Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Design and analysis of a coherent memory sub-system for FPGA-based embedded systems

Vahid Roostaie

Abstract

Cache coherence and memory consistency are of the most decisive and challenging issues in the design of shared-memory multi-core systems that influence both the correctness and performance of parallel programs. In this thesis, we identify and analyze the problem of designing a coherent/consistent memory subsystem in general and then focus on FPGA-based multi-core embedded systems containing general purpose CPUs and dedicated hardware accelerators. We narrow down the range of the problem by targeting only the stream-based applications and developing dedicated application-specific solutions. A flexible Windowed-FIFO communication pattern is proposed to be used by the parallel programs being run on the multi-core system. The software APIs for the FPGA platform are implemented and tested, a customized streaming cache memory is designed, implemented and tested based on the proposed communication pattern and in the end, example embedded systems are developed and tested on the FPGA platform to prove the correct functionality of the APIs, the cache memory and the coherent data communication between the cores. All the tests are done on a Xilinx Spartan3dsp development board and all the hardware and software aspects of the FPGA platform are studied and their influence on the memory system is analyzed. The simulations and analyses show that the developed solution has less complexity and more scalability and portability comparing to existing solutions while it provides a flexible range of functionality that different streaming parallel applications can benefit from.

CE-MS-2011-16

Faculty of Electrical Engineering, Mathematics and Computer Science

Design and analysis of a coherent memory sub-system for FPGA-based embedded systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Vahid Roostaie
born in Tehran, Iran

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design and analysis of a coherent memory sub-system for FPGA-based embedded systems

by Vahid Roostaie

Abstract

Cache coherence and memory consistency are of the most decisive and challenging issues in the design of shared-memory multi-core systems that influence both the correctness and performance of parallel programs. In this thesis, we identify and analyze the problem of designing a coherent/consistent memory subsystem in general and then focus on FPGA-based multi-core embedded systems containing general purpose CPUs and dedicated hardware accelerators. We narrow down the range of the problem by targeting only the stream-based applications and developing dedicated application-specific solutions. A flexible Windowed-FIFO communication pattern is proposed to be used by the parallel programs being run on the multi-core system. The software APIs for the FPGA platform are implemented and tested, a customized streaming cache memory is designed, implemented and tested based on the proposed communication pattern and in the end, example embedded systems are developed and tested on the FPGA platform to prove the correct functionality of the APIs, the cache memory and the coherent data communication between the cores. All the tests are done on a Xilinx Spartan3dsp development board and all the hardware and software aspects of the FPGA platform are studied and their influence on the memory system is analyzed. The simulations and analyses show that the developed solution has less complexity and more scalability and portability comparing to existing solutions while it provides a flexible range of functionality that different streaming parallel applications can benefit from.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-16

Committee Members :

Advisor:	Dr. Ir. Said Hamdioui, CE, TU Delft
Advisor:	Dr. Ir. Jos Van Eijndhoven, Vector Fabrics B.V. Eindhoven
Chairperson:	Dr. Ir. Said Hamdioui, CE, TU Delft
Member:	Dr. Ir. Stephan Wong, CE, TU Delft
Member:	Dr. Ir. Rene van Leuken, CAS, TU Delft

To my beautiful and kind wife; "Aisan", for her true companionship and honest friendship over the years. Also to my mother for all her sacrifices since I was a child.

Contents

List of Figures

List of Tables

Acknowledgements

When I first went to Vector Fabrics office for an interview about the thesis assignment, I got a very positive impression about the technical drive of the company's projects and also the attitude of Dr. Jos Van Eijndhoven, that is why I eventually chose Vector Fabrics over my other options. During the project, I learned a lot discussing with Jos and observing his work style. He was not only technically strong and a practical up-to-date engineer, but also theoritically deep and personally patient and easy going. I would like to sincerely thank him for his helps and useful advices during the project. I would also like to show my appreciation to Vector Fabrics for the opportunity and their nice and friendly work environment. I am sure we will hear more from Vector Fabrics in near future. I additionally express my gratitude to my advisor in TU Delft; Dr. Said Hamdioui. He accepted to supervise my work even though I asked him after a few months that the project had started and I needed to change my supervisor due to organizational issues. I benefited a lot from his valuable remarks specially on project management and work discipline. And last but not least, I have to deeply thank my wife for her great patience during my whole master study at TU Delft. I was away from home a lot and she was always supporting, caring and underestanding despite all the difficulties I put her through.

Vahid Roostaie
Delft, The Netherlands
December 5, 2011

1 Introduction

With the tremendous demand for speed and efficiency in digital systems, multi-processors and parallel systems have been playing an important role in the development of recent embedded systems for high throughput applications. Meanwhile, over the past decade, FPGAs have had a huge growth in terms of logic capacity and speed so that nowadays, a multi-processor system can be implemented on a commercial FPGA fabric. Hardware acceleration, inter-processor communication, shared resource management and many other concepts and design issues have standard solutions proposed by FPGA vendors and their development environments. However, in some applications and areas, more work is needed to come up with a tailored solution and implementation. These solutions will be still developed, compiled and built using the standard tools of the FPGA vendors and this makes them easily portable and reusable. An embedded system in general consists of two major parts: a low-level software (we may call it firmware) part and a hardware part. The hardware part could be a co-processor or accelerator that is supposed to boost up the overall performance of the whole system. Having the entire system on an FPGA proposes some restrictions specially on area comparing to big multi-processor platforms with multiple chips and general interconnection networks. One of the most complex and important features of a shared-memory multi-processor system is having a coherent and consistent memory sub-system [1]. Memory system is a major performance and power bottleneck in embedded systems as the performance gap between processors and memories is ever increasing [2]. The global memory or shared memory on these systems is the main data and program storage and having different cores reading and writing to the same place concurrently and simultaneously requires a conceptual model for the semantics of memory operations to let programs correctly use the shared memory [3]. This model is called *memory consistency model* or in short, *memory model*. Beside memory consistency, we have the cache coherence issue in multi-core systems. These two main issues are closely tied together and have mutual influences on one another. The memory consistency model of a shared-memory multi-processor system affects both the performance and the programmability of the system [4]. One of the goals of this thesis project was to study and analyze coherence and consistency issues at all the levels of FPGA-based embedded systems. Throughout the rest of this report, we might see these issues and their impacts at the level of any software or hardware component of the system.

The remainder of this chapter is organized as follow: Section 1.1 briefly introduces Vector Fabrics B.V. which is the company that this thesis project was carried out at. Section 1.2 defines the main problem that this thesis project is addressing. Section 1.3 discusses the related work and compares the existing solutions. Section 1.4 details the tasks and contributions done in the project and eventually Section 1.5 clarifies the structure of the rest of the report.

1.1 Vector Fabrics B.V.

Vector Fabrics B.V. is a start up company active in the field of embedded systems tool development. The company was founded in 2007 and just a few month ago officially released its first product. Vector Fabrics tools facilitate the creation of a parallel embedded system and helps the users to parallelize their sequential codes and map them on a specific multi-core platform. Parallel programming is a tough and very time-consuming task and using VF tools the users can easily analyze their programs, generate an embedded system, parallelize their codes and eventually port them to a heterogeneous X86 platform or an embedded FPGA-based multi-core system. This thesis assignment was carried out in Vector Fabrics office in Eindhoven. More information about Vector Fabrics B.V. and its tools can be found on Vector Fabrics website.

1.2 Problem statement

The embedded systems created by Vector fabrics tools in general contains multiple CPUs and co-processors, connected to some global system bus infrastructure. The target platform could be an FPGA fabric. The bus infrastructure implements a global memory map over a distributed memory implementation. Co-processors are dedicated hardware units that implement an application-specific function with high throughput. The hardware platform used to develop and test the embedded systems and the proposed solution for this project is a Xilinx FPGA. Figure 1.1 illustrates the general architecture of these embedded-systems consisting of Microblaze processors and dedicated hardware accelerators all accessing the shared external memory through their local cache memories and the global bus. In principle, there is no limit to the number of CPUs and accelerators except the logic capacity of the particular FPGA chip being used.

The local memory allows frequent and fast access to local data. Caches allow a more efficient use of the system bus bandwidth. However, the various co-processor caches and the CPU caches may introduce the system level issues of *memory consistency* and *cache coherence*. In traditional computer architectures, these issues are usually solved by a rigorous implementation of a hardware cache-coherence control. However, for more function-specific embedded systems on FPGAs, this traditional and generic solution is rather expensive to implement. Furthermore, embedded processors often do not provide standard cache-coherent interfaces for customized co-processors. In this project we seek to use application analysis results to create a more dedicated solution tailored for the class of stream-based applications. This will result in a 'cooperative' method, where the co-processors as well as the remaining application code on CPUs are already aware of required data synchronization points and critical sections to explicitly issue coherence control commands or take care of the consistency. A cache controller needs to be designed that is suitable for VF tools target applications. The application-specific solution works based on a *design pattern* that defines the nature of the shared memory accesses across

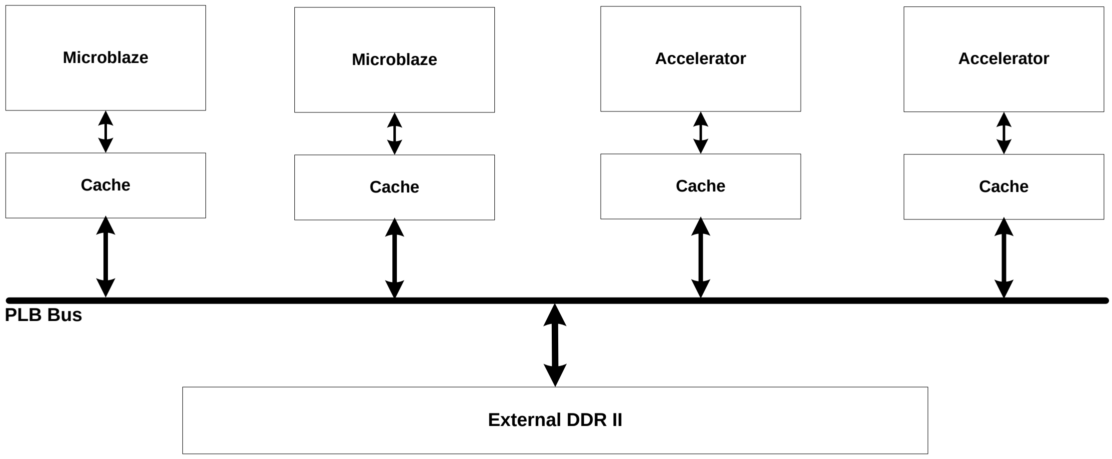

Figure 1.1: FPGA-based embedded systems of Vector Fabrics

the cores. In principle, the design pattern determines the type of memory accesses of the cores in the system with respect to each other and the way the data is being exchanged or shared between the cores. The term *design pattern* is originally used in the field of software engineering where it refers to description of communicating objects and classes that are customized to solve a general design problem in a particular context [5].

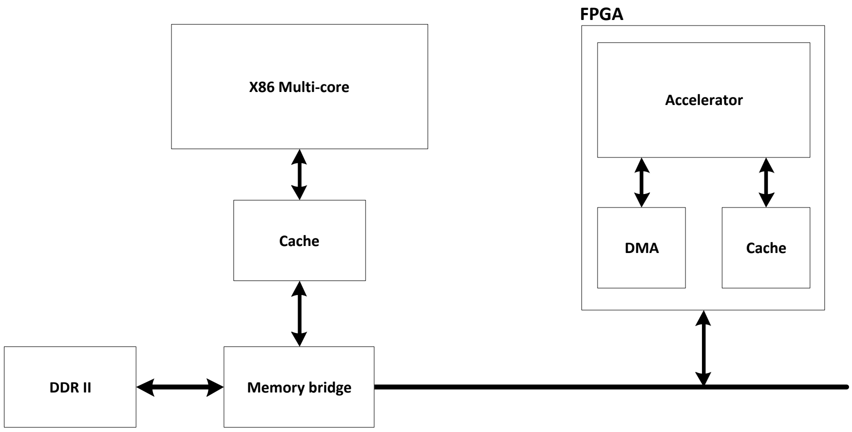

Figure 1.2: Heterogenous platforms of Vector Fabrics

The general design problem that is addressed in this project is creating parallel application codes with a coherent/consistent data communication and the context is in general a CPU/FPGA platform. The applications are stream-based applications

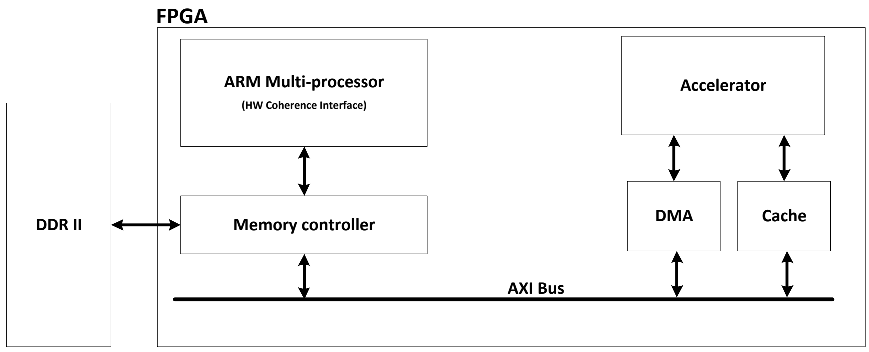

Figure 1.3: ARM-based embedded systems of Vector Fabrics

like multimedia, encoding, signal processing, etc that are typically characterized by their excessive data memory accesses but with regular access patterns for stream data structures [2]. FPGA-based multi-processor systems are viable solutions for stream-based embedded applications [6]. The proposed solution in this thesis can be still usable on a heterogeneous platform like the one in Figure 1.2. or an ARM-based multi-core system using AXI bus [7] as represented in Figure 1.3. It is obvious that some modifications are needed to be applied to the implementation of the solution on each on of these platforms, but the design pattern and the basic concepts are the same and the same level of functionality is achieved. These two platforms (Figure 1.2 and Figure 1.3) are also supported by VF tools, however, as mentioned before, the target platform of this thesis project is only a Xilinx FPGA-based embedded system. Summarizing all the aforementioned points and putting them all into one statement, it would be that "we want to have the same solution based on the same design pattern on the same kind of memory system architecture.

