Politecnico Di Torino

Master's Degree course in
Electronic System

Master's Degree Thesis

A reduction-capable AXI XBAR for fast M-to-1 communication

Supervisors
Dr. Luca Colagrande
Prof. Dr. Luca Benini
Prof. Dr. Maurizio Martina
Prof. Dr. Ahmed Hemani

Candidate
Lorenzo Leone

Academic year 2023-2024

Acknowledgements

I would like to express my gratitude to Professor Benini for giving me the opportunity to join the IIS lab and for consistently reviewing my work on a weekly basis, providing valuable insights that contributed to achieving good results. I am also thankful to Luca C. for proposing this intriguing project to me and for the unwavering support throughout this short period. Your patience in addressing my numerous questions was truly appreciated, and you served as a mentor in every aspect, helping me overcome technical challenges and more. I extend my thanks to all the talented people within the PULP team with whom I had the privilege to exchange ideas and engage in insightful discussions over a cup of hot coffee during the cold days in Zurich. Special appreciation goes to Prof. Martina and Ahmed for imparting their technical knowledge, which was instrumental in my exploration of the fascinating world of ASIC.

As I approach the end of this five-year journey, I am grateful for the opportunity to have crossed paths with many other people whose support has been invaluable. Thank you Gaspare and Federico for the long phone calls and for all the chats we had all together with some communication problems that we somehow managed to overcome. Thank you, Roberto and Andrea, for the time spent on projects, always with smiles and positive vibes that uplifted me. Thank you, Matteo, for the support and life lessons you often imparted. Thank you, Andres, for the amazing time we spent in freezing Stockholm. I am also thankful to the friendly people I met in Zurich, particularly Elisa, for supporting me during fragile times and motivating me to look ahead.

A huge thank you goes to my family, who have always stood by my side, trying their best for me, even during the most difficult times.

Finally, I would like to express my gratitude to a person who has always supported and believed in me, never putting a brake on my desire to study, always prioritizing what was best for me. Thank you, Paola, for the time we spent together.

Abstract

Collective communications are widely used in parallel computing applications and create one of the main bottlenecks in large system architectures due to their overhead. With the rise of Graphic Processing Units (GPUs), Many-core, and Multi-core architectures, the importance of these operations and their impact on the system's energy and performance is continuously increasing.
This project aims to develop a reduction-capable crossbar to allow reduction operations during data transmission, accelerating M-to-1 communications. Regardless of the general-purpose capability of the developed crossbar, an in-depth study was conducted on global barrier mechanisms which can be implemented with reductions. The device was synthesized for GlobalFoundries 12nm with different configurations, leading to a considerable area overhead. Additionally, the crossbar was integrated into Occamy, a system with 216 RISC-V processors, to run various kernels evaluating the performance speedup. Analysing the global barrier under different conditions, the maximum achieved speedup was 6.9, while in the worst condition, the hardware version is still 15% faster than the software. To observe the impact on a real application, the developed feature was exploited in the bitonic sort algorithm, reaching a speedup of up to 2.43.
To our knowledge, this is the first work where reduction capabilities have been integrated inside a crossbar for general-purpose systems.

Contents

List of Figures

List of Tables

List of Acronyms

AMBA . . . . . Advanced Microcontroller Bus Architecture

AXI . . . . . Advanced eXtensible Interface

ALU . . . . . Arithmetic Logic Unit

API . . . . . Application Programming Interface

BFS . . . . . Breadth-First Search

CMP . . . . . Chip Multi-Processors

CPU . . . . . Central Processing Unit

CSR . . . . . Control Status Registers

CUDA . . . . . Compute Unified Device Architecture

DMA . . . . . Direct Memory Access

DUT . . . . . Device Under Test

FIFO . . . . . First-In-First-Out

FFT . . . . . Fast Fourier Transform

FPU . . . . . Floating-Point Unit

GE . . . . . Gate Equivalent

GPU . . . . . Graphic Processing Unit

HBM . . . . . High Bandwith Memory

ISA . . . . . Instruction Set Architecture

LSB . . . . . Least Significant Bit

LSU . . . . . Load-Store Unit

NoC . . . . . Network-On-Chip

PE . . . . . Processing Element

PPA . . . . . Power, Performance, Area

RISC-V . . . . . RISC-V

RTL . . . . . Register Transfer Level

R-XBAR . . . . Reduction-capable XBAR

SIMD . . . . . Single Instruction Multiple Data

SoC . . . . . System-on-Chip

TCDM . . . . . Tightly Coupled Data Memory

XBAR . . . . . Crossbar

Chapter 1

Introduction

1.1 Motivations

In today's rapidly evolving technological landscape, the widespread adoption of heterogeneous systems, such as System-On-Chips (SoC), Many-Core architectures, and Graphic Processing Units (GPU), is reshaping the electronic systems' paradigm. These advancements not only provide unparalleled computational power but also pose significant challenges in managing efficiency and communication among system components.
In this context, "Collective Communications" [1] emerge as a critical element for optimizing performance and fully harnessing the potential of these cutting-edge architectures. A fundamental aspect within this framework is the necessity of a robust communication system, such as a reliable crossbar, to facilitate efficient and rapid communication among various cores and system components.
Numerous application fields today leverage collective communications, especially in reduction operations. An illustrative example is found in AI applications like Deep Neural Networks (DNN), which extensively use these operations to construct neural models. Similarly, Machine Learning (ML) systems often exploit Graphic Processing Unit (GPU) to perform parallel operations that must later be consolidated on a central core. The challenges of large-scale systems become pronounced when considering the exchange of data among masters and slaves. The communication bottleneck introduced by high traffic and limited bandwidth poses a significant hurdle. For instance, in distributed computing environments, the efficient exchange of information between different processing units is essential for seamless coordination and optimal performance, emphasizing the need to address this bottleneck for scalability and effectiveness. Moreover, the increasing demand for computational power is noteworthy. Operations such as reduction are currently executed through software techniques, employing specialized program paradigms like OpenMP [2] or CUDA [3].
An example of reduction operations widely used in parallel computation is barrier synchronization mechanisms. Both aforementioned paradigms allow barriers among

multiple cores, but the synchronization overhead can reduce the theoretical speed-up estimated by Amdahl's Law. Several studies have been done to accelerate software implementations on different architectures [4, 5, 6], and specific hardware supports have been designed for some reduction operations [7, 8, 9, 10].
Introducing the possibility of performing these operations at the hardware level, maintaining the general purpose structure of the system, can potentially alleviate the strain on bandwidth, meeting application demands more efficiently. This shift to hardware-level execution offers a promising avenue to not only enhance performance but also optimize the utilization of computational resources.
In this project, an Advanced eXtensible Interface (AXI) crossbar developed in the PULP team was extended to perform reduction operations during the transmission of data. This approach aims to reduce the exchange of information among cores in a multi-core system, increasing the available bandwidth and the system performance.

1.2 Contributions

The key contributions of this project are:

1. Design of a reduction-capable AXI XBAR for fast M-to-1 communication. The architecture was designed for general purpose system and therefore it can be extended to support different reduction operations. The actual version supports only logic AND that can be used for barrier mechanism.

2. Investigation of the area cost introduced with the reduction logic for GlobalFoundries 12nm technology.

3. Integration of the designed interconnection infrastructure in a real multi-core system based on RISC-V processors further extended to issue reduction request.

4. Analysis of the global barrier mechanism with and without hardware support in different situations and conditions to identify advantages and drawbacks of the hardware support, bottlenecks and limitations.

1.3 Project Overview

This thesis description is organized as follows. In Section 2, all the information and background knowledge collected at the beginning of the project, necessary to fully understand the problem and find a solution, are presented with a focus on related works. Then, in Section 3, the methodology to design the reduction-capable crossbar, together with the testbench implementation, is described, focusing on all the rules and limitations of the developed system. Subsequently, Section 4 explains the steps followed to integrate the crossbar into a real multi-core system, including the

extension of the internal cores to issue reduction requests. In the same chapter, the main software applications used to test and measure the performance introduced by the reduction crossbar are described. Section 5 summarizes all the results obtained during the project. It includes both the outcomes retrieved from the synthesis of the crossbar and all performance metrics derived from various experiments. Finally, Section 6 concludes the thesis and outlines future works to improve the system.

Chapter 2

Background

This chapter provides essential background information to grasp the concepts of reduction operations and their key properties (Section 2.1). Additionally, considering that the project builds upon an existing implementation of a communication crossbar [11], the architectural structure of the original device is detailed in Section 2.3, while the protocol that the crossbar must adhere to is explained in Section 2.2. Subsequently, as the latter part of the project aims to integrate the designed XBAR into a real multi-core system, Section 2.4 delves into the system organization details and describes the internal cores' structure. Lastly, the State of the Art is examined in Section 2.5 to provide an overview of current methodologies employed for performing reduction operations.

2.1 Collective Communications

Collective communications, as delineated in [1, 12], entail the exchange of information among different processing elements. This communication paradigm involves collaborative efforts among various actors to process a dataset following specific patterns.
There are two primary subgroups under which Collective Communication operations can be classified:

- **Redistributions:** These are simple data transfer, without any additional logic or arithmetic operation.
- **Reductions:** These are data transfers that also involve some computations.

2.1.1 Redistributions

The first subset comprises mechanisms for transferring data from one or multiple nodes to one or more nodes. Several operations fall under this category, including:

Broadcast: A single Processing Element (PE) possesses information that needs to be disseminated to all other nodes within the system. The sender is

referred to as the "root," and all other nodes in the system act as receivers. This operation is also known as "All-to-One" communication, resulting in all nodes having a copy of the transmitted data.

Gather: Each node k possesses a sub-vector x_{k}. Every PE sends its sub-vector to a central node (the "root"), which aggregates all sub-vectors into a final vector x.

Allgather: An extension of the "gather" mechanism, where each node receives the final vector x obtained by combining all sub-vectors x_{k}.

Numerous other *Data Redistribution Operations* exist.

2.1.2 Reductions

Nodes often collaborate to compute a shared result. Partial results are independently evaluated by different nodes and then must be reduced to a single value, yielding a final result. Collective communication operations in this case involve not only data transfer but also logical/arithmetic operations on the collected data.
Not all operators are suitable for reductions. A generic operator $\oplus$ can be used for reduction applications if it is *associative*, i.e., $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. This requirement allows for parallelization of operations, as partial results can be computed concurrently and reduced together independently on the order. If the reduction operator $\oplus$ is also *commutative*, i.e., $a \oplus b = b \oplus a$, more efficient implementations are possible. Examples of reduction operations include addition, multiplication, maximum, minimum, and logical OR, AND, XOR.
The final result of the reduction can be distributed among nodes in different manners:

Reduce: Similar to "gather," this mechanism performs an operation on all data (OR, AND, sum, etc.) instead of merely concatenating the data sent by different nodes at the root, with the final result collected only by the root.

Allreduce: Similar to "reduce," but the final result is distributed among all nodes.

2.1.3 Barrier Synchronization

Another type of collective communication operation, often distinguished from the aforementioned families, is *barriers*. These mechanisms are crucial in parallel computing applications, requiring synchronization among nodes executing the program. Barrier mechanisms halt program execution at any node that reaches a designated point until all other actors reach the same stage of the program.
In barrier synchronization, there is no data transfer among nodes; instead, a stall

mechanism is employed upon each node's arrival at the barrier until the last node also arrives.

In this Degree Project, not all mentioned mechanisms will be supported by the reduction-capable XBAR. The thesis aims to develop an interconnection crossbar capable of handling *M-to-1* reductions, with a focus on the logical AND operator, which can also support the "Barrier Synchronization" mechanism among different masters connected to the same interconnection system. Additionally, leveraging the intrinsic nature of the AXI4 protocol [13], "1-to-All" communication can be implemented for Barrier scenarios where no information is shared among the processing elements.

2.2 AXI4 Protocol

The reduction-capable XBAR developed during the project is based on the *AMBA AXI* (Advanced eXtensible Interface 4) protocol [13]. This protocol defines interface specifications among different IPs that need to exchange information.
In this section, the main characteristics of the protocol are explained, focusing on aspects considered during the design of the crossbar. The terminology used throughout this report differs slightly from that in the official documentation. The main terms are defined in the glossary.

2.2.1 Channels Description

An AXI compliant interface features five distinct channels, as depicted in Fig. 2.1:

AW: The *Address Write* channel is used by masters to specify the address and control information for outgoing write transactions.

W: The *Write* channel transfers data to be written at the destination.

B: The *Write Response* channel is used by the receiving slave to notify the master of the completion status of the write transaction.

AR: The *Address Read* channel is employed to specify the address and control information for outgoing read transactions.

R: The *Read* data channel is used by the slave to send both data and the read response, indicating the completion of transactions.

2.2.2 Handshake

All channels in the interface feature different signals necessary for communication, detailed in [13] **A2**. Regardless of the channel type, communication between masters

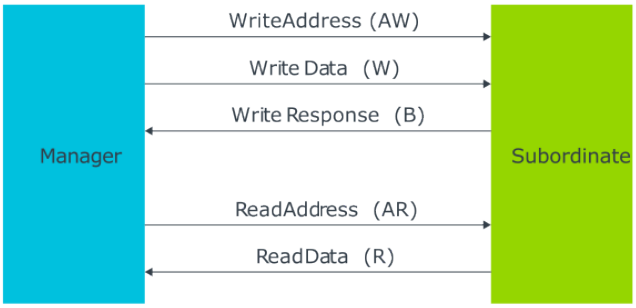


Figure 2.1: AXI Interface

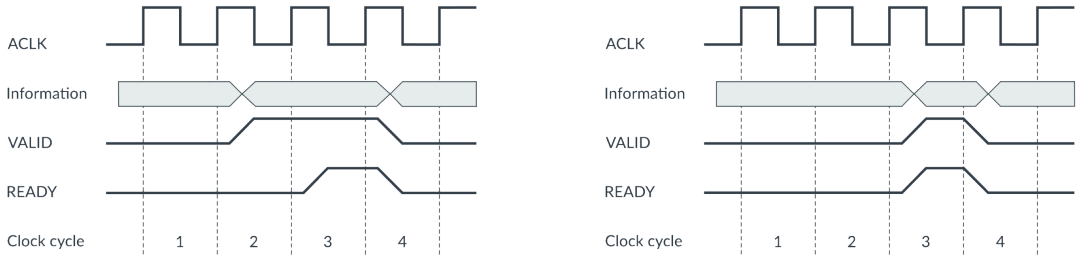
and slaves is enabled by a handshake protocol. All AXI channels have **VALID** and **READY** signals. When information is transmitted, whether data or control signals, the source asserts the VALID signal to indicate availability, and the destination asserts the READY signal when ready to receive. Transfer occurs only when both VALID and READY signals are active. The master's role as a source or destination depends on the channel. For example, in the AW channel, the master is the source, while the slave is the destination. Conversely, in the B channel, used by the slave to notify the master of the completion of a write transaction, the VALID signal is tied from the slave to the master, while the READY signal operates in the opposite direction.
To illustrate a channel transfer, consider the timing diagram depicted in Figure 2.2a. At the end of the first clock cycle, the source of the information has a VALID data, asserting the VALID signal. The source must maintain the information stable until the transfer is complete. By the third clock cycle, the destination is READY to sample the data and thus asserts the READY signal. Transfer occurs at the beginning of the fourth cycle when both handshake signals are active. Upon completion of the transfer, the data can be removed from the interface, and both the VALID and READY signals are reset.
It's important to note that the order in which the VALID and READY signals are asserted can vary. The READY signal can be asserted before the VALID if the destination can sample the data before it's available, or both signals can be activated simultaneously in the same clock cycle. Naturally, the fastest transfer occurs when both signals are asserted simultaneously, as depicted in Figure 2.2b.

2.2.3 Write Transaction

The timing diagram of a write transaction is depicted in Figure 2.3, illustrating a master sending a single data to a slave unit. The transaction involves the following steps:

1.

2. First, the master sends the address of the data to be written in the destination



(a) VALID before READY (b) Simultaneous VALID and READY

Figure 2.2: Examples of handshake transfers.

and other information to specify data length, burst type, etc. This step, known as *Address Write*, utilizes the AW channel. As shown in Figure 2.3, the address becomes available by the second clock cycle, at which point the master also asserts the VALID signal associated with the AW channel. The data must remain stable until the completion of the address write transfer, and during the third clock cycle, the slave asserts the READY signal, resulting in the transfer occurring during the fourth positive edge of the clock. After this, all handshake signals and the address are reset.

3. The next step is the *Write Data*. At cycle $n+2$, when the master has the data ready, it notifies the slave by asserting the VALID signal associated with the W channel. Additionally, the master sends a control signal called WLAST, indicating when the sent data is the last of the write data transfer. If the transfer involves burst transfers, the WLAST signal would be kept low for all transfers except the last one. Since the slave already asserted the READY signal in cycle n, the data transfer completes in cycle $n+3$.

4. The final step is the *Write Response*, during which the slave sends back information to the master to indicate the success or failure of the write transaction. As the B channel is driven by the slave, the VALID signal is asserted from the slave during cycle $n+3$, and the handshake associated with the B channel completes at the end of the same cycle.

Upon completion of the B transfer, the write transaction concludes.

2.2.4 Transaction Identifiers

All transactions in the AXI4 protocol are tagged with an identifier. This identifier is crucial for determining whether transactions should be handled in-order or out-of-order. Transactions with the same ID must remain in-order, while transactions with different IDs can have responses that are out-of-order.

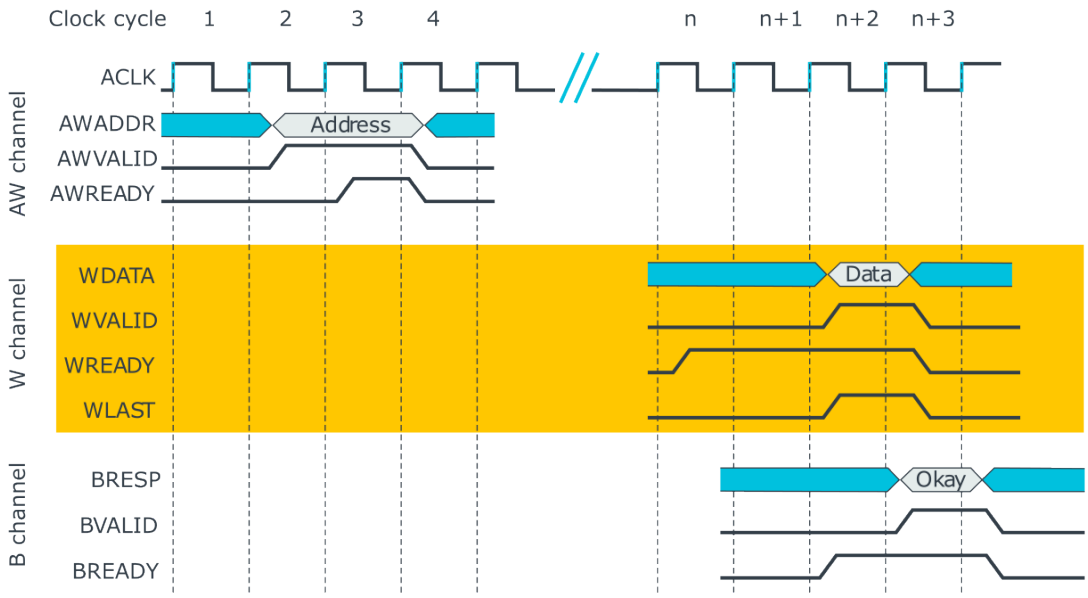


Figure 2.3: Example of a complete write transaction using the AXI interface protocol.

This aspect is of utmost importance in the design of the AXI crossbar (see Section 2.3.2).

2.3 AXI XBAR

Given the aim of the thesis to develop a reduction-capable crossbar based on an existing version [11, 14], this section offers an overview of the original crossbar. In 2.3.1, a general explanation is provided on how the system is organized and its main features. Subsequently, detailed explanations of the demultiplexers and multiplexers implemented within the crossbar are presented in sections 2.3.2 and 2.3.3, respectively.

2.3.1 XBAR Overview

The AXI crossbar serves as an interconnection system facilitating communication among various masters and slaves. A simplified diagram of the device is depicted in Fig. 2.4.
The slave ports of the crossbar are where the masters can connect, while the master ports are utilized by the slaves. This terminology is chosen because, from the perspective of masters, the device behaves as a slave (slave ports), and conversely, it acts as a master for the slaves connected downstream (master ports). This terminology remains consistent throughout the thesis (see GLOSSARY).
A fundamental component of the system is the Address Map, used by the Address

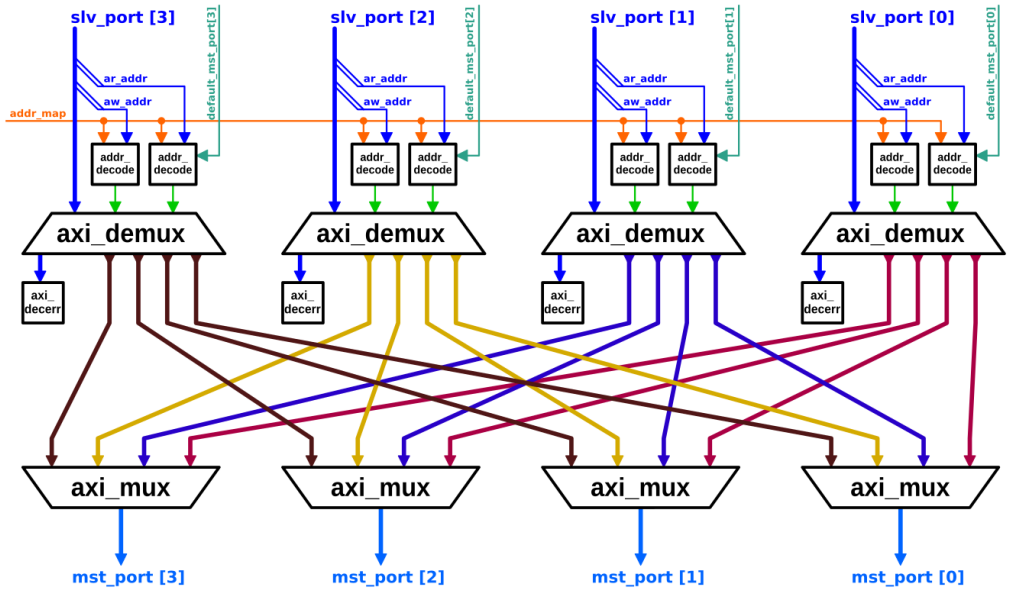


Figure 2.4: AXI crossbar block diagram.

Decoder to determine the target slave for incoming requests. The address map functions as a table where each entry represents an address range mapped to a specific master port. In cases where transactions are directed to non-existing destinations, two possibilities arise [11]:

1. **Default Port:** A master port can be designated as default. When the input address doesn't match any entry in the address map, the request is automatically redirected to the default port. This option proves useful in hierarchical systems, as explained in 2.4.

2. **Decode Error:** If the default port option isn't utilized, and the input address doesn't match any address map entries, the request is forwarded to the decode error module, which sends an error message back to the requesting master.

Upon address decoding, the input request and associated signals are transmitted to a Demultiplexer, responsible for selecting which master port the request should be forwarded to. Subsequently, all requests converge at all master ports via Multiplexers. Each multiplexer listens to input requests from masters and arbitrates them, forwarding one request at a time to the slave.
While demultiplexers and multiplexers serve as the main elements of the crossbar, they incorporate additional components to implement the AXI protocol, making them more than simple logic multiplexers/demultiplexers.

2.3.2 Demultiplexer

One of the pivotal modules in the crossbar is the demultiplexer, responsible for receiving write/read input requests from a slave port and forwarding them to one of

the master ports. Fig. 2.5 illustrates a detailed diagram of this module.
Starting from the slave ports, each of the AXI channels features a spill register used to cut the combinational path. These registers are optional and can be configured using synthesis parameters. Both the AW and AR channels possess a select signal determining the destination master port, typically driven by an external address decoder.
For write transactions channels (AW, W, and B), as depicted in the left half of the diagram in Fig. 2.5, two primary components manage the transactions: a FIFO memory and an *axi_id_counter*.
The FIFO stores the select signals of incoming transactions, which are later used to select the data direction on the write channel W. Conversely, the axi_id_counter is a specialized module handling transactions with different IDs and their corresponding B responses. In the AXI protocol [13], transactions with distinct IDs can complete out-of-order. To manage this scenario, the axi_id_counter comprises parallel counters, one for each ID. Upon an AW handshake, the counter associated with the transaction's ID is incremented, and when a B response returns, the associated counter is decremented. This ensures that all transactions with the same ID are handled in-order. Considering a system with *AxiIdWidth* bits to encode transaction IDs, a total of $2^{\text{AxiIdWidth}}$ counters are necessary for the write channels. The same number is required for read channels, summing up to $2 \times 2^{\text{AxiIdWidth}}$ counters. To mitigate costs, users can opt to utilize only a subset of the ID bits, known as *AxiLookBits*, to determine if two transactions share the same ID. This choice reduces the axi_id_counter's area but may lead to increased conflicts among transactions, as different IDs may share the same least-significant AxiLookBits.
B responses are routed back using a round-robin arbiter.
The read channels (AR, R) are structured similarly to the write channels, with the exception that there is no FIFO. An ordering constraint has been introduced during the design to reduce the crossbar's complexity: "when one slave port receives two transactions with the same ID and direction but targeting two different master ports, it will not accept the second transaction until the first has completed"[14]. Consequently, if multiple outstanding R responses must return, they originate from the same master port, and thus, the select signal used to route back these responses remains constant.

