays orderly.

The important point is that selector logic remains the central control element: it is the selectors that determine which sources each PE uses in a given computation cycle. This keeps the matrix independent from any specific algorithm and makes it extremely adaptable to various problem classes.

3.9 Chapter Conclusions

The previous sections have shown how the CGRA design is structured around a set of precise choices: modular components, structural regularity, clear separation of responsibilities, and a parametric matrix. Each level of the architecture plays a specific role:

- the DMA feeds the pipeline with coherent, ordered data through AXI-Stream;

- the frame loader reconstructs 2D planes and guarantees continuous operation via the ping–pong mechanism;
- the Processing Element executes simple but essential operations, keeping the pipeline short and easily replicable;
- the parametric $N \times N$ matrix merges computation and interconnection, natively supporting four different topologies.

The introduction of configurable topologies makes the CGRA an extremely useful platform both for architectural exploration and for comparative analysis of structural choices. In particular, the differences among FULL, MESH4, D-MESH, and D-TORUS highlight how the balance between connectivity and complexity is central in parallel computation. On one side lies maximum flexibility; on the other, maximum regularity. The architecture developed in this project allows exploring both extremes (and intermediate points) with a single RTL codebase.

Overall, the CGRA shows a mature balance between area, scalability, and predictability. The choice of using small, easily replicable blocks and a fully parameter-generated matrix reflects a design philosophy oriented both towards research and towards potential integration in more complex systems. The foundations laid in this chapter will be further developed in the following ones, dedicated to functional verification, synthesis, and quantitative analysis of the obtained results.

Capitolo 4

Functional verification and testing methodology

4.1 Functional Verification and Testing Methodology

Functional verification has been, without doubt, the most "invisible" part of the whole work: the section where nothing looks particularly exciting from the outside, yet where one truly understands whether the architecture holds together or not. As long as everything lives on paper or inside RTL files, it is surprisingly easy to believe that each block will behave exactly as intended. The real challenge begins when the DMA, the frame loader, and the parametric matrix are connected. At that point, even a tiny detail can generate behaviours that no one expected. For this reason, a significant amount of time was devoted to simulation and the observation of how data flow moves inside the system. The goal was to reproduce, as faithfully as possible, what would happen in a realistic scenario—where values do not always arrive in perfect order, and where the design needs to withstand stalls, misaligned signals, and all the small irregularities that naturally occur in a computing pipeline.

4.1.1 Verification Goals

The aim of verification is not simply to confirm that the design "works." That is a shallow objective and, in practice, nearly useless. Instead, the goal is to understand how the system reacts under different conditions, including those slightly irregular situations that often occur in real hardware. The CGRA consists of modules which, taken individually, are not particularly complex. Yet when combined, they create an emergent complexity that goes well beyond the sum of their parts.

The main goals of the verification phase were:

- ensuring that the AXI-Stream DMA produces a coherent and protocol-compliant flow, with a particular focus on the correct handling of `TLAST`;
- checking that the frame loader reconstructs the GEN0 and GEN1 planes without dropping values or terminating a plane too early;
- verifying that the ACTIVE/NEXT double-buffering mechanism does not introduce ambiguous states during the swap phase;
- confirming that valid propagation inside the matrix follows the rules imposed by the selected topology;
- observing that Processing Elements never operate on "dirty" or invalid values, especially those located at the edges of the matrix.

Another important objective was to confirm that the architecture remains stable while the algorithm switched between different parameter configurations. As the design reshapes itself when either the dimension **N** changes or the **topology** changes, the verification must ensure correctness across all these combinations.

4.1.2 Simulation Environment

For the simulation environment, I deliberately chose a straightforward approach. Instead of relying on large verification frameworks, I opted for a handcrafted testbench, which gives complete control over the signals and makes every behaviour immediately visible. The testbench instantiates the `cgra_single_top` module and a simple DMA emulator. The goal was not to mimic a full production setup, but rather to build a clear and readable environment where anomalies stand out instantly.

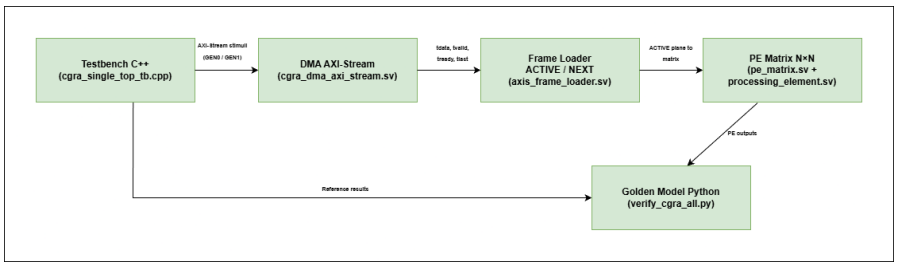

Figura 4.1: Overview of the simulation environment. The C++ testbench based on Verilator instantiates the CGRA top module, provides AXI-Stream stimuli via a DMA emulator, and monitors key Control signals, while a Python-based golden model is utilized for its validation.

The environment includes:

- a stable clock common to all modules so that cycle-by-cycle examination of the pipeline;
- a synchronous reset, active only in the first cycles;
- a source module that emulates the **AXI-Stream DMA**, generating two $N \times N$ planes: **GEN0** and **GEN1**;
- monitors for the main frame loader signals: `tready`, `planes_ready_o`, the `active_bank_q` bit, and any possible error outputs;
- a dump of the main signals in **FST** format, chosen for its efficiency and readability.

One behaviour that I explicitly wanted to test involved the DMA reaction to a low `tready`. Since the frame loader occasionally needs time to store incoming values, it can generate backpressure. The simulations showed that the DMA properly freezes its output when `tready` goes low, exactly as the AXI-Stream protocol requires. This might look like a small detail, but it is crucial: even a single misplaced advance of the DMA would shift the entire GEN0 or GEN1 plane, producing an input map entirely different from the intended one.