Coherence and consistency issues for us have two major meanings at two different levels: one is inside the proposed cache memory and the other one is at system level when it is about the global memory interactions/requests issued from different cores and the way the memory sub-system appears to them. More details will be given in the following chapters of this report.

1.3 Related work

In this section we will review and compare the state of the art with respect to the main problem addressed in this project and then explain that how the proposed solution removes their shortcomings. We first review the related work for the cache design and then cover the communication patterns and models.

Streaming cache design

Cache design for stream-based systems has had so many different solutions that in general fall into two main categories:

1. **Data prefetching**: these solutions are based on the key technique of data prefetching to hide some part of the latency of the extensive continuous memory accesses [8, 9]. In these systems, there must be an algorithm to prefetch some data blocks that are not currently used but will be very likely requested by the processor in near future. The main idea is to reduce the rate of *mandatory misses* or *cold misses* in the cache by already bringing them in before they want to be accessed for the first time.

    (a) **Hardware-based techniques** implement complex prediction logics for prefetching the data. [10] and [11] are two examples of such systems. [12] has also proposed a hardware-based prefetching scheme that makes use of reference prediction tables to predict the next memory access.

    (b) **software-based algorithms** are basically based on an exhaustive static analysis of the code and inserting prefetching instructions at specific locations of the program. [8] reviews and compares several examples of such algorithms

    (c) **Locality optimizations [9]** are another kind of software-based techniques that use run-time transformations to change the data layout of the program and hence the probability that the next requested data is already in the cache. In case of success, the average latency and bandwidth usage will be reduced due to a reduction on memory accesses.

2. **Multiple cache modules**: In [2] and [13] we see a methodology to design the memory hierarchy with specific memory and cache modules each used for storing a specific kind of variable. They work based on some initial analysis to discover the memory access pattern of the program and use specific memory modules to accommodate different parts of their data. The result is multiple physical memory modules.

The above solutions have some shortcomings comparing to the proposed solution in this thesis that can be briefly summarized as follows:

1. The solutions based on the use of multiple cache modules are complex and based on an exhaustive analysis of the data locality. They analyze the input code to classify its memory access pattern to propose a cache system consisting of multiple stream caches each caching a part of the memory, while in our approach we want to take advantage of the fact that the final parallel program is generated by VF tools and its structure in terms of shared data access, critical sections and synchronization points is already known based on the design pattern. As a result, contrary to [2] and [13], we do not produce a different memory sub-system for a different program and always use the same cache system but take advantage of the WFIFO pattern and the stream-based cache memory to have less cache misses and exchange data coherently.

2. Hardware-based prefetching solutions are complex and have high area and power overheads and so not very economic to implement in more critical embedded systems. These methods work dynamically in run-time and could be more efficient and faster comparing to software-based solutions. Software-based prefetching solutions have the advantage of saving the area of the chip and reducing the complexity of the hardware but might not perform well in some situations as the prediction is not possible in compile-time in many cases. Both hardware-based and software-based solutions target the *mandatory misses* and are rather complex and may introduce lower performance in case of incorrect prediction, while the proposed cache architecture in this thesis focuses on reducing the *conflict misses*. It has a simple and novel architecture that does not have a huge area overhead and avoids using sophisticated prefetching algorithms. It has been specifically implemented to work perfectly for streaming data communication on FPGA-based embedded systems.

Streaming communication models

Different implementations and solutions are available for stream-based communication patterns and inter-core data communication:

1. In [14] and [15], task-level APIs and interfaces are presented for structured design and programming of embedded multi-processor systems and flexible token-based data communication between processors and tasks. However, they still lack a strong cache management and also a hardware-software communication support.

2. [16] has presented a framework to build FPGA multi-processor systems for stream-based applications by generating homogeneous networks of Microblaze processors connected by buses and direct FSL links. This approach is depending on specific hardware FIFOs. Furthermore, it has an area overhead if the number of cores is increased. A big disadvantage of FIFO-based communication models are their restricted functionality as data items can be read and written once and can not be skipped or read and written in different lengths.

3. In [6] a hardware windowed-FIFO implementation in Xilinx FPGAs is introduced. It offers a great deal of functionality, however it has some shortcomings: it can not be easily ported to other platforms, it uses block RAMs of the FPGA and is not easily scalable as the area overhead and memory usage grows if we want to use multiple channels, it is rather an on-chip point-to-point solution and not a system level solution through the global memory that deals with cache memories and their coherence.

The flexible windowed-FIFO communication pattern proposed in this project for our parallel programs works in conjunction with the proposed cache memory and communicates data coherently via external global shared memory. This solution has important advantages over the existing solutions described above and removes their shortcomings:

1. It has *less complexity* comparing to prefetching and locality optimization solutions.

2. It removes the restrictions of a sole FIFO communication and establishes the windowed-FIFO channel in both software-software and software-hardware systems.

3. It has less area overhead and improves the *scalability* by making use of the global external memory as the main data storage. It means that multiple WFIFO channels can be easily defined in the shared system for data communication between multiple groups of cores regardless of the fact that they are general purpose CPUs or hardware accelerators.

4. It handles the cache coherence at a high level without needing the same kind of standard cache coherence protocol interface on the cores.

5. It can be ported to other platforms with global shared memory architecture with minimum modifications as it is not too dependent on a specific hardware component.

1.4 Thesis contributions

The main tasks and contributions done in the thesis project can be summarized as follows:

1. Different architectures of cache controllers were studied and investigated and a suitable architecture for VF embedded systems was proposed.

2. A pipelined cache controller was designed, implemented and verified for the accelerators using synthesizable parametrized Verilog.

3. Cache coherence and memory consistency issues were studied and analyzed in general and documented for Vector Fabrics for its future use.

4. A cooperative memory consistency model and a communication pattern suitable for stream-based application-specific embedded systems on FPGAs was investigated and proposed.

5. Software APIs of the coherent communication pattern were developed and tested on the FPGA platform.

6. AXI-based and ARM-based systems were analyzed and compared against PLB-based systems in Xilinx FPGAs in terms of memory behavior and consistency/coherence issues to be used as future reference by Vector Fabrics.

7. Several test applications and their corresponding embedded systems were designed and created for demonstrating the correct memory consistency and cache coherence on the FPGA board.

1.5 Thesis organization

The rest of the thesis is organized as follows:

Chapter 2 reviews the cache coherence and memory consistency issues in more details and as general issues in mult-core computer systems. Chapter 3 focuses on the design pattern and reviews the proposed communication model to address the consistency issues in the target platform of this project. Chapter 4 reviews the Xilinx development environment and its main features. Chapter 5 discusses the architecture and implementation details of the proposed cache controller for the accelerators. Chapter 6 explains the APIs and software libraries of a SW-SW embedded system. Chapter 7 explains how the WFIFO communication channel is mapped and works on a SW-HW embedded system. Chapter 8 presents the test methodologies and results, while Chapter 9 summarizes the report, reviews the potential unresolved issues and proposes some ideas for future work.

2 Background

In uni-processor systems, the semantics of the memory sub-system is simple, all the memory requests are normally issued in order by the processor and are completed in order by the memory sub-system. In such systems, applying optimizations in compiler or in the hardware will not impair the correct functionality of the programs. Reads always return the last written values to the variables and the CPU always has a consistent and intuitive view of the memory. Out of order execution used in dynamic execution techniques or instruction re-ordering at compile-time can be still used to increase the performance. The only limitation is to prevent these optimizations when accessing the same memory location with different load and stores. This will actually result in the illusion of sequentiality in the programs in uni-processor systems which is all the programs need to observe with respect to memory behavior. As long as there is no dependency among different memory access instructions, the compiler can statically or the CPU can dynamically re-order them without affecting the correctness of the program. Many of these techniques have been developed over the past twenty years and all of them take advantage of this key observation. [17] and [18] are of the best resources for understanding how these techniques work. In multi-core systems however, the situation is totally different. When a HW/SW system has more than one core, it certainly has to provide some ways for these cores to communicate and exchange data. Every core might have its own cache memory that basically could cache a part of memory that is also cached by other cores. The path between the cores to the memory may consist of many components like bus interconnects, NOCs, bridges, memory controllers, etc. The processors themselves might also want to apply some optimizations to improve the performance. In this situation, the memory contents may be seen differently from different cores point of views and this is a potential threat to the correctness of the parallel program. It is obvious that these issues start to show up when multiple cores aim to access the same data or when they want to communicate with each other. If there is not any shared data access (which is not a very realistic case), then there is not any coherence or consistency issue. We will first review the memory consistency and its related concepts and then see its difference with cache coherence. In the following chapters we will see how the proposed design pattern will eliminate these issues in the class of applications targeted by this thesis project.

The rest of this chapter is organized as follows: Section 2.1 describes what memory consistency is and defines its principles, Section 2.2 reviews and explains a few example pseudo codes about memory consistency, Section 2.3 specifically discusses the need for synchronization in multi-core systems and explains how it is related to the behavior of the memory system, Section 2.4 presents the basics of memory berries and how they can be used to maintain program orders in parallel programs, Section 2.5

reviews the impact of the compiler and CPU optimizations on the memory behavior and its consistency, Section 2.6 reviews some of the most common memory consistency models and summarizes their features, Section 2.7 focuses on cache coherence issue in multi-core systems and explains its principles and finally Section 2.8 explains the difference between memory consistency and cache coherenc.

2.1 What is memory consistency

Symmetric shared-memory multi-processor systems are the most common architecture of multi-core systems nowadays. They have some advantages over message passing or private memory multi-processors like scalability [17] or simplified data partitioning and dynamic load distribution [3]. Physically distributed logically shared memory multi-processors or Distributed shared-memory multi-processor systems are another form of shared memory multi-processor systems that have two main advantages: they are more cost effective in terms of bandwidth when most of the accesses are to local memories of the cores and they reduce the latency to access the local memory of the nodes [17]. Shared memory architectures can be used for systems with relatively smaller number of cores as the shared memory system can still satisfy the memory demands of the cores. All the cores might use their cache memories to reduce the bandwidth request on the shared global bus even more, however, this leads to issues with cache coherence that have different software and hardware solutions. The consistency model of a multi-processor defines the program's view of the time-ordering of events that occur on different processors. These events include memory read and write operations, and synchronization operations [19]. Allowing different cores to read and write from/to the same memory location makes it more complex to predict the memory behavior in different runs of the program. Consider the piece of code shown in Figure 2.1.

Processor A	**Processor B**
A = X;	While(!Flag)
Flag = 1;	B = A;

Figure 2.1: A simple flag on a dual-processor system

Suppose that both A and flag are initially equal to 0. This piece of code is a simple producer-consumer pattern when Processor B must always see the new value of A and set B to X. The flag is simply used to synchronize the programs and enforce a predetermined order of events. However, Processor B might see the old value of A despite the synchronization. Here are two reasons that this can happen:

- If processor A simply reorders the writes to A and flag because it detects no dependency between them from its own point of view.

- If A is physically allocated in a memory module that takes more time to be updated and flag is for instance in a local memory that is faster.

In both cases above, processor A gets past the while loop and reads the value of A while it might have not been yet updated in the memory. This simple example shows the necessity of having a comprehensive and intuitive model to define the memory behavior and how the cores observe it. The most common and natural memory consistency model is *sequential consistency* [20] originally defined by *Leslie Lamport* as follows:

Sequential consistency:A multi-processor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [21].
In the case of the program of Figure 2.1, this definition makes sure that processor B always sees the new value of A at any execution of the program. Sequential consistency is the most desirable and simplest memory model from the programmer's point of view. It guarantees the most natural way that programmers expect to see the memory system. However, its implementation differs from platform to platform and it creates so many restrictions to the system at many levels. An important observation of the program in Figure 2.1 is in fact one of the significant differences between a uni-processor system and a multi-processor system: in a uni-processor system, the memory access instructions can be executed out of order to achieve more performance as long as there is no dependency between them, however, this reordering might be detected by other cores who are having access to these memory locations.