2.3.3 Multiplexer

All write/read requests from the slave ports converge to all master ports through the multiplexer modules. Fig. 2.6 illustrates the block diagram of this module. Each multiplexer unit connects the slave ports to a single output master port where the requests can be forwarded.
This module is also responsible for forwarding responses (B, R) from the master to the slave ports. The transaction ID associated with the input request alone is

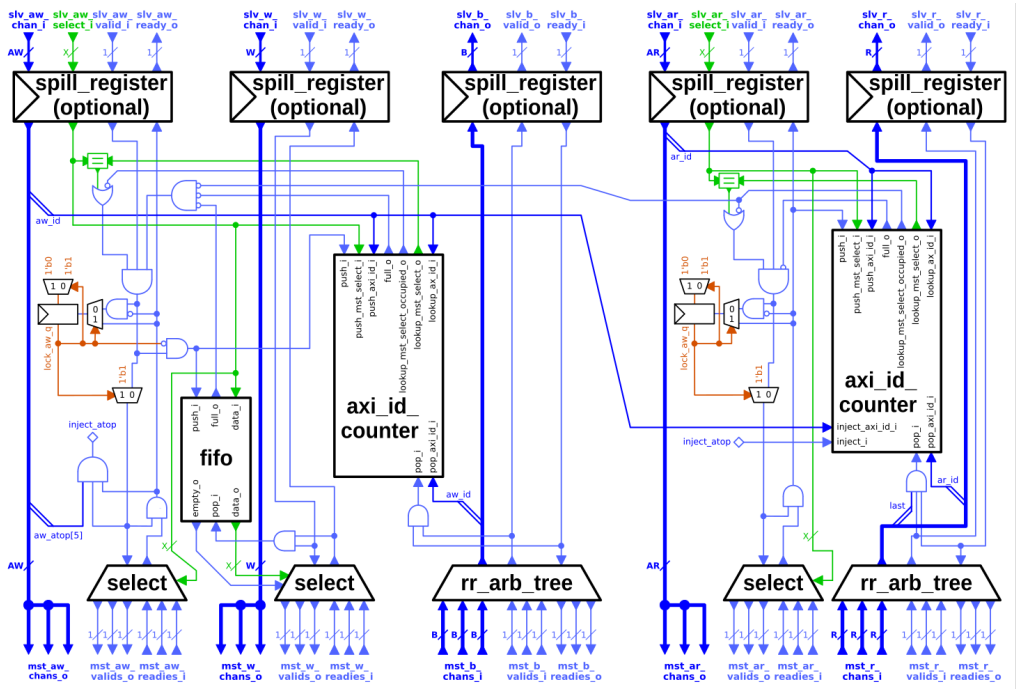


Figure 2.5: XBAR demultiplexer block diagram.

insufficient to route responses back to the requesting master. Consequently, the multiplexer internally extends the ID by appending the index associated with the slave port from which the request was accepted. As a result, the output AW and AR feature a wider ID, with $\lceil \log(\text{NoSlavePorts}) \rceil$ bits introduced as the most significant bits to the input IDs.

The AW (AR) input requests are selected using a round-robin arbiter. When an AW request is accepted, the prepended IDs are stored in a FIFO to determine which of the W channels must be forwarded to the master port. The FIFO is updated upon an AW handshake and is cleared when the associated data transfer completes, i.e., when the last data transfer is finished.

In conclusion, when the slave sends back the B (R) response, a logical demultiplexer routes that signal back to the correct slave port. This is accomplished through the most significant bits associated with the B response, which were appended when the request was accepted.

2.4 Occamy-Snitch PEs

The xbar serves as an interconnection system facilitating communication among various cores and clusters within multi-core systems. One such system utilizing this crossbar design is Manticore [15], a 4096-core RISC-V architecture optimized for efficient floating-point computing. Based on this architecture, a subsequent system named Occamy [16] was developed to explore scalability and performance. Both Occamy and Manticore rely on numerous Snitch processors [17].

To assess the functionality of the reduction-capable xbar and evaluate its perfor-

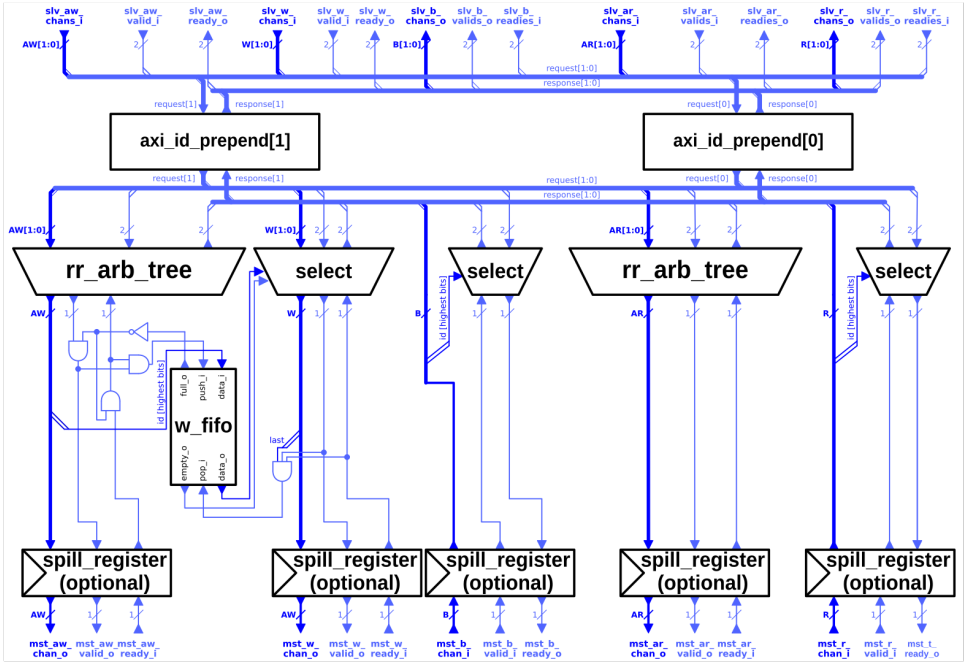


Figure 2.6: XBAR multiplexer block diagram.

mance under diverse conditions, the newly designed crossbar version presented in this thesis has been integrated into the Occamy system. This integration necessitated extensions in both the internal modules of Occamy and within the Snitch Cores to enable the issuance of reduction requests.
In this section, a brief explanation of Occamy and the main features of Snitch processors are provided in sections 2.4.2 and 2.4.1, respectively.

2.4.1 Snitch Core

Zarube et al. [17] developed a RISC-V-based processor called Snitch, incorporating custom instructions to enhance floating-point utilization. Illustrated in Fig. 2.7, the Snitch block diagram delineates its internal structure.
At the smallest level lies the *Snitch Core Complex*, comprising a RISC-V core. This core is partitioned into integer and floating-point sides. The integer side implements the integer base RV23I instruction set, with the majority of instructions executable in a single clock cycle. Multiply/divide instructions are offloaded to a shared multiply/divide unit among Snitch CC. The Arithmetic Logic Unit (ALU) is fully combinatorial, and executes operations within a single cycle. The Load-Store Unit (LSU) monitors issued loads, with responses from memory required to return in-order.
Conversely, the FP Subsystem manages operations involving floating-point data, encompassing arithmetic/logic operations and load/store operations of floating-point

numbers. Unlike the integer subsystem, the FPU is pipelined, and floating-point instructions entail multiple cycles. These instructions are dispatched from the integer to the FP subsystem via the accelerator interface, enabling the single-issue core to behave as dual-issue by overlapping independent integer and floating-point instructions.
A collection of Snitch Cores forms the *Snitch Hive*, where cores within the same hive share an L1 instruction cache and multiply/divide unit. These hives are then aggregated into a *Snitch Cluster*. Here, interconnection occurs via a XBAR, linking the cores to a shared Tightly Coupled Data Memory (TCDM), cluster peripherals, and other external Snitch clusters, as observed in multi-core systems like Occamy (Section 2.4.2).
In conclusion, the Snitch architecture introduces two ISA extensions called Stream Semantic register (SSR) and Floating-Point Repetition instruction (FREP) to achieve high floating-point utilization.

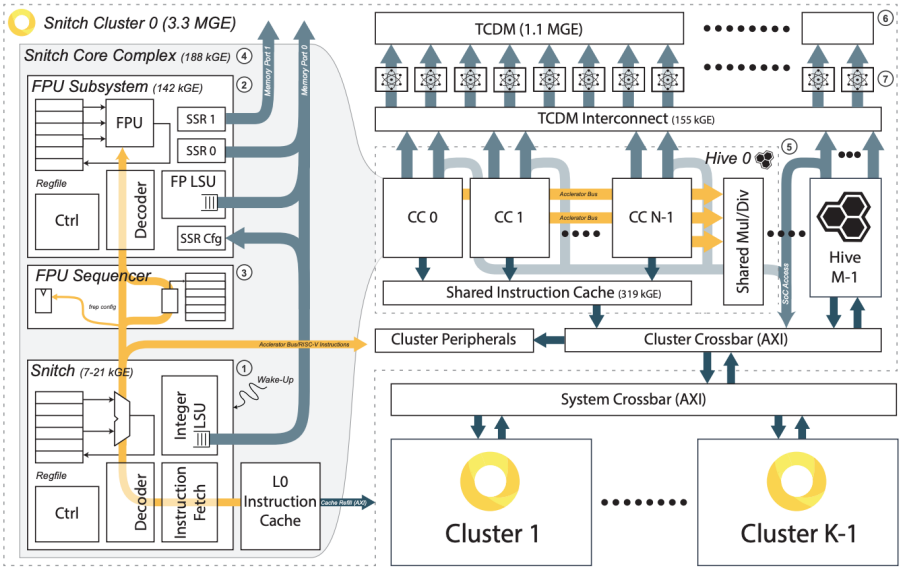


Figure 2.7: Snitch processor internal architecture.

2.4.2 Occamy

Occamy is a 216-core chiplet architecture developed to assess the efficiency and scalability of the PULP RISC-V-based architecture [16].
The system's top view, as illustrated in Fig. 2.8, showcases its architecture, which comprises six groups, each containing four clusters. Fig. 2.8b outlines the internal group organization, where four Snitch Clusters (see Section 2.4.1) share a constant cache and a zero memory. Additionally, a narrow 64-bit and a wide 512-bit crossbar are employed for the LSU and for the instruction cache and Direct Memory Access (DMA), respectively.
Within each cluster (Fig. 2.8c), eight worker Snitch cores reside, each equipped

with an integer core, a small L0 instruction cache, and a double-precision FPU. Additionally, a ninth core controls a DMA engine for cluster coordination. A shared L1 instruction cache is provided among all the cores, along with a multiply/divide unit. Moreover, a wide AXI XBAR is shared between the DMA and L1 cache to access the global memory system, accessible to all Snitch cores via a narrow 64-bit crossbar.
Each cluster also features a 128 KiB TCDM and a 64 KiB Zero Memory, where all read accesses return a zero value, and all writes are accepted without actual data write occurring. Table 2.1 depicts the address space of the first cluster within the first group, with each cluster mapped to a 256 KiB address space, resulting in a total of 1 MiB address space for the entire cluster.

Table 2.1: Address space of group 0 cluster 0

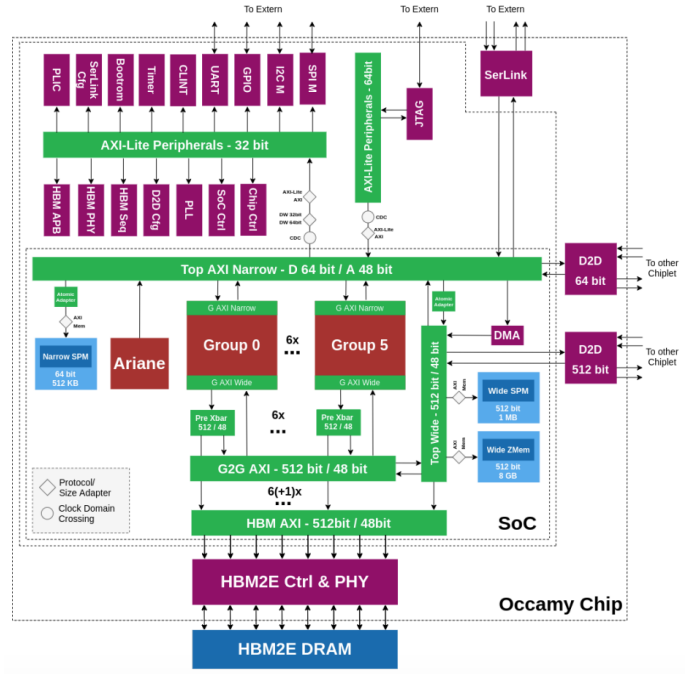
Name	Size	Status	Start	End
TCDM	128 KiB	used	0x1000_0000	0x1001_ffff
PERIPHERAL	64 KiB	used	0x1002_0000	0x1002_ffff
ZERO MEMORY	64 KiB	used	0x1003_0000	0x1003_ffff

2.5 Related Works

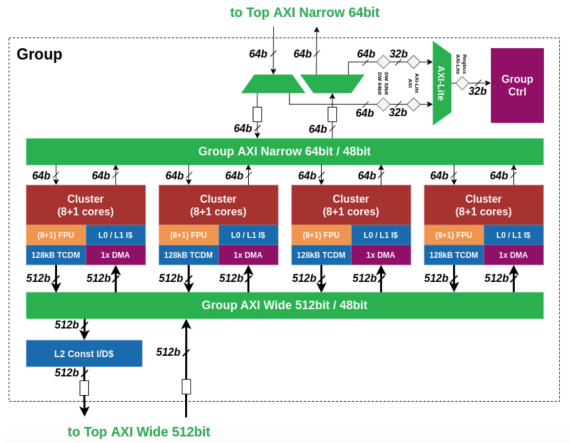
In this section different related works aiming to support reduction in hardware and software are reviewed.

2.5.1 Hardware

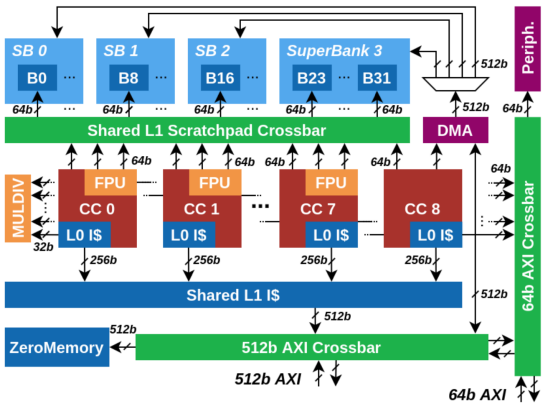
In the past decade, several hardware solutions have been developed to address the bottleneck arising from the use of collective communication in modern Chip Multi-Processors (CMP) systems.
Ma et al. [7] introduced a new framework and routing algorithm to manage multicast and reduction communication in cache coherence protocols. In a directory-based system, when a cache line is upgraded, an invalid signal must be sent to all other nodes in the system (1-to-M), followed by acknowledgment signals (ACK) from all destinations (M-to-1) before the cache line can be modified by the source. Their hardware solution for reduction relies on a Message Combination Table (MCT) present in all routers and a message framework appended to the packet's header. This framework includes two key pieces of information: identifying the last node in the network to replicate the message and utilizing a 3-bit field as a message ID to manage multiple outstanding multicast packets from a common node. Upon replication, an entry is added to the Message Table, encoding all necessary information, and the header is updated with the current node ID. Additionally, a counter tracking the number of destinations reached from that node is included in the entry to count ACK signals during the reduction phase. Each destination then sends an ACK signal back to



(a) Top View



(b) Internal group view



(c) Internal cluster view

Figure 2.8: Occamy system block diagram

the last replication node, updating the associated MCT entry upon arrival and forwarding the signal to the next node encoded in the MCT entry when the counter reaches zero.
Krishna et al. [8] adopted a different approach, focusing on improving efficiency in 1-to-M forking and M-to-1 aggregation in cache coherence applications. Their Flow Aggregation In-Network (FANIN) logic aims to aggregate incoming ACKs at each node before forwarding a single signal to the next node. Unlike the previous approach [7], FANIN doesn't employ tables to track responses. The first arriving ACK, termed the "master", is buffered and determines the port to poll, identified using the reverse Whirl, i.e. the reverse path of the routing algorithm used in the coherence protocol's first phase. Each incoming ACK carries a counter field indicating the number of aggregated responses. The master updates its counter, drops the incoming flit, and forwards the flit to the next node once all ACKs from the tracked port have arrived.
The proposed reduction-capable XBAR differs from previous solutions in several ways. Firstly, while previous approaches targeted traditional mesh NoC topology, the AXI XBAR is a fully connected crossbar used in a tree structured interconnection system which can be classified as a hierarchical star topology. Additionally, prior works aimed to enhance multicast and reduction for specific applications like message passing in cache coherence protocols, whereas the extension proposed in this thesis aims to remain a general-purpose solution capable of various reduction operations applicable in diverse contexts.
One significant class of reduction operations is the barrier synchronization mechanism commonly used in parallel computing applications to synchronize multiple processors before proceeding with computations. Various approaches have been proposed to accelerate global barriers, such as the barrier filter introduced by Sampson and Gonzalez [10]. This solution leverages cache line access and the subsequent stall before line filling. A hardware module called the "Barrier Filter" is integrated into the cache memory controller. When a thread participating in a barrier is ready, it sends an invalidation signal to notify its arrival by accessing the shared "arrival address tag" among all threads in the barrier. The Barrier Filter detects this operation and blocks thread fill requests until the last thread arrives, allowing the cache line fill request to proceed. Finally, each thread signals its exit to participate in a new barrier. According to the authors, this solution doesn't require any ISA extension since all required operations are commonly included in current architectures. In contrast, the reduction-capable XBAR requires a minor extension of the Processing Element (PE) to issue reduction requests. However, no new instructions are needed, as the request issuance utilizes the Control Status Registers (CSR) supported in the RISC-V standard (4.2). Moreover, while the arrival address must be assigned to each thread by the operating system in the barrier filter approach, the method used in the XBAR extension doesn't rely on an operating system. Consequently, integrating the barrier filter into some shared level of memory may introduce significant

communication latency, particularly in large scale systems.
In the late 1990s, several parallel computers were designed with dedicated networks for barrier synchronization mechanisms or more general reduction operations. For instance, the CM5's control network [18] comprised three different interconnection networks, one of which was dedicated to broadcasting and combining operations, supporting five reduction operations: bitwise OR and XOR, signed maximum and both signed and unsigned addition. Instead, in the reduction XBAR, only bitwise logic AND is currently supported, but the architecture is designed to be easily extended for additional operations.
Additionally, IBM developed the Blue Gene/L massively parallel system [9, 19] in the early 2000s, featuring 65,536 dual-processor compute nodes interconnected via a three-dimensional torus network with five networks. Among these, a global barrier network is specifically dedicated to facilitating hardware barriers.

2.5.2 Software

Given the significance of multi-core processors and the need for rapid reduction and synchronization operations, various research efforts have focused on accelerating these mechanisms in software. For example, Gao et al. [4] conducted an extensive study on OpenMP synchronization performance on ARMv8-based multi-core systems. While OpenMP synchronization performance had previously been studied for x86 CPU architectures, Gao et al. performed experiments to illustrate the barrier synchronization overhead on ARMv8 systems. Subsequently, they designed an optimized algorithm based on the "f-way tournament" to align with the hardware architecture. Similarly, an extended version of the butterfly barrier was developed for Intel and AMD microprocessors in [5]. A similar approach was employed during the benchmarking of the reduction XBAR to ensure that the software barrier performed well in the Occamy system. This underscores the importance of adapting software implementations to the underlying architecture to identify the most suitable algorithm that minimizes synchronization overhead.
Another common scenario where synchronization mechanisms and reduction operations are widely employed is in Graphic Processing Unit (GPU) architectures. Particularly, the Compute Unified Device Architecture (CUDA) paradigm enables parallel computing across multiple multithreaded SIMD processors. Typically, threads collaborate on a common algorithm, necessitating data sharing and, consequently, synchronization. Harris [6] presented seven techniques and programming strategies to enhance the performance of reduction operations in parallel computing. Additionally, the granularity of synchronization mechanisms is a crucial factor in parallel computing. With the introduction of the cooperative groups model [20], CUDA programmers now have increased flexibility and can synchronize groups of threads smaller than thread blocks. A similar objective is achieved with the reduction XBAR, as each master driving the crossbar's slave port can request reduction. Thus,

in a multi-core system like Occamy, where a hierarchical interconnection XBAR is utilized, reduction can be performed at the processor, cluster, and group levels.

Chapter 3

Methods I: Reduction-XBAR

In the following chapter, the complete process necessary to develop and test the functionalities supported by the Reduction-capable XBAR (R-XBAR) will be explained. In section 3.1, an overview of the extended crossbar is provided, focusing on design choices, limitations, and rules that users must fulfil to employ the device. Subsequently, section 3.2 delves into the details of the RTL implementation, demonstrating how all the AXI channels have been extended to support reduction requests. To verify the functionalities, an exhaustive testbench described in section 3.3 has been developed. Finally, the R-XBAR has been synthesized to explore Power, Performance, Area (PPA) overheads, with the synthesis steps discussed in section **??**.

3.1 Overview

Reduction operations constitute a specific class of collective communications, wherein an associative operation, denoted $\oplus$, is applied to several elements to obtain a single final result, expressed as $res = el_1 \oplus\ el_2 \oplus \cdots el_{\mathrm{n}}$. Examples of reduction operations include addition, bitwise logic AND/OR, as well as maximum/minimum computations.
In parallel computing applications, the algorithm workload is distributed among various PEs that operate in parallel. When the application necessitates a reduction among all the results computed by the different PEs, data must be transmitted from one processing unit to another, where a partial reduction is performed. This process iterates until the final result is obtained. However, this conventional approach often results in significant traffic within the interconnection system. To mitigate this issue, this thesis proposes extending the interconnection system to facilitate reduction operations directly during data movement. In essence, instead of transferring data among cores, each processing unit can transmit its own element along with a specialized request to the XBAR. Upon receiving all elements, the interconnection system will perform reduction on the aggregated data, and the resulting value will be transmitted to a designated destination specified in the reduction request control

signals.

3.1.1 Many-to-1 or 1-from-many

To support reduction two different approaches can be employed:

- *Many-to-1:* All the masters involved in the reduction can issue a write reduction request. The XBAR will listen to these requests, and upon data availability, it will perform the reduction and store the result in the destination address.

- *1-from-Many:* A master module initiates a read reduction request, specifying the involved masters. The XBAR forwards the read request to multiple destinations (multicast). Upon receiving the data back into the interconnection system, the crossbar performs the reduction and completes the read transaction by forwarding the result to the requesting master.

Both approaches are valid; the primary difference lies in the source that triggers the operation request. In the *Many-to-1* approach, the producers of the elements to be reduced initiate the transaction, whereas in the *1-from-Many* approach, it is the destination that sends the reduction request. In other words, with the *Many-to-1* approach, each producer can start a write transaction (AW), and all the data are reduced inside the XBAR, with the result forwarded to the destination port. Upon receiving the result, the slave can send back the B response signal, which must be routed back to all the participants (multicast). Conversely, using the *1-from-Many* approach, the consumer initiates a read transaction (AR), which must be forwarded to all the participants. Subsequently, when all the producers send back the data on the R channel, the XBAR can reduce them and provide the results to the requesting slave port.
The two approaches are almost dual, but some crucial differences can be identified:

- In the second method, the transaction is initiated by the destination. In the AXI protocol, slaves cannot initiate any transactions. Hence, this approach requires a module connected simultaneously as a slave and as a master to use the master side to send the read request, and then internally store the data in the correct location. Such modules are common; for example, in Occamy, the snitch clusters can drive the XBAR as masters, and the internal TCDM is connected as a slave.

- Even though clusters have internal memory, it may be necessary to store the reduction result in external locations, such as the HBM integrated into the Occamy chip. In this case, the memory cannot initiate the reduction request. Therefore, the workflow to perform the reduction requires first reducing the elements and storing the result inside one TCDM. Later, the cluster holding

the result must perform a normal write transaction to the main memory. This approach leads to an increased latency.

- Considering the specific case of global barriers, using the write channels to perform the reduction allows all the masters that need to synchronize to simply send a reduction request upon arrival at the barrier. Only when all the requests arrive the reduction is forwarded to the slave, which will then reply with the B response. This strategy can be easily used to create a blocking reduction request for synchronization purposes. In contrast, using the second approach makes synchronization more complex because the reduction is requested by the destination. In this case, it is necessary to design a specific module to connect to the XBAR, which must request reduction from all the masters to synchronize and continuously poll the reduction result. Only when it is set to a specific value, this module can wake up the participants. This second approach can require more extensions to the system, while the first approach is intrinsically based on the AXI protocol rules.

In conclusion, considering all the drawbacks deriving from the *1-from-Many* approach, the *Many-to-1* method was chosen for this project.