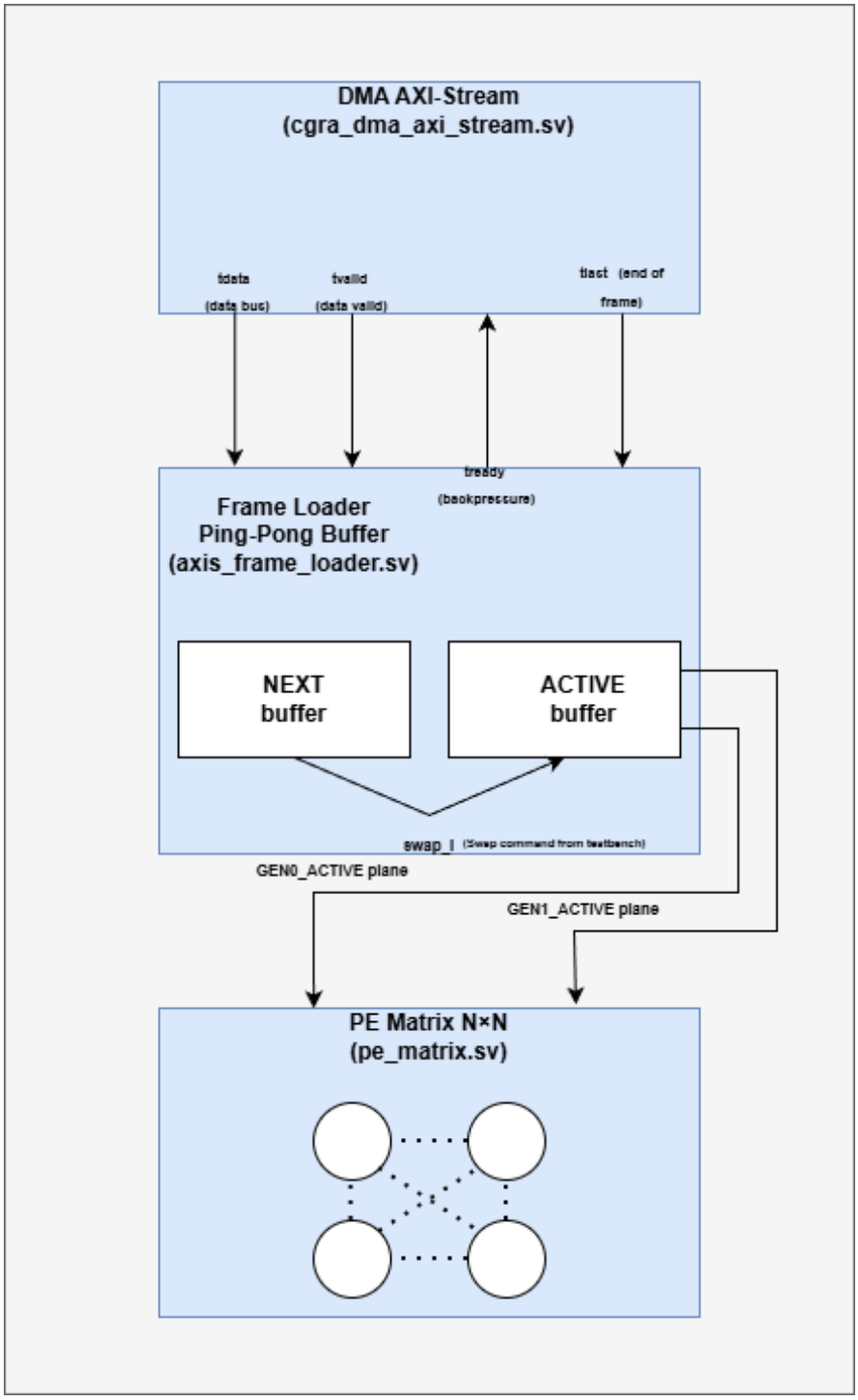

Figura 4.2: AXI-Stream data path from the DMA emulator to the CGRA. The DMA generates tdata, tvalid and tlast, while the Frame Loader asserts tready and reconstructs the incoming frame into the ACTIVE and NEXT buffers. Once a frame is complete, the ACTIVE buffer is fed to the PE Matrix for computation.

4.1.3 Input Pattern Generation

Pattern generation looked like a secondary task at first, but it quickly became one of the most important aspects of the verification. If input data are not sufficiently recognisable, identifying errors becomes unnecessarily hard. Random numbers, for instance, are rarely helpful: seeing a "12" where a "13" was expected does not immediately communicate whether something is wrong or not.

For this reason, different pattern types were used:

- **Incremental sequences**, which make even a single-cycle misalignment immediately visible;
- **Patterns tailored for border testing**, especially useful in MESH4 and D-MESH, where some neighbours do not exist and must be replaced by null values with `valid = 0`;
- **Region-divided patterns**, where different numerical areas exist within the same plane, useful for observing how information spreads across highly connected topologies such as FULL.

This variety made it possible to distinguish routing-related issues from logical problems or from effects simply caused by the chosen topology.