Figure 2.2 depicts the concept of sequential consistency model. The system simply consists of multiple uni-processors *P1* to *Pn* all sharing the same global memory. The concept of this picture can be seen as a non-deterministic or arbitrary interleaving between executions of the instructions of the programs being run on the cores while each program is issuing and completing its memory requests in order. Each memory operation also appears atomically to the other cores. In other words, it is like having an arbitrary selection of all the operations of all the cores and running them in one single sequential order.

For a program result to be *sequentially consistent*, there must be at least one execution on the conceptual sequentially consistent system that generates the exact same result like the execution of that program. It means that the result of the execution of a parallel program could be considered as sequentially consistent even if it does not necessarily enforce the requirements of the sequential consistency model at each and every one of the operations being executed in the system. For example, consider the simple piece of code shown in Figure 2.3. The result of the program is still sequentially consistent as long as there are some ways to guarantee that processor B is always seeing the updated value of variables A and B. It means that other operations on both processors might be executed out of order. In other words, the result of the execution of a program is considered as the values returned by the reads in that program and not necessarily the final state

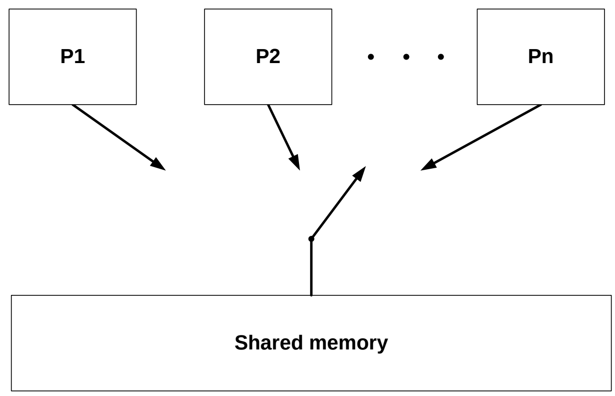

Figure 2.2: Conceptual representation of sequential consisteny

of the memory. Therefore, if the result of the execution of a program do not violate the sequential consistency model, then we conclude that the program is sequentially consistent. In fact, we are only talking about the order in which the operations appear to execute. This fact is the beginning point of adding some optimizations to sequential consistency model.

Processor A	Processor B
A = B;	Mul(...);
	.
B = D;	.
.	.
.	
.	While(!Flag)
	X = A;
Flag = 1;	Y = B;
.	.
.	.
.	.

Figure 2.3: Flag-based synchronization on a dual-processor system

2.2 Consistency examples

In this section, we will look into some more example pseudo codes to clarify more aspects of sequential consistency. For simplicity we can label the result of the program execution as (x,y,z). In Figure 2.4 a result like (0,0,0) or (1,1,1) is sequentially consistent, it is clear that a result like (0,1,1) is also sequentially consistent because for all these executions there is a total order of all the operations that is matching with the sequential consistency model which is (A1, A2, B1, B2, B3, A3) for (0,1,1). Now, consider a result like (1,0,0), this one is not sequentially consistent because if c is updated by A3 on processor A and returned by B1 on processor B then the next two reads (B2 and B3) must also see the new values updated by processor A. This result naturally should not be allowed on a sequentially consistent system because it is not expected by a program like Figure 2.4.

Processor A	Processor B
A1: a = 1	B1: x = c
A2: b = 1	B2: y = b
A3: c = 1	B3: z = a

Figure 2.4: Dual-processor code example one

Figure 2.5, shows another example. In this simple program, a result like (0,0) or (1,1) is not sequentially consistent but (1,0) and (0,1) are sequentially consistent with program orders (B1,B2,A1,A2) and (A1,A2,B1,B2) respectively.

Processor A	Processor B
A1: a = 1	B1: b = 1
A2: x = b	B2: y = a

Figure 2.5: Dual-processor code example two

Figure 2.6 shows a three-processor example. In this program, a (x,y,z) equal to (1,1,1) or (0,1,1) is sequentially consistent, however, a result like (1,1,0) is violating the sequential consistency model. The reason is that if processor B has seen the new value of variable a updated by processor A and then updated variable b to 1 and this has also been seen by C1 in processor C to update variable y. Consequently, processor C, must also see the new value of a executing operation C2. In simpler words, processor B has seen variable a and then updated variable b for processor C. So, if processor C sees the new value of y, it also has to see the new value of a.

Another example is shown in Figure 2.7. In this example, we have four processors and (x,y,w,z) equals to (1,0,1,0) has a conflict with sequential consistency. Because it basically means that Processor B has seen the write to a happened before the write to b, but processor D has seen the opposite order. In sequential consistency, all the processors

Processor A	Processor B	Processor C
A1: a = 1	B1: x = a	C1: y = b
	B2: b = 1	C2: z = a

Figure 2.6: Three-processor code example

should see the same order of all the writes in the whole system.

Processor A	Processor B	Processor C	Processor D
A1: a = 1	B1: x = a	C1: b = 1	D1: w = b
	B2: y = b		D2: z = a

Figure 2.7: Four-processor code example

2.3 Synchronization

One important concept in multi-processor environments and programs is synchronization. Synchronization might be used for a few different reasons in parallel systems. Usually, every multi-processor platform provides some standard synchronization primitives that are guaranteed by that platform to have a reliable result. Both hardware and software parts of an embedded system are involved for a synchronization procedure to complete successfully. In this section, we will review the concept of synchronization and how it might need to be used under sequential consistency. Even thought sequential consistency guarantees the program order to be kept sequential among the operations of one processor, it still allows the operations of other cores to arbitrary interleave at the instruction granularity. Synchronization is needed when we need to enforce a deterministic order of the operations of all the cores. Mutual exclusion algorithms like Dekker's or Peterson [22] can be used for synchronization purposes (the typical usage of these algorithms is to protect a shared critical section). All the parallel platforms often provide a set of atomic instructions like *set and lock*, *read-modify-write* or *load-locked store-conditional* to implement synchronization among operations. Microblaze processor which is the processor used in this project provides the pair of LWX and SWX instructions for implementing exclusive accesses, synchronization and semaphore mechanisms like *test and set*, *compare and swap*, *exchange memory*, and *fetch and add* [23]. ARM processor which is the next generation of the platforms that is going to host the proposed solution has an instruction pair *LDREX* and *STREX* [24] for these purposes. To clarify the situation let us look at a few simple examples. Figure 2.8 is the first example that illustrates the concept of using a test-and-set instruction pair. The instruction pair is supposed to be atomic. This atomicity is actually a service provided and guaranteed by the platform.

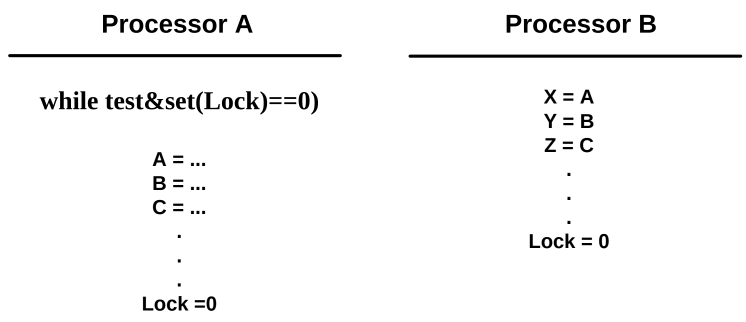

Figure 2.8: Using test and set between two processors

In the above example, the test-and-set instruction pair is executed together and atomically and only one of the two competing processors eventually succeeds to enter the critical section. Mutual exclusion is not the only use of synchronization. In fact, we more focus on the other application of synchronization which is maintaining a particular order of the operations among the processors to communicate data. The piece of code in Figure 2.9 makes this more clear. It is a simple producer-consumer pattern. Processor B is waiting for the flag to be set by processor A and then it can read the updated value of A. It must be noted that this flag is a conceptual representation. It means that the main purpose of this flag as a concept is to notify the other processor as the consumer that the data is ready to be read or consumed. The actual implementation of this flag could be by writing into a specific memory location or by sending a particular value into a special hardware interface or anything similar. However, in any case, the most important point is that synchronization must guarantee that the consumer processor always sees the new value of the updated memory locations. The example of Figure 2.9 is with this assumption that sequential consistency is already performed in the system so that the writing of Flag in processor A is always seen by other cores before seeing the write to A. More information about synchronization concepts, hardware requirements and their implementation techniques could be found in [25], [26] and [27].

2.4 Memory Barriers

Another mechanism for maintaining a particular order in the program is using *memory barriers* or *fence instructions* [28]. Memory barriers are special instructions explicitly used by programs at special places to make sure that all the outstanding memory operations are completed and all the output or store buffers are empty before issuing any new memory request after the barrier instruction. It also causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before

Processor A	Processor B
A = X(p,q)	while(Flag==0)
.	B = Y(A)
.	
Flag =1	

Figure 2.9: Flag-based synchronization

and after the barrier instruction. It is in fact a barrier in the order of the program to enforce a determined order at critical points in the program sequence. Memory barriers should be used with utmost care and there are some conditions for them to work properly. Their successful completion depends on some factors and how and where in the program they are being used or on what bus interconnect the previous memory access to the barrier is being executed. Consider the program in Figure 2.10. Suppose that there is no sequential consistency in the system that the program is running on. We want to use the *mem_bar* instruction to make sure that the write to A is finished and then update the flag for the other core to see the new value of it. We also assume that A and Flag are located in two different memory modules that have different access times. Compiler and system designers need to consult the reference manual of the processor(s) for the details of the operation of memory barrier instruction, however, if the system (the compiler with its system intrinsics) offers the *memory barrier* intrinsic, then the user can assume that it operates as it should. If A is accessed on a bus interface that has absolutely no way to make sure that the store is updated or at least received by the memory module, then the barrier instruction might fail to accomplish its mission. This fact shows another aspect of memory consistency that how it could depends on different bus protocols and communication infrastructures and the way they are implemented in the system. We will review this in more details later. Microblaze processor provides a memory barrier instruction named *mbar* [23] and ARM processor provides *IMB* instruction [24, 29].

Processor A	Processor B
A = X(p,q)	While(!Flag)
Mem_bar	B = A
Flag = 1	

Figure 2.10: Memory barrier

2.5 The impact of architectural and compiler optimizations

In order to achieve sequential consistency in our multi-core environment, we have to guarantee two fundamental requirements: *program order* and *atomicity.*

- **Program order requirement**: All the operations are issued and completed in order on individual processors, or in simpler words, they at least appear to be executing in program order.
- **Atomicity requirement**: All the operations of individual processors must appear atomic with respect to others.

Over the past few decades, modern processors have been using several architectural optimizations to improve their performance. Meanwhile, more advanced interconnect infrastructures like NOCs and high speed buses have been being used for the same reason. Faster memory modules that come with more complicated memory controllers with multiple channels and ports are quite common in recent systems. These factors have all added more complications to the issue of memory consistency.

2.5.1 Program order requirement

Processors might issue their instructions out of the original program order. Dynamic execution is used in many processors to hide the latency of some slow instructions that may be waiting for a resource to become available or a message to be received. If the instructions are issued out of order, they are going to be completed out of order and this might ruin the sequential consistency. Consider the simple program in Figure 2.4. If processor B issues the first read operation for whatever architectural reason, then it might read the wrong value and a result like (1,0,0) is possible which violates sequential consistency. A similar outcome is possible if the writing processor issues the writes out of order. The simple conclusion to this is that processors must make sure to always issue their memory operations in order and always wait for them to be completed before issuing the next one to prevent other processors in the system to detect an unwanted scenario. Even if the processors maintain their program order, memory operations might still get reordered after they leave the processor. The operations may have to take a long journey all the way down to the memory through a complex NOC and a memory controller. There might be scheduling policies on different routes of the interconnect network or arbitration mechanisms on the ports of the memory controller. Putting it all together, instructions might end up being reordered even if they have been issued in order by the processor due to network optimizations and this will create a situation that destroys the program order requirement. In more complex systems with distributed memory models and huge interconnection networks, another issue can show up. Consider the program in Figure 2.3 again. If flag is allocated in a memory that is faster than the memory which has accommodated A and C or the write to the flag has to travel a shorter path on the network comparing to A and C, it is possible for processor B to see the new value of flag but reads the old value of A and C. Having an interconnect network with multiple paths is the main reason for this kind of reorderings that leads to a sequentially inconsistent result. In order to eliminate the reordering described above, the solution is to have all the processors issue their own instructions in program order and then wait for them to be completed before issuing another one. In most systems this means that the processors need an explicit *acknowledgment* message from the memory module indicating that the write has actually taken place in the memory. This facility depends on the organization of the interconnect network being used and the memory subsystem architecture and so is not available on all platforms.