3.1.2 Reduction Mask

Once the AXI transaction type for performing reduction has been chosen, the next crucial step was to devise a strategy for encoding the list of participants. When a reduction needs to be executed, multiple requests will arrive at the XBAR, and the device must recognize them as reduction requests. Additionally, the R-XBAR must ascertain the list of participants to poll their slave ports until all the masters are ready for the reduction.
To achieve this, the AXI AW user field can be utilized. As described in the official protocol documentation [21, p.200], a user field can be added to each AXI channel. It is not mandatory for this field to be introduced in every channel, thus only the AW channel was extended.
For identifying the list of participants, an encoding strategy similar to the *Bit String* introduced by Chi-Ming and Lionel [22] can be employed. The main principle involves adding a *reduction mask* alongside the address that matches a specific port of the crossbar. This mask, used together with the address, represents a range of addresses corresponding to the list of masters participating in that specific reduction. When a bit in the mask is set to 1, the corresponding bit in the address field can be considered as a don't care, while a mask bit set to 0 indicates that the same bit in the address field must be considered.
Let's consider the example shown in Fig. 3.1. In the depicted system, there are four modules connected as both master and slaves. These modules could, for instance, be Snitch Clusters [17], where the crossbar's slave port is driven by the Cluster's

XBAR, while the master port is connected to the TCDM (Fig. 2.7). Assuming each memory has an address range of 256 KiB as shown in Table 2.1, if master A and B want to participate in a reduction, the reduction mask to provide is as follows:

$$\text{RedMask} = 0\text{x}1004\_0000 \tag{3.1}$$

Indeed, when comparing the start addresses of the two modules, they differ in the 18^{th} bit. By masking the result of the comparison with the mask 3.1, and considering that the 18^{th} bit is set to 1, the equivalent bit in the result is treated as a don't care. If the same mask is used to compare the address of module A with that of module C, the two addresses don't match because they differ in the 19^{th} bit, which is not masked. Additionally, if C and D participate in the same reduction, mask 3.1 can be used to match the two addresses.
Hence, to determine when some masters participate in the same reduction, three pieces of information are necessary:

- *Redmask*: The reduction mask to be applied after comparing the masters' addresses.
- *FirstAddr*: The start address of the first master sending the reduction request.
- *CurrAddr*: The start address of the current master requesting a reduction.

With these parameters, Equation 3.2 can be used to ascertain if two or more masters participate in the same reduction:

$$\text{match} = \ \& \left[\ \overline{(\text{FirstAddr} \ \oplus \ \text{CurrAddr})} \ + \ \text{RedMAsk} \ \right] \tag{3.2}$$

where $\oplus$ denotes the logical XOR operation, $+$ represents the logical OR operation, and & stands for bitwise AND.

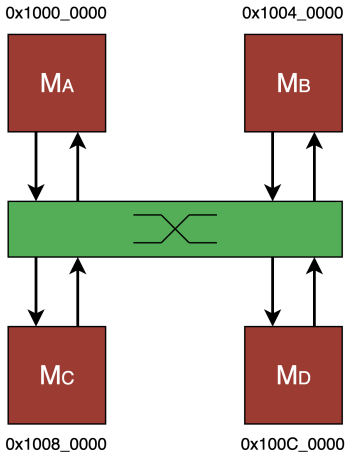


Figure 3.1: Interconnection XBAR with four modules connected both to slave and master ports. Each master port is associated to and address space of 256 KiB.

In conclusion, with the explained method, each master port of the XBAR is associated with an address space that can be represented either by the tuple (*start address*, *end address*) or by the tuple (*start address*, *bit mask*).

3.1.3 Design Rules and Limitations

With the reduction mask logic described earlier, certain features can be supported by the R-XBAR, but users must adhere to specific design rules. Additionally, to ensure the proper functioning of the crossbar, limitations have been introduced to manage the complexity of the architecture, aiming to strike a balance between flexibility and simplicity. The following is a comprehensive list of rules and considerations.

Address Space

With the introduction of the *Reduction Mask* logic, we have a method to identify multiple address spaces simultaneously. Since the logic is mask-based, not all ranges can be represented, but certain rules must be satisfied:

- **Memory Map:** To define the participant list, it's necessary to associate an address space with each master. In the original XBAR, the slave ports are not identified with any addresses, since only the AXI transaction destinations (slaves) require them. However, with the reduction extension, the only way to encode the participant lists is by defining some imaginary addresses. Often, cores like those used in Occamy are connected to the XBAR both as master and slave. Instead of introducing a dedicated memory map for the masters only, it has been decided to associate the slave ports with the same address space as the master port with the same index. Hence, when different reduction-capable modules are connected to the XBAR, as shown in Fig. 3.1, each module must be connected as a slave and master to ports with the same index: slave port i^{th} and master port i^{th}. In this fashion, the i^{th} entry of the address map can be used for both ports.

- **Number of Participants:** Given a start address and a reduction mask with N bits set to 1, 2^N address spaces can be represented with the same tuple. Therefore, all masters identified by this tuple must participate in the reduction. Consequently, reduction operations can only be performed with a group of power-of-two masters. For instance, in Fig. 3.1, if Redmask = 0x100C_0000, all four address spaces match the mask, requiring all four masters to participate in the reduction; otherwise, the R-XBAR will indefinitely wait.

- **Stride:** The reduction mask introduces the possibility of synchronizing non-contiguous masters. For example, if master A and C from Fig. 3.1 want to participate in the same reduction, setting Redmask = 0x1008_0000 is sufficient. Therefore, the mask-based logic enables the synchronization of masters with a certain stride.

Errors Detection

In the AXI protocol, AW transfers are characterized by several parameters. Among these are the W transfer *size*, which denotes the number of bytes in each W data transfer; the write *length*, indicating the number of transfers in a W transaction; the *burst* type, identifying how the address provided must be incremented between consecutive transfers in the same transaction; and finally, the *destination address* of the slave where the data must be written.
In the case of reduction requests, all these parameters must be the same among the AW requests arriving from the different sources involved in the reduction. In the designed R-XBAR, these fields are assumed to be identical among the requests, with no error detection mechanisms. Therefore, it is the user's responsibility to ensure that all participants set the same value for the mentioned parameters. Additionally, reduction transactions can only have a zero length; in the current reduction-capable crossbar, burst mode transactions are not supported. However, this extension may be introduced in the future to move larger portions of data while performing reduction, further improving the overall system's performance.
Some error detection mechanisms can be easily introduced in the current version, albeit at an additional area cost.

Transaction ID

As specified in the AXI protocol [21, p.93], each transaction is identified with an ID. As explained in sections 2.3.2 and 2.3.3, these IDs serve to maintain the order among transactions with the same ID. In a normal write transaction, the ID is received through the AW channel from the slave, and during the B response step, the same ID is sent back to the XBAR and internally used to route the response to the correct demultiplexer. However, in a reduction transaction, multiple requests are received from the XBAR, but only a single response is sent from the destination slave. Since this single B response must be routed back to all participants in the reduction, two possible design strategies have been considered:

1. All masters involved in the same reduction must use the same ID for that transaction. This allows the same ID encoded in the B responses to be sent back to all requesting masters.

2. Each master can use a different ID for the reduction request. This approach requires the introduction of extra hardware to store the received ID, which will be routed back to the master upon arrival of the B response.

Both solutions are valid. The first one introduces an extra rule for the user but saves some area, while the second approach provides more flexibility at the cost of larger devices. For the purpose of this thesis, the first approach was chosen.

Opcode

The implemented R-XBAR aims to perform generic reduction operations. Therefore, a reduction request also requires an *opcode* to specify which of the possible reduction operations must be performed. This opcode is embedded in the least significant bits of the AW user field. The width of this field can be set through synthesis parameters, allowing for the inclusion of more operators in the system. Currently, the only supported operator is the bitwise logic AND, which is sufficient to implement barrier synchronization mechanisms and test the functionalities of the crossbar.

3.2 Architecture Design

Since the XBAR needs to listen and wait for data coming from all the involved masters to perform reduction, the architectural extension primarily affects the crossbar's multiplexer. This is the location where multiple master requests converge, and only here can the XBAR detect reductions and properly handle them. In section 3.2.1, the small variations introduced in the demultiplexer are explained. Due to the complexity of the multiplexer, section 3.2.2 focuses solely on the multiplexer AW channel. In section 3.2.3, the elements introduced to support reduction during data transmission are explained, and finally, section 3.2.4 delves into the details of the B channel to multicast responses back.

3.2.1 Demultiplexer

While the original demultiplexer could support reduction, the design of the Reduction-capable XBAR (R-XBAR) prompted the following considerations: In a large-scale system with numerous cores, such as the one depicted in Fig. 2.8a, the processing elements are interconnected through a hierarchical XBAR structure. Consequently, it is possible for two masters engaged in the same reduction to be linked through a higher-level XBAR. In such cases, a reduction request should only be treated as a reduction if other masters connected at the same level also seek reduction. If no such masters exist, it indicates that the participant is connected to the next level, and thus the request can be processed as a normal transaction.
To implement this logic, the multiplexer was extended with the introduction of the logic illustrated in Fig. **??**. In the depicted block diagram, the module named *is_in_reduction* simply applies equation 3.2 to determine if any of the address rules match the reduction mask. If a match is found, the LSB of the OPCODE is set to 1 to signal the downstream demultiplexer that the incoming request is indeed a reduction. Conversely, if none of the slave ports connected to the XBAR participate in the reduction, the original OPCODE LSB is set to 0.

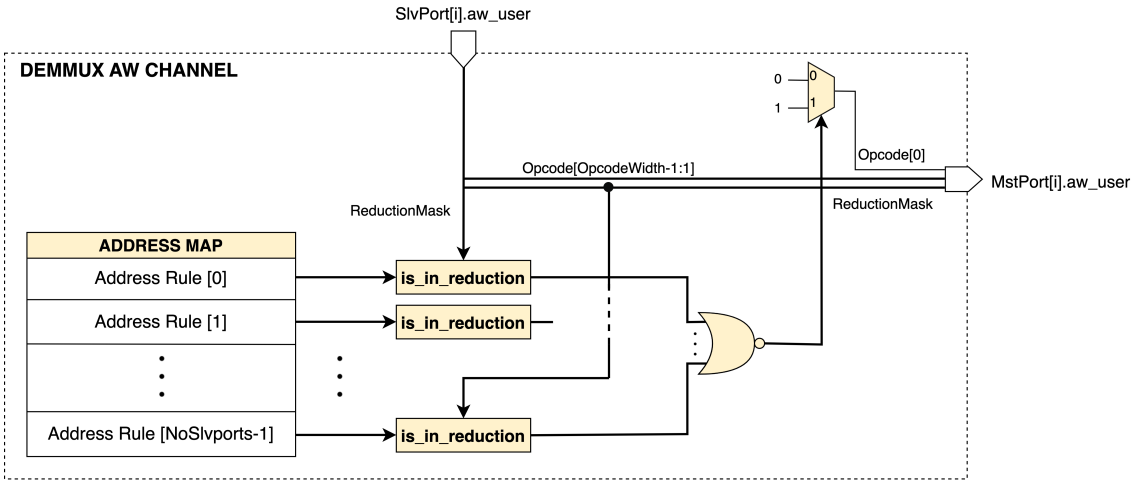


Figure 3.2: Unicast reduction management logic inside the demultiplexer AW channel

3.2.2 Multiplexer: AW Channel

Fig. 3.3a illustrates the internal organization of the extended AW channel for reduction support. To describe how the crossbar manages reduction requests, only the main building blocks are shown due to the complexity of the system. Before delving into the design details, some information is provided to better understand the schematic:

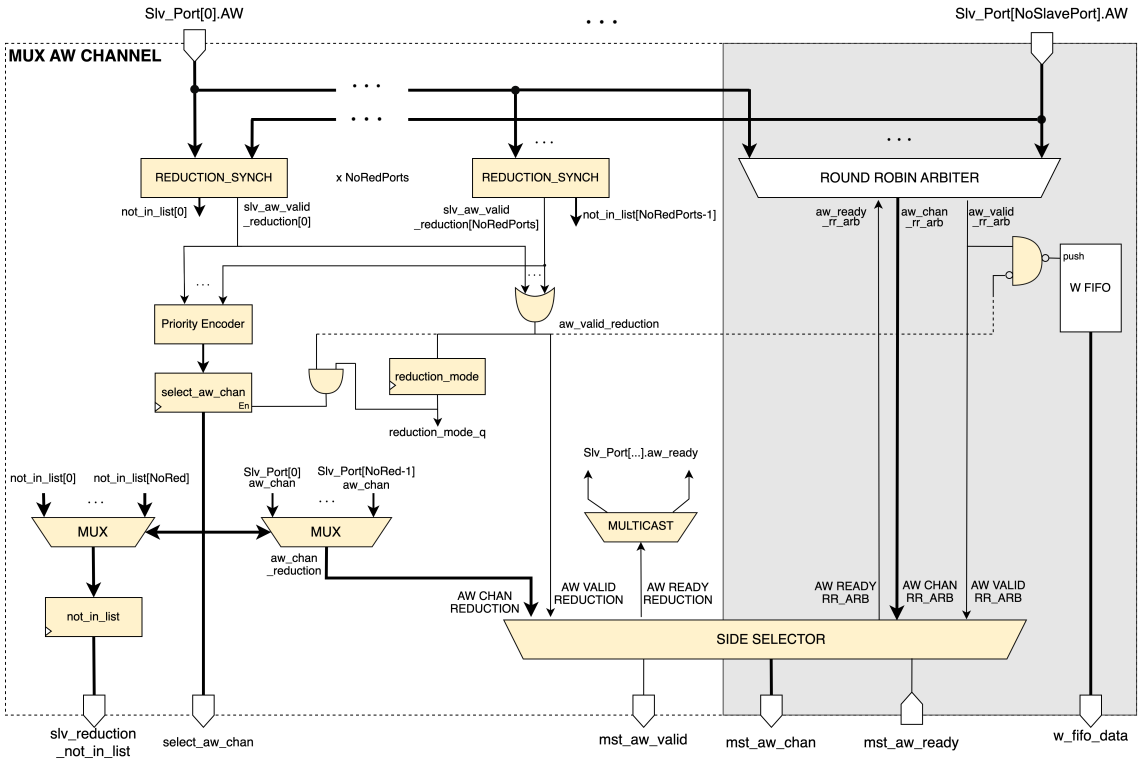
- Thin arrows represent single-bit signals.
- Thick arrows denote buses carrying various signals.
- When buses are not specified by any width, it indicates that they comprise signals with different meanings. For example, the AW channel encompasses a user field, a length field, an address field, etc.
- Yellow elements have been introduced in the reduction extension.
- The term *AwUserWidth* is a synthesis parameter used to configure the AW channel user's width to encode the reduction mask and the opcode.
- All signal and module names correspond to those used in the SystemVerilog RTL description.

As depicted in Fig. 3.3a, the majority of the elements necessary to manage reductions were introduced alongside the original system. The grey region represents the round-robin side where input requests were scheduled, as explained in Section 2.3, while the white part is necessary only for reductions.
Starting from the input ports, all the *NoSlavePorts* are connected to the reduction side through a module named *REDUCTION_SYNCH* and to the original side through a round-robin arbiter. To detect if the input request is a reduction, it's

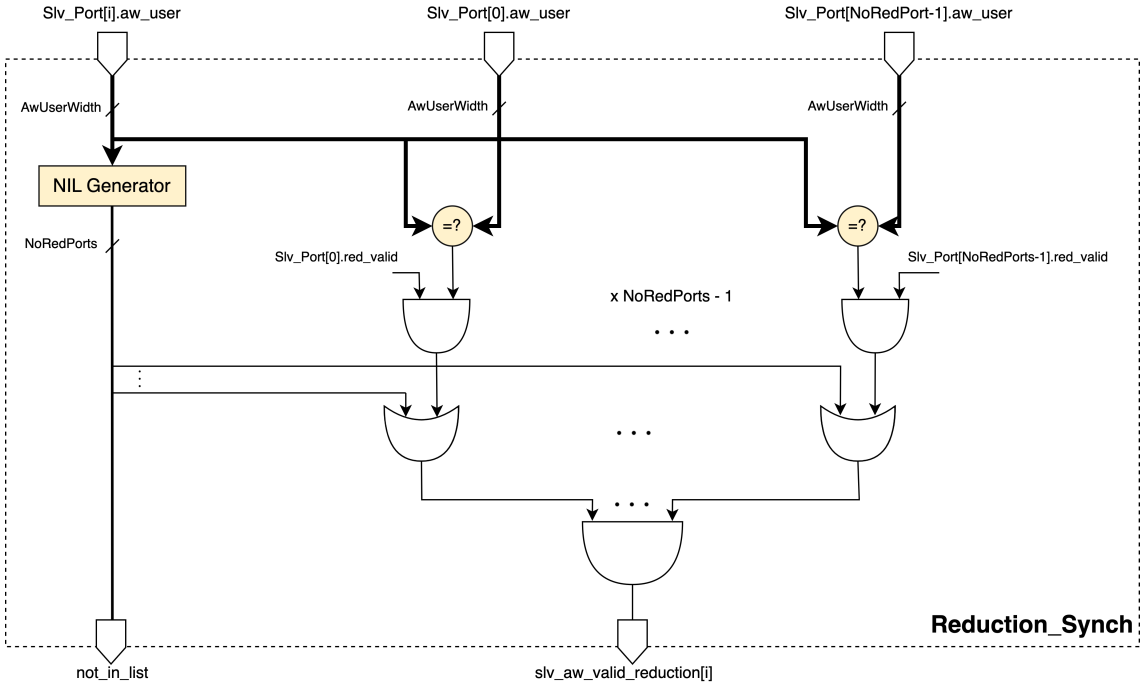
sufficient to check the user field: when it's zero, the transaction is normal; otherwise, the input request is a reduction. Only normal requests are arbitrated from the round-robin side, while the others are handled by the reduction one. The reduction synchronization module, described later in Section 3.2.2, detects all the reduction requests and asserts the AW VALID signal after all the reduction participants have asserted their valid signal. It's also responsible for computing the participants list (*slv_reduction_not_in_list*) that will be used by the W channel to perform the reduction operation only among the data sent by the involved masters.
Upon assertion of the AW VALID on the reduction side (*aw_valid_reduction*), a priority encoder determines which of the input channels must be forwarded to the master port. The channel can be any of the participants' channels because the AW control signals must be the same among all the involved masters, as explained in Section 3.1. Once the reduction AW VALID signal is asserted, the multiplexer enters the so-called "reduction mode". Therefore, no other requests, neither reduction nor normal transactions, can be accepted until the end of the actual reduction operation. For this reason, the reduction valid signal is used to disable the First-In-First-Out (FIFO) push signal.
Finally, a selector is used at the interface with the master ports to choose between the AW signals coming from the reduction or round-robin side. In the case of reduction, the AW READY signal is back-propagated to all the involved masters. This multicast operation is performed by using the Not In List (NIL) signal selected from the priority encoder.

Reduction Synchronisation Module

The reduction synchronization module depicted in Fig. 3.3b serves as the pivotal element of the entire Reduction-capable XBAR (R-XBAR). The main aspect to consider when performing reduction is the synchronization among the various requests. Since each master operates independently from the others, reduction requests can arrive at different times. The synchronization module has been designed to monitor each slave port, and upon arrival of a reduction request, it checks whether the other participants have already sent the associated reduction request or not. When all requests have been detected, the module asserts the AW VALID signal downstream, which is subsequently propagated to the output as described previously (section 3.2.2).
Using equation 3.2, the synchronization module determines, for each input port, the Not In List (NIL) signal that defines which masters are involved. Each bit in NIL is associated with one slave port: when the i^{th} bit is asserted, the i^{th} slave port does not take part in the reduction; otherwise, it does. Hence, the NIL signal is used to mask all the non-involved reduction requests.
In conclusion, by utilizing the logic AND/OR tree and comparing the reduction masks from all the masters, the output AW VALID signal can be asserted only when all participants are ready.



(a) Detailed block diagram of the reduction-capable crossbar AW channel



(b) Internal architecture of the Reduction_Synch module

Deadlock

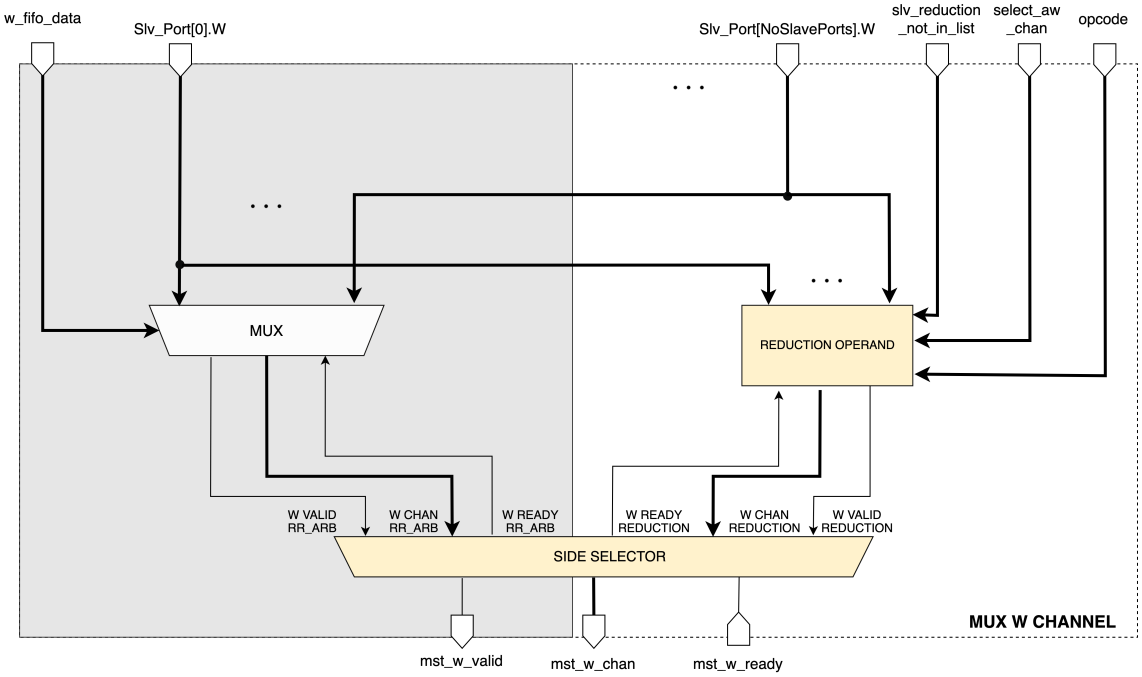
The design considerations for the AW channel are crucial to understanding the final architecture, particularly regarding potential deadlock conditions.
Initially, the plan was to employ only one synchronization module in each multiplexer, intending to grant control of the module to the first arriving request at the multiplexer. However, this approach posed a risk of deadlock. Consider a system with three masters: M_A, M_B, and M_C. Suppose two reductions are to be performed: the first between B and C (R_{BC}) and the second between A and B (R_{AB}). Since each master operates independently, there is a chance that the first request issued is from (R_{AB}), thereby giving module control to master A. As B has not generated the same request, the AW VALID signal cannot be asserted, and the module will await B. However, when the request for the reduction between B and C arrives, the module is already under the control of master A. Consequently, the request cannot be handled. Thus, as R_{BC} cannot be processed, and B will never send the request R_{AB}, the system will indefinitely stall.
To avoid this scenario, it is necessary to utilize one reduction synchronization module for each slave port. This ensures that each port has its module to monitor the arrival of all participants, mitigating the risk of deadlock.

3.2.3 Multiplexer: W Channel

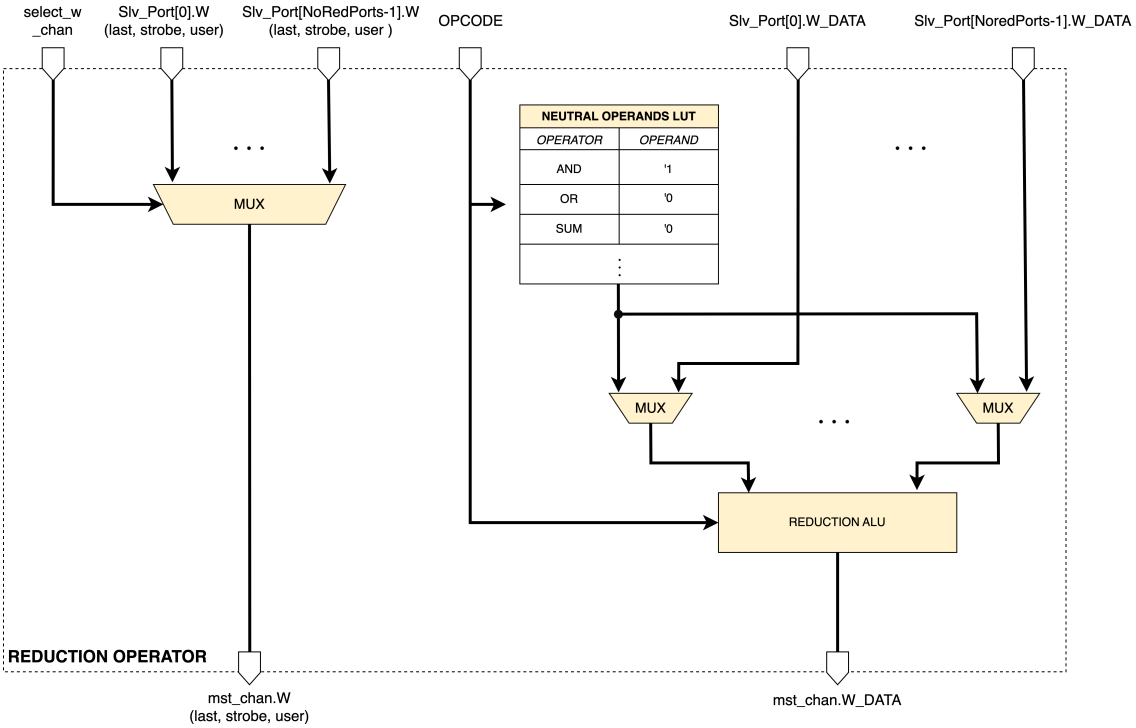
To manage sent data, the system depicted in Fig. 3.4a has been developed. The grey side represents the original implementation already described in Section 2.3.3, where input W channels are selected using a multiplexer driven by the First-In-First-Out (FIFO) data from the AW channel. However, when reductions must be performed, this approach is not sufficient because the data from different sources can arrive asynchronously. Hence, the Reduction-capable XBAR (R-XBAR) must have the capability to wait for all involved masters, and only upon the arrival of all items, it can finally reduce them. Additionally, considering that the reduction list is not known a priory, the system must implement some masking operations to mask data from the rest of the W channels.
The module responsible for all mentioned tasks is called *REDUCTION_OPERATOR*.