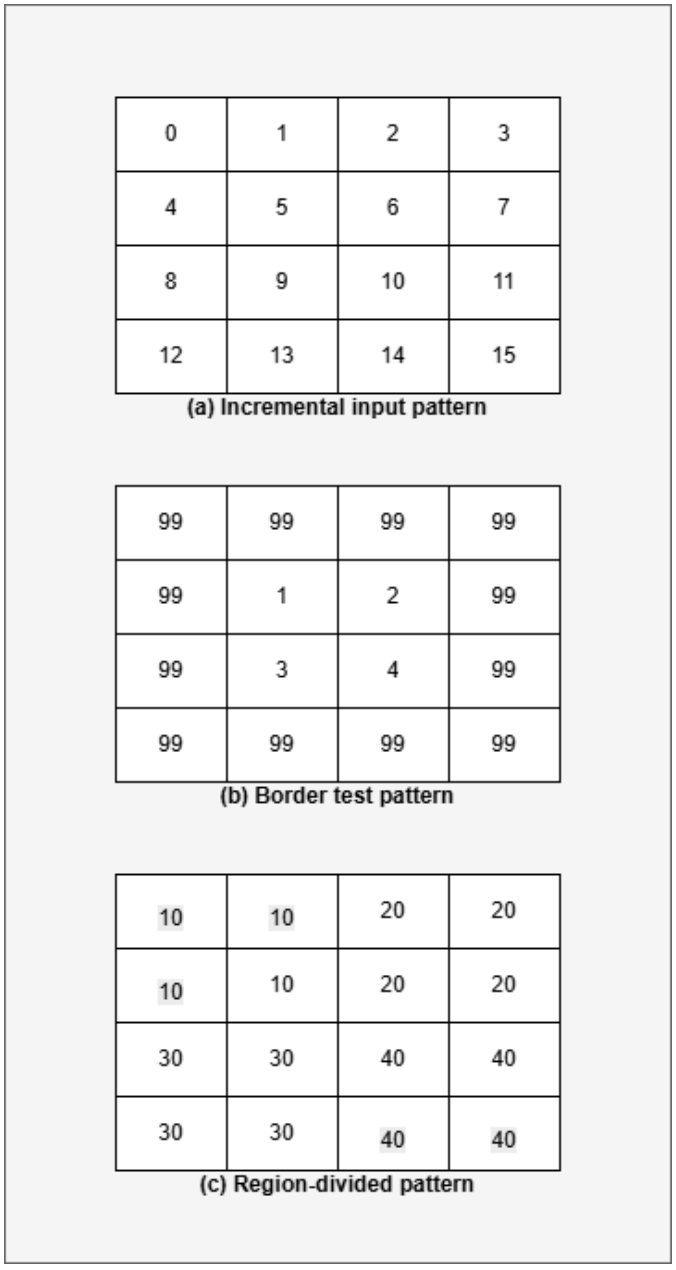

Figura 4.3: Example input patterns used during functional verification. Incremental, border-based, and region-divided patterns are employed for Highlight alignment issues, boundary conditions and reconstruction problems. in the Frame Loader as well as along the data path.

4.1.4 Logging and Data Tracing

The logging system was designed to be meaningful rather than verbose. Only signals that genuinely help interpret the behaviour of the system were recorded: `planes_ready_o`, `active_bank_q`, error signals from the frame loader, and, inside the matrix, the propagation of valid.

The evolution of the valid signal is, more than anything else, the key indicator of how the topology shapes the movement of data. Each PE can update its value only if its neighbours provide valid inputs, and following these paths allowed a clear understanding of whether the logic behaved as expected.

4.1.5 Verification Scripts and External Tools

To avoid repeating long sequences of manual tasks, I integrated several Python scripts into the verification workflow. These scripts made it significantly easier to prepare the inputs, analyse the outputs, and compare the behaviour of multiple simulations.

Their main functions are:

- Creating AXI-Stream sequences starting from a Python matrix, placing `TLAST` in the correct position automatically;
- Analysing the result produced by the CGRA and comparing it against a Python-based golden model mimicking the PE logic;
- Automatic comparisons between the simulations for different **N** or topology configurations.

The golden model was extremely useful, since it re-implements the PE logic with full visibility and without pipeline constraints. In case of a mismatch, the scripts save a small log with the row, column, expected value, and actual value, making it unnecessary to manually browse the FST file to locate the source of the issue.

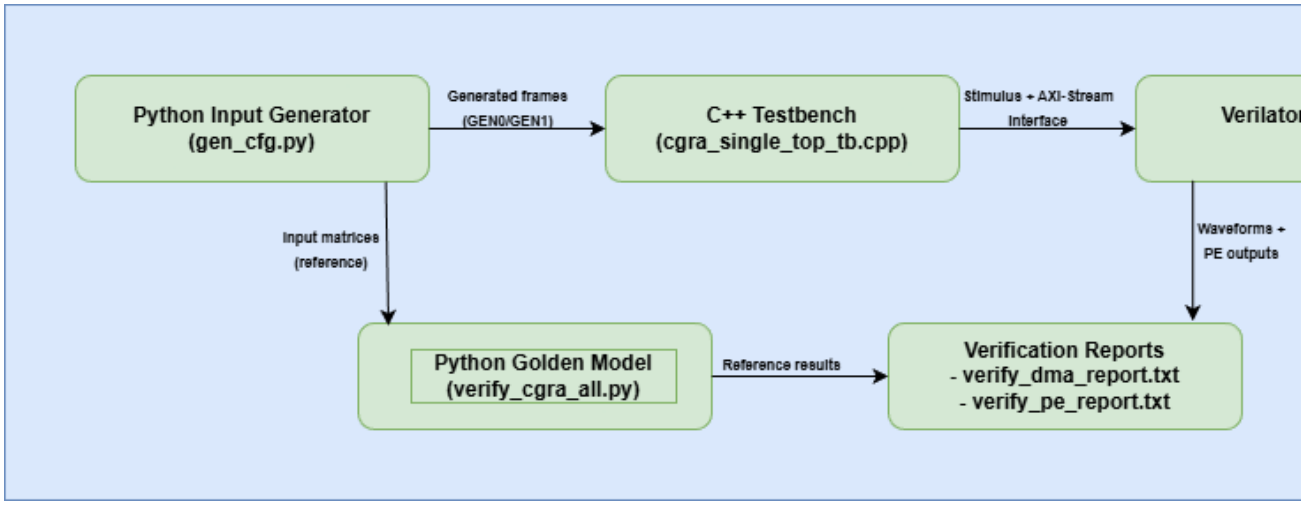

Figura 4.4: Verification workflow. Python scripts generate input frames, the C++ testbench drives the Verilator simulation, and a Python-based golden model validates the CGRA outputs and produces verification reports.

4.1.6 Functional Verification Results

The verification highlighted several interesting dynamics which, while not errors, helped shed light on the internal behaviour of the architecture. The frame loader consistently handled the transition from GEN0 to GEN1, with both the word counter and `TLAST` behaving as expected. The only nuance to keep in mind is the slight temporal shift caused by backpressure: when `tready` is low, the AXI flow pauses, and this may delay the precise moment when the ACTIVE/NEXT swap occurs. This is not a bug, but simply a natural outcome of how AXI-Stream works.

About the matrix, the simulations have confirmed what RTL suggested: each topology generates its unique characteristic propagation behavior. In the **FULL** configuration, valid spreads quickly and almost uniformly. In **MESH4** it moves more slowly and is strongly dependent on the starting position, whereas in **D-TORUS** the wrapping effect makes the matrix much more uniform, as edge cells receive inputs from the opposite side.