2.5.2 Atomicity requirement

The second main requirement for sequential consistency is atomicity. This requirement is more associated with systems which have cache memories for individual cores. Caching the shared data creates another issue called cache coherence which will be reviewed in section 2.7. Under *SC*, we need the write operations in the system to appear atomic to other cores. It is obvious that in reality, these instructions are not atomic and it becomes worse in presence of data replication in different cache memories. In multi-processors with caches, when a processor updates a shared memory location it will also send either an *invalidate* or an *update* message to other cores to make sure that all the other copies of the shared data is properly re-evaluated. To make this more clear we can take a look to the example in Figure 2.11 from [3].

Processor A	Processor B	Processor C	Processor D

A = 1
B = 1
	A = 2
	C = 1
		While(B != 1) {...}
		
		While(C != 1) {...}
		
		X = A
			While(B != 1) {...}
			
			While(C != 1) {...}
			
			Y = A

Figure 2.11: Atomicity requirement on a four-processor system

Assume that A,B,C and D are initially 0. We also assume that the processors execute their memory operations in order. It is still possible to violate sequential consistency if the updates/invalidates of the writes to A reaches to processors C and D in different order and X and Y get different values. This can happen in systems that have a big interconnect network with multiple paths so that messages travel along different paths towards the destination. Such violations are fixed by imposing a condition named *write serialization* [30]. It basically means that writes to the same location are all seen in the same order by all the processors. Achieving this serialization in reality has some complications. One solution could be to have all the invalidates or updates originate from the same location (like a central directory) and also make the network to preserve the ordering of such messages. Another approach would be delaying an invalidate or update to be sent out until any other invalidate or update issued by a previous write to the same location is acknowledged [3]. Even having write serialization is not yet enough to maintain sequential consistency. Consider the simple program in Figure 2.12. This program is a simple flag-based synchronization between three processors.

Processor A	Processor B	Processor C
A = 1	If (A) B = 1	If (B) C=A

Figure 2.12: Flag-based syncrhonization between three processors

Again, we assume that we are using an update protocol and all the variables are initially cached by all the processors. In addition, we assume that processors issue their memory operations in order and wait for them to be acknowledged before sending out the next one. The writes are properly serialized as described above. It is still possible to violate sequential consistency. Consider this scenario:

1. Processor A writes to A and broadcasts the corresponding update to the the other cores.

2. Processor B receives the new value of A before processor C receives it.

3. Processor B writes to B and broadcasts the corresponding update to the other cores.

4. Processor C receives the new value of B while it has not yet received the new value of A.

5. Processor C sees B as 1 and then proceeds to set C to the old value of A.

The complicated scenario described above can happen in systems using an interconnect network with multiple paths and leads to a non-atomic write operation. The violation in this case is allowed to take place because processor B has the permission to return the value of the write to A before processor C has seen the update generated for this very write operation. One solution to this problem could be preventing a read from returning a value newly written until all the cache copies have been properly acknowledged the reception of the invalidate or update messages generated by the write [3].

2.5.3 Write buffers

Some systems have write buffers or something similar for performance boost purposes. In these systems, a write is placed into the write buffer and the processor considers the write completed and goes on with the rest of the program. In a uni-processor system, if a following read wants to access the same address, the write buffer is first gets checked if it has a copy of the data requested by read and will return the value to the processor if it does, and if it does not, the write buffer can be bypassed without affecting the uni-processor data dependency status. However, in a multi-core system where all the processors have write buffers, it is possible that this bypassing optimization would make the sequential consistency to be violated. Let's take a look at the famous Dekker's mutual exclusion algorithm in Figure 2.13.

Processor A	**Processor B**
F1 = 1	**F2 = 1**
If(F2==0)	**If(F1==0)**
Critical section	***Critical section***
.	.
.	.
.	.

Figure 2.13: Dekker's mutual exclusion algorithm

We assume that F1 and F2 are initially 0. If both processors put their writes to the flags to their write buffers and continue to the read operation afterward, they will both see a 0 in response to the read and enter the critical section. This has a simple conclusion: under sequential consistency, we can not just simply put the write in a write buffer and continue executing the program [17]. Analogous situations can happen in case of using a victim cache or having write-merging or overlapped-writes, etc. For instance, in a multi-core system with caches and store forwarding as depicted in Figure 2.14, a scenario can happen where there are two copies of a variable in both the cache and the store buffer and even if the data copy in the cache gets invalidated or updated under a cache coherence protocol, the processor might gets the old value in its store buffer upon the execution of a following read. The solution is to make the store buffers and caches synchronized.

2.5.4 Compiler optimizations

Besides architectural optimizations in the hardware, compilers use optimization techniques too. Many of these optimizations that are the result of some static analysis during compile-time eventually load to reordering of the operations. The violation of SC here is the same as it is in hardware reordering. Some more advanced compilers might have more intelligent algorithms to deal with reordering and synchronization points in the program, but in the absence of such algorithms the solution is to make the compilers to avid any reordering in the program. As a result, optimizations like code motion, register allocation ,loop blocking or software pipelining are restricted. In summery, many of the compiler optimizations that can be easily employed in a uni-processor system must

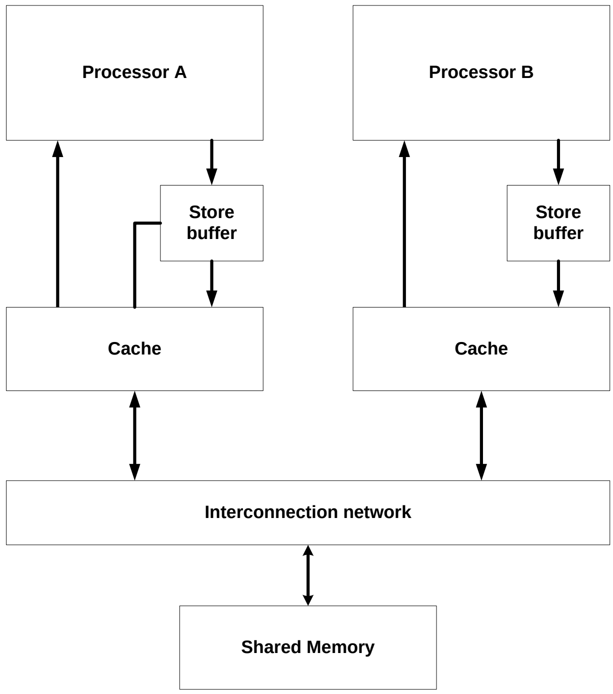

Figure 2.14: Store forwarding

be avoided in a multi-processor system. Compilers that are specifically designed and developed for parallel systems and multi-processor environments have enough in-depth knowledge about the parallelism of the program and its shared data status so that they can efficiently apply essential optimizations without harming the sequentially consistent behavior of the whole system. There are some algorithms to analyze the program and find the locations at which we have to prevent reordering to keep the sequential consistency. These algorithms are complicated, limited and depend on a comprehensive dependency analysis of the program, [31] and [32] are two examples of such algorithms.

2.6 Common consistency models

So far, we have seen one consistency model; sequential consistency. In order to implement sequential consistency, we have to impose many restrictions to entire system and this will defeat the performance improvement and all the advancements and modern techniques that have been proposed in the past few decades. It is clear that we want to avoid it.

2.6.1 Relaxed consistency models

To eliminate the restrictions imposed by sequential consistency on performance, *relaxed memory models* were introduced. These models actually relax some constraints of the sequential consistency model and allow some of them to happen out of order to have the opportunity to employ the optimizations blocked by sequential consistency. Before we proceed with relaxed models we take a look at sequential consistency from two different perspectives. As it was discussed before, there are some minimum requirements for sequential consistency. In [21] and [33], these requirements are described respectively as follows:

- **Lamports Requirements for Sequential Consistency:**
  1. Each processor issues its memory requests in the order specified by its program.
  2. Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of placing the request in this queue.
- **Scheurich and Dubois Requirements for Sequential Consistency:**
  1. Each processor issues memory requests in the order specified by its program.
  2. After a write operation is issued, the issuing processor should wait for the write to complete before issuing its next operation.
  3. After a read operation is issued, the issuing processor should wait for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation.
  4. Write operations to the same location are serialized in the same order with respect to all processors.

Now, we can have a better definition of *write completion*. A write is considered to be completed when it has updated the memory and all the other copies in other processors caches are properly invalidated or updated. We can further simplify this based on Lamport definition that a write can be considered complete as soon as it reaches the memory input FIFO. It is a very important conclusion that if we are sure that a write has reached the memory subsystem and queued in its buffer or FIFO, and we are also sure that all the following requests are going to be buffered after the write, we can consider the write completed even if it has not yet really updated the physical memory location. A more aggressive implementation of write completion would be the idea of *lazy caching* first was introduced by *Yehuda Afek* [34]. The idea is that the processor that updates or invalidates a cache block can consider it completed if the invalidate or update has been received and buffered by the nodes who have a copy of that block. All of these solutions rely on having an acknowledgement mechanism on the network to signal the reception of the message. Scheurich and Dubois's definition is a more generic one comparing to Lamport's and it deals with systems with caches and interconnection networks. In relaxed consistency models, the programmer should have a better understanding of the memory semantics and the underlying hardware. That is why sequential consistency is the easiest and most desirable memory model from programmer's viewpoint as it takes care of the sequential behavior of the program every where and the programmer would not get involved with the complications of the memory operations across the cores. However, we have to make a trade-off to be able to still take

Relaxation	W → *R*	W → *W*	R → *RW*	Read other writes early	Read own write early
BM 370	Yes	No	No	No	No
TSO	Yes	No	No	No	Yes
PC	Yes	No	No	Yes	Yes

Table 2.1: A brief description of three relaxed consistency models

advantage of the optimizations that are eliminated by sequential consistency. Relaxed memory models may relax some program orders: Read to Write/Read order(R-RW), Write to Read order(W-R) and Write to Write order(W-W). We briefly look at three famous relaxed memory models. Other memory models may be available and used by different multi-processor systems. Table 2.1 shows a brief description of three common relaxed consistency models.

- **BM 370:** a read can complete before an earlier write to a different address, but a read can not return the value of a write unless all processors have seen the write.
- **SPARC V8 Total Store Ordering (TSO):** a read can complete before an earlier write to a different address, but a read can not return the value of a write by another processor unless all processors have seen the write (it returns the value of own write before others see it).
- **Processor Consistency (PC):** a read can complete before an earlier write (by any processor to any memory location) has been made visible to all.

A very common relaxed memory model is *weak ordering* or *weak consistency* that relaxes all the constrains in Table 2.1 except read other writes early. Under weak ordering, operations are classified as *data* and *synchronization* and reordering can happen in-between synchronization points. For example, a counter can keep track of the number of outstanding data operations and a synchronization is not issued until the counter is zero. Data operations are not allowed to begin unless the previous synchronization has been completed. This memory model enables many optimizations and relaxes the sequential consistency to a high extent, however, it is not always the best choice as it can not be used with every kind of program and data pattern. We will take advantage of this model in this project. It will be explained in chapter 3. Weak consistency yields high performance and alleviates the latency established by sequential consistency but the programmer needs to analyze, understand and identify the synchronization points and data operations in the program and this can be a very cumbersome task in systems with a high number of cores. *Release Consistency(RCpc, RCrc)* extend weak consistency. Operations are first distinguished as *ordinary* and special. These two categories loosely correspond to data and synchronization categories in weak consistency [3]. RC works based on *acquire* and *release* concepts. Under release consistency, a special variable or address must be first acquired to be written to. It will be later released and only after that can be accessed by other cores for reading. Most of the commercial systems are not sequentially consistent and use some kind of relaxed model. They rely on the programer at some points to take care of the correct functionality of the program by writing consistent and well -designed programs. The extent to which the user needs to take care of the consistency depends on the relaxed model being used by the multi-core system and the architecture of the program itself. In [35] a performance analysis and comparison and benchmarking is done between different memory consistency models. It can be consulted to understand more about their differences and suitability for different contexts and platforms.