Reduction Operator

Figure 3.4b illustrates a block diagram of the reduction operator module. Two main functionalities can be distinguished. Firstly, it must select which of the input channels carrying the strobe, last, and user signals must be forwarded. To achieve this goal, the *select_w_chan* signal, derived from the AW side *select_aw_chan* signal, is used to select only one of the involved channels. It is not important which of them is selected since they must have the same configuration signals while participating in the same reduction.



(a) Detailed block diagram of the reduction-capable crossbar W channel



(b) Internal architecture of the Reduction_Operand module

Additionally, the module is responsible for the reduction itself. As mentioned earlier, the current version of R-XBAR supports only the logic AND operation, although the system has been designed for future extensions. Therefore, in the following description, more operators are considered even though they have not been implemented. Before performing the arithmetic/logic operation, all data channels not involved in the reduction must be masked. For this purpose, a look-up table filled with neutral operands for all operators can be used. The operand is selected by the opcode coming from the AW control signals: the neutral operand for the logic AND is a data with all bits set to one, for the OR it is a zero value, and similarly, in the case of addition, a zero data can be used, while in the case of maximum and minimum operators, it depends on whether the numbers are signed or unsigned. This table can be easily extended with more operators if necessary.
Then, the signal carrying the list of participants already evaluated on the AW side (*not_in_list*) is used to select for each channel between the neutral operand or the original data coming from the slave port. Finally, the opcode selects the operation to be performed inside the reduction Arithmetic Logic Unit (ALU), which can be extended to support various operators.

W Valid/Ready

The W VALID signal is asserted downstream only after detection of all valid signals coming from the involved slave ports. The list of participants evaluated in the AW channel is used to listen only to the necessary ports and mask the others. On the other hand, the W READY signal is multicast back to the participant ports. Since the data can arrive asynchronously along with their associated valid signals, the ready signal is rooted back only when the handshake is detected on the master port side. Therefore, the W READY sent from the slave is sent back to all participants only upon the arrival of W VALID signals from all the masters.
In conclusion, similarly to what is done on the AW side, a selector is used to choose between the original side and the reduction channels. This selection is performed by examining the status of the R-XBAR, i.e., by observing the *reduction_mode_q* signal that is asserted on the AW side only after acceptance of the AW VALID and completion of all outstanding normal transactions.

3.2.4 Multiplexer: B Channel

To back propagate the B response from the master port to the slave ports, the architecture depicted in Fig. 3.5 is utilized. In this diagram, the arrows are bidirectional to underline that the VALID signal is replicated from the master port to the slave ports, while the ready signal is reduced in the opposite direction. Some main modules can be identified:

- **B_COUNTER:** It is an UP/DOWN counter used to track the outstanding normal transactions. As explained in Section 3.2.2, before assertion of the

AW VALID signal for reduction requests, all outstanding transactions must be completed. The term "outstanding" refers to all write transactions which are in-flight, i.e., those transactions which have been accepted because the AW handshake already occurred, but didn't receive the B response yet. The First-In-First-Out (FIFO) in the AW channel can be used to determine how many of the accepted AW requests already sent the associated data, while the B counter is used to track all the transactions from the moment when the data is sent (W_LAST = 1) to the reception of the associated B response. Only when the FIFO is empty and the counter is zero, no normal transactions are outstanding, and a reduction request can be accepted, leading the R-XBAR into the reduction mode.

- **REPLICATION:** When the B response associated with a reduction request is received, it must be routed back to all the participants in the reduction. This operation can be easily performed using the *not_in_list* signal computed on reception of the AW reduction request. Care must be taken with the handshake on the B channel. The B VALID signal is driven from the slave, while several B READY signals come from multiple masters and they can be asserted at any moment regardless of the other ready signals. Therefore, as soon as the valid is asserted, it is routed to all participants. But in each B channel, upon reception of the B READY, the valid is removed to avoid multiple valid detections from the masters. Only when all the B READY signals have been asserted, the ready is also activated downstream on the master port side to notify the completion of the B response to the slave. In this way, the slave removes the valid only after reception from all the involved masters, and each master detects the valid signal for a single cycle as required by the Advanced eXtensible Interface (AXI) specification.

Reduction ID limitation

As described in Section 3.1, to have a working system, each AW request associated with a specific reduction must have the same transaction ID. This ensures that the same ID received from the slave can be sent back to all participants while receiving the B response. To provide flexibility to the user, a synthesis parameter named *EnableRedId* can be set to introduce enough registers to store the IDs coming from all the masters involved in the reduction. This allows each request to have a different ID, and when the B response is backpropagated, the correct IDs to be routed back to each slave port are held in specific registers. As mentioned before, this extension will increase the final area.

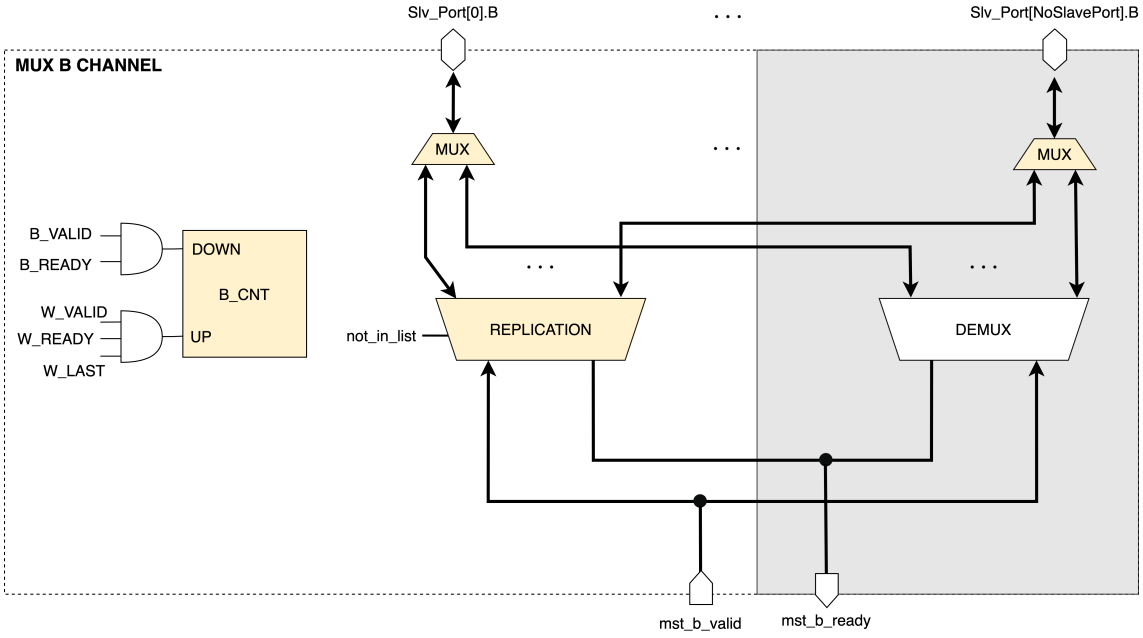


Figure 3.5: Block diagram of the reduction-capable crossbar B channel

3.3 Verification

To verify the functionality of the reduction-capable crossbar, the original testbench has been extended to generate reduction requests. Before delving into the details of the extended version, a short explanation of the general organization is provided in Section 3.3.1, and the complete version capable of generating reductions and checking for their correctness is explained in Section 3.3.2.

3.3.1 Testbench Overview

Figure 3.6 illustrates a simplified structure of a common testbench for Device Under Test (DUT) verification. Several modules can be identified:

- **Driver:** This unit is responsible for generating stimuli sent to the device.
- **DUT:** This is the designed system to be tested and verified.
- **Monitor:** This module is responsible for monitoring the input and output signals.
- **Scoreboard:** This element compares the obtained results with those expected. The expected results are usually determined from the scoreboard by implementing a gold model of the DUT and applying it to the driven stimuli.

The same structure has been used for the Crossbar (XBAR). Figure 3.7 depicts the complete organization of the testbench. Each master and slave is driven by independent drivers to simulate real conditions where devices asynchronously send and receive requests. Each driver generates requests randomly, i.e., with random

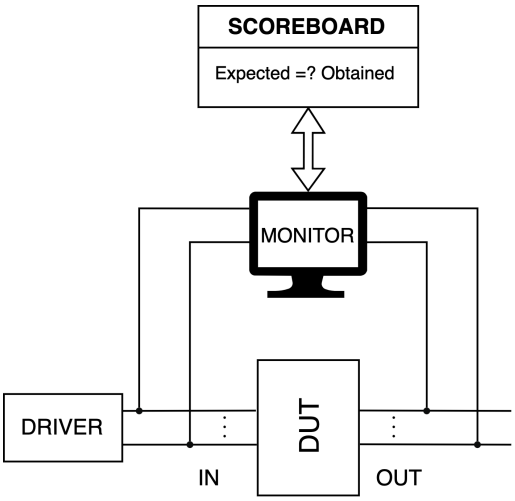


Figure 3.6: Diagram of a generic testbench for verification purposes

AW configuration signals (length, size, burst, etc.), and issues them at a random time. Similarly, slaves accept arriving requests and send responses back in a random fashion while still being compliant with the Advanced eXtensible Interface (AXI) protocol. Alongside the drivers and the DUT, a monitor tracks the input stimuli and evaluates the expected signals at the output ports of the XBAR, which are the master ports for the request signals and the slave port for the response signals. The gold model and the checking functionality are directly embedded inside the monitor unit.

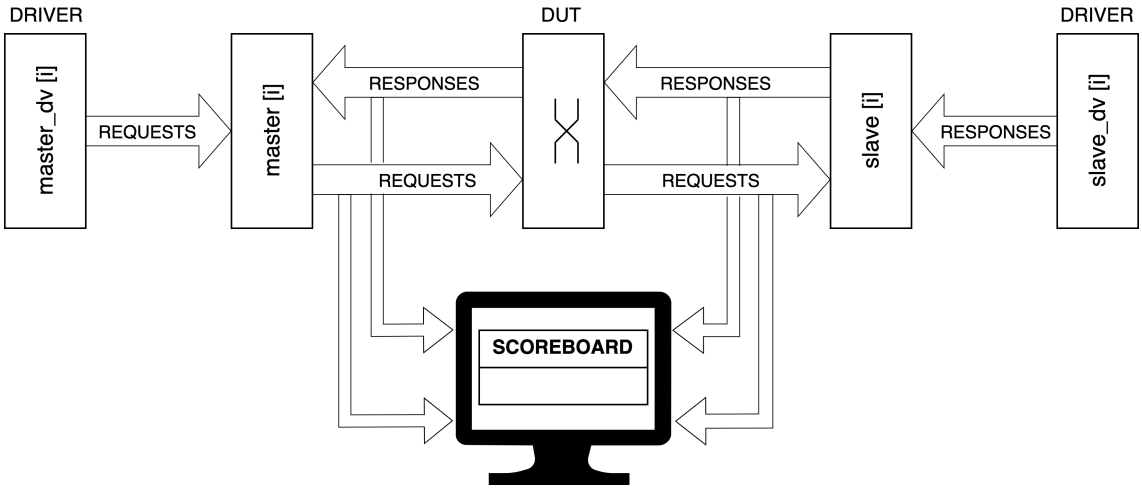


Figure 3.7: Complete diagram of the XBAR testbench

3.3.2 Testbench Extension

The original testbench didn't consider many aspects that became crucial with the introduction of reduction support. The main extensions have been introduced in the drivers and in the monitor.

The main challenge encountered in the driver extension came from the lack of correlation among masters. They were modelled with objects, each with its own attributes used to generate the requests. Each object is associated with a thread and therefore they run concurrently. Originally, there was no necessity to have inter-master communication mechanisms, while the introduction of reduction support makes this aspect crucial.

Drivers extension

As previously explained, each master generates its write/read requests independently of the others. This approach cannot be used to model reduction requests because a reduction is a shared operation among various participants. In a real-world system, this aspect is obvious: considering a multi-core architecture, each core executes a specific part of a parallel program and then the reduction is issued when necessary by all participants. Therefore, in the model, it is necessary to ensure that before sending a reduction request, all the involved masters know they must take part in the same reduction. Additionally, even though a communication mechanism among the involved masters is introduced, the testbench must ensure that each master still issues its request whenever it wants, at a random time, without being affected by the other masters, to test the Reduction-capable XBAR (R-XBAR) functionalities with asynchronous requests.
To solve all these problems, a shared queue-based structure has been used among all the masters. This structure is depicted in Fig. 3.8. It can be seen as a set of queues, one per master. Every time a master wants to issue a request, this can be normal or a reduction. The probability to choose one or the other can be defined in the testbench. When a master generates a reduction request, it checks its queue and two scenarios can occur:

1. *Empty Queue:* In the case of an empty queue, the master is not part of any outstanding reductions. Consequently, it's free to generate a random mask and all the AW control signals to associate with the request. These must satisfy the rules imposed by the R-XBAR design. Then, it must communicate to all participants that they will take part in the reduction as soon as they need to issue a reduction request. This communication is performed by pushing all the information associated with the request into all participants' queues. This update operation must be arbitrated using a *semaphore* because each master is an independent thread and it might interrupt the execution of all the others. At the end of this process, all the involved masters will have the queue filled with the necessary information to issue a reduction request coherent with that already sent.

2. *Full Queue:* If the queue contains at least one element, it indicates that the master is part of a reduction initiated by another master. In this case, the

master reads the information from the queue's entry, forwards the request, and removes the element to prepare for potential subsequent reductions.

With this structure and the use of a semaphore to arbitrate queue accesses, each master can randomly decide when to issue a reduction request while maintaining consistency with the rest of the participants.
As illustrated in Fig. 3.8, each queue entry contains all the information shared among AW reduction requests. This includes details such as the write transfer length, size, and burst. While the current version of the R-XBAR does not support burst transactions, and therefore the size and burst type are fixed, they have been included in the queue data structure for future extensions of the crossbar. This design choice ensures that the testbench can be easily adapted and extended in the future.

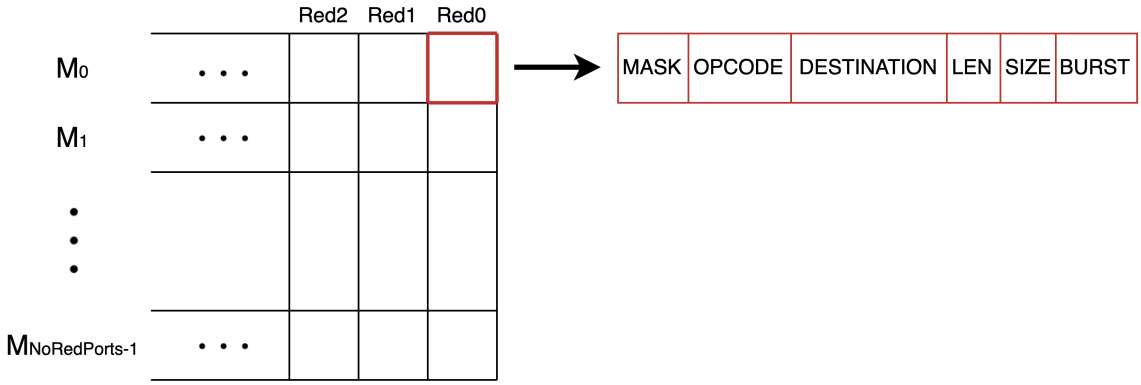


Figure 3.8: Queue-based structure to maintain coherence among the various threads

Monitor extension

The monitor has also been extended to ensure the correctness of reductions. The main change involves the control of the W DATA field. Previously, without the reduction extension, the XBAR solely transferred data from the input to the output port, requiring verification that the output data matched the input. However, with the introduction of reductions, this is no longer sufficient. Inside the R-XBAR, multiple operations can now be performed based on the OPCODE set during the AW transfer. Therefore, the monitor has been augmented to track all reduction requests, wait for the arrival of all the associated data, and subsequently verify that the output data meets expectations.

Verification Status

In conclusion, numerous tests with different configurations have been conducted to verify the correct functionality of the system. All combinations ranging from 2 to 8 master and slave ports have been tested successfully in simulations. In each test scenario, every master issued a total of 600 requests, with half being write requests

and half read requests. Additionally, among the write requests, a portion were reductions, with the probability of generating them set to 50%.

Chapter 4

Methods II: System Level

After designing the reduction-capable XBAR, the next step in the project involved its integration into a real system called Occamy [16]. Section 4.1 analyzes the implications of integrating the R-XBAR into the real system and explains the further extensions introduced. Subsequently, section 4.2 demonstrates how the Snitch processor was extended to issue reduction requests. Finally, to test the entire system and evaluate performance improvements, various applications have been developed, and their structure is explained in section 4.3.

4.1 Occamy Integration

In this 216 core-based architecture depicted in Fig.2.8, crossbars are utilized to interconnect the internal cores in a hierarchical structure. Each Snitch core is grouped within a cluster comprising eight Snitch CC units. Four of these clusters are further grouped together and interconnected via both a 64-bit and a 512-bit XBAR. Finally, inter-group communication is facilitated by a last level crossbar that connects the six groups within the Occamy chip.
To facilitate the hardware instantiation of all necessary modules, a Python script named *occamygen* [23] is utilized. This script relies on a configuration file that users can modify to test different versions of the Occamy system, adjusting parameters such as the number of clusters in a group or the overall system. It generates the SystemVerilog files required to build the chip, ranging from the cluster level depicted in Fig.2.8c to the SoC view in Fig.2.8a. To accommodate the reduction-capable crossbar, the script has been appropriately modified to allow users to instantiate either a reduction or a normal crossbar. However, due to the complexity of the system, the R-XBAR architecture outlined in section 3.2 did not fully meet certain constraints imposed by the Occamy architecture. Therefore, further extensions were necessary to integrate the interconnection infrastructure. The following list details the extensions made and highlights some limitations introduced to achieve a fully functional system.

4.1.1 Master and Slave ports

During the initial design phase of the R-XBAR, it was assumed that the interconnection system would accommodate an equal number of master and slave ports, i.e., NoSlavePorts = NoMasterPorts. However, as illustrated in Fig.2.8, this assumption does not hold in real systems, where a crossbar may have a varying number of masters and slaves connected to it.
This discrepancy impacts another aspect of the original design: the encoding of the list of participants in a reduction. As mentioned in 3.1, slave ports must be associated with an address space to encode the list of participants. To avoid duplicating the address map, it was decided to assign the same address rules to both master and slave ports. Consequently, reduction-capable masters are connected as both master and slave units, a configuration commonly seen in systems like Occamy, where clusters, groups, and Snitch cores are connected to both master and slave ports. However, the possibility of having a different number of interconnected masters and slaves introduces the option for devices to be connected solely as master or slave units. For example, in the architecture depicted in Fig.2.8a, the purple modules function solely as slaves and cannot issue requests, while the *Ariane* RISC-V CPU is connected exclusively as a master unit.
To address this issue, modifications were made to the R-XBAR as depicted in Fig.4.1, 4.2a, and 4.2b. The added interconnections are highlighted in red. A new synthesis parameter, *NoRedPorts*, has been introduced to identify the number of interconnected masters capable of issuing reduction requests. Modules connected to these reduction ports must adhere to the following rules:

- **Ports Duality:** A reduction-capable master connected to a reduction port must also have a connection to a slave port with the same index.
- **Connection Order:** Reduction ports must be associated with the lowest indexes in the address map. Therefore, the first *NoRedPorts* address rules are associated with both slave and master ports.

These rules ensure consistency in the connections. However, it means that Ariane cannot issue reduction requests. This limitation is not significant because, for parallel computation, Snitch cores are typically utilized. In many-core systems, each core may have its own internal memory and thus would be connected to both a slave and a master port. The only restriction is the connection order.

4.1.2 Default Port

The XBAR specifications [11] allow users to designate a default master port where requests are forwarded when the destination address does not match any address map rules. In Occamy, this feature is used to implicitly define the address space of the port where the next XBAR level is connected. Consider the group view in

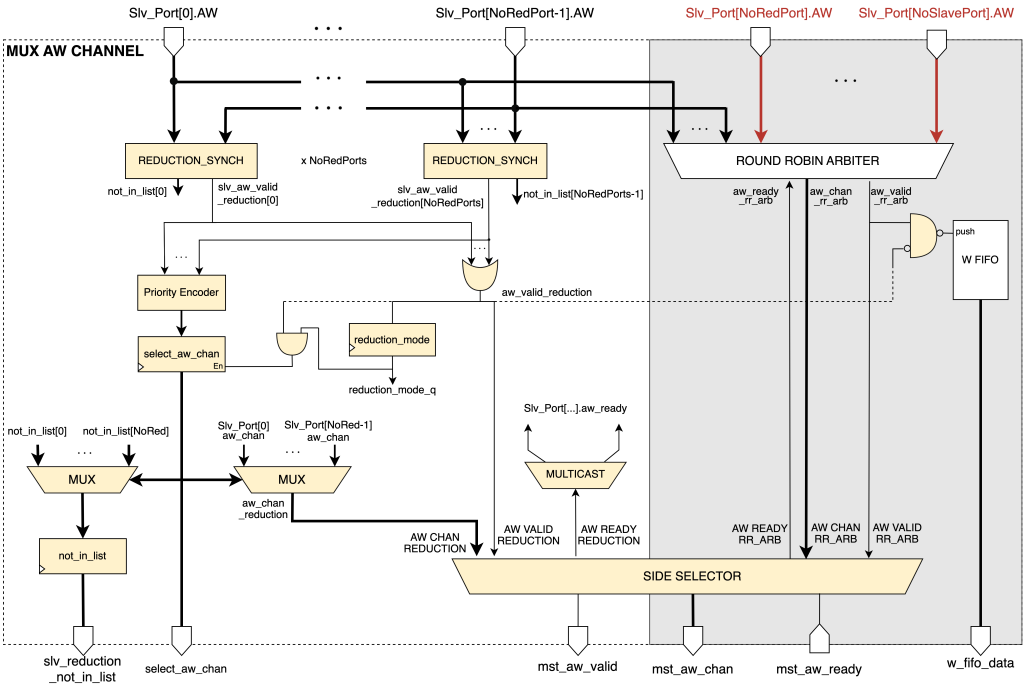


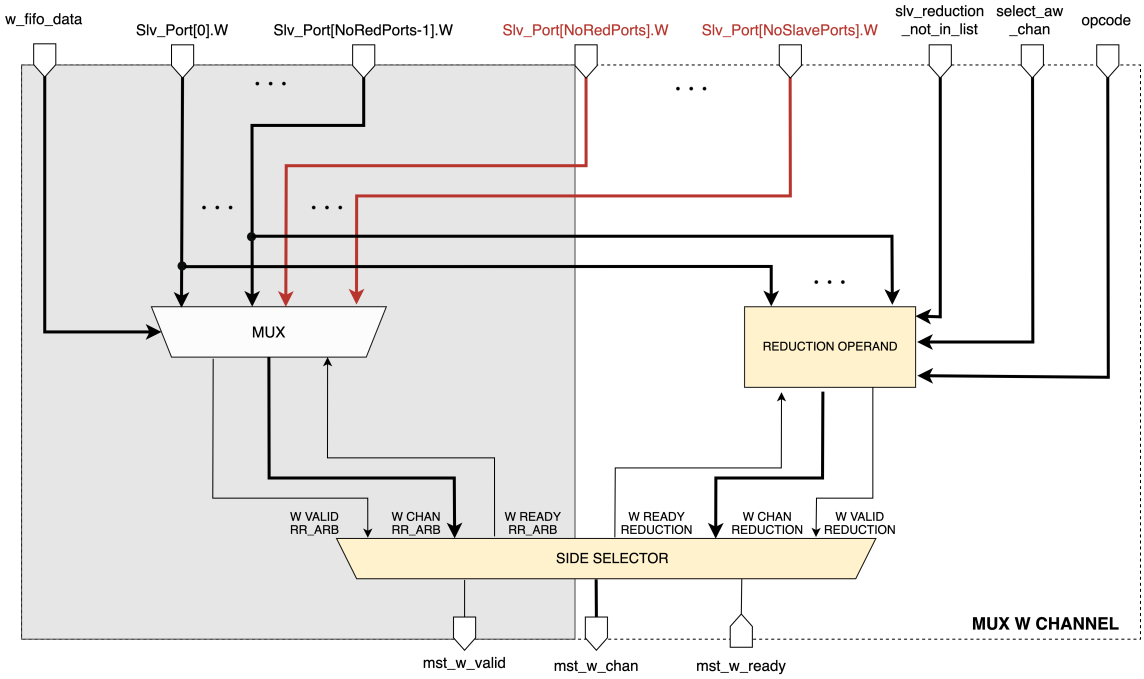
Figure 4.1: AW channel extension to support mixed type ports, reduction and normal slave ports

Fig.2.8b. Each group is linked to the SoC crossbar through dedicated master and slave ports. To simplify the address space representation, these ports are not associated with any address rules. By enabling the default port, any requests that do not match any rules are automatically routed to these ports.
However, this approach conflicts with the reduction rules introduced earlier. To represent the participants list, all reduction ports must be associated with an address rule (3.1). Therefore, the SoC slave port cannot be used to issue reduction requests because the current R-XBAR implementation cannot identify that port as a reduction source. Nevertheless, this port can be utilized as a destination for reductions, enabling reduction operations among all clusters and groups in the system, facilitating global synchronization mechanisms. The drawback is that reduction operations must always traverse the entire interconnection subsystem.