Another interesting observation concerns PE cells located at the corners. In MESH4, they receive null inputs where neighbours do not exist; in D-TORUS, they instead receive valid data coming from the opposite edges. Verification confirmed that both behaviours are handled correctly, and that no PE performs operations on invalid data.

The comparison with the golden model did not reveal structural errors. A few small mismatches emerged when using particularly unusual patterns, but

repeating the same tests with more regular inputs made those differences disappear, confirming that they were not design issues.

4.1.7 Final Considerations on Verification

The verification phase proved to be essential in validating the architecture, especially in those areas where modules with different roles interface with each other. Parametric CGRAs are naturally prone to indexing mistakes, missing connections, or incomplete topology selection signals. The fact that the system behaved consistently across several structured simulations is a strong indication of the soundness of the overall design.

One of the most positive aspects to emerge was the robustness of the pipeline despite stalls, slight delays, or imperfectly aligned signals, the sys- TEM preserves data coherence. At the same time, verification showed that certain topologies—**D-TORUS** in particular—offer more uniform data propaga- tion compared to **MESH4** or **D-MESH**, where position strongly affects timing and flow.

In conclusion, this phase did more than simply confirm that the implementation works. It provided a deeper understanding of how the CGRA behaves internally and revealed dynamics that were not immediately evident when reading the RTL alone. Overall, the design shows a level of consistency and stability that justifies moving confidently to synthesis and implementation analysis.

Capitolo 5

5.1 Synthesis Flow and Implementation Considerations

The transition from functional verification to synthesis is one of the key turning points of the whole project. As long as everything stays in the simulation world, every signal can be stopped, inspected, and interpreted calmly, and there are no real physical constraints to worry about. Once the design enters the synthesis phase, however, the perspective changes: RTL modules are translated into standard cells, timing constraints are applied, and the architecture starts to resemble what it would look like in a concrete ASIC implementation. Even though this CGRA has been designed from the beginning as a modular and parametric system, synthesis is always the moment when one discovers how "robust" the design really is, how much logic hides behind each connection, and how strongly the topological choices affect area, timing, and the overall quality of the result.

5.1.1 Synthesis Goals

The synthesis phase was guided by two main objectives. On the one hand, it was necessary to verify the physical feasibility of the design; on the other, the goal was to compare the different topologies and understand how the structure of the PE graph impacts area and timing. Synthesis was not used as a tool to squeeze out every last nanosecond or to reach extremely high operating frequencies. Instead, the intention was to observe how the architecture behaves when it is mapped onto real cells of a 65 nm technology, with its actual delays and area cost.

More concretely, the goals were:

- getting a full implementation of **cgra_single_top** for each topol- ogy;
- checking that the design is synthetically clean — no unintended latches, no unresolved combinational paths, no critical warnings;

- analyzing the overall area, combinational versus sequential logic split, and the impact of the registers compared to the PEs and the frame loader;
- evaluating slack and identifying critical paths, especially those located inside the matrix;
- building a fair comparison among **FULL**, **MESH4**, **D-MESH**, and **D-TORUS** while keeping the RTL code completely the same.

This last point is arguably the most important. The fact that the matrix and all its connections are generated through parameters and `generate` constructs ensures that the comparison among topologies is truly fair. There are no behavioural differences in the source code, only structural differences derived from the value of the `TOPOLOGY` parameter.

5.1.2 Synthesis Environment

The synthesis flow is based on Synopsys Design Compiler, using standard-cell libraries representative of a 65 nm CMOS technology. The project already includes the required `.db` files, so it was sufficient to organise a Tcl script that performs the main steps: RTL reading, library setup, constraint application, optimisation, and final report generation.

The environment is composed of:

- standard-cell libraries for logic and flip-flops;
- memories modelled as synthesised RAMs (the internal memories of the frame loader are the most delicate part, since in the absence of real SRAM macros Design Compiler implements them as arrays of registers);
- a script split into several blocks (`set_libs.tcl`, `set_constraints.tcl`, `dc_script.tcl`) that coordinates the whole process.

The timing constraint is the same for all topologies, so that no artificial advantage is introduced for any configuration. In all the reports, design reveals a small negative slack, of the order of one nanosecond. This should not be interpreted as a structural defect: it is just a consequence of the fact that the design has not been aggressively optimized or deeply pipelined in its most critical points. Here the goal is not maximum peak performance, but understanding how the internal structure affects the critical paths.

A crucial characteristic of the synthesis setup is that the flow is repeated under exactly the same conditions for each topology. The only parameter that changes is `TOPOLOGY`, and the fact that the scripts automatically generate separate reports for $N = 8$ makes it possible to build a homogeneous and reliable comparison.

5.1.3 Impact of Topologies on Synthesis

The most interesting aspect of the synthesis process concerns the impact of the topologies on total area and on the complexity of the internal routing. The four available topologies – FULL, MESH4, D-MESH, and D-TORUS – differ not only in the logical "shape" of the connections, but also in the actual number of signals and multiplexers required to route the inputs to each PE.

The **FULL** topology is, as expected, the most expensive in hardware terms. Each PE can receive inputs from practically every relevant direction and the selection logic has to consider more possible sources. This is clearly reflected in the reports: in FULL the total area exceeds one million units, with marked increase compared to the other topologies. This is not surprising: even if the PE itself is simple, having more inputs means more multiplexers, more wires, and more combinational load.

On the other end of the spectrum, **MESH4** is the lightest topology. Each PE is connected only to its four orthogonal neighbours and the border logic is relatively simple: external nodes receive null values with `valid = 0`, which keeps the selector logic small. As a result, the total area is significantly lower than in FULL and the critical path is less congested.

The D-MESH topology introduces only the diagonal connections, increasing connectivity without reaching the full complexity of FULL. The reports show that the area of D-MESH is slightly higher than that of MESH4, but it never reaches the values observed in FULL. This makes D-MESH a reasonable compromise between flexibility and cell cost.