2.7 Cache coherence

In this section, we will review the cache coherence issue in multi-core systems. It is not intended to explain all the details of the current cache coherence protocols. We may only refer to them and rather analyze and compare them at system level. In multi-core systems that communicate through the same shared global memory, we have the issue of cache coherence(cache consistency) besides memory consistency that is a significant performance limiting factor [36]. The problem simply arises from the fact that each core might have its own private cache memory to improve its performance. This local caching is also helping to save the bandwidth requests on the shared global interconnect network to the shared memory. Nevertheless, as long as there is not any shared data between the cores, there is no cache coherence issue. It all starts when we have more than one copy of a shared data in more than one cache memory of the system. Processors might want to read or write the shared data and this will take place in their own cache memories in the first place. After this moment, there will be mismatching multiple copies for the same data in multiple caches. We need to have a mechanism to resolve this issue to guarantee that different cores in the system can always reliably access the correct and updated values of the shared memory locations. Although processors logically access the same memory, on-chip cache hierarchies are crucial to achieving fast performance for the majority of memory references made by processors. Thus, a key problem of shared-memory multi-processors is providing a consistent view of memory with various cache hierarchies [37]. Cache coherence can also have impacts on memory consistency. We will first look into the cache coherence as an individual design issue. Figure 2.15 depicts a conceptual representation of the shared memory in multi-core systems and the root of the coherence issues. Figure 2.16 shows a simple situation that incoherent cache management creates unwanted results.

suppose that the value of x is initially zero in the memory and none of the cores already have read it. After some time both cores have read the variable and copied it into their own cache. If later on processor 1 updates x (in this case the shared memory is also updated as it is assumed that the cache is using write through policy) then the other core has no clue that the copy of x that it has is no longer valid. The coherence control protocols in general aim to resolve these kinds of issues in all the situations. In general, there are two major kinds of cache coherence protocols: hardware-based and software-based.

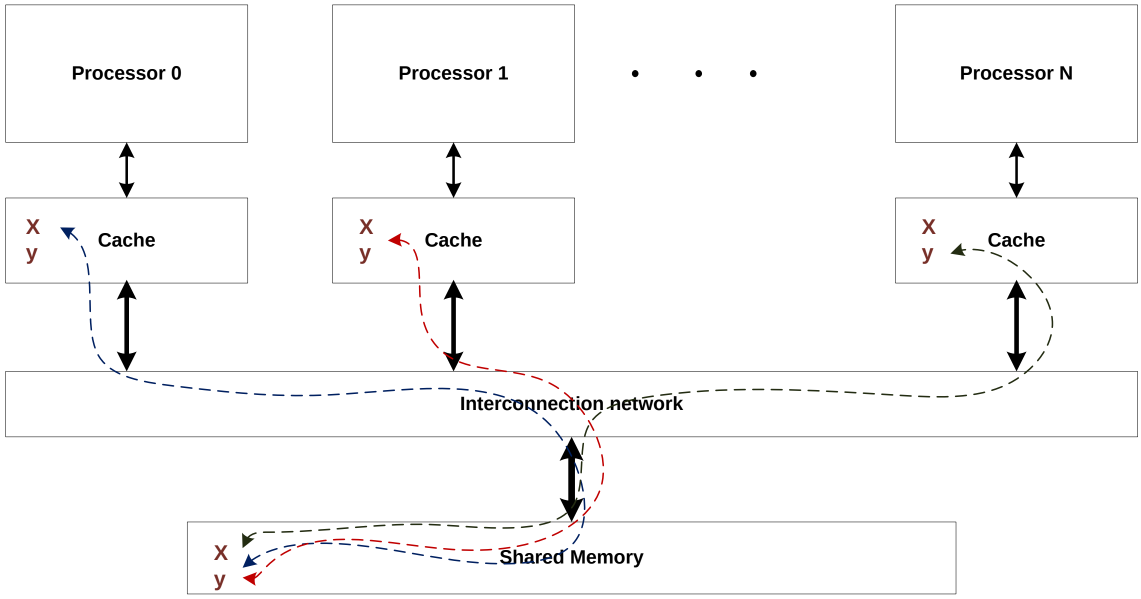

Figure 2.15: Data replication and cache coherence in shared memory multi-core systems

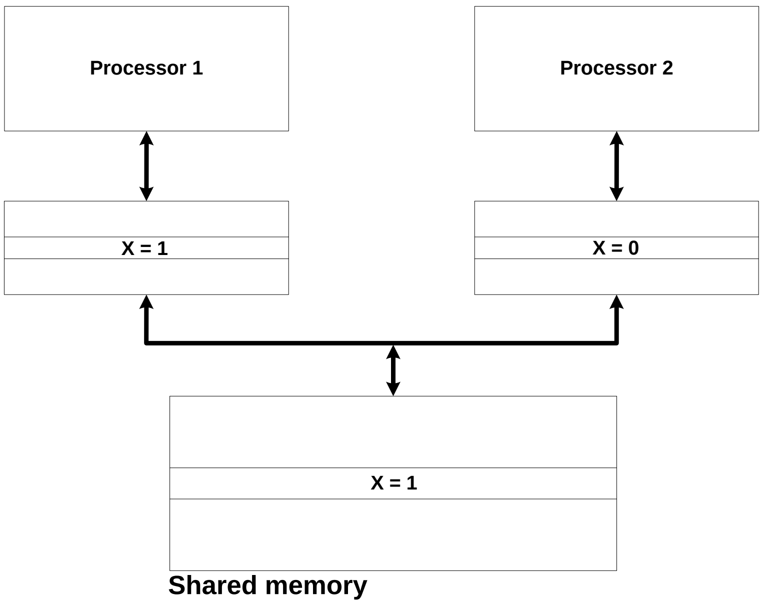

Figure 2.16: Incoherent local copies of data due to data replication

2.7.1 Hardware-based protocols

There are three major types of hardware cache coherence protocols:

- **Snooping:** caches keep track of the sharing status of all the blocks, No centralized state is kept in the system, when a processor detects on the bus that a block of data is updated by another core, it will invalidates its own copy of that data block. This category is associated with write-invalidate protocols [38].
- **Directory-based**: sharing status of any block in memory is kept in one location: the directory. First described in [39].
- **Snarfing**: like snooping except the cache controller checks the data and address buses both and will update its own copy of the data block when it detects a write into that address. This category is associated with write-update protocols,

Another classification of hardware-based cache coherence protocols is:

- **Write-invalidate protocols:**
  - Guarantees that only one writer has a valid copy of a block.
  - Other copies in other caches are invalidated.
- **Write-update protocols:**
  - When a processor writes to a block, it also broadcasts the new value to other cores.

Each one of the above protocols have their own pros and cons.

- **Snooping/snarfing protocols [38]:**
  - Advantages:
    * easier to implement.
  - Disadvantages:
    * uses a lot of bandwidth of the global bus due to broadcasts needed on a cache miss.
    * harder to use with higher number of cores as the broadcast medium has to support a higher degree of broadcasts.
- **Directory-based protocols [40]:**
- Advantages:
  - More scalability, easier to use in systems with higher number of cores.
  - Demands less bandwidth of the centralized shared memory.
- Disadvantages:
  - More complex to be implemented.
  - Uses more area and memory for the directory specially when the number of cores and shared memory blocks grow.

There are several hardware-based snooping and directory-based protocols. Examples of snooping protocols are:

Software-based protocols

- MSI (Modified, Shared, Invalid)
- MESI (Modified, Exclusive, Shared, Invalid)
- MOSI (Modified, Owned, Shared, Invalid)
- MOESI (Modified, Owned, Exclusive, Shared, Invalid)

directory-based protocols

[41]:

- Bit-vector/Coarse-vector
- Dynamic Pointer Allocation
- Scalable Coherent Interface (SCI)

All the protocols briefly reviewed above are generic and complex protocols capable of effectively resolving cache coherence issues in hardware using huge state machines and controllers to continuously check and control the status of individual cache blocks. The task of keeping the cache coherent in multi-core systems could be time consuming. Depending on the application, the cost for maintaining a coherent view of the memory may easily account for half of the execution time or more [42]. In this thesis however, instead of trying to solve the problems for any random parallel program with arbitrary memory accesses to the shared memory in general purpose systems, we are looking for more application-specific solutions that fix the issues of a limited class of applications that are implemented based on a more organized and predictable architecture on FPGA-based embedded systems. In embedded systems, area and throughput are just too precious and need to be spent very carefully. On such systems, we may concentrate only on a given set of applications [2].

2.7.2 Software-based protocols

To alleviate the complexity of hardware-based cache coherence protocols, software-based protocols may be used. Software cache coherence is attractive because the overhead of detecting stale data is transferred from run-time to compile time, and the design complexity is transferred from hardware to software. However, software schemes may perform poorly because compile-time analysis may need to be conservative, leading to unnecessary cache misses and main memory updates [43]. The goal of software-based protocols is to prevent the existence of an inconsistent data in the cache by only caching a data item at safe times. It means that an analysis phase is needed to identify *cacheable* or *non-cacheable* data. This is the job of a complex compiler. One simple and relatively naive solution would be making all the shared data non-cacheable. A better way is to make sure that the date is shared when it is safe and the compiler has to analyze and identify the time intervals when this safety is established. The main idea behind software-based cache coherence protocols is the static analysis or *cacheablity marking* done by the compiler [44]. The compiler has to identify the shared data and categorize the accesses to it into some main classes like the following:

1. Read-only for an arbitrary number of processes.
2. Read-only for an arbitrary number of processes and read-write for exactly one process.
3. Read-write for exactly one process.
4. Read-write for an arbitrary number of processes.

Basically, the main task of the compiler is to analyze the data dependencies and issues appropriate cache control commands like invalidates and flushes to maintain the data in the cache consistent with the global memory. The static data partitioning takes place during compilation of the programs and the way they are shared between computational units is identified and the accesses to shared data from different computational units are managed by inserting cache control commands at appropriate locations in the program. The excellent paper by Hoichi Cheong and Alexander V. Veidenbaum [45] can be used for more information. Also in [46] a very useful classification between software-based cache coherence solutions is presented. We will review this subject in a more specific way tailored to the context of this thesis project in chapter 3.

2.8 Cache coherence vs memory consistency

So far, we have reviewed two main issues in multi-core systems: cache coherence and memory consistency. At this point, It is worthwhile to emphasize on their differences to avoid any confusion. A system is cache coherent if any read to a specific address returns the latest data written to it. Cache coherence protocols are responsible to make sure that all the writes to the same location in the system are serialized and eventually seen by all the cores in the same order(write serialization). However, this is not yet enough to guarantee a sequentially consistent memory system. Under sequential consistency, a) all the operations of one processor need to complete in the program order and b) all the writes to all the locations must be seen in the same order by all the processors. In other words, SC deals with the question of when the data is consistent or when other cores can access the right data. Sequential consistency gives more accuracy to the meaning of latest data. Consider the simple example code in Figure 2.17 from [4]:

Two processors are writing to two locations and two processors are reading those locations. Assume that a and b are initially 0 and consider the following two results of the program for (x,y,z,w):

1. **(1,2,2,1)**:
   This result is not possible under cache coherence nor it is under sequential consistency, because the writes on processors A and B are to the same location.

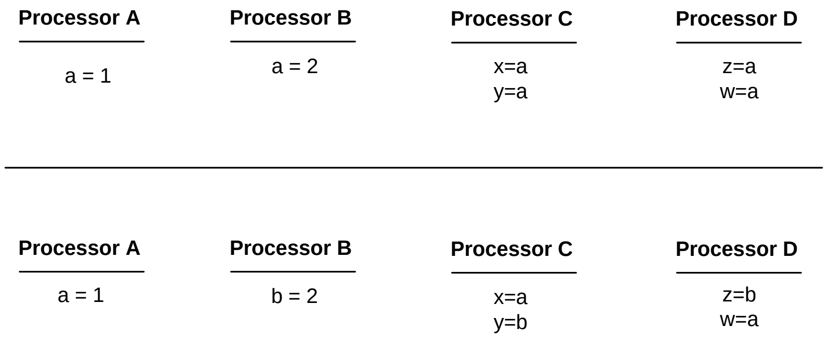

Figure 2.17: Cache coherence vs memory consistency

2. **(1,2,2,1)**:
   This result is possible under cache coherence but is not possible under sequential consistency, because the writes on processors A and B are to different locations and the cache coherence is not violated.

In this thesis project, a cache controller (cache memory) was designed and implemented (chapter 5) and was used to design a coherent and consistent memory sub-system on Xilinx-based multi-core embedded systems. The hardware components, software components and the design pattern together constitute the consistent memory sub-system of our embedded system. In the following chapters more details will be given on each of these aspects.

Design pattern 3

A pattern describes a problem that happens in our target environment and then talks about finding the core solution to the problem in such a way that we can use it over and over again without actually overdoing it in the same way twice [5]. In this project, we aim to define a design pattern that in fact determines how our context looks like and what exactly the problem is. Our design pattern defines the way our parallel programs or their architecture appear in terms of memory accesses, inter-core communication and data sharing. It is a set of communicating tasks, exchanging data through streaming communication, enriched with some global shared-memory features, to support creating applications that fit this style. We will focus on the applications that can be developed and constructed based on this design pattern. Consequently, we will describe the problems existing in the system and propose solutions to solve them and eventually implement the solutions. After this point, we will be able to have a correct execution of any program that has been developed based on the design pattern and is being run in the context of our applications. The design pattern might be even called communication pattern as we are actually focusing on solving the problem of data communication and transfer between cores. However, the term design pattern is what we will mostly use in the rest of this report.

The remainder of this chapter is organized in four sections: Section 3.1 presents the concept of program specific approach in defining a memory consistency model and how it has been used in this project, Section 3.2 reviews several common communication models and compares them and finally Section 3.3 explains the features and principles of the proposed design pattern used in this project.