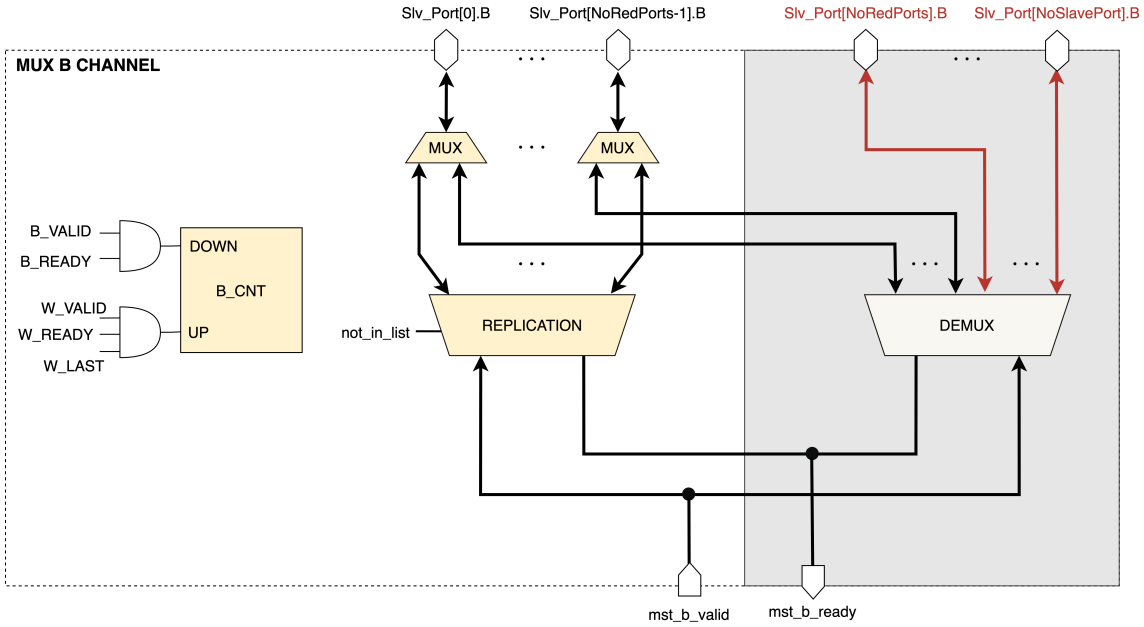
Let's consider a scenario where a reduction needs to be performed among all clusters from group 0 and 1, with the result stored inside the first cluster's TCDM. The optimal approach for such a reduction would be as follows:

1. The items from the clusters inside group 0 reduce in the first XBAR level, yielding a partial result. Simultaneously, the items from the second group of clusters reduce inside their group.

2. Since the final destination is inside the first group, there is no need to forward the partial result from the first group to the second XBAR level. Instead, it's necessary only to move the result from the second group upward to the SoC



(a) W channel extension



(b) B channel extension

Figure 4.2: Extensions to support mixed type ports, both reduction and normal slave ports.

level and then downward to the first group.

3. Finally, the XBAR can evaluate the final result in the first group XBAR and store the data in the destination.

This workflow is illustrated in Fig.4.3a, where the red arrows identify the path covered by the partial results, while the blue arrow represents the path taken by the final result to reach the destination. This approach minimizes unnecessary data exchanges with the next level, resulting in shorter latency and a more balanced load across the network. It also helps reduce the number of hotspots in the interconnection system. However, to forward the partial result from the second group deep into the first group's crossbar, the slave port connecting the SoC XBAR with the first group must be configured as a reduction port. Since this port is defined as the default in the L1 XBAR, this cannot be done.
Thus, the current version of R-XBAR does not support this feature. To execute the described reduction, the destination port must reside within the SoC XBAR. In this scenario, partial results are received from both groups, and the ultimate value is calculated and stored in any of the connected slave units. The data flow is illustrated in Figure 4.3b.

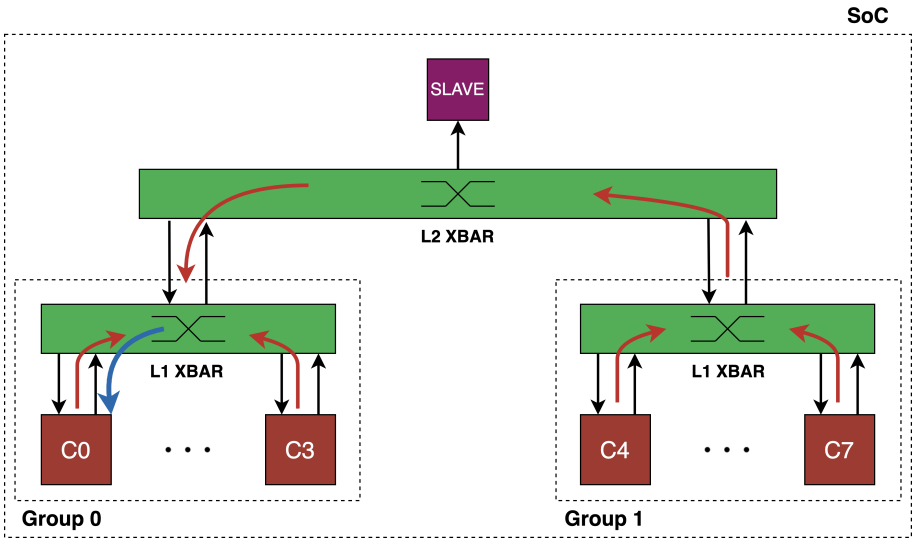
In conclusion, the Python script has been modified to fulfil all the aforementioned requirements. The outlined rules do not impose significant restrictions on the original system; they slightly limit the flexibility compared to the original plan. However, further extensions can be incorporated into the Reduction-capable XBAR (R-XBAR) to eliminate such limitations, thereby enhancing the final performance even further.

4.2 Snitch Extension

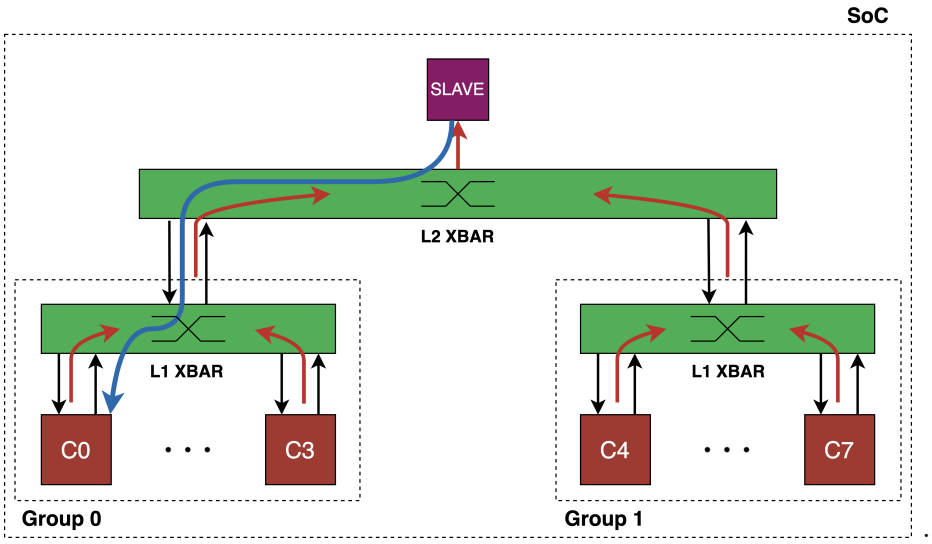
In order to utilize reduction operations within the Reduction-capable XBAR (R-XBAR), the Occamy extension alone was insufficient. Additionally, the internal Snitch cores required an extension to issue reduction requests. Specifically, the designed crossbar can execute reductions when the input write request is associated with a user field carrying the list of participants. This information needs to be set by the user during the software development process to be executed on the architecture. Therefore, several strategies for setting the reduction mask have been considered. Two solutions have been analysed:

1. **ISA Extension:** One possibility involves extending theISA by adding some instructions for reduction management.

2. **CSR:** Another possibility is to leverage the CSR features already supported in the RISC-V ISA.



(a) Asymmetric reduction tree



(b) Symmetric reduction tree

Figure 4.3: Example of reduction data flow in a simplified version of the Occamy system. In Figure 4.3a, the final result is computed directly in the L1 Crossbar (XBAR) close to the final destination. This asymmetric structure reduces the latency required to store the result in the final destination. In Figure 4.3b, the final result is evaluated in the L2 XBAR, stored in a slave, and then retrieved back from Cluster 0, which is the actual destination. This second approach results in greater latency.

The first approach would necessitate extending the entire ISA to introduce a few instructions. Conversely, with the second approach, no further instructions need to be introduced in the microarchitecture: the RISC-V ISA already supports CSR [24, Chapter 2], which can be utilized to set the mask and opcode for crossbar reduction requests. Hence, the second approach has been chosen to extend Snitch capabilities.

4.2.1 Control Status Registers

The RISC-V architecture encompasses both privileged and unprivileged Instruction Set Architecture [24, Chapter 1]. At any given time during execution, a RISC-V hart (hardware thread) is running code at some privilege level. These levels are useful for ensuring protection between different elements in the software stack. Three main levels can be identified:

- **User:** The *U* mode is intended for application codes.
- **Supervisor:** The *S* mode is used by the operating system.
- **Machine:** The *M* level has the highest privilege and is mandatory for any RISC-V architecture.

The RISC-V "Zicsr"extension supports a separate address space for 4096 Control Status Registerss. These registers can be used for various functionalities, and some of them can be utilized for custom read/write operations. Table 4.1 depicts the instruction format for Control Status Registers handling. The first 12 bits in the

Table 4.1: Control Status Registers instruction format

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

CSR format represent the address at which the register is mapped. The upper 4 bits are utilized to identify whether the register is read/write or read-only. Additionally, bits `csr[9:8]` encode the lowest privilege level that can access the register.
For reduction purposes, `CSRRW` and `CSRRWI` are the main operations that can be used. To send reduction requests, the core requires two pieces of information: the mask and the reduction opcode. Two control status registers can be employed to hold their values, and they can be set and cleared using the aforementioned instructions. Specifically, `CSRRW` atomically swaps values between the source and control status register. With `CSRRWI`, an immediate value can be used instead of the source register, facilitating the cleaning of the CSR.

Therefore, two control status registers have been allocated. The first, called `CSR_MREDMASK`, holds the reduction mask encoding the list of participants, while the second register, named `CSR_MOPCODE`, is dedicated to storing the reduction opcode to specify which of the reduction operations must be executed inside the R-XBAR. Both CSRs have been selected from the machine-level list. Thus, no protection is provided against incorrect application codes.

4.2.2 Core Extension

Once the technique for setting the reduction mask and opcode was identified, the core was extended to support reduction requests.
Initially, the need for the AW user field necessitated extending the internal interfaces to include this field as well. Moreover, the introduction of the aforementioned control status registers was insufficient to issue reduction requests. As explained in Section 3.1, during the design of the R-XBAR, it was decided to utilize write transactions to issue reduction requests. From the processor's perspective, these transactions are simple *STORE* operations. Therefore, the information to be sent alongside the write operation must be routed through the LSU. Additionally, this module needed to be modified to forward the contents of the Control Status Registers (CSR) while adhering to the expected protocol at the LSU interface.

Load-Store Unit

Before delving into the details of the Load-Store Unit extension, a brief description of the original structure is provided below.
Two FIFOs are utilized to handle load/store requests. The first one is essential for storing all the information related to load operations. This includes the destination register *tag* to identify the r_d where the loaded data must be written back, the *signed* flag to properly shift the value upon reception, the address location *offset*, and the data *size*. All this information is necessary for managing the data during the write-back stage. In the case of a store operation, such information is not required.
A second FIFO is then used to detect whether the accepted request was a store or a load. This information is necessary to determine how many loads are outstanding and to assert the response valid signal at the interface between the core and the LSU only upon completion of a load; stores always proceed without a response to the core.
The protocol used is a custom interface called *reqrsp_interface*, based on common principles found in other interconnects such as AXI. It is characterized by a request and response channel, respectively named q and p. In the case of a request, the initiator (i.e., the core) asserts the valid signal, and the receiver asserts the ready signal whenever it is ready to receive the transaction. When both signals are active, as per the AXI protocol, the transaction is deemed successful.

LSU Extension

Considering that one of the primary applications where reductions can be exploited is to implement synchronization barrier mechanisms, the Load-Store Unit has been extended not only to forward the values of Control Status Registerss for generic reduction operations, but also to stall the core on barrier requests until the arrival of all participants. To achieve this, a specific reduction opcode has been associated with barriers. When this opcode is detected by the LSU, the store operation is treated as blocking. Consequently, when a core wants to synchronize with others, it can issue the reduction request and then will be automatically stalled by the LSU. Recalling the working principle of the R-XBAR, only when all the involved cores have sent the reduction request can the transaction reach the destination, which can then reply with the AXI B response. Upon receiving the store response, the LSU removes the stall signal, allowing the cores to begin executing the next instructions. Several signals have been introduced at the LSU interface:

- **red_mask_i:** This is the concatenation of the *CSR_MREDMASK* and *CSR_MOPCODE* used to set the AW user field expected by the R-XBAR.
- **red_mask_o:** This is the output user field sent alongside the store request.
- **glb_barr_stall_o:** This signal is used by the LSU to stall the core on a barrier operation.

If the core is not stalled, the input reduction mask is forwarded to the output without any delay. Therefore, when the user wants to issue a reduction request, the control status registers must be set first, and then the reduction store operation can be sent. Subsequently, the CSRs must be cleared; otherwise, all future stores would be identified as reduction requests because they would have a non-zero user field.
When the input reduction mask has the opcode associated with the barrier operation, the barrier stall signal is asserted as soon as the store handshake is detected. The stall signal can only be removed upon receiving the store response. Since multiple outstanding stores are allowed in the system, the introduction of a counter was necessary to track their responses and identify the response associated with the barrier request. Thus, every time a store is accepted, the counter is incremented, while the arrival of a store response will decrement its value. The condition to set the barrier stall signal low is that the counter has a value equal to one, and the load-store unit detects a store response. By ensuring that the counter is equal to $CNT = 1$, we ensure that the stall signal is removed in the same cycle as the response reception, allowing the core to immediately start executing the next instruction.

In conclusion, once the Snitch core has been extended to issue reduction requests and the AW user field has been expanded in interfaces from the core level upwards

to the cluster level, the Occamy system and its internal cores were ready to utilize the reduction features supported in the reduction-capable XBAR.

4.3 Software Development

Once the system was ready for testing, the benchmarking phase of the project could commence. This phase aims to assess the correct behaviour of the developed Reduction-capable XBAR (R-XBAR) and measure the performance of different applications to identify the speedup gained with the extended crossbar. In this section, various setups are presented to understand how the results shown in chapter 5 were obtained. As explained in section 3.2, the designed crossbar currently supports only logic AND reductions. However, this operation can be leveraged to implement barrier mechanisms for global synchronization. For this reason, all the experiments study the speedup of the global barrier and programs that exploit this synchronization mechanism for parallel computations.
Before delving into the details of the experiments, section 4.3.1 illustrates how the synchronization mechanism can be implemented either in software or hardware. Then, three experiments have been set up: two aimed at assessing the performance of the global barrier under different conditions, explained in sections 4.3.2 and 4.3.3, while the last experiment, analysed in section 4.3.4, utilizes a sorting algorithm to evaluate the impact of barrier acceleration on a real application.

4.3.1 Barrier Synchronisation

Barriers are specific examples of reduction operations [12] commonly used in parallel computing to synchronise actors working on parallel tasks towards a common goal. Barriers delineate a specific step in the algorithm where each processing unit cannot proceed further until all other participants have reached the same point. A common paradigm in parallel programs is the "fork-join" model, where a task is divided into subtasks that are executed concurrently (forked) and then joined back together to form the result or resume sequential execution. In many cases, all parallel units must be synchronized. For example, some graph algorithms like Breadth-First Search (BFS) [25] can be parallelized using the fork-join model, where different nodes are investigated in parallel by multiple tasks. However, at each level of the BFS traversal, all tasks must synchronize to ensure that the next level is processed correctly. Another example is parallel matrix operations, where each processor computes a sub-matrix product. In this case, some processors may need to wait for results from others before they can proceed with their computation, and global barriers can be exploited to synchronize all processors after each phase of the computation.
Therefore, sometimes the percentage of time spent on synchronization is comparable to computation time, constituting a non-negligible overhead of the fork-join model.

For this reason, the global barrier mechanism has been chosen as an example to demonstrate the performance improvement introduced by hardware support for reduction operations.

Software Implementation

The common way to implement a barrier is by means of a shared counter. Each time a processing element reaches the barrier, it increments the counter. Only when the counter's value corresponds to the number of participants does it indicate that all units can proceed to the next step in the program. The pseudo-code in Algorithm 1

Algorithm 1 Non optimised software barrier

Require: $shared\_cnt \leftarrow 0$

```
_atomic_add(shared_cnt)
if shared_cnt = N then
  shared_cnt ← 0
else
  while shared_cnt != 0 do
    ;
  end while
end if
```

depicts an example of how the barrier can be implemented. As illustrated, once the counter is updated, all cores enter a loop except the last, which resets the counter to free the other units. The main reason why the software barrier can be time-expensive is the increment of the shared counter. Indeed, this operation must be done atomically, but atomicity might create a lot of contention. If two or more cores try to access the counter simultaneously, only one of them can be served, and thus the others must wait. This results in serial access to the shared variable, which can increase the overall synchronization time.
Additionally, the above algorithm has some other problems:

- **Loop Condition:** The inner loop condition stalls all units until the last core arrives. If only a single counter is employed, some cores might be stalled indefinitely. For instance, if two consecutive barriers are performed, it's possible that after resetting the counter, some units reach the barrier before others from the previous synchronization step exit the loop. If those units increment the counter value, the loop condition would remain valid, preventing the cores associated with the previous barrier from exiting the loop. The only way to avoid this situation is to use a second counter to track the "barrier iteration," as shown in Algorithm 2.

- **Memory Latency:** Considering the system architecture in Fig. 2.8, the shared counter is stored in the HBM. This memory location is far from all

the Processing Elements (PEs), and therefore the latency to atomically read and increment its value can be high. To mitigate this issue, the counter can be stored inside the first cluster's TCDM, where all other units can access it faster.

- **Instruction Cache Miss:** Another significant issue arises from the possibility of instruction misses. To accurately estimate the execution time for the software barrier, it's crucial that all phenomena not related to the execution of the algorithm don't affect the outcomes. The counter is reset by the slowest core, but if the barrier is executed multiple times, at each iteration, a different core might clean the counter. Consequently, with this version of the algorithm, it's challenging to control that no instruction misses occur because theoretically, for each of the N barriers executed, a different core might reset the counter, resulting in a cold miss because it has never fetched the instruction before. To address this, the pseudo-code can be modified as shown in Algorithm 2, where the task to reset the counter is always assigned to the same processing unit.

Algorithm 2 Optimsed software barrier

Require: $shared\_cnt \leftarrow 0$, $shared\_barrier\_iter \leftarrow 0$ // Stored in TCDM

```
_atomic_add(shared_cnt)
loc_barr_iter ← shared_barrier_iter
if core number = 0 then
  while shared_cnt != N do
    ;
  end while
  shared_cnt ← 0
  _atomic_add(shared_barrier_iter)
else
  while loc_barr_iter = shared_barrier_iter do
    ;
  end while
end if
```

In the optimized version of the software barrier, all units check the value of a barrier iterator except for the first core. This first core is stalled until the shared counter reaches the maximum value. Then, it resets the counter and increments the barrier iterator, freeing the other units. In this way, all cores are guaranteed to exit from the loop properly synchronized.
Additionally, the memory latency problem is addressed by storing the shared counter and barrier iterator in the first cluster's TCDM. This ensures faster access for all units. Furthermore, cache misses can be easily mitigated by repeating the code twice. Since at the second iteration, all cores will have the necessary instructions already in the cache, potential cache misses are effectively removed.

Hardware Implementation

To perform the barrier in hardware, the algorithm in Algorithm 3 can be utilized. With the `GCC Extended Asm`, inline assembly instructions can be directly used in the C code. For instance, the `set_mask` function simply utilizes the RISC-V `CSRRW` instruction to set the value of the CSR reduction mask. Similarly, the CSR opcode is defined with the `set_opcode` function. Finally, for both control status registers, two C functions can be used to reset them. These functions are based on the `CSRRWI` RISC-V instruction, with the immediate field set to zero.
Thanks to this bare-metal code, the user can exploit the reduction capabilities of the developed system. For global barrier synchronization, the user only needs to define the reduction mask following the methodology explained in section 3.1 to select the appropriate list of participants. Additionally, the opcode must be set to zero, which corresponds to the barrier operation code.

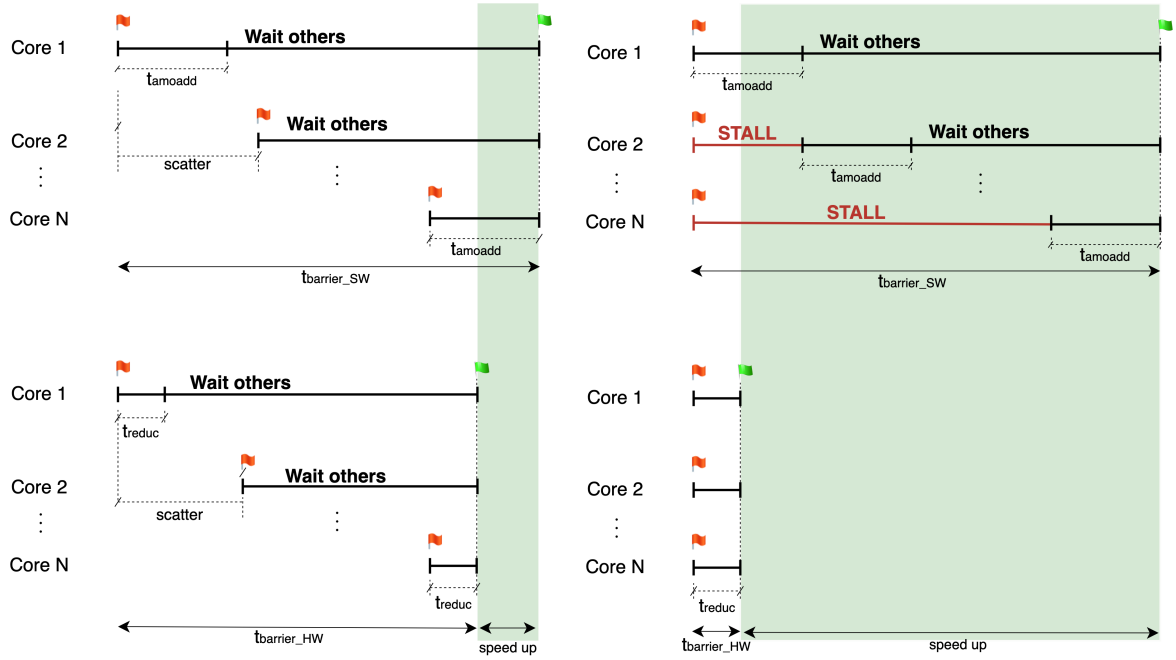
Algorithm 3 Hardware barrier

$set\_mask(RED\_MASK)$
$set\_opcode(RED\_OPCODE)$
$clear\_mask()$
$clear\_opcode()$

4.3.2 Barrier arrival times

Once all necessary functions to support the software and hardware barriers in C code were developed, experiments could be conducted to evaluate performance and verify the correct behaviour of the entire system.
The first interesting aspect to evaluate was the dependency of the barrier speedup on the arrival time of each participant at the barrier. The main drawback of the software barrier is the usage of `atomic` operations, which can create contention and force parallel code to be serialized because only one computation unit can access the shared variable at a time. Conversely, the hardware barrier was implemented to handle parallel requests. Therefore, the arrival time of each core at the barrier constitutes a critical aspect of the overall application.
As depicted in Fig. 4.4a, if the participants are scattered enough during arrival at the barrier, the overhead introduced by the use of atomic accesses becomes negligible because each unit has enough time to fetch and increment the shared counter before the next PE requires access to the same memory location. Thus, the speedup introduced by the hardware implementation is solely in the last step between the arrival of the slowest unit and completion of the synchronization mechanism. When the number of cores is high, that speedup may be small compared to the overall application.
In contrast, the worst scenario for the software barrier occurs when all cores arrive

simultaneously. As illustrated in Fig. 4.4b, regardless of whether all cores arrive at the same time, the atomic operation can be performed by only one unit at a time. Therefore, each core stalls waiting for the previous one to fetch and increment the shared counter. Consequently, if t_{amoadd} represents the time necessary to atomically fetch and increment the variable, the last core will wait for a total of $t_{\text{N wait}} = (N-1) \times t_{\text{amoadd}}$ before it receives permission to access the counter. Instead, the hardware barrier is capable of managing all requests in parallel, and therefore the execution time of the total barrier is simply AXI write transaction round-trip delay. To test the speedup gained with the hardware barrier under dif-



(a) Scatterred arrival time

(b) Simultaneous arrival time

ferent conditions, the arrival time of each core can be varied in a controlled fashion, and the performance evaluated. Each core can generate a random delay drawn from a uniform distribution, wait for that time, and then send the barrier request to notify its arrival. Algorithm 4 outlines the main steps of this process.