Finally, the **D-TORUS** topology adds wrapping at the borders. From a pure logically speaking, wrapping does not increase the number of inputs by much beyond **D-MESH**, since the number of sources remains similar. However, the routing of signals becomes more involved and would introduce longer paths in a real layout, increasing physical complexity. In RTL synthesis, however, D- TORUS is not heavily penalized and the total area remains close to that of D-MESH.

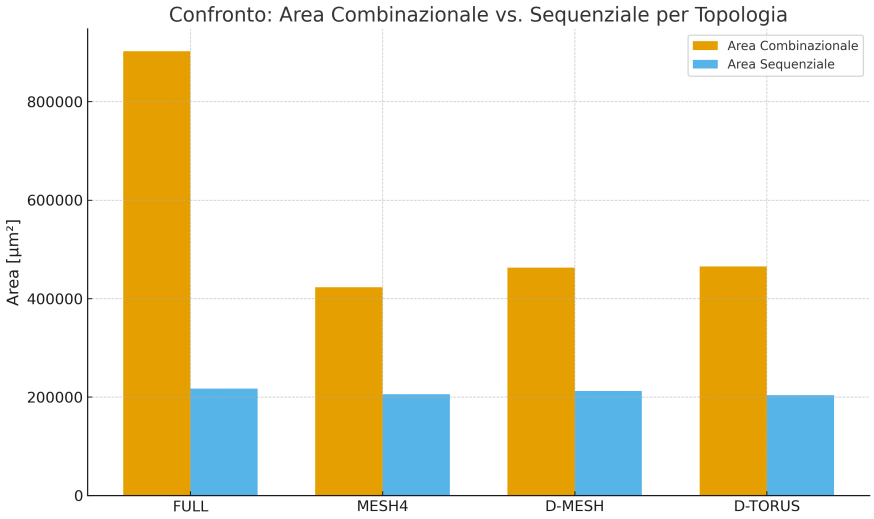

Figura 5.1: FULL indeed features a much larger total cell area due to its dense network of interconnects, which are implemented through a large amount of multiplexers and routing logic. By contrast, the MESH4, D-MESH, and D-TORUS topologies feature much smaller and similar areas; the reduction compared to FULL is more than 40% in all three cases.

What clearly emerges from the reports is that the raw computational block of a CGRA – the PE itself – weighs relatively little. The real difference comes from the interconnection fabric. The richer it is, the more one pays in multiplexers, combinational networks, and, ultimately, in critical paths.

5.1.4 Analysis of Synthesis Results

Once the four variants of the design (one per topology) were generated, the first striking element was the difference in area. Even without looking at the numerical reports, it is intuitive that a highly connected topology will produce a heavier internal network. However, only the concrete figures make this intuition precise. The FULL topology, for example, largely exceeds one million area units, whereas the other three configurations are around 600 to 700 thousand. Such a gap does not come from the PEs, which are identical in every variant, but from the interconnection logic, which is far more numerous and articulated in the FULL case.

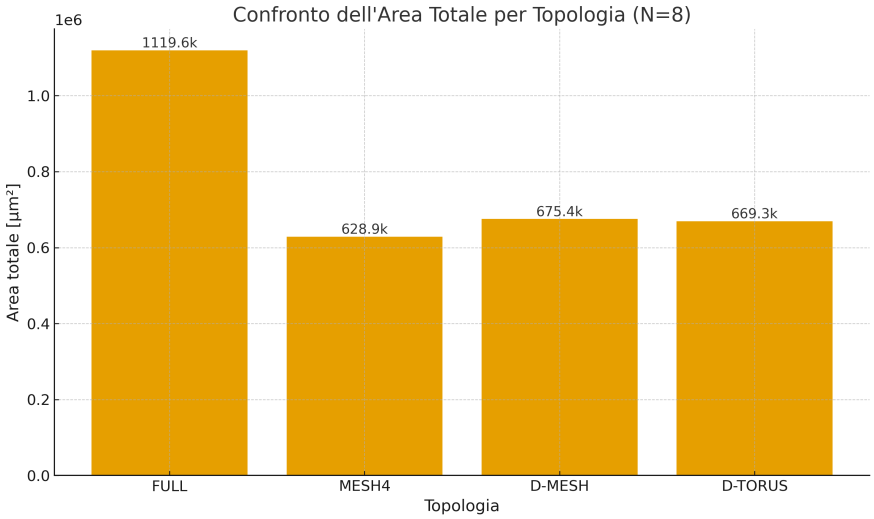

Figura 5.2: The combinational logic clearly dominates the total in all cases, especially in the FULL topology where it accounts for over 80% of the total. Sequential area remains roughly constant between configurations and serves to confirm that the main source of overhead is indeed the interconnection fabric rather than the processing elements themselves.

Absolute area does not in itself give any indication about quality. For this reason, the relative weight of combinational logic and registers was also considered. The reports clearly show that in **FULL** the combinational portion grows significantly: more inputs per PE mean more multiplexers, more paths and more selection networks. In **MESH4**, the combinational logic is cleaner and more regular. Each PE only has four main sources, with the rest null values at the borders. The resulting matrix is less flexible from a topological stand- point, but it synthesises more compactly & produces a more contained internal structure.

With respect to slack, all topologies show similar negative values, within a relatively narrow range. This means that the global complexity of the architecture does influence timing, but not as dramatically as it impacts area. The main critical paths appear to be concentrated around the input selection logic of the PEs, rather than in the arithmetic core itself, which is simple and inexpensive. In other words: the PEs “cost little”, while the interconnection network “costs a lot” in timing terms.