3.1 Program specific approach

The necessary requirements presented so far are used to maintain the sequential consistency in parallel systems. However, it is not always essential to keep the order at any part of the program to have a sequentially consistent result. It means that one approach could be defining a standard methodology to gather and analyze some information about the program and its shared data with other cores and enforce the consistency requirements whenever it is necessary. These solutions are more complicated and need a comprehensive static analysis of the programs, they also complicates the compilers. The result of this analysis can be used by the hardware and/or compilers to apply reordering when it is safe. As an example, consider the simple program in Figure 2.3. It is obvious that because of synchronization, the expected sequentially consistent result is (x,y) = (1,1), however, it is not essential to keep the order between the write to A and the write to B on processor A, this processor can reorder these writes and still the final outcome of the program is sequentially consistent. There are some algorithms and methodologies to find the critical instruction orders in the programs that really need to be kept to guarantee a sequentially consistent result. In [32], a method is presented for this purpose. These methods are not perfect. In the case of [32] for instance, writes are assumed to be atomic, while in reality, it is not always the case, specially in the presence of local caches and data replication. In this thesis project, we use program specific approach. In other words, we rely on program specific analysis results and define our design pattern based on them. How this analysis is done is another topic beyond the scope of this thesis. The work in this thesis actually starts from the point that the analysis is done on the program which is supposed to be ported on the multi-core system and the synchronization points and critical sections are identified. After this analysis, it is clear that how the data is shared and who is accessing it at a given point during the execution. *Vfanalyst* [47] is the tool from Vector Fabrics that does the analysis on the sequential C codes received from the user and seeks for the parallelization opportunities in it. This tool together with *Vfembbeded* [48] are used to map the parallelized program to the hardware platform. The final parallel program is generated by VF tools has a structure defined by our design pattern. Vfembedded supports a few different platforms and the knowledge acquired during this thesis and its outcome is used in the implementation of this tool-set, specifically on the FPGA-based platforms supported by the tool. As mentioned before, the proposed solution is reusable on other platforms by adding some modifications to its implementation.

The main target application class of VF embedded tools are stream-based applications like multimedia, coding-decoding, signal processing, etc. The focus is on the applications which are potential candidates for performance improvement using block-level functional parallelism. Vfanalyst is responsible for the analysis and detection of the appropriate communication pattern of the program. In general, both data parallelism and functional parallelism are supported in Vector Fabrics tools. After the static analysis of the sequential codes, it is possible to request a data parallel version or a functional parallel version of the application. The data parallelized programs are typically ported on GPU-accelerated platforms which are very effective in this domain due to their high number of independent parallel functional units, and the functional parallelized programs are normally ported to hardware accelerated platforms on FPGAs or a heterogeneous X86 platform with multiple CPUs and dedicated hardware units (accelerators). The functional parallelized programs have a producer-consumer-like structure. So, the final program being run on the platform is not supposed to have an arbitrary non-predictable memory access pattern.

3.2 Review of some communication models

During this thesis project, a lot of time was spent on studying different communication protocols and standards to come up with a suitable one for our target FPGA platform and applications. It is not possible nor intended to discuss all the details of the studied standards, however, a brief review is given in this section for the sake of future reusability.

3.2.1 MPI(Message Passing Interface)

MPI or message passing interface is a set of library specifications and associated APIs proposed as a comprehensive message passing protocol in complex multi-process contexts. The simple intention of this interface is allowing the data to be passed between processes in a distributed memory environment. MPI supports heterogeneous parallel architectures and offers a great deal of functionality. MPI libraries and routines can be included and its functions and APIs can be used in C or FORTRAN programs. Under MPI, a data item can be exchanged between two processes as a message. The fundamental assumptions in MPI are:

- A parallel program consists of multiple processes each with its own data memory and no process can directly access the memory of other processes.
- Data sharing only takes place via message passing. The data needs to be explicitly sent as a message to the other node.

MPI is a comprehensive standard. Using MPI it is possible to define different types of messages and communications and because it is standardized, the codes are supposed to be portable on other supported platforms. In MPI, a point-to-point communication has the following principles:

- It happens between two processes.
- The source process sends the message to the destination process.
- The destination process receives the message.
- Communication takes place within a communicator.
- The destination process is identified using its rank in the communicator.

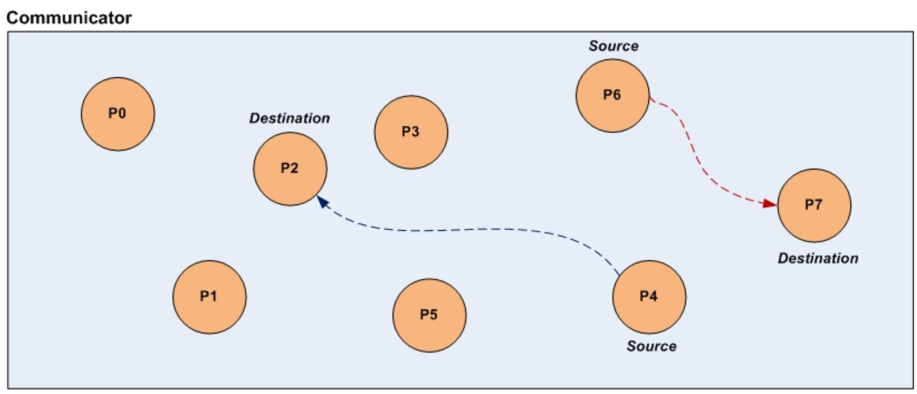

Figure 3.1: MPI communication environment

Figure 3.1 shows a conceptual representation of MPI system. MPI standard documentation [49] can be consulted for more information. After understanding MPI it was concluded that it was not a suitable option for our platform. The main reasons can be summarized as follows:

- It is originally proposed and implemented for multi-task and multi-process environments.
- It is too complicated to be easily ported to our FPGA-based embedded system.
- It is originally dedicated to distributed memory systems and message passing contexts that are not exactly our main target in this project.
- It does not directly deal with cache coherence issues.
- It is originally developed and more suitable for message passing, while we need extensive data communication and transfer.

Despite above reasons, it is clear that technically, porting an MPI compatible interface to our FPGA-based platform is still possible. However, it is not easy and economic. Specially when there are hardware accelerators present and they use their own cache memories. What was needed for this project was a simpler standard that is able to handle the communication through the central shared memory, is powerful enough to support the target applications and can be used by hardware accelerators more easily. Message passing schemes are typically more compatible with distributed memory multi-processor systems.

3.2.2 FIFO channels

FIFO channels are another communication pattern that can be used for data exchange between the cores, they can be implemented in software or hardware. Sending the data to the consumer via some available FIFO interface is possible on the FPGA. In Xilinx FPGAs, two hardware FIFO interfaces are available that are supported by Microblaze processor and can be also easily interfaced on the accelerator. Fast Simplex Link(FSL) [50] and AXI-stream [51] are the supported hardware FIFO interfaces. During this thesis project, these two interfaces were tested and simulated using some simple examples.

FSL link

- **Main features:**
  - Implements uni-directional point-to-point communication.
  - FIFO depth is configurable ranging from 1 to 8K.
  - Supports unshared and non-arbitrated communication mechanism.
  - Provides extra one control bit to be used by the receiver core.
  - Supports both synchronous and asynchronous modes.

– Can be implemented using Xilinx block RAMs or LUT distributed RAMs.

The functional description and signaling of FSL interface can be found in [50]. This interface is proposed by Xilinx and supported in its tool-set as the standard interface for the communication between the processor and the user peripheral modules added to the embedded system. It can also be used as a channel for two processors to communicate. An example is the Microblaze-based multi-processor system implemented in [52]. Using FSL in Xilinx FPGAs is straight forward and simple as it is fully standardized and all the APIs, drivers and documents are available. One Microblaze can support up to 8 FSL links and adding and removing them is easily done in Xilinx EDK tool.

AXI stream FIFO

AXI-stream is newly supported by Xilinx. It is based on AXI-lite [7] from ARM and can be used as a streaming interface, it allows memory mapped access to an AXI streaming interface. Xilinx provides standardized IP cores that can be used to write and read data packets in a FIFO fashion without getting directly involved with AXI signaling. Similar to FSL links, AXI-stream interface is fully supported by Microblaze and can be easily instantiated and used in the FPGA designs within Xilinx tools. The only limitation is that all the AXI family interfaces are only supported by Xilinx starting from Virtex6 and Spartan6 families and ISE 12.3. Figure 3.2 taken from [51] illustrates the core block diagram of AXI-stream FIFO interface.

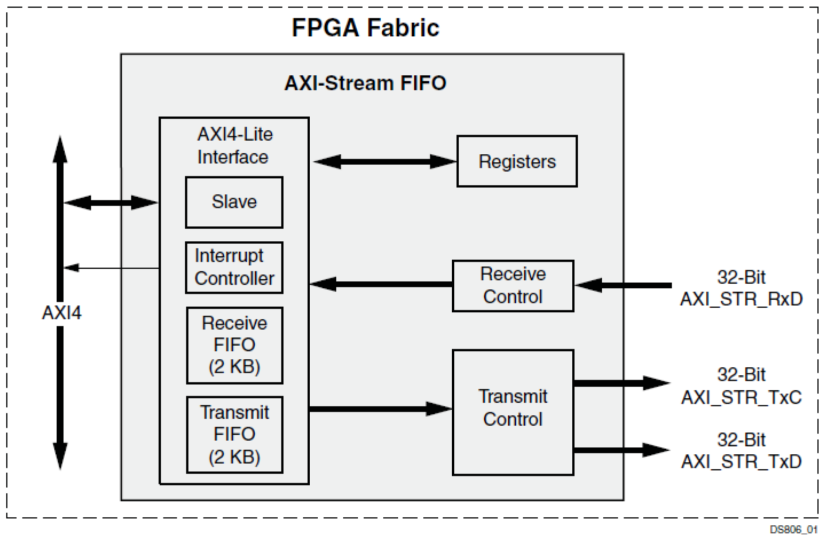

Figure 3.2: AXI-Stream FIFO Core Block Diagram

- **AXI_STR_TXD-**Axi_Stream Transmit Data.
- **AXI_STR_TXC-**Axi_Stream Transmit Control.

- **AXI_STR_RXD-**Axi_Stream Receive Data

Both of the FIFO interfaces described above can be easily used as an inter-core communication channel. However, they are not the perfect choices as the main basis of our design pattern. The reason is that they are vendor-specific and would defeat the original purpose of portability of the solution. FSL links and AXI-stream are not available on all platforms and in future they might be replaced with another interface or might not even be available on Xilinx platform at all. Making our design solutions developed based on these interfaces is not a wise choice and so was avoided in this thesis project. Another drawback is the area overhead in case the number of cores grows. In this case, we need to use several hardware interfaces.

3.2.3 Ping pong buffers

In a producer-consumer fashion, blocks of data may be exchanged between the producer core and the consumer core. A ping pong buffer may be simply interpreted as something like Figure 3.3.

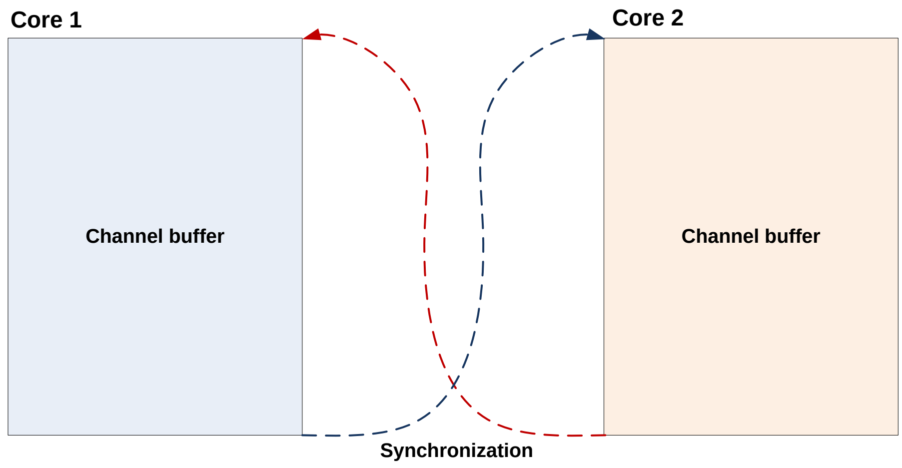

Figure 3.3: Ping pong buffer data pattern

Channel buffers are arrays in the memory that contain the data. Two separated buffers are accessed by the producer and consumer at different times. While one buffer is being used by the writer, the reader is working with the data from the other buffer. The ownership of the buffers are exchanged at the end of the processing using some synchronizations. Needless to say that this synchronization might be a standard primitive provided by the platform or implemented by the designer. In both cases, it has to be coherent and atomic so that it can guarantee to the other core that the buffer being synchronized does contain the updated valid data after the synchronization takes place. This model is a simple pattern that can be easily implemented in many platforms.