Algorithm 4 Speed up test with different arrival times

$delay \leftarrow prng\_delay(MaxDelay)$
$hw\_barrier()$
$wait\_for(delay)$
$barrier\_request()$

Delay generation

The generation of delays constituted a crucial step in the experiment. In order to maintain sufficient control over the setup, a pseudo-random number generator func-

tion was devised. The Leapfrog Method [26] was employed for generating numbers from a uniform probability density function ranging from zero to MaxDelay, a constant value adjustable between experiments. For instance, to examine the scenario where all cores reach the barrier simultaneously, MaxDelay could be set to zero.

$$delay \sim U(0, MaxDelay) \tag{4.1}$$

Synchronisation mechanism

Controlling the arrival time requires synchronizing all cores at the outset. Hardware global barriers rely on the AXI B response. When all cores issue the reduction request and the B response is sent back from the destination, it arrives simultaneously at all participants. Consequently, if each core is assigned the same workload, all computational units will remain synchronous after receiving the B response. Therefore, to ensure participant synchronicity before awaiting the computed delay time, the hardware barrier mechanism can be utilized.

Waiting routine

After computing the delay time, each core must be stalled for that duration. This task can be accomplished by employing a polling loop, which essentially halts core execution for a certain number of iterations proportional to the delay time. Before measuring barrier performance, multiple executions were conducted to determine this proportionality term. Analysing the generated assembly code enabled the determination of the required number of cycles to execute one iteration of the loop. Subsequently, using this information, the total number of iterations could be computed based on the generated delay.
It's worth noting that the constant value correlating the number of iterations with the delay depends on the target machine and the compiler used. Hence, all pertinent information necessary for replicating the experiment will be detailed in the Results chapter 5.

Barrier request

After cores have waited for the computed delay, the barrier request can be issued. To assess the speedup introduced by the hardware barrier, both the software and hardware algorithms explained in section 4.3.1 can be utilized.

4.3.3 Number of participants impact

Another aspect to consider when analysing barrier performance is the number of participants. As previously noted, the primary overhead in the software version stems from contention generated by the `amoadd` instruction. Consequently, increasing the

number of participants leads to more accesses and, consequently, more time spent on synchronization. Conversely, the hardware version should remain unaffected by the number of participants because the design was implemented in a manner such that all requests in the same reduction can be detected simultaneously, thanks to the tree structure illustrated in Fig. 3.3b.

Only the cores involved in the computation issue the barrier. Recalling the limita-

Algorithm 5 Speed up test for different number of participants

```
if core_idx < NumParticipants then
  hw_barrier()
  barrier_request()
end if
```

tions of the R-XBAR, the number of participants must be a power of two. Additionally, since in this experiment the effect of different arrival times is not important, before initiating the barrier request used to measure performance, a hardware barrier is executed to ensure that all cores are synchronised and the delay among them is zero.

4.3.4 Bitonic Sort algorithm

In conclusion, after studying the performance of global barriers, a real application was implemented to assess the impact of the provided synchronization acceleration on the overall program. Various applications utilize barriers, such as the BFS algorithm, the FFT, certain matrix multiplications, or sorting algorithms. To evaluate performance under real conditions, the *Bitonic Sort* algorithm was selected.

Basis

First devised by Batcher [27], the bitonic sort is renowned as one of the fastest sorting algorithms [28]. A sequence of numbers is defined as bitonic if it comprises two subsequences, one in ascending order and the other in descending order (or vice versa). An example is the sequence s shown in equation 4.2, where the first four elements are ascending while the second half is descending.

$$s = 1, 2, 9, 13, 11, 8, 5, 3 \tag{4.2}$$

The sorting algorithm can be generalized for any length L that is not a power of two. However, for the purpose of this experiment, the length is always a power of two. Given a generic bitonic sequence of length $L = 2 \times N$, two bitonic sequences s' and s'' can be created by applying the rules in equation 4.3. It can be demonstrated that none of the elements in s' are greater than those in s''. By repeating this procedure for each of the obtained sequences, four bitonic sequences of length $N/2$ are

obtained. This procedure can be repeated for $\log_2(L)$ steps, and in the final step, L bitonic sequences of length 1 are obtained, leading to a sorted sequence.

$$s' = \{\min(s_1, s_{N+1}), \min(s_2, s_{N+2}), ..., \min(s_N, s_{2N})\} \quad (4.3)$$

$$s'' = \{\max(s_1, s_{N+1}), \max(s_2, s_{N+2}), ..., \max(s_N, s_{2N})\} \quad (4.4)$$

Implementation

Figure 4.5 illustrates the sorting algorithm applied to the example 4.2. The arrows represent comparison operations between two vector elements. At each step, a total of $L/2$ comparisons are performed, and they can be executed in parallel by $L/2$ cores. However, before proceeding to the next step, a barrier must be employed to ensure that all involved units have completed their tasks, as each core might work on elements swapped by another before. The total number of barriers required is equal to $\log_2(L)$. To evaluate the algorithm's performance more effectively by

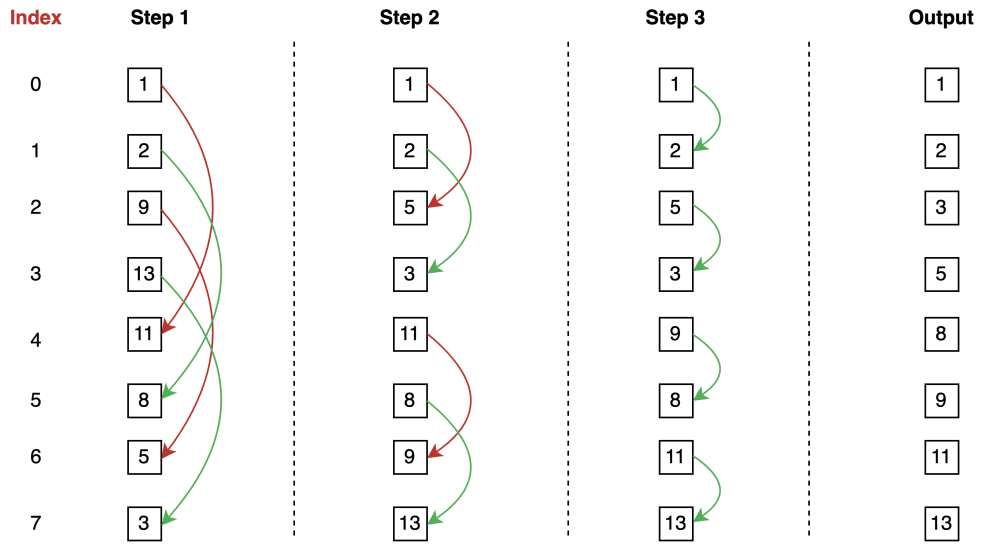


Figure 4.5: Bitonic sort algorithm applied to the sequence 4.2

exploiting the hardware barrier, an initial implementation was developed, and the corresponding pseudo-code is presented in Algorithm 6. The indexes of the elements to be compared can be computed as a function of the core index, as shown in equation 4.5, where $core_idx$ represents the core index obtained from a built-in function, *stride* indicates the distance between the elements being compared, and % denotes the modulo two operation. As depicted in Fig. 4.5, in the first step, the stride is equal to half of the input sequence length, then it becomes one fourth, and so forth.

$$seq\_idx = \lfloor core\_idx/stride \rfloor * len + core\_idx\%stride \quad (4.5)$$

Once the algorithm was implemented, the performance evaluation yielded unsatisfactory results, with a speedup of less than 3% for a sequence length of 8 elements.

Algorithm 6 Non-optimal bitonic implementation

```
if core_idx < len/2 then
  while len > 1 do
    stride ← len/2
    seq_idx ← core_idx/stride * len + core_idx%stride
    val1 ← sequence[seq_idx]
    val2 ← sequence[seq_idx + stride]
    if val1 > val2 then
      swap(val1, val2)
    end if
    barrier_request()
    len ← len/2
  end while
end if
```

Upon analysing the assembly code, it was discovered that the time spent on synchronization was negligible compared to the rest of the program. The primary issue stemmed from each core's access to the sequence. Specifically, the sequence was stored in the HBM (Fig. 2.8a), which is shared among all cores in the Occamy chip. The utilization of this shared memory led to two critical considerations:

- The latency could be significant since the HBM is located far from the cores.
- All cores attempted to access the memory simultaneously, creating contention at the interface and further increasing access time.

Consequently, optimizing the algorithm for the Occamy architecture became fundamental to minimize computation time as much as possible and assess the impact of barrier acceleration on the overall program's execution time.
Since the primary inefficiency stems from the sequence's location, the optimization revolves around storing the elements inside the TCDM of each cluster to improve their locality. Essentially, at the beginning of the program, each core moves the first pair to be compared from the HBM into its TCDM. From that point onwards, each core performs the comparison and instead of writing the data back to the main memory as done in Algorithm 6, it directly writes them into the TCDM of the cluster, which will work on the same elements in the next step.
Each core should identify the next location by simply knowing the current step. Unlike the first version, computing the next core index cannot be achieved with a closed-form equation like in 4.3 because the floor function $\lfloor\ \rfloor$ is non-injective, meaning various inputs can yield the same results. To discern a pattern for determining the next core index based on the algorithm step, the diagram in Fig. 4.6 was devised. This schematic represents the data flow between cores for a sequence of 16 elements. The values within each node denote the element indexes on which each core is operating at a specific step. For instance, during the first step, the first core

will compare element zero with the 8^{th}; in the second step, the comparison will be between 0 and the 4^{th}, and so forth. In each step, every cluster must transmit only

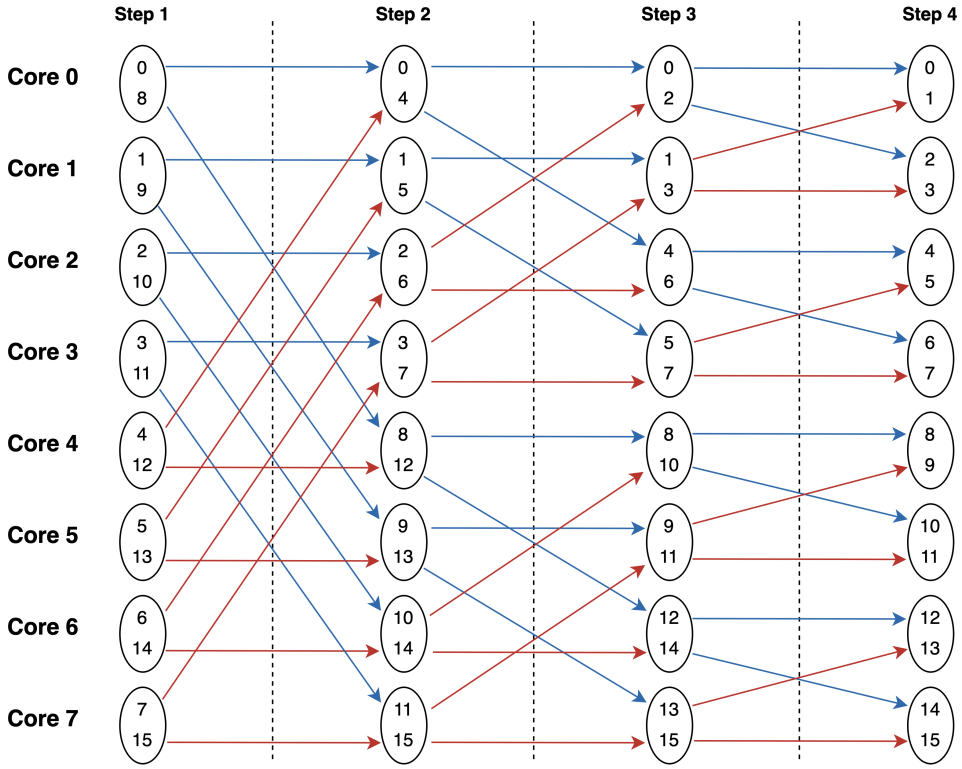


Figure 4.6: Bitonic sort algorithm data flow for a sequence of 16 elements.

one of the two elements, which can be either the first or the second. Examining the first iteration, the first half of the cores retain the first element and transmit the second to the core situated four indexes away ($Core_0 \rightarrow Core_4$, $Core_1 \rightarrow Core_5$, etc.). Conversely, the second half retains the second element and transmits the first to the core located four indexes before ($Core_4 \rightarrow Core_0$, $Core_5 \rightarrow Core_1$, etc.). As mentioned in [27], after the first step, two bitonic sequences are obtained, allowing the same algorithm to be applied to them independently. Moreover, since none of the elements in the first sequence is greater than those in the second, there won't be any data exchange between the first and second half of the cores. Focusing on the first four cores, the same data flow described previously can be observed, i.e., the first half retains the first element and transmits the second, while the second half does the opposite. With the flow illustrated in Fig. 4.6, each core can determine in which TCDM the processed data must be written, enhancing their locality and consequently reducing the synchronisation-free region time. This optimized version of the bitonic algorithm enables the evaluation of the speedup gained with the hardware barrier.

The tasks assigned to each core are not overly complex; they simply involve comparing two values and determining the address of the next TCDM location where the data must be written. Therefore, the percentage of code devoted to synchronization is comparable to the synchronisation-free region. To observe the effect of compu-

tation time on the final speedup, the sequence length can be increased to ensure that the number of comparisons to be performed exceeds the available cores in the system. This increases the workload distributed to each core and artificially extends the length of the synchronisation-free region. In conclusion, the original implementation was further expanded to support cases where the system lacks sufficient cores to parallelize the entire code.

Scaling

With the designed algorithm, various types of studies can be analysed. In parallel computing applications, two scaling models can be evaluated [29]:

- **Strong Scaling:** In this scenario, the parallel code is executed on more computational units without altering the total problem size. Consequently, the workload assigned to each unit decreases.
- **Weak Scaling:** In this model, not only is the number of cores increased, but the problem size is also enlarged. This results in a constant workload per PE.

These two models aim to examine different aspects. Strong scaling applications are those categorized as *CPU-bound*, where the majority of time is spent on computation. Conversely, weak scaling is often used for *Memory-bound* applications. The bitonic sort algorithm does not necessitate extensive computation since the tasks performed are limited. However, a significant portion of the total time is spent on data movement between cores and synchronization. Both scaling aspects can be analysed with the described setup:

1. The strong scaling can be studied by utilizing the feature introduced in the latest version, i.e., selecting a specific sequence length and executing the code on different numbers of cores. The workload assigned to each core will vary, but the problem size remains constant.
2. Conversely, weak scaling can be examined by simultaneously increasing the number of cores and the sequence length.

From this study, it's possible to determine which, between the software and hardware barrier, scales better with the problem and architecture size.

To summarize, this chapter has presented various applications and experimental setups. All results and performance metrics are detailed in Chapter 5.

Chapter 5

Results

The synthesis and performance results are presented in the following chapter. Section 5.1 focuses on the outcomes obtained after synthesizing the designed reduction-capable crossbar, with a primary focus on the area overhead introduced by the extension. Subsequently, in section 5.2, the performance metrics associated with the software experiments described earlier are analysed. Specifically, the impact of arrival time at the barrier is studied in 5.2.1, while the effect of the number of participants on the gained speedup is detailed in section 5.2.2. Finally, all considerations and outcomes derived from the bitonic sort algorithm are summarized in 5.2.3.

5.1 Synthesis

In this section, the implementation of several reduction-capable XBAR instances is analysed in terms of area and performance.
The crossbar interconnection system was synthesized for GlobalFoundries 12nm technology using Synopsys Fusion Compiler 2022.03. The place and route was carried out with the same tool. For this technology node, one Gate Equivalent (GE) is equal to $0.121\mu m^2$. Table 5.1 summarises all the synthesis parameters: To synthe-

Table 5.1: Synthesis Parameters

# Masters	$\in [2, 4, 8, 16]$
# Slaves	$\in [2, 4, 8, 16]$
Enable Reduction	$\in [0, 1]$
Reduction Id Registers	$\in [0, 1]$
Spill Registers	$\in [0, 1]$
Operating Corner	SS/0.72 V/125°C TT/0.80 V/25°C FF/0.88 V/-40°C
Target Frequency	1 GHz

size the system, Synopsys Fusion Compiler has been used to perform both the logic and physical synthesis. In the latter phase, not all the steps usually included in the

physical implementation have been executed. In the *floorplan* step, a utilization of 0.2 has been chosen, and no other constraints were given to the tool. Then, all the steps towards the *routing* have been performed except for the *clock tree synthesis* (CTS). The main objective for the project was to obtain area metrics of the R-XBAR to be compared with the original version. Therefore, many of the physical synthesis steps have been executed, providing the possibility to obtain more realistic results. Indeed, in a fully connected crossbar, routing can be the critical aspect, and logic synthesis alone would not be detailed enough to evaluate this aspect thoroughly.
All results shown in the following sections are relative to the TT/0.80 V/25°C corner.

5.1.1 Area

As shown in table 5.1, different R-XBAR instances have been synthesized to determine the area overhead introduced by the reduction logic. Only symmetrical crossbars with the same number of master and slave ports were synthesized. Given the possibility to have spill registers either at the input or output ports, the two extreme cases with and without registers were analysed. Additionally, recalling from section 3.2.4 the limitation on the reduction IDs, both versions with and without ID registers were synthesized to evaluate the cost in terms of area introduced by the more flexible version.

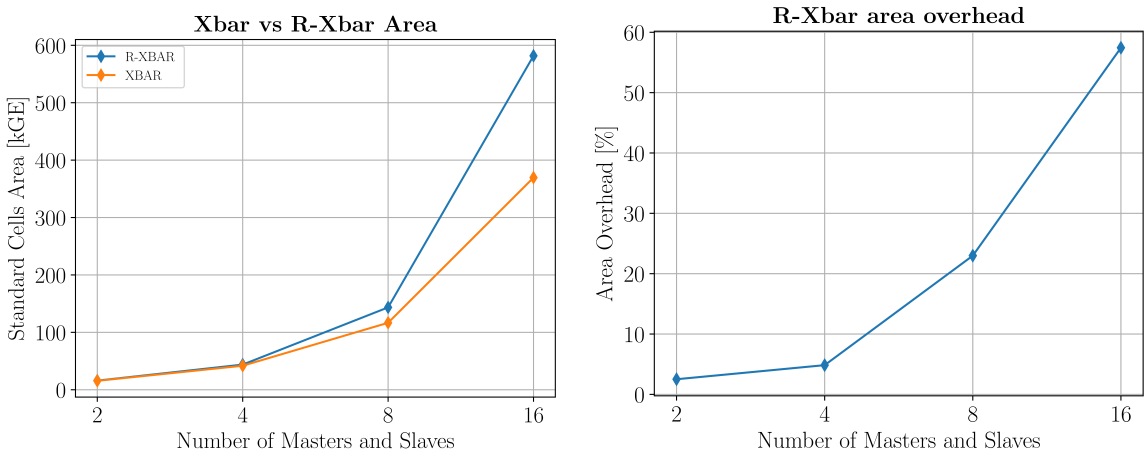
No Spill registers

The first synthesized version doesn't include spill registers. Figure 5.1 illustrates the area of the original and reduction-capable XBAR for a number of master and slave ports ranging from 2 to 16. From the regression diagram, it can be observed that the overhead is below 5% for a crossbar interconnecting less than 4 masters and slaves. For an 8×8 configuration, the reduction-capable version occupies 23% more area, while for large systems with 16 masters and slaves interconnected to the same XBAR, the reduction logic can lead to an interconnection system 57% bigger than the original version. Regarding the latter condition, two considerations must be made:

- First, it's not common to have a 16×16 crossbar in a large system because the total area and the interconnection latency can be too large. Considering multi-core systems like Occamy, a more realistic situation is to use crossbars with fewer than eight masters and slaves; then, all cores are grouped in a hierarchical architecture, and these levels are interconnected through hierarchical XBARs as depicted in Fig. 2.8.

- Additionally, the synthesized version doesn't include spill registers. Although they are not mandatory, they are often utilized to cut the combinational path, which otherwise would be too big and might reduce the performance of the

overall system. Therefore, with spill registers, the reduction logic might introduce a lower overhead since these registers constitute a big portion of the interconnection infrastructure, and the reduction logic does not affect their dimensions too much.



(a) Area comparison

(b) Area overhead

Figure 5.1: Synthesis outcomes for reduction and original crossbar without spill registers.In both diagrams the depicted area is the total one, i.e. including sequential cells, combinational logic and also the buffer cells introduced during the synthesis to meet all the constraints.

Moreover, from the overhead diagram, it is possible to notice that the introduced logic increases quadratically with the number of master and slave ports. This result is consistent with the expectations. From the design step in section 3.2.2, it's clear that the reduction synchronization module in Fig. 3.3b has been instantiated in each multiplexer for a total of times equivalent to the number of slave ports. Therefore, in a symmetric crossbar, the total overhead can be computed as shown in equation 5.1:

$$R_{\text{XBAR\_overhead}} = RedSynch_{\text{area}} \times (NoSlavePorts \times NoMasterPorts) \tag{5.1}$$

where:

- $\boldsymbol{RedSynch}_{\text{area}}$: area of the introduced logic to manage reduction requests.
- $\boldsymbol{NoSlavePorts}$: number of masters (red_synch) connected to the XBAR.
- $\boldsymbol{NoMasterPorts}$: number of slaves (mux) connected to the XBAR.

Already from this analysis, it's clear that the majority of the introduced logic is combinational. To better study the obtained outcomes, table 5.2 identifies, for each configuration, the total area, showing how much of it is due to sequential elements and how much to combinational logic without considering the area associated with

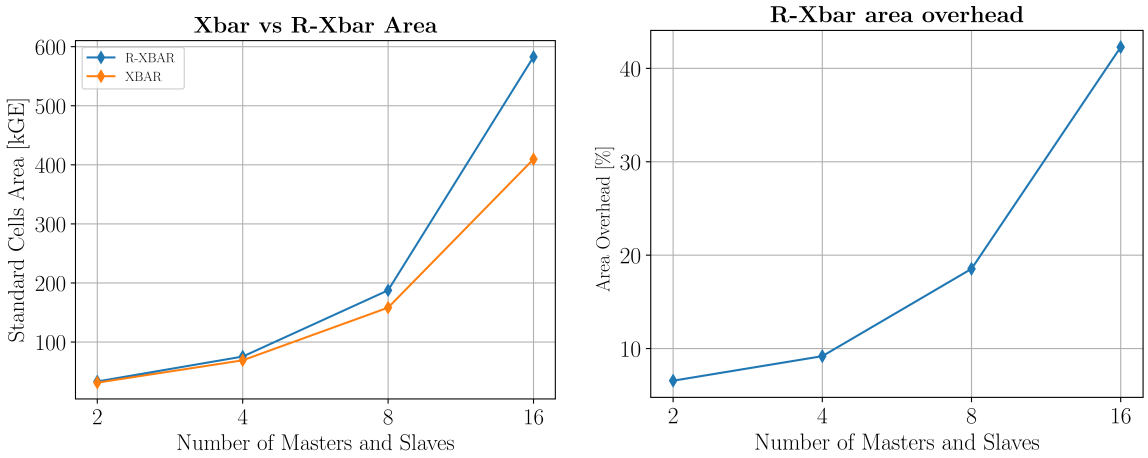
the buffers inserted during the synthesis. Let's focus on the 8×8 XBAR. The sequential area is not affected by the introduction of the reduction logic. It's a different case for the combinational area: the original system was made by 71671 GE combinational cells whereas the reduction crossbar is made up of 94121 cells, 31% more.

Table 5.2: Detailed area overview without spill registers. All results are provided in GE and the combination area does not include buffers.