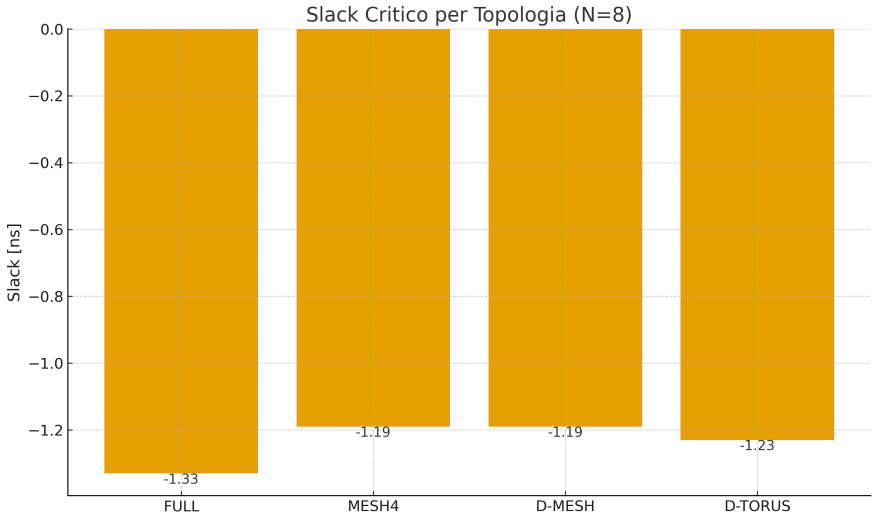

Figura 5.3: All topologies present a little negative slack value, since no aggressive timing optimization was applied. FULL is the most timing-constrained topology owing to its deeper logic levels and higher fan-in. MESH4, D-MESH, and D-TORUS present very similar slack values, reflecting their lower combinational complexity.

An interesting detail is that D-TORUS, despite using wrapping at the borders, does not show a dramatic slack degradation compared to D-MESH. This suggests that wrapping, while introducing longer connections from one side of the matrix to the other, does not heavily affect the logical critical path in a purely RTL context. In a full physical implementation, where geometric distances matter, D-TORUS would likely be more challenging in terms of placement and routing. RTL synthesis does not see the physical chip dimensions, so this aspect only appears partially in the reports.

Another area of interest is the most "fragile" part of the design – the **frame loader**. In all topologies, the loader accounts for a moderate fraction of the area, yet it is still a crucial zone since it applies internal memory without optimised macro-RAMs. The result is a dense area of flip-flops that contributes to the total area. However, its behavior is stable and the dominant critical paths are almost always associated with the **PE network**, not with the loading logic.

Overall, the analysis of the synthesis results confirms a pattern that often appears in CGRA designs: area grows with connectivity, and timing tends to degrade not because of the computations themselves, but because of the proliferation of multiplexers and selection signals required to support multiple routing options.

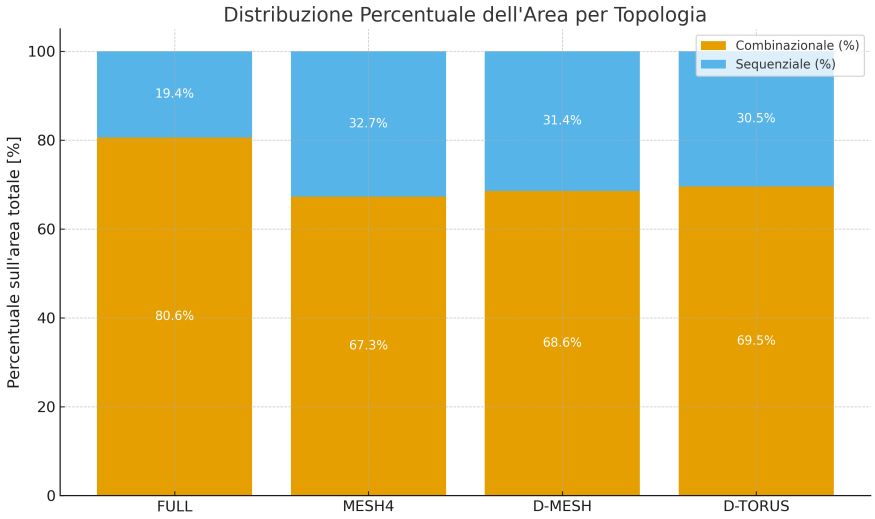

Figura 5.4: If FULL topology devotes about 80% of the area to combinational logic, this really underlines the heavy impact of its extensive connectivity. The other topologies have a more even distribution, where combinational logic occupies something like 67–70% of the area, so they are much more structurally efficient and easier to scale.

5.1.5 Scalability and Implications for a Real ASIC Implementation

Scalability is one of the crucial aspects of a parametric CGRA architecture. If the results for $N = 8$ already show significant differences among topologies, increasing the dimension makes these gaps even more evident. Doubling the matrix dimension means quadrupling the number of PEs and, at the same time, multiplying the complexity of the interconnection network. It is easy to imagine that FULL becomes quickly impractical from a physical viewpoint, while more contained structures such as MESH4 or D-MESH remain manageable even for larger values of N.

From a real ASIC implementation point of view, three aspects are par- ticularly important:

- **Physical routing (place & route).** A topology like **FULL** implies connections between PEs that, though "free" in RTL, have long physical wires in silicon. It is comparatively easy to connect a cell to its four neighbours; connecting it with many more points needs longer tracks, crossing nets the matrix, and higher congestion. This translates into real delays that do not appear in pure RTL reports.

- **Power consumption.** More multiplexers and more signals mean not only more area, but also more switching activity. In a real context, where dynamic power is a constraint, choosing a topology with excessively high connectivity can soon become very unaffordable. **MESH4** and **D-MESH** tend to behave better in this respect.

- **Timing stability.** RTL synthesis ignores physical distances, whereas physical synthesis does not. It is reasonable to expect that D-TORUS, despite having acceptable logical timing, would suffer more in place & route. Wrapping introduces connections between opposite sides of the matrix, which, on silicon, correspond to cells placed far apart. This cost emerges fully only during the physical stages.

On the positive side, the PE architecture itself is very light. Its simplicity and linear structure avoid major bottlenecks, and the absence of complex operations helps keep power consumption under control, even for medium-to-large matrices. The real limiting factor is therefore the connection fabric.

The overall picture that emerges is that the design is well suited to large-scale experiments, provided that a topology is chosen in line with the target application. FULL is extremely powerful but very expensive; D-MESH and D-TORUS offer interesting trade-offs; MESH4 is the most predictable and scalable. Knowing in advance how these choices affect area and timing makes it possible to build an architecture that can be adapted to different scenarios while preserving a single, unified RTL code base.