Nevertheless, it is not a powerful one. It is not flexible and systematic. It also seems too simplistic to support complicated parallel structures and does not provide a wide range of functionality.

3.2.4 Windowed-FIFO

A windowed FIFO (WFIFO) is in fact a FIFO with extended functionality and more flexibility. Even though a FIFO pattern is an acceptable communication model for streaming applications and has also been used in Vector Fabrics tools before, it still lacks flexibility and presents some restrictions. In a normal FIFO channel, data have to be always read one at a time and exactly in the same order that it is written. It is not possible to skip or remove a data item from the FIFO. Another limitation is that in a FIFO channel, it is normally not possible to read the same data item multiple times. WFIFO removes these limitations from normal FIFO model and provides more functionality and flexibility. WFIFO can be implemented in hardware like [6] or software. A conceptual representation of windowed-FIFO is shown in Figure 3.4.

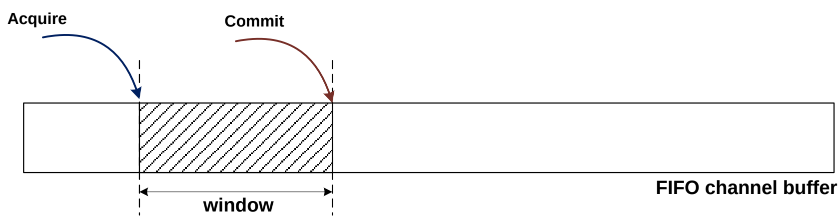

Figure 3.4: Concept of Windowed-FIFO

A window which is a range of data is acquired for data access and then is released. The window is sliding through the FIFO channel buffer and can wrap around it. Both the writer and the reader are performing a procedure like what is illustrated in Figure 3.4. Using WFIFO, parallel programs can exchange data more efficiently and read and write the items from/to windows with more freedom. They can skip a data item if they do not want it. A positive point about WFIFOs is that reads are not destructive like normal FIFOs.

3.3 The proposed design pattern

Up to this point, several communication models were reviewed and compared. Since the main target applications of this thesis project in particular and a wide range of the supported applications of VF tools in general are streaming applications and multimedia programs with determined memory access patterns, a FIFO-like design pattern is a wise basic choice to define the structure of the programs. Obviously, it needed to be discussed in Vector Fabrics and approved within the company whether they prefer this model or not. The KPN (Kahn Process Networks) model [53] is the high level description of how the multi-processor applications look like at process level. Under this model, the application is decomposed into several individual processes communicating through unbounded FIFO channels. A producer-consumer like communication is also interpretable based on this model where the producer process or processor is producing data for the consumer process and these data is being transferred through FIFO channels. We change these FIFO channels to WFIFO channels to gain more functionality and efficiency like described in previous section. WFIFO could be considered as a great extension to a simple ping pong buffer in terms of data exchange and to normal FIFOs in terms of communication pattern.

The final design pattern used in this thesis project is based on the above principles. The WFIFO control interface and administrative structures were implemented both in software and hardware. The channel buffers themselves are allocated in the shared memory. The hardware interface is used by the accelerators and their cache controllers and the software interface implemented as software libraries and APIs is used by CPU and the C code running on it. The final implemented design pattern can be considered as a software-based WFIFO communication. Regarding cache coherence, we know that for relatively well-structured and deterministic programs, software schemes perform significantly better than hardware schemes [43]. So, Software-based cache methods are used for controlling the cache coherence of the memory sub-system during FIFO operation and data transfer. Back to the background knowledge discussion in chapter 2, we can conclude that the consistency model used in the proposed design pattern is in fact a special case of weak consistency named release consistency model as the coherence and consistency is taken care of at synchronization points of the windows at the time of acquiring and releasing. We will see later how exactly this happens in our embedded systems. The formal definition of Release consistency says: The system is said to provide release consistency, if all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter acquires it. Figure 3.5 depicts a functional overview of the proposed design pattern. Many tests and analyses needed to be done to identify and understand the exact behavior and characteristics of all the software and hardware components, Microblaze processor libraries and APIs, the bus infrastructure and controllers and memory and memory controllers and their impact on the coherent and consistent operation of the implemented design pattern. In the following chapters, more detailed information will be given on all of these steps. Back to the original issue of cache coherence and memory consistency, we can now see that in this project, these issues are not managed at the granularity of all the operations since the programs being run on our FPGA-based multi-processor system has the high level structure in terms of data sharing that is described by the design pattern. So the coherence and consistency of all the data need to be taken care of at block levels and synchronization points and this has to happen at both the CPU side and the accelerator side regardless of the internal architecture of their cache memories. in chapter 5, the features and architectural details of the proposed and implemented cache memory for accelerators will be completely explained. In chapter 6, the main features of the proposed design pattern and their associated APIs will be reviewed.

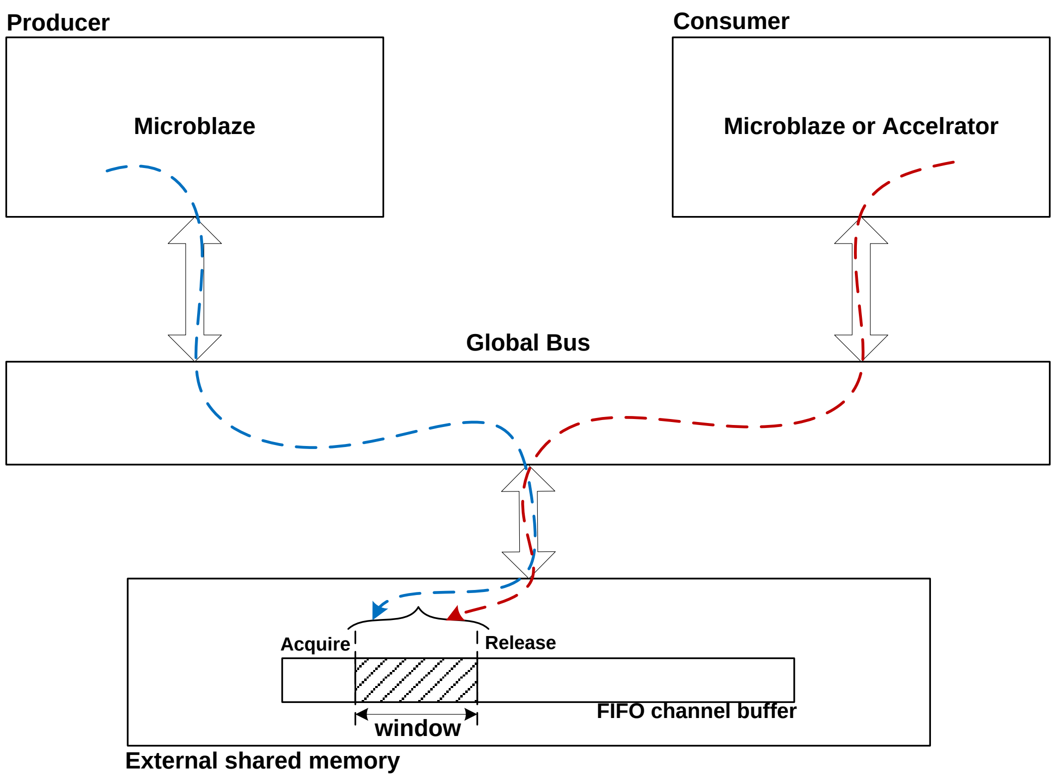

Figure 3.5: The proposed design pattern

In principle, the system has a master processor that starts the communication channel. The other core could be another processor or a hardware accelerator. In both cases, the proposed FIFO channel interface is used to access the data items and control the FIFO administration. Multiple channel buffers can be defined in the shared memory and they can be accessed by different cores. Because the proposed FIFO channel is implemented in shared memory, it can be easily scaled and even ported to other platforms. This solution is not depending on specific features of the hardware platform or any special IP or component. It has to be reiterated again that minor changes to the implementation or taking advantage of different kinds of synchronization primitives or memory accesses on another platform are inevitable and can not be considered as significant platform dependability. But, if the solution was totally dependent on a special hardware FIFO, an internal resource of some specific FPGA series or a particular memory module or architecture, it would be something more platform-dependent that we did not want.

4 Development platform

All the hardware and software developments were done using Xilinx tools. Xilinx EDK or Embedded Development Kit [54, 55] is the development environment for embedded system design and test on Xilinx FPGAs. Before embarking on the hardware and software implementation of our embedded systems, a lot of time was spent during this thesis project on understanding and learning Xilinx embedded solutions and acquiring in-depth knowledge of the procedure of creating, simulating and testing a complete embedded system consisting both hardware and software sides and their interface. In this chapter, we will review the most important aspects of the development environment and explain how they are used. We also review the technical features of the basic constituent hardware components of our systems. This chapter is a good introductory reference for future re-usability in Vector Fabrics. However, it does not cover all the details and it is recommended to consult Xilinx manuals and literature for more information.

The remainder of this chapter is organized as follow: Section 4.1 explains more details about the development environment and the main tools from Xilinx for software and hardware developments, Section 4.2 introduces and compares the kinds of memory modules available on our embedded system, Section 4.3 reviews the available bus and interconnection networks used in our embedded systems, Section 4.4 illustrates the general features of the hardware accelerators and their role in our embedded systems, Section 4.5 reviews the main features of the general purpose processor in our embedded systems and explains its influence on the consistency and finally Section 4.6 illustrates the architecture of our target FPGA-based embedded systems and its memory issues.

4.1 Xilinx environment

The main synthesis and place and route tool of Xilinx is ISE [56]. It was used for RTL implementation and simulation, RTL synthesis, FPGA configuration and on-board tests. EDK is part of ISE. We will also see in next chapter that the cache controller was implemented using HDL guidelines from Xilinx. To create a complete embedded system to be configured on Xilinx FPGAs, we have to first design and create the hardware platform. The hardware platform consists of all the hardware components of the system and their connections through buses and hardware interfaces. Addresses of the components are defined and all the hardware components are configured if it is needed. These steps are all done in XPS (Xilinx Platform Studio) which is part of EDK. There are already lots of standard supported modules available in the tool that can be added to the hardware platform. Their APIs and software drivers are also provided so that they can be accessed and used from within the C code running on the processor. Figure 4.1 shows how XPS hardware platform creation window looks like.

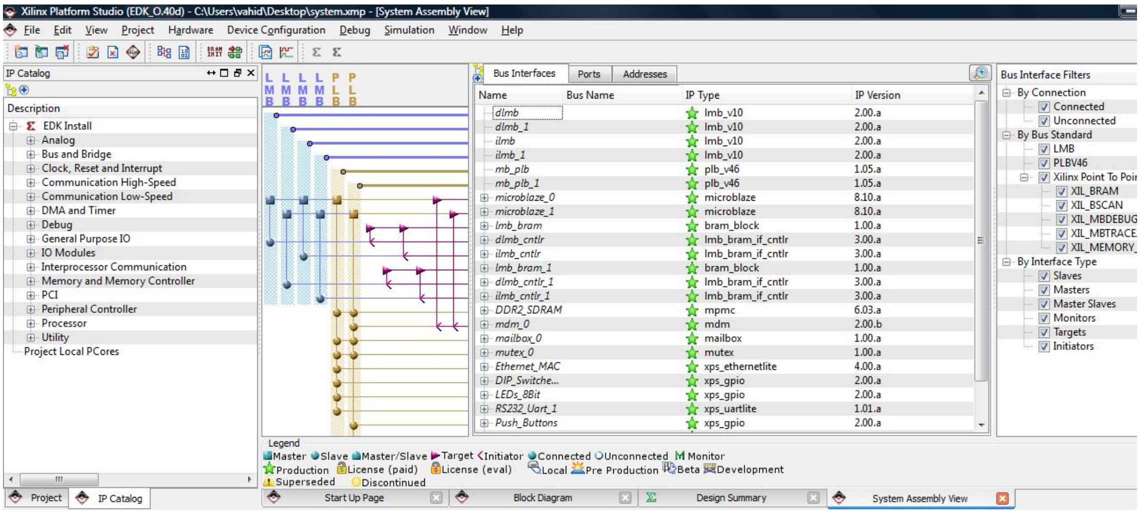

Figure 4.1: XPS(Xilinx Platform Studio)

In XPS, it is also possible to import and add user defined peripherals and connect them to the rest of the embedded system. XPS user interface and wizards make it very straight forward to manage the hardware components and their connection. It is possible to create the embedded system for a specific Xilinx development board or to do it from the scratch for a custom board. In this project, a standard *Spartan-3dsp 3400A* [57] board was used. Using a standard development board means that EDK already knows all the available interfaces and components on the board and how and where they are connected to the FPGA IO pins. So the user has no trouble defining the pin connections of the FPGA chip to the outside world. We can define uni-processor systems or multi-processor systems. It is possible to add as many processors or any other component to the system as we need. The only real limitation is the logic capacity of the FPGA chip being used to accommodate the whole embedded system. Figure 4.1, depicts a standard dual-processor system with two Microblaze processors in XPS. In general, Xilinx FPGAs support two kind of processors. On some FPGA families like Virtex there are power PC processors [58] available. And in almost all the FPGA families from Xilinx it is possible to add and use the *softcore* Microblaze processor. At system level, these two processors are both used in the same way in the tools. However, their features and architecture are definitely different. In this thesis project, Microblaze processors are used as first, they are very easily portable to all the FPGAs because they are softcores. And second, they are the standard processors used in embedded systems generated by VF tools. As it is clear in Figure 4.1, in *system assembly* view, all the components added to the system are shown in the main window and their connections via buses are shown next to them. Connections to the buses can be easily disconnected and reconnected and new addresses can be defined in the *address tab*. The *IP catalog* tab contains the available modules that can be inserted into the system. If we define a new user specific peripheral we need to provide its HDL implementation and it will be also shown in this window and can be treated like other standard peripherals. Multiple buses and interconnects can be also added to the system like any other component. The embedded system of Figure 4.1 has two Microblaze processors. The *block diagram* view of XPS gives a nice view of the blocks in the system. Figure 4.2 is the block diagram of the embedded system of Figure 4.1.