Size	**XBAR**		**R-XBAR**	
	Combinational	Sequential	Combinational	Sequential
2×2	6802	5093	7074	4683
4×4	21995	11848	23773	11262
8×8	71671	27113	94121	26607
16×16	253899	62343	411016	64311

Spill registers

Figure 5.2 illustrates the overhead for a crossbar with spill registers both at the input and output ports. The introduction of a user field in the AW channel to send the reduction mask alongside the write request also influences the spill registers' dimensions. Indeed, these modules are decoupling interfaces that hold the information associated with the AXI channels at the crossbar's interfaces. Therefore, increasing the AW user field leads to wider registers and more sequential elements in the final interconnection system.
In the synthesized R-XBAR, the AW user field has a width of 34 bits, where 32 bits are allocated for the reduction mask and the two LSB are dedicated to the opcode. In a real system, the user field can be narrower. Indeed, the reduction mask can be encoded in fewer bits than the AW address width. Each port is associated with an address space, and the least significant bits of the address rule identify the locations that match the port. Therefore, these bits are not necessary to encode a reduction list because a request coming from any port will reduce data coming from some other ports with the same address space size. Considering Occamy as an example, all clusters are mapped with a 256 kiB address space, and hence the 18 LSB of the address can be omitted from the reduction mask since only the others are useful to encode the list of participants as explained in Section 3.1. Considering the synthesized version with a 34-bit AW user field, two different behaviours can be observed. For an interconnection system with less than 8 masters and slaves, the overhead introduced by the reduction logic is greater than that obtained without spill registers. Indeed, a 4×4 R-XBAR is 9% larger than a normal crossbar, whereas before it was 4.8% times bigger than the original version. For small interconnection systems, the introduction of spill registers can further increase the final area since the reduction logic added in each multiplexer is comparable to the sequential cells at the input/output ports.



(a) Area comparison

(b) Area overhead

Figure 5.2: Synthesis outcomes for reduction and original crossbar including spill registers. In both diagrams the depicted area is the total one, i.e. including sequential cells, combinational logic and also the buffer cells introduced during the synthesis to meet all the constraints.

A different behaviour occurs for configurations greater than an 8×8 crossbar. In this case, the overhead introduced by the reduction logic, considering also the AW user field expansion inside spill registers, decreases to 18%. Indeed, in big systems, the area associated with spill registers is critical, and it outweighs the combinatorial reduction logic, reducing the overall overhead. Table 5.3 illustrates for each configuration both the area occupied by the combinational and by the sequential logic.

Table 5.3: Detailed area overview including spill registers. All results are provided in GE and the combination area does not include buffers.

Size	**XBAR**		**R-XBAR**	
	Combinational	Sequential	Combinational	Sequential
2×2	7298	20303	7214	21958
4×4	19251	42448	20783	46155
8×8	53963	88787	72614	96806
16×16	185442	187027	318440	206068

ID Registers

Another configuration that has been analysed is the reduction crossbar, both with and without ID registers. This feature, as previously explained in section 3.2.4, offers increased flexibility to the user. Its implementation necessitated the instantiation of various registers to monitor the IDs associated with incoming reduction requests. Contrary to expectations, the introduction of these registers does not significantly increase the overall area, especially in the specific case of R-XBARs with spill registers. This is because each multiplexer necessitates $NoSlavePorts$ registers, each

with a width equal to the transaction ID, which, in the synthesized crossbars, was set to 4 bits. Consequently, for a 16×16 R-XBAR with spill registers, the incorporation of ID registers results in $16 \times 16 \times 4 = 1024$ flip-flops, constituting almost **1.8%** of the sequential area of a R-XBAR of the same size but lacking ID registers. In conclusion, this feature can be supported at a low hardware cost.

Considerations

To summarise table 5.4 illustrates the synthesis outcomes in different conditions. Two main aspects must be taken into account:

Table 5.4: Synthesis results for different crossbar configurations

Size	**Area Overhead**	
	No Spill Registers	Spill Registers
4×4	4.8%	9%
8×8	23%	18%
16×16	57%	42%

- **AW user**: In a real system, the user field width can be reduced, decreasing the sequential area overhead introduced by the extension. Therefore, the R-XBAR with spill registers can be smaller than those that have been synthesized.
- **Mux logic**: The main contribution to the reduction extension comes from the combinatorial logic inserted in each multiplexer. As mentioned before, equation 5.1 identifies the overhead introduced by this logic. Recalling the limitations in Section 4.1.2, in the current version of the reduction-capable crossbar, only symmetrical reductions are allowed. Therefore, considering Fig. 4.3b, there won't be any situations where the reduction request comes from a higher interconnection level downwards to a lower level. Hence, the reduction logic can be instantiated only inside the multiplexer that connects the L1 XBAR with the next hierarchical level. This will reduce the introduced overhead by a factor equal to the number of master ports, i.e., the overhead becomes equation 5.2. For the 8×8 crossbar with spill registers, the overhead might decrease up to 2.2%.

$$R_{\text{XBAR\_overhead}} = RedSynch_{\text{area}} \times NoMasterPorts \tag{5.2}$$

5.1.2 Frequency

The target frequency for the synthesized system was 1 GHz. All tested configurations met the timing, resulting in a positive slack. From a detailed analysis of the outcomes obtained after routing, it has been noticed that for large systems such as the 16×16 crossbar, the original version is characterised by a positive slack greater than the one obtained from the reduction version. Therefore, a further analysis was

conducted to evaluate the maximum frequency that could be met for different configurations. Figure 5.3 illustrates the maximum frequency for various XBAR sizes. In the case of small systems, the frequency is almost the same between the reduction and original version, while larger R-XBARs meet a slower target frequency. In the diagram, for each point, an error bar is also shown, corresponding to 5% of the target frequency. As stated in [30], Synopsys employs heuristic methods to derive the outcomes from synthesis, and this is the reason why results can differ between different runs. Therefore, a rule of thumb is to "allow 5% uncertainty in synthesis results".
The only configuration in which the original and reduction XBAR target frequencies differ by more than 5% is the 16×16. Analysing the timing report, it has been identified that the reduction logic was inserted in the critical path. In particular, increasing the crossbar dimension will also increase the reduction synchronization tree depth (Section 3.2.2), leading to a bigger propagation delay. A possible way to limit this drawback is by inserting pipeline registers in the reduction tree to cut the critical path.
Since the objective for the thesis was to reach a frequency of 1 GHz and this timing constraint is met by all configurations, the aforementioned critical path issue is left as future work.

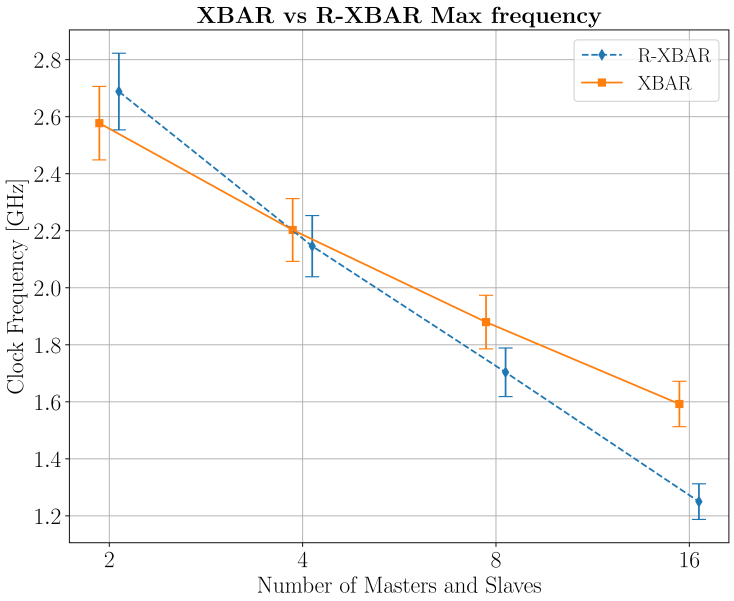


Figure 5.3: Maximum frequency for different interconnection sizes. These outcomes refer to the TT/0.80 V/25°C corner.

5.2 Benchmark

In the following sections, all results obtained from the experiments outlined in section 4.3 are presented and analysed.
To measure performance and analyse the hardware barrier, the built architecture depicted in Fig.2.8 is utilized, with the difference that 8 quadrants have been instantiated instead of 6, resulting in a 288-core architecture. Another important consideration is the granularity of the barrier. In the Snitch CC, there was already the option to execute a barrier in hardware among the 8 + 1 cores within each cluster. The introduction of the R-XBAR adds the capability to execute the barrier among different clusters and groups as well. Hence, in the following experiments, only one core per cluster is utilized, allowing for a maximum of 32 cores. This approach aims to ascertain the performance gained solely due to the reduction-capable crossbar, without any influence from the use of the already implemented hardware barrier mechanism. In the future, the hardware support within each cluster might be eliminated, and the cluster's XBAR might be replaced with a reduction-capable one to observe the impact.
In conclusion, table 5.5 summarizes all the necessary tools to conduct the same experiments and their respective versions.

Table 5.5: Tools versions

Name	Version
GCC	9.2.0
G++	9.2.0
LLVM	0.12.0
RISC-V GCC	8.3.0
QuestaSim	2022.3

5.2.1 Barrier arrival times

As elucidated in section 4.3.2, the initial experiments aimed to scrutinize the barrier's speedup for different delays among the participants. In this scenario, the maximum number of cores was utilized, i.e., each of the DM cores inside each cluster issued a reduction request. Initially, all cores were synchronized, and subsequently, after a random delay drawn from a uniform distribution $U(0, MAX_DELAY)$, a second barrier was initiated for performance evaluation. The kernel was executed twice to mitigate cold misses that could impact the results. Additionally, due to the randomness of the delay, multiple measurements were conducted, and the average was considered.
The execution time was measured as the duration between the arrival of the first core and the completion of the barrier, i.e., when the last core exits the barrier.
Figure 5.4 illustrates the obtained results in two distinct situations. Specifically,

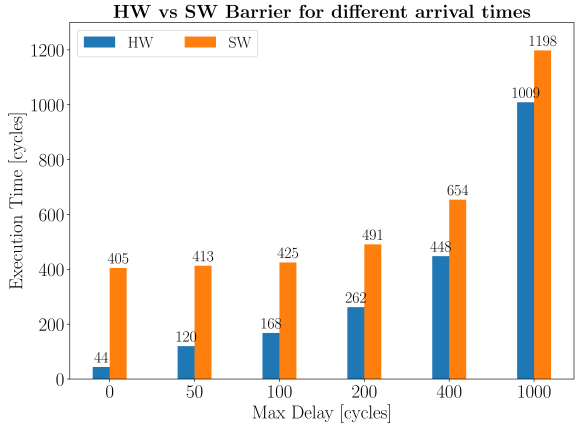
the bar diagram in Fig. 5.4a illustrates the performance of both the software and hardware barriers in the first iteration, while the second plot in Fig. 5.4b depicts the execution time obtained in the second iteration. By comparing the two diagrams, it is evident that both the hardware and software versions are affected by cache misses. Therefore, the second diagram was utilized for performance evaluation.
As anticipated, as the delays among the different arrival times increase, the speedup diminishes. In the best-case scenario, where all requests are issued simultaneously, the R-XBAR introduces a speedup of $s_{\text{bar}} = 304/44 \simeq 6.9$. However, this value decreases to $s_{\text{bar}} = 1124/978 \simeq 1.15$ when the last core issues the request after 1000 cycles from the first synchronization. These outcomes lead to several considerations:

- To achieve the maximum speedup, it is crucial to balance the workload as evenly as possible among all working units. In this manner, the PE can be synchronized first, and subsequently, the scattering among them is kept very low. This approach is feasible only with the hardware barrier because the release phase in the software version is executed by polling a shared counter, resulting in each core accessing it at different instances. Conversely, in the hardware implementation, the release phase is executed using the AXI B response signal, which is simultaneously detected by all cores if the crossbar is not congested.

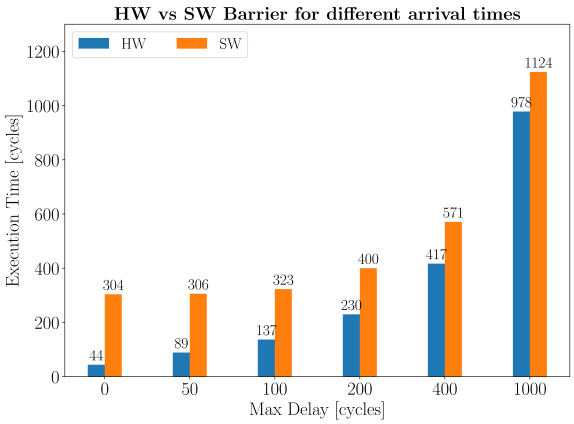
- From the result associated with $MAX\_DELAY = 0$, it is discernible that the round-trip latency for the reduction request is approximately 44 cycles.

5.2.2 Number of PEs

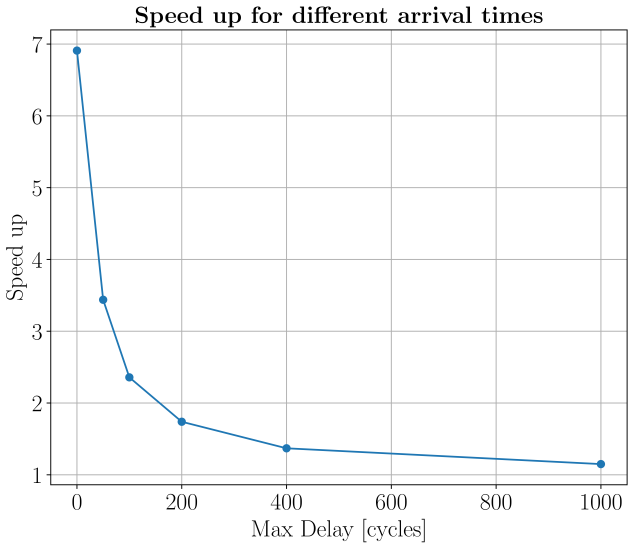
In the second experiment, the objective was to evaluate how the speedup is affected by the number of PE participating in the synchronisation. In this case, the delay between arrival times is maintained at zero to eliminate its effect from the outcomes. The utilization of a reduction mask in the R-XBAR to encode the list of participants imposes certain limitations, as elucidated in section 3.1. One of these limitations is that the number of participants must be a power of two. Consequently, only four configurations were tested, ranging from 4 to 32 cores.
Figure 5.5 illustrates the obtained results. As expected, with an increase in the number of participants, the software barrier performs poorer due to a higher number of accesses occurring simultaneously to the shared counter, which must be accessed atomically. The speedup is equal to 2.32 in the worst-case scenario and can reach a maximum value of 6.9, consistent with the findings of the previous experiment. From this study, it can be observed that the performance of the hardware barrier does not depend on the number of participants. This outcome aligns with the design, as the R-XBAR has been designed to handle all requests within the same reduction operation in parallel. This represents one of the primary advantages introduced by the reduction-capable crossbar compared to the software implementation. However,



(a) Performance evaluation affected by instruction cache misses.



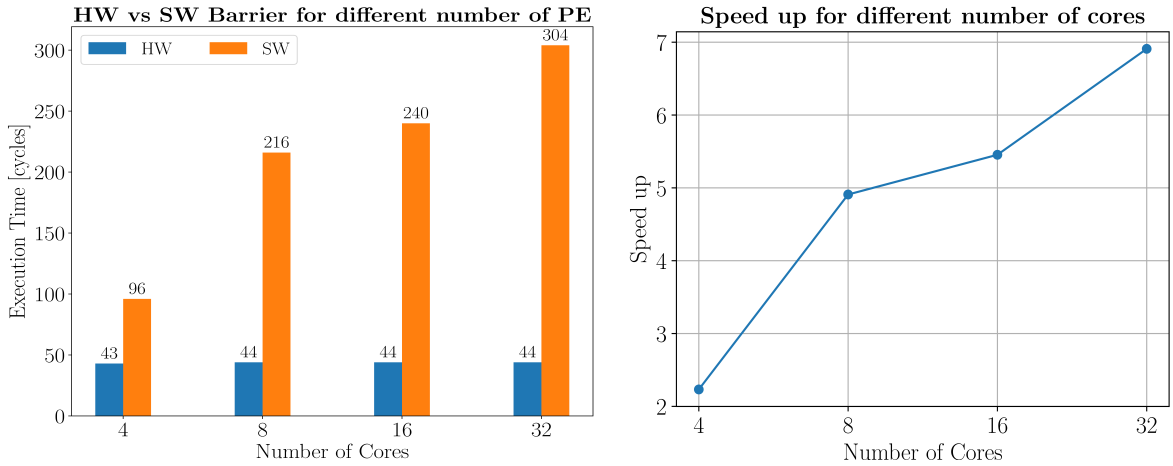
(b) Performance without cache misses.



(c) Speedup without cache misses.

Figure 5.4: Performance comparison between software and hardware barrier. In Fig.5.4a the software barrier performs worst than Fig.5.4b because some cache misses occur increasing the overall execution time. Instead the hardware barrier never experiences those misses. Plot 5.4c depicts the speed up without misses.

it's worth noting that in this study, the only operations performed by the working cores are barriers, resulting in relatively low traffic in the interconnection system. The latency of reduction requests might increase if other cores issue write requests directed to the same destination as the barrier, as explained later in the bitonic sort experiment in section 5.2.3.



(a) Execution time

(b) Speedup

Figure 5.5: In 5.5a is depicted the execution time of the global barrier for different number of participants. In 5.5b the speedup introduced by the hardware support is illustrated.

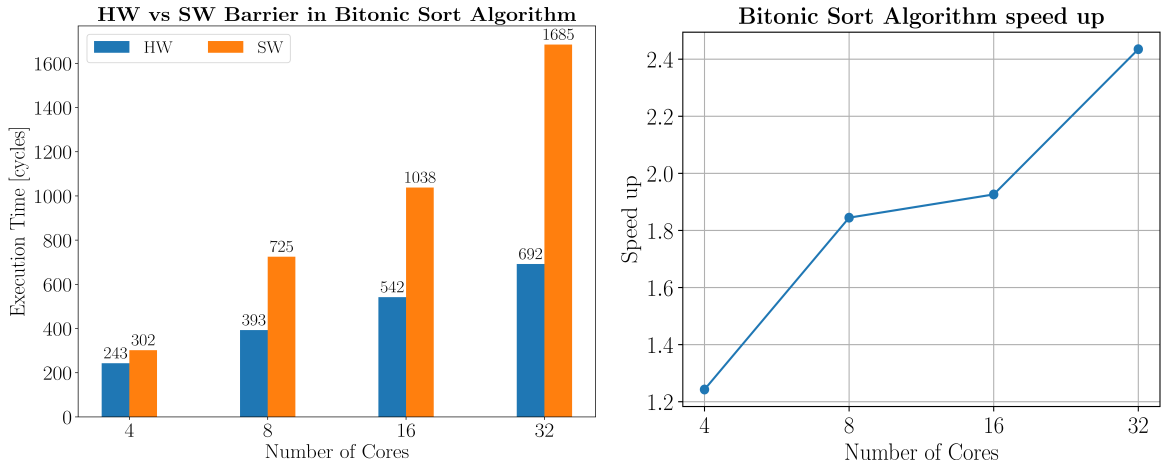
5.2.3 Bitonic Sort algorithm

The final experiment aimed to assess the impact of the barrier's speedup on a real application, specifically the bitonic sort algorithm described in section 4.3.4. Various measurements were taken to analyse different aspects.

Weak Scaling

Figure 5.6 illustrates a comparison between the performance obtained with both software and hardware barriers for synchronizing multiple cores during the sorting algorithm. This diagram illustrates the weak scaling problem for the scenario where the sequence length corresponds to twice the number of available cores. As described in 4.3.4, the number of comparisons to be performed in parallel corresponds to $SeqLength/2$, with each core executing a single comparison per step. This configuration is characterized by a quick synchronization-free region, highlighting the importance of accelerating the synchronization phase. The maximum speedup is equal to $1685/692 \simeq 2.43$ and is achieved with 32 cores, corresponding to a sequence of 64 elements, the largest problem size that can fit in the employed architecture.
Let's examine the case with 4 cores. Upon analysing the assembly code, it was observed that the execution time of the software barrier was not equal to the expected 96 cycles, as depicted in Fig. 5.5a, but instead was almost equal to 80 cycles. The

reason behind this discrepancy lies in the arrival time of each core at the barrier. With the software version, cores exit the barrier at different times, making it rare for them to arrive simultaneously at the next barrier. Consequently, even though the expected speedup is 2.23, the effective speedup at the barrier was measured to be 1.6, resulting in an overall speedup of $302/243 \simeq 1.23$. In conclusion, weak scaling has



(a) Bitonic Sort execution time

(b) Bitonic Sort speed up

Figure 5.6: Bitonic sort analysis in weak scaling condition. The workload of each core is kept constant by doubling the sequence length together with the number of working units.

been investigated with different problem sizes. For example, if the sequence length is equal to 16 but only 4 cores can be utilized, the workload of each core will double compared to the previous study. Consequently, the time spent on synchronization across the entire code will decrease, leading to a decrease in speedup as well. Figure 5.7 illustrates the weak scaling trend for three different workloads. The blue plot depicts the scenario where the total problem size can fit in the architecture, allowing each core to perform a single comparison. The orange trend represents a sequence length four times the number of available cores, resulting in two comparisons per core. Finally, the green line illustrates the case with four comparisons per core.
As anticipated, the speedup decreases with the increasing problem size since the time spent on synchronization reduces as well. The three trends exhibit almost the same dependency on the number of cores.

Strong Scaling

Another aspect analysed is the strong scaling speedup. In this study, the problem size is kept constant while the number of working units is varied, thereby altering the workload distribution among cores.
Let's begin by considering the case with 32 cores. Increasing the sequence length beyond the maximum problem size ($Seq_Len = 64$), the total speedup decreases, as depicted in Fig. 5.8. In this experiment, the problem sizes are such that they cannot be completely parallelized because the number of comparisons to be performed

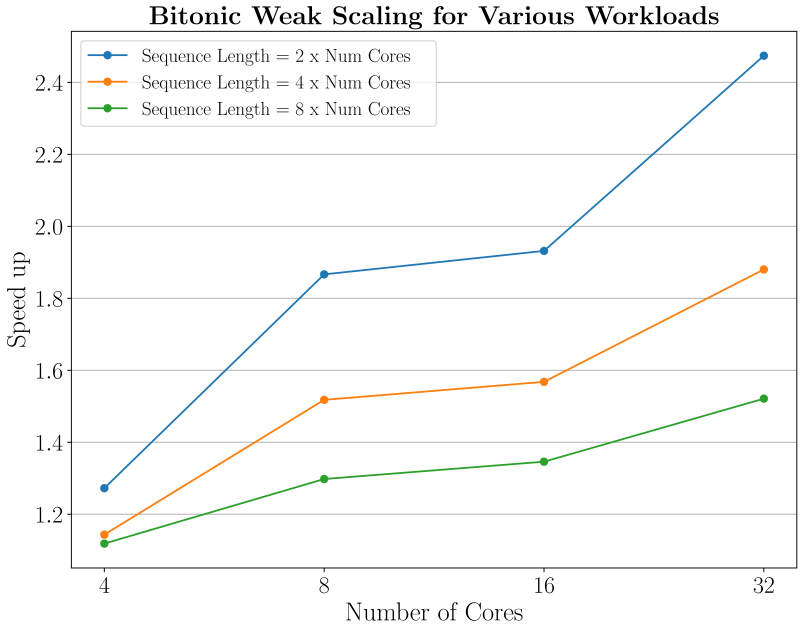


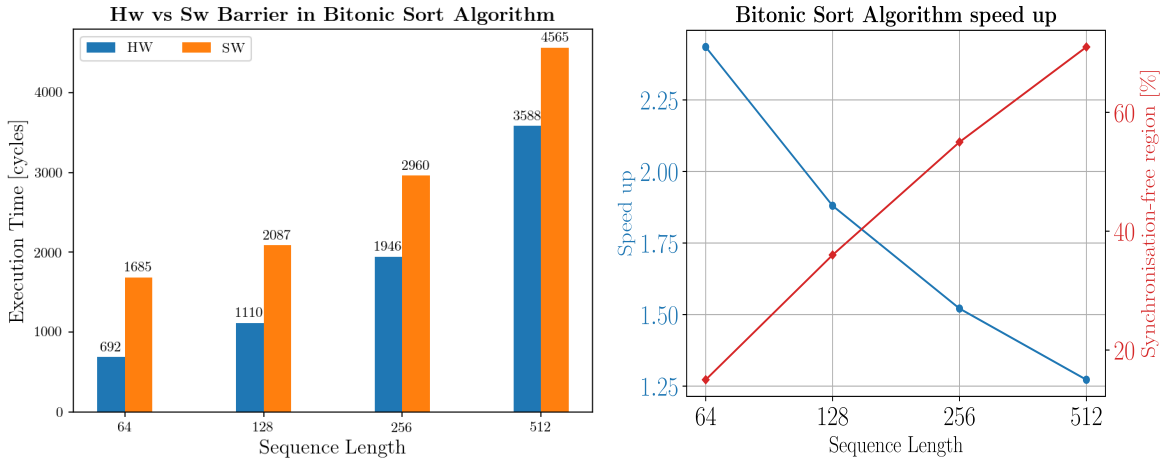
Figure 5.7: Weak scaling analysis for different workloads.

exceeds the number of cores. Consequently, the time spent on the synchronization-free region increases, reducing the application speedup. As shown in Fig. 5.8, the minimum speedup is equal to 1.27, achieved with a synchronization-free region of almost 72%. Focusing on the best and worst cases, the following observations can be made:

- ***Seq_Len* = 64**: In this scenario, the percentage of time spent on synchronization corresponds to 86%, indicating that the maximum achievable speedup is $1/0.14 \simeq 7$. However, the execution time with the hardware barrier is 2.43 times the original. The discrepancy from the ideal speedup is due to unbalanced workload distribution. Although each core must perform the same number of comparisons, some may require swapping operations while others may not, resulting in non-uniform workload distribution and varying arrival times at the barrier. Upon analysis, a delay of almost 30-40 cycles has been identified. Using the inverse of Ahmdal's law and the obtained bitonic sort speedup, the actual acceleration of the barrier was evaluated, resulting in around 3.21, consistent with the outcomes shown in Fig. 5.5b, where the expected speedup for a barrier with a maximum delay of 50 cycles is approximately 3.5.