Capitolo 6

6.1 Experimental Results and Critical Discussion

The experimental evaluation phase is a fundamental step, because it shows how the design behaves beyond theory and functional simulation. As long as one stays in the RTL domain, it is possible to form a reasonably clear idea of what will happen, but only a complete synthesis flow provides a credible picture of the real impact of the architecture in terms of area, timing, connectivity, and overall quality of the result. In particular, a direct comparison between the four implemented topologies makes it possible to understand which design choices are truly sustainable and which ones, although appealing from a purely theoretical point of view, turn out to be too expensive once the design is mapped onto a physical technology.

In this chapter, three aspects are analysed: the comparison between topologies, scalability with respect to the matrix dimension N, and a discussion of the current limitations of the design, together with some reflections on possible future extensions. The goal is not just to display numbers, but also to interpret them, so as to build a more solid understanding of how the CGRA actually behaves when the internal network structure changes.

6.1.1 Objectives of the Experimental Analysis

The experimental analysis was structured to achieve two main goals. On the one hand, it aims to provide an objective evaluation of the area and timing costs of the four topologies. On the other hand, it seeks a more qualitative understanding of *why* these differences appear. It is not enough to say that one topology is "larger" or "slower": what really matters is identifying which part of the architecture is responsible for that difference, and whether it is a structural limitation or something that could potentially be optimised.

More concretely, the objectives of the analysis can be summarised as follows:

- comparing total area, the distribution between combinational logic and registers, and the impact of individual sections (PEs, loader, interconnection network);
- observing slack and identifying the slowest paths inside the CGRA;
- evaluating how the behaviour changes when the topology increases or decreases connectivity;
- understanding which structures are more efficient and which ones quickly become impractical in a realistic ASIC context.

A further, and perhaps even more important, goal is to provide a clear picture of how effective the parametrisation of the design actually is. Since the matrix is generated completely through dynamic constructs, the experimental results do not depend on ad hoc code written for each topology, but on a single hardware description that reconfigures itself automatically. This is a significant advantage and allows the impact of structural choices to be evaluated in a much more neutral and controlled way.

6.1.2 Comparison Between Topologies

The comparison between topologies is at the core of the experimental analysis. The synthesis reports reveal clear and, in many respects, intuitive differences: FULL is the heaviest, MESH4 is the lightest, and between them lie D-MESH and D-TORUS, which increase connectivity without reaching the extreme complexity of FULL.

The FULL topology, with more than 1.1 million units of area, represents the worst case in terms of cost. The logic required to manage the multiplicity of connections is substantial: each PE must be able to select among many possible sources, which leads to a proliferation of multiplexers and combinational paths. The reports clearly show that the percentage of combinational area grows significantly with respect to the other topologies, and timing degrades accordingly. It is not surprising to observe negative slack on the critical paths, since such a rich network naturally tends to produce delays that are difficult to optimise.

MESH4, by contrast, stands at the opposite end of the spectrum. With around 620–630 thousand area units in total, it is the most compact and the most regular topology. Connectivity is limited to the four orthogonal neighbours, and the border nodes behave in a very simple way, thanks to automatically inserted null values. The result is a stable and predictable network that requires fewer resources and produces shorter critical paths. From an ASIC perspective, this is the topology with the best chances of scaling cleanly to larger matrix sizes.

D-MESH acts as a compromise: it adds diagonal connections, providing a richer data flow without introducing the excessive complexity of FULL. The

synthesis reports show that area increases with respect to MESH4, but remains well below the values of FULL. Timing is also affected, but only moderately, since the combinational logic grows without exploding.

D-TORUS exhibits a behaviour very similar to D-MESH in terms of area. Its distinctive feature is the wrapping at the borders, which allows PEs at the edges to "see" the nodes on the opposite side. In RTL synthesis this has a relatively small cost, because it mainly consists of adding a few extra connections. In a real physical implementation, however, these connections would span the entire layout and have a much more significant impact. It is interesting to note that, despite its higher conceptual complexity, the area of D-TORUS remains close to that of D-MESH, indicating that wrapping alone has limited influence at the pure logic level.

Overall, the comparison shows that the cost of the CGRA is not primarily determined by the PEs themselves, which are extremely light, but by the interconnection fabric. Each increase in connectivity translates almost linearly into a larger number of inputs, multiplexers, and combinational paths surrounding the PEs. This phenomenon is particularly evident in the complete reports: in FULL, the multiplexer network becomes so large that it accounts for almost half of the total area.

6.1.3 Scalability with Increasing N

One of the most important aspects of a parametric CGRA architecture is understanding how it behaves as the matrix size grows. Even though synthesis tests were conducted for $N = 8$, it is fairly clear that some of the trends observed will scale in a predictable way, while others will degrade more abruptly.

The first point to consider is that the matrix contains N^2 PEs. Increasing N from 8 to 16 means going from 64 to 256 PEs, a factor of 4. This quadratic growth makes it immediately clear that the densest topologies quickly become difficult to sustain. If FULL already requires more than 1.1 million area units for $N = 8$, it is reasonable to expect it to easily exceed 4 million for $N = 16$. This does not only mean more area, but also a dramatic increase in internal routing complexity, with a strong impact on timing and power consumption.

The MESH4 topology, on the other hand, scales much more gracefully. Each PE maintains exactly the same number of connections as the matrix grows. As a consequence, the area increases in proportion to the number of PEs, without additional explosions caused by the interconnection fabric. In application scenarios where extremely rich connectivity is not required, MESH4 is the most sensible choice for building large matrices.

A similar argument holds for D-MESH. Its growth is slightly steeper than that of MESH4, because each PE has more connections, but the overall behaviour remains

reasonably predictable. D-TORUS occupies an intermediate position: from an RTL point of view it scales similarly to D-MESH, but in a real physical context the "wrap-around" connections would have a strong impact on routing, introducing longer physical paths and consequences for the maximum achievable frequency.

Another important factor concerns the internal memory required by the frame loader. This memory also grows proportionally to N^2, and since it is implemented with synthesised flip-flops, its area weight becomes increasingly significant. While for $N = 8$ the internal memory remains manageable, for larger values a real ASIC implementation would almost certainly require macro-SRAM blocks to avoid excessive resource usage.