- ***Seq_Len* = 512**: In this scenario, the application speedup is equal to 1.27, and since the synchronization-free region corresponds to 72% of the entire execution time, the maximum achievable speedup is equal to $1/0.72 \simeq 1.38$.

$$s_{bitonic} = \frac{1}{(1 - p_{barr}) + \frac{p_{barr}}{s_{barr}}} \quad \rightarrow \quad \frac{p_{barr}}{\frac{1}{s_{bitonic}} - (1 - p_{barr})} \tag{5.3}$$



(a) Bitonic Sort execution time

(b) Bitonic Sort speed up

Figure 5.8: Bitonic sort analysis for $Seq\_Length > 2 \times NumCores$. In this experiment the workload of each core is not constant, it increases together with the sequence length. The red trend represent the percentage of code spent on the synchronisation-free region, while the blue plot depicts the seed up of the bitonic algorithm for different problem sizes.

Theoretical Model

To assess the validity of the results, a theoretical model (5.4) has been derived to estimate the code execution time under different conditions:

$$T_{\text{ex}} = \log_2(L) \times \frac{L}{2N} \times T_{\text{sfr}} + \log_2(N) \times T_{\text{sync}} \tag{5.4}$$

Where:

- $\boldsymbol{T}_{\text{ex}}$: Execution time.
- $\boldsymbol{L}$: Sequence length.
- $\boldsymbol{N}$: Number of available cores.
- $\boldsymbol{T}_{\text{sfr}}$: Synchronisation-free region time.
- $\boldsymbol{T}_{\text{sync}}$: Synchronisation region time

From the experiment in Fig. 5.5, it was observed that the hardware barrier does not depend on the number of involved cores, indicating that $T_{\text{sync\_HW}}$ remains constant. Conversely, the software barrier is strongly dependent on the number of participants, meaning that $T_{\text{sync\_SW}}$ is a function of N. To determine this relationship, the experiment described in section 5.2.2 was repeated for the software version with fine-grained synchronization from 1 to 32 cores. Figure 5.9 illustrates the obtained outcomes. The trend shows two distinct hops: the first from 1 to 2 cores and the second from 4 to 5 cores. When only the first core issues the barrier request, the shared counter being stored inside the first cluster TCDM, the latency to access

it is minimal. However, when a second core in a different cluster participates, its store request must be forwarded from the internal cluster crossbar upward to the group XBAR (Fig. 2.8c) and then downward to the first cluster. This path incurs a longer latency compared to the previous one. Similarly, when fewer than 5 cores synchronize, the store requests remain confined inside the first quadrant. However, when the 5^{th} core issues the request, it is routed to the TCDM passing through the top-level XBAR (Fig. 2.8a).

Once all the parameters in equation 5.4 were known, the expected execution

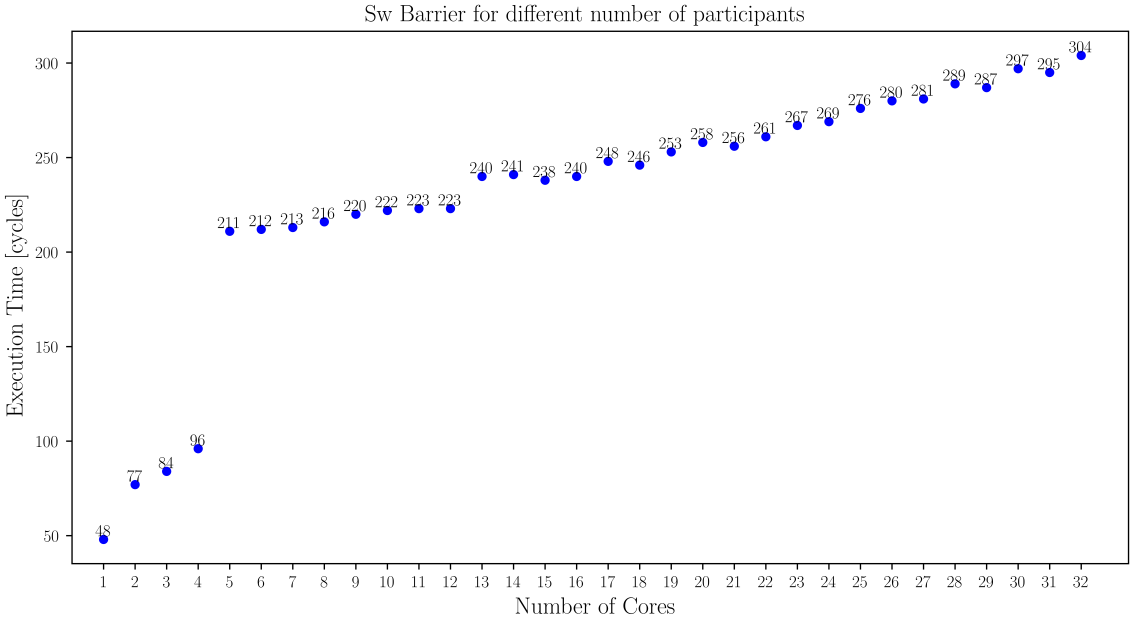


Figure 5.9: Execution time of the Software barrier for fine grained synchronisation.

time in the strong scaling condition was evaluated for three different problem sizes: $Seq_Len = 128$, $Seq_Len = 256$, and $Seq_Len = 512$. The three plots are depicted in Fig. 5.10. As illustrated in Fig. 5.11, the model provides a reasonably accurate prediction of the real outcomes, especially starting from 8 cores. However, it overestimates the execution time when only 4 cores are used. This discrepancy arises from the interpolation performed to estimate the software barrier execution time with 4 cores based on the data from 5 to 32 cores. However, the hardware barrier prediction also does not match the obtained results when only 4 cores are utilized. The theoretical speedup, represented by dashed lines, appears larger than the real speedup, indicated by continuous lines.

Investigation into the assembly code revealed the cause of this phenomenon. As the number of cores decreases, each core is tasked with performing more operations in the algorithm. Specifically, each operation involves a comparison between two values. If they are already correctly ordered, the core can proceed to the next pair; otherwise, it must perform a swap operation before proceeding. Consider a scenario with a sequence length of 512 elements, resulting in 256 comparisons in each step of the algorithm. With only 4 cores, each core is responsible for 64 comparisons.

Consequently, there may be cases where all 64 comparisons for one core are in the correct order, while those associated with another core are all in the wrong order. In the latter case, each comparison is followed by a swap operation, which consumes nearly 3-4 cycles. This discrepancy in workload distribution leads to a variation in arrival times at the barrier, approximately 200 cycles apart.
From the previous study in Fig. 5.4, when cores have an arrival delay of 200 cycles, the barrier speed-up becomes 1.8. This discrepancy elucidates why the observed speedup is lower than expected.

Another aspect that can be identified in Fig. 5.11 for a $Seq_Len = 128$, is the

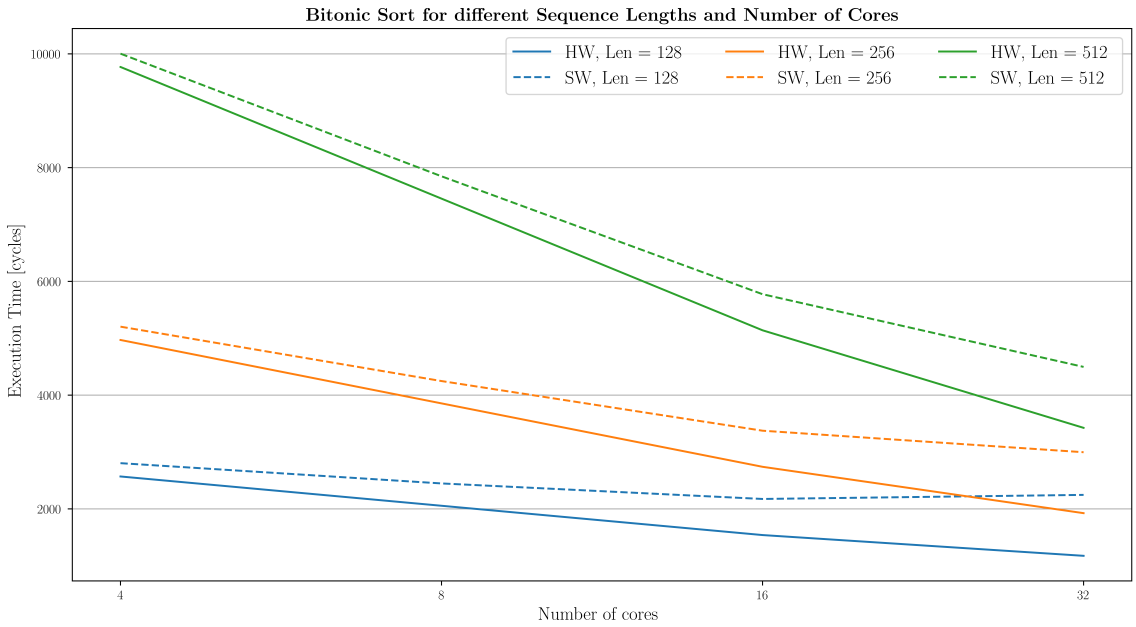


Figure 5.10: Prediction of the bitonic sort execution time for different problem sizes and architecture configurations.

presence of a minimum execution time for the software version at 16 cores, after which the execution time starts increasing with 32 cores. In contrast, the hardware version continuously decreases in execution time. This highlights the advantage of the hardware implementation, which remains unaffected by the number of cores.
In conclusion, since from the previous results the theoretical model seems to be a good tool to approximate the expected execution time, except for a 4-core system, it can be exploited to estimate the execution time trend for configurations that cannot be simulated. For instance, it has been used to evaluate the trend up to 256 cores as depicted in Fig. 5.12. Recalling that in all the experiments done so far, one core per cluster has been used, 256 cores would mean 256 clusters. Imagining a system organization like the one in Occamy (9 cores per cluster), such a system would result in 2304 cores. Perhaps, this hypothetical system would have more hierarchical levels in the interconnection subsystem, something like Manticore [15].
The software barrier exhibits an optimal number of cores beyond which the synchronization overhead outweighs the benefits of parallelization. In this evaluation, the

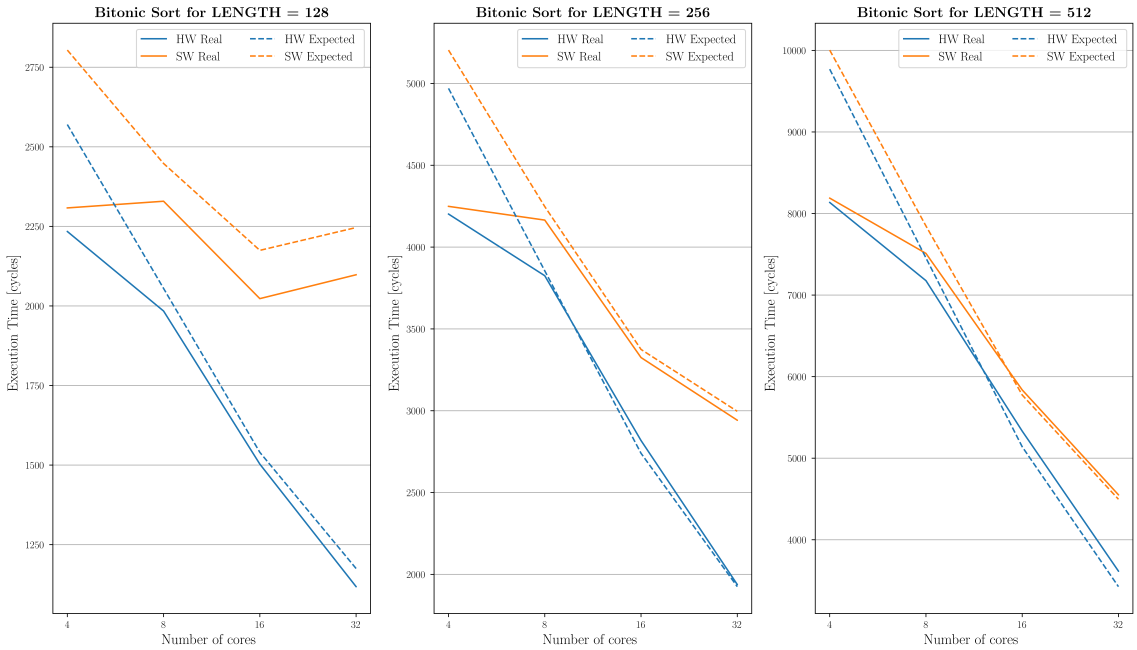


Figure 5.11: Comparison between the expected and measured execution times.

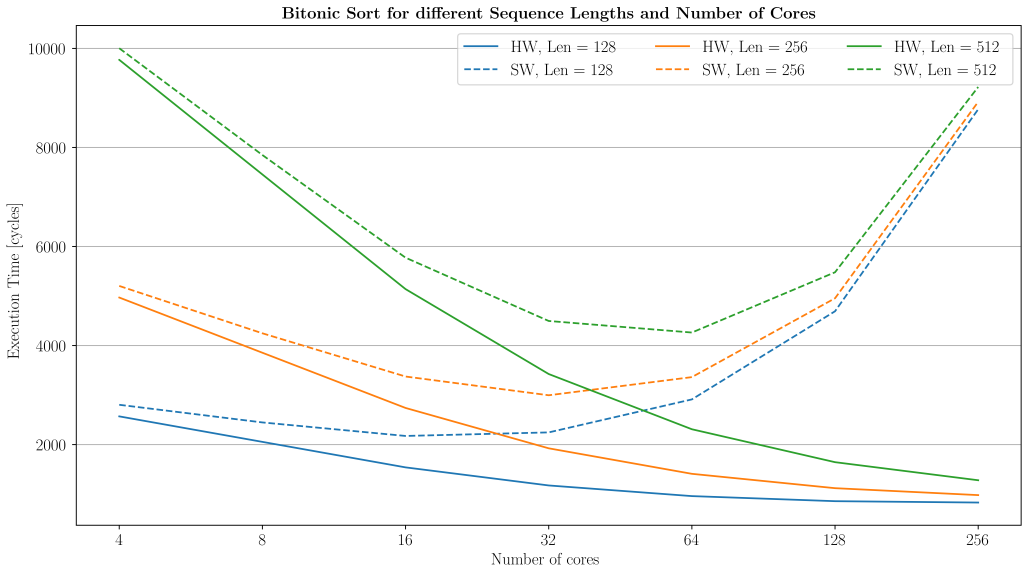


Figure 5.12: Prediction of the expected execution time for both hardware and software global barrier in very large systems.

relationship between the synchronization time $T_{\text{sync\_SW}}$ and the number of cores N obtained previously was used. This relationship holds reasonably well for a system with only two hierarchical levels (clusters and groups). However, in a larger system with 2304 cores, which would likely have more levels of hierarchy resembling a structure akin to Manticore, the trend might deviate. The expected trend is similar to that depicted in Fig. 5.9, where after 32 cores, a new drop in performance should occur. This expectation arises from the arrangement of clusters and groups, where after a certain point, the 9^{th} group may be connected via a different chip, necessitating communication through a third-level crossbar. Consequently, the plotted trend may overestimate the actual execution time.
Conversely, the hardware version demonstrates increasingly improved performance as the number of cores increases until reaching a saturation point determined by the required number of cores, i.e., half of the sequence length. However, in a system with a larger number of cores, such as 2304, the assertion that the time to perform a hardware barrier does not depend on the number of cores may no longer hold true. With an additional level in the interconnection system, the latency for performing a hardware barrier may increase. Assuming a round-trip time of 50 cycles for the barrier, with 25 cycles each for the reduction request to reach its destination and for the "restart" signal to travel back to the cores, and considering two hierarchical levels, each XBAR level may introduce a 25-cycle delay. Thus, in the hypothetical 2304-core system, an estimated latency of around 75 cycles may be anticipated.
In conclusion, while this theoretical study overlooks factors such as off-chip communication latency, it serves as a preliminary exploration of the benefits of the hardware barrier in alternative system configurations.

Chapter 6

Conclusions

In this report, a reduction-capable crossbar designed to accelerate fast M-to-1 communications has been presented. The AXI-based XBAR, already developed by the PULP team, was extended to support reduction operations in a general-purpose manner. The current version is capable of performing the logic AND operation, which is sufficient for barrier synchronization mechanisms in parallel computing systems. The interconnection infrastructure was designed flexibly to allow for future extensions, enabling the integration of more operations. An extensive testbench was implemented to verify the correct functionality both with and without reduction generation.

Regarding area efficiency, post place-and-route results were presented using GlobalFoundry 12nm technology. Different configurations were synthesized, resulting in an overhead ranging from 3% for a 2×2crossbar to 42% with 16 master and slave ports.

Subsequently, the R-XBAR was integrated into Occamy, a 216-core system for parallel computing applications. This integration necessitated the extension of the internal Snitch cores to issue reduction requests. The RISC-V-based processing elements were extended to exploit AXI functionalities and reduction logic AND for barrier mechanisms capable of stalling each unit until the arrival of all participants.

In conclusion, an extensive benchmarking process was conducted to assess the gained performance under various circumstances. Firstly, the impact of arrival times at the barrier was analysed, resulting in a maximum speed-up of up to 6.9 times in the best-case scenario. Secondly, an experiment was conducted to evaluate the impact of the number of synchronizing modules on the hardware barrier speed-up, revealing that with only four cores, the speed-up was around 2.2 times, while with 32 cores, the hardware barrier could be 6.9 times faster than the software version. Finally, the Bitonic Sort algorithm was employed to study the effective acceleration introduced by the hardware barrier in real conditions, yielding an overall program speed-up ranging between 1.27 times and 2.43 times.

Looking ahead, several extensions can be introduced in the developed R-XBAR to perform other reduction operations such as addition, logic OR, maximum, and min-

imum. Furthermore, the optimization strategy outlined in Section 5.1.1, aimed at reducing area overhead, could be implemented to instantiate the reduction logic selectively on the necessary master ports, thus potentially improving efficiency. Moreover, expanding the scope of barrier benchmarking to encompass a wider range of applications, each with distinct workloads that utilize synchronization mechanisms, would facilitate more comprehensive functional testing and enable the collection of additional performance data. This would provide a more holistic understanding of the RXBAR's impact across various scenarios. Finally, considering that the current project primarily focused on M-to-1 communication, an alternative avenue for exploration could involve modifying the reduction logic to adopt a 1-from-many approach. This adjustment might yield different outcomes in terms of both area utilization and performance, warranting further investigation and experimentation.

Glossary

Master In an interconnection system, the term Master refers to the module that can initiate any transactions, whether it's a write or read transaction. The AXI documentation utilizes the term Manager to refer to what is called Master in this report.

Slave In an interconnection system, the term Slave refers to the module where a master can write data to or from which it can read information. In other words, slaves are the destination of the write and read transactions. The AXI documentation utilizes the term Subordinate to refer to what is called Slave in this report.

Synchronisation Free Region In parallel computing, this term represents the portion of code executed by each core in the computation.

Synchronisation Region In parallel computing, this term represents the portion of code executed by each core in the synchronization with the other participants.

Transaction A transaction is the set of transfers necessary for a Master to send a complete message, either a write or read, to a Slave. A transaction comprises several transfers. A write transaction consists of a write address request, followed by a write data transfer, and finally, it concludes with a write response received back from the Slave. Similarly, a read transaction is composed of a read address request and the response received back from the Slave, which sends the requested data along with a response signal.

Transfer A transfer refers to a single-cycle communication on an AXI channel. A write transaction consists of three transfers: write address request, write data, and write response. A read transaction consists of two transfers: read address request, read data.

Bibliography

[1] R. van de Geijn and J. Träff, *Collective Communication*. Boston, MA: Springer US, 2011, pp. 318–327. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_28

[2] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[4] W. Gao, J. Fang, C. Huang, C. Xu, and Z. Wang, "Optimizing barrier synchronization on armv8 many-core architectures," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 542–552. [Online]. Available: https://ieeexplore.ieee.org/document/9556046

[5] A.-K. Mohamed El Maarouf, L. Giraud, A. Guermouche, and T. Guignon, "Combining reduction with synchronization barrier on multi-core processors," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 1, p. e7402, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7402

[6] M. Harris, "Optimizing parallel reduction in cuda," 2007. [Online]. Available: https://cuda.uga.edu/docs/reduction.pdf

[7] S. Ma, N. E. Jerger, and Z. Wang, "Supporting efficient collective communication in nocs," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[8] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 71–82. [Online]. Available: https://doi.org/10.1145/2155620.2155630

[9] G. Almasi, C. Archer, J. G. Castanos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D.

Steinmacher-Burow, W. Gropp, and B. Toonen, "Design and implementation of message-passing services for the blue gene/l supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 393–406, 2005. [Online]. Available: https://ieeexplore.ieee.org/document/5388785

[10] J. Sampson, R. Gonzalez, J.-f. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 235–246.

[11] A. Kurth, W. Rönninger, T. Benz, M. Cavalcante, F. Schuiki, F. Zaruba, and L. Benini, "An open-source platform for high-performance non-coherent on-chip communication," *IEEE Transactions on Computers*, vol. 71, no. 8, pp. 1794–1809, 2022. [Online]. Available: https://doi.org/10.1109/TC.2021.3107726

[12] J. Duato, S. Yalamanchili, and L. Ni, "Chapter 5 - collective communication support," in *Interconnection Networks*, ser. The Morgan Kaufmann Series in Computer Architecture and Design, J. Duato, S. Yalamanchili, and L. Ni, Eds. San Francisco: Morgan Kaufmann, 2003, pp. 207–286. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558608528500080

[13] ARM, "Axi protocol overview." [Online]. Available: https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview

[14] GitHub, "Axi systemverilog modules for high-performance on-chip communication," 2023. [Online]. Available: https://github.com/pulp-platform/axi

[15] F. Zaruba, F. Schuiki, and L. Benini, "Manticore: A 4096-core risc-v chiplet architecture for ultraefficient floating-point computing," *IEEE Micro*, vol. 41, no. 2, pp. 36–42, 2021.

[16] PULP, "Occamy," 2024. [Online]. Available: https://pulp-platform.org/occamy/

[17] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1845–1860, nov 2021.

[18] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the connection machine cm-5 (extended abstract)," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 272–285. [Online]. Available: https://doi.org/10.1145/140901.141883

[19] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates, "An overview of the bluegene/l supercomputer," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002, pp. 60–60. [Online]. Available: https://ieeexplore.ieee.org/document/1592896

[20] M. Harris and K. Perelygin, "Cooperative groups: Flexible cuda thread programming," Oct 2017, [Accessed 31.01.2024]. [Online]. Available: https://developer.nvidia.com/blog/cooperative-groups/

[21] ARM, *AMBA AXI Protocol*, Sep 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest/

[22] C. M. Chiang and L. M. Ni, "Multi-address encoding for multicast," in *Parallel Computer Routing and Communication*, K. Bolding and L. Snyder, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 146–160.

[23] PULP, "Occamy," 2024, [Accessed 22.11.2023]. [Online]. Available: https://github.com/pulp-platform/occamy

[24] RISCV, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*, December 2021. [Online]. Available: https://riscv.org/technical/specifications/

[25] A. Bundy and L. Wallen, *Breadth-First Search*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 13–13. [Online]. Available: https://doi.org/10.1007/978-3-642-96868-6_25

[26] "Distributed random generators," [Accessed 20.12.2023]. [Online]. Available: https://www.mcs.anl.gov/~itf/dbpp/text/node119.html

[27] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, p. 307–314. [Online]. Available: https://doi.org/10.1145/1468075.1468121

[28] L. Hans Werner, *Bitonic Sort*, 3rd ed., 2012, [Accessed 04.01.2024]. [Online]. Available: https://www.hwlang.de/algorithmen/sortieren/bitonic/bitonic.htm

[29] HPCWIKI, "Scaling," [Accessed 30.01.2024]. [Online]. Available: https://hpc-wiki.info/hpc/Scaling

[30] DZ, "Performance differences due to versions," Feb 2022, [Accessed 07.03.2024]. [Online]. Available: http://eda.ee.ethz.ch/index.php?title=Synopsys#Performance_Differences_due_to_versions