In summary, scalability is not limited by the PEs, but by the network that links them together. The PEs are simple enough to be replicated many times, whereas the selection logic and multiplexers often grow faster than N^2. This makes it clear that, in a concrete design scenario, the lighter topologies are the ones that guarantee a more stable evolution of the architecture.

6.1.4 Discussion of Design Limitations

Like any reconfigurable architecture, this CGRA exhibits a number of inevitable limitations that become evident when one considers synthesis and scalability.

The first limitation concerns the double buffering of the frame loader. The logic is solid and functionally correct, but it relies entirely on synthesised memory. As already mentioned, this leads to a non-negligible area cost. For an academic prototype this is an acceptable compromise, but in a real chip the structure would likely need to be revised in order to integrate more compact memory blocks.

A second limitation involves the management of valid signals inside the matrix. In very dense topologies, propagation is extremely fast and tends to produce behaviours that depend strongly on the location of the initial data. This is not a bug, but a natural consequence of a highly connected network: a value can "travel" across the matrix in just a few cycles, which sometimes makes simulation results harder to interpret. In a real application context, the flow of data would need to be orchestrated carefully, in order to avoid situations where dependencies spread too quickly and create unintended effects.

Another limitation is the overall timing performance. All topologies show negative slack, which clearly indicates that, without specific optimisations, the design is not capable of operating at very high frequencies. This is a fairly common outcome for architectures where PEs are connected purely combinationally. To improve timing, some form of internal pipelining would be required, for example by inserting registers between PEs or by splitting the network into multiple stages.

Finally, there is a more conceptual limitation related to the difficulty of evaluating real applications without a dedicated compiler or mapping tool. The CGRA is fully

functional from the hardware point of view, but without a system that translates high-level operations into topology configurations and data flows, its potential cannot be fully exploited. This is not a defect of the design itself, but it does represent a practical barrier when trying to use the architecture in a concrete scenario.

6.1.5 Possible Future Extensions

Despite the limitations highlighted, the architecture is an excellent starting point for future developments. A first natural extension would be the introduction of real macro RAMs in the frame loader. This would not only reduce the total area, but also improve timing and make it easier to handle larger matrices.

Another promising direction involves optimising the intermediate topologies. At the moment, the implemented topologies are static and predefined. It would be interesting to explore a more dynamic approach, where the topology can be modified at configuration time, similar to how more advanced CGRAs use programmable switch boxes. In this way, one could obtain a connectivity level close to FULL without paying its full cost in area.

The PE operation set could also be extended by adding logical operations or small combinational functions without significantly increasing latency. This would make the CGRA more suitable for real-world applications and would allow experimentation with more sophisticated mapping strategies.

A further improvement concerns timing optimisation. By inserting strategically placed registers, it would be possible to build a more stable pipeline and reach higher operating frequencies. Naturally, this would change the functional model, since the network would no longer be completely combinational, but the potential performance gains could be substantial.

6.1.6 Conclusions on the Experimental Results

The experimental analysis provides a clear picture: the topology structure is the dominant factor in determining area, timing, and internal behaviour of the CGRA. The PEs are light, scale well, and never represent the real bottleneck. The interconnection network is what really makes the difference. Simpler topologies are also the most stable and scalable, whereas richer structures offer theoretical advantages but come with much higher costs in terms of area and combinational depth.

Overall, the design behaves consistently with the original architectural choices and confirms that the parametric approach makes it possible to compare very different variants without modifying the RTL. This property makes the architecture

an excellent starting point for experimenting with new topologies, connection algorithms, and optimisation strategies targeted at realistic scenarios, while preserving a single, unified RTL code base.

Bibliografia

[1] Artur Podobas, Kentaro Sano e Satoshi Matsuoka. «A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective». In: *IEEE Access* 8 (2020), pp. 146719–146743. DOI: 10.1109/ACCESS.2020.3012084 (cit. alle pp. 6, 7).

[2] Rubén Rodríguez Álvarez, Benoît Denkinger, Juan Sapriza, José Miranda Calero, Giovanni Ansaloni e David Atienza Alonso. «An Open-Hardware Coarse-Grained Reconfigurable Array for Edge Computing». In: *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF '23)*. Bologna, Italy: ACM, 2023, pp. 391–392. DOI: 10.1145/3587135.3591437 (cit. alle pp. 7, 9, 11).

[3] Daniel Vázquez, José Miranda, Alfonso Rodríguez, Andrés Otero, Pasquale Davide Schiavone e David Atienza. «STRELA: STReaming ELAstic CGRA Accelerator for Embedded Systems». In: *CoRR* abs/2404.12503 (2024). arXiv:2404.12503 [cs.AR] (cit. alle pp. 8, 9).

[4] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann e Brandon Lucia. «RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture». In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 546–564. DOI: 10.1109/MICRO56248.2022.00046 (cit. a p. 8).

[5] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia e Nathan Beckmann. «SNAFU: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture». In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021. DOI: 10.1109/ISCA52012.2021.00084 (cit. alle pp. 8, 10).

[6] Rohan Juneja, Pranav Dangi, Thilini Kaushalya Bandara, Zhaoying Li, Dhananjaya Wijerathne, Li-Shiuan Peh e Tulika Mitra. «Building an Open CGRA Ecosystem for Agile Innovation». In: *CoRR* abs/2508.19090 (2025). arXiv:2508.19090 [cs.AR], invited paper (cit. alle pp. 8–10).

[7] Satyajit Das, Kevin J. M. Martin, Davide Rossi, Philippe Coussy e Luca Benini. «An Energy-Efficient Integrated Programmable Array Accelerator and Compilation Flow for Near-Sensor Ultralow Power Processing». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.6 (2019), pp. 1095–1108 (cit. alle pp. 9, 11).

[8] Nicolò Carpentieri, Juan Sapriza, Davide Schiavone, Daniele Jahier Pagliari, David Atienza, Maurizio Martina e Alessio Burrello. «Performance Evaluation of Acceleration of Convolutional Layers on OpenEdgeCGRA». In: *Proceedings of the 21st ACM International Conference on Computing Frontiers (CF '24)*. arXiv:2403.01236 [cs.AR]. Ischia, Italy, 2024 (cit. a p. 11).