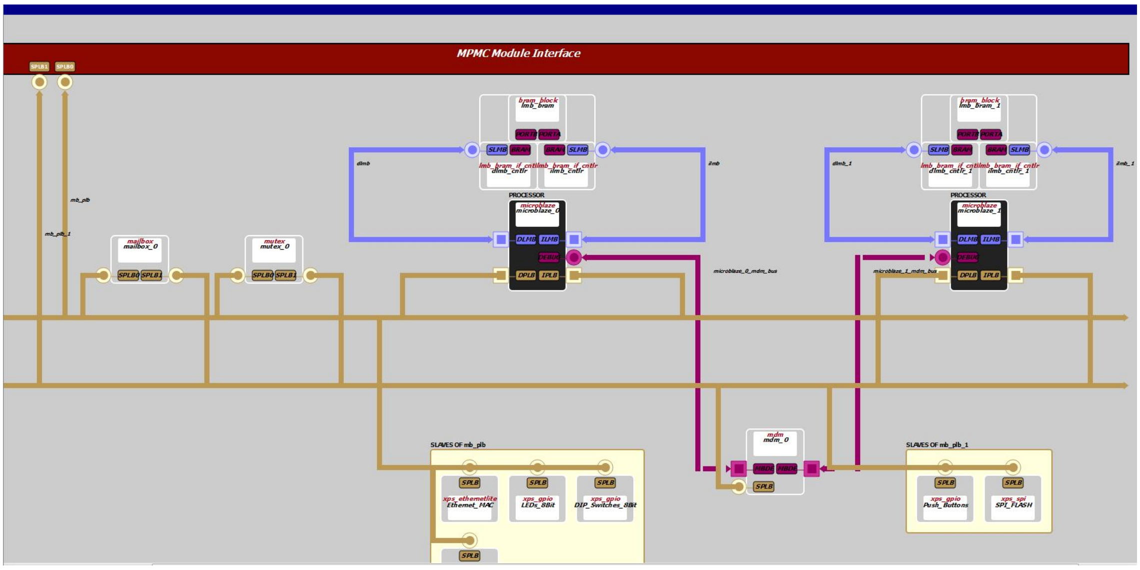

Figure 4.2: Block diagram veiw of the embedded system of figre 4.1

By double clicking on any hardware component in the system assembly view, a new window specific to that component will open. All the available options for this component can be seen in this window and configurations can be done here. There is also a link to open the data sheet of the component to get more information about its features. After building the hardware side of the embedded system in XPS, we can save it in standard Xilinx formats and project files. It can be always reopened for further modifications. So many standard and customized examples were done in this thesis project to gain enough knowledge of the tool and the way it can be used. After the hardware platform of our embedded system is ready, we can start writing our programs to be executed on it. Our C/C++ programs are built on and linked against the hardware platform libraries and software drivers. Xilinx provides a separated tool for embedded software development named SDK (Software Development Kit) [54, 55] which is another part of EDK. The designed hardware platform is exported in the form of a XML file to be later on imported into SDK. SDK provides an eclips-like software development environment and is used for writing the codes and compiling them with *GCC-mb* which is the Microblaze C compiler. In SDK, we can also configure the FPGA with the hardware bitstream (generated by XPS), download and launch the executable image files and even debug the programs that are running on the board through the JTAG connection between the SDK software on the PC and the board and see the results on the PC screen. XPS and SDK together offer a complete suit of hardware/software co-design development, simulation and debugging environment for Xilinx embedded systems. Figure 4.3 shows the appearance of the SDK. On the left side (project explorer) window, the hardware platform and all the software projects defined for this hardware platform can be seen. The third component in SDK that completes our embedded system is defining the *board support package*. It is the complete set of the drivers and libraries compiled for this specific hardware platform. The codes that we write will be compiled and built on top of this package. In the example system shown in Figure 4.3, two board support packages are defined: standalone$_{b}sp_{0}andstandalone_{b}sp_{1}.Thesearealsovisibleintheprojectexplorerwindow.$

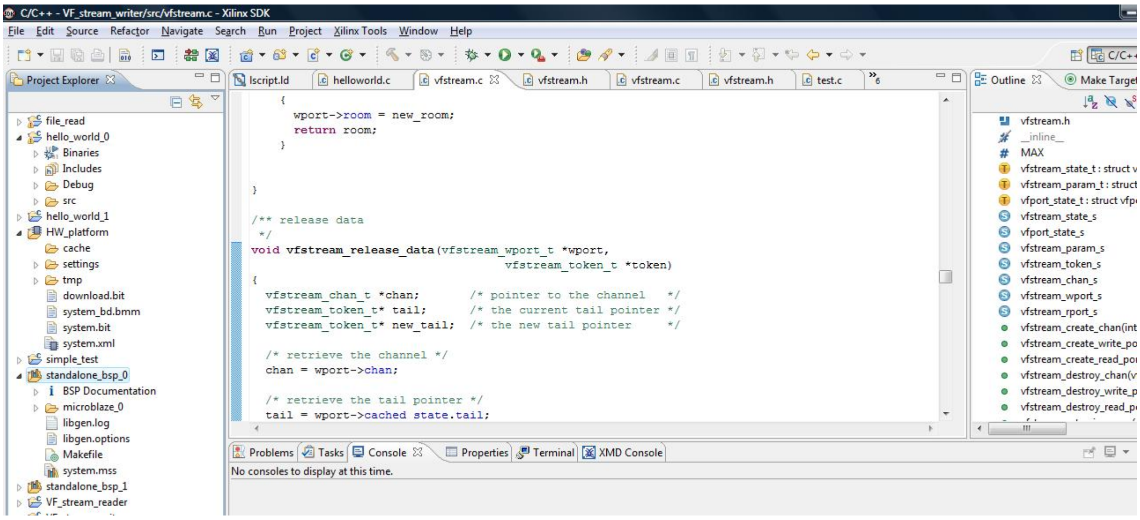

Figure 4.3: Main window of Xilinx software Development kit (SDK)

4.2 Memory system

In our FPGA-based embedded system, we have two major types of memories: local RAMs and External RAM. They can in general be used for any type of data or program storage.

4.2.1 Block RAMs

Local RAMs have two types: BRAMs(Block RAMs) [59] and distributed RAMs. The block RAMs are built-in memory blocks inside the FPGA that can be instantiated in the system with different sizes and widths and distributed RAMs are actually small memory pieces distributed throughout the chip in LUTs(look-up tables). BRAMs are usually used by Xilinx tools when bigger memory blocks are required and distributed RAMs are used otherwise. It is also possible to explicitly determine if we prefer either one of them. Simulation models and behavioral VHDL/Verilog descriptions of these memories are available for verification and RLT development purposes. Normally, in our embedded systems, BRAMs are used as small local memories for storing boot-up program images and specific local data. BRAMs can be used to store any program or data but due to their small size they are usually not considered as the main storage place for application codes or data blocks.

4.2.2 External memory

The main global memory of the system is the shared external DDR II or DDR III module on the board. The external memory can be used to store the data or the instructions. In this project, a 256 MB DDR II memory module available on the spartan3dsp board was used. The external memory can be accessed using a fully parametrized memory controller module provided by Xilinx [60]. This multi-port memory controller is a versatile module with configurable number of ports and a wide range of supported interfaces. The way this module works is very important that may have serious impacts on the correctness and performance of the system. It needed to be studied and tested using some example systems for some behavior analysis before being used in the final solution.

- MPMC can have up to 8 input ports.
- Each port is configurable as PLB, XCL, MIB, etc.
- The arbitration policy between the cores can be configured by user.

All the memory requests to the external memory are eventually delivered to MPMC for actual execution. As a result, it is extremely important to understand how this module process these requests. Transactions are buffered and serviced in order on a single port. Across multiple ports however, there is no guarantee that transactions issued by different ports will complete in order [60]. It is possible to change the arbitration policy so that a specific port is favored over the others, but this is only a mechanism to partially influence transaction orderings and not enough to guarantee a specific order in the whole system. In the design of the hardware architecture of the proposed solution, we take advantage of the fact that transactions are guaranteed to complete in order on one port. We will see more details on this shortly in this chapter.

4.3 BUS infrastructure

All the components of the embedded system are connected to a global bus infrastructure. The global address space is defined over this global bus. There are also some other bus interconnects in the FPGA that can be used for other partial connections.

4.3.1 LMB (Local Memory Bus)

LMB [61] is a simple synchronous bus that is used to connect to BRAMs. Fast access to local memories are managed on this bus. In Figures 4.1 and 4.2, it is clear that LMB buses are used to connect the processors to their local data and instruction memories.

4.3.2 FSL Links

As discussed in the previous chapter, FSL links are high performance hardware FIFO channels.

4.3.3 XCL(Xilinx Cache link)

XCL interface [23] is a high performance channel for connecting the processor to external memory. It is designed based on integrated FSL links. This interface is only activated on the Microblaze when the caches are enabled. It is only dedicated to accessing the *cached* memory area of the external memory. This interface does not reorder memory requests as it has a FIFO-based nature. The main purpose of XCL in Xilinx is having a fast and direct connection to memory controller(MPMC).

4.3.4 PLB(Processor Local Bus)

PLB bus [62, 63] is the global bus infrastructure in Xilinx FPGAs and also in our embedded systems. It is a 128-bit wide high performance bus that can be used to connect an optional number of masters and slaves to the same address space. The original PLB bus standard is simplified in Xilinx. The Xilinx version of the bus has the following features:

- Arbitration support for a configurable number of PLB master devices.
- PLB address and data steering support for all masters.
- 128-bit, 64-bit, and 32-bit support for masters and slaves.
- PLB address pipelining (supported in shared bus mode or point-to-point configuration).
- Supports a configurable number of slave devices.
- PLB watchdog timer.
- Selectable round robin or fixed priority arbitration.

PLB bus does not reorder the transactions. All the memory requests complete in the same order that they are issued with respect to the issuing master. PLB bus signaling and protocol details are rather complicated. In this project, a customized *PLB bus master* module was used to connect to PLB system. The bus master simplifies the data transfer process and acts as a bridge between the accelerator and the bus interconnect. Microblaze processor has its own PLB interface port. In the next chapter, more information will be given on PLB bus master and how it has been used in the system with accelerators.

4.3.5 AXI BUS

Starting from *Spartan-6* and *Virtex-6* FPGA families and ISE design software version 12.3, Xilinx started to support the high performance AXI bus standard from ARM [64, 7]. AXI or *Advanced Extendable Interface*, is part of the famous ARM AMBA bus family specification. AXI was first released in 2003 with ABMA 3.0 and the last version was released with AMBA 4.0 in 2010. there are three types of AXI interfaces:

- AXI4: for high performance memory mapped requirements.
- AXI4-lite: for simple, low-throughput applications.
- AXI4-stream: for high speed streaming data.

Having three types of interfaces, AXI is a perfect and flexible kind of standard that seriously improves the productivity of the FPGA-based embedded systems. Is also provides more availability as it is a very well documented and accepted bus standard worldwide, and designers can have access to all kinds of support and documentation. At the time of doing this thesis project, there was still no development board available that supported AXI at Vector Fabrics. Furthermore, the original intention of the thesis was to address the memory issues and propose solutions for the PLB-based systems of Vector Fabrics tools. However, AXI bus was studied and analyzed in Xilinx systems and in this report the hypothetical role of this bus in VF FPGA-based embedded systems and how it influences the proposed design pattern and solutions are reviewed as an introduction to the next generation of these systems. AXI bus and many AXI-based IPs and modules are at the moment available in Xilinx tooling to be used with Microblaze. In near future, Xilinx is going to start supporting ARM processor in its FPGAs. Having hardcores of ARM inside the FPGA is a very excellent facility. Vector Fabrics is supporting the ARM platform without hardware accelerators now and is going to travel to ARM-based Xilinx FPGAs after it is launched. AXI4 bus will be the standard global bus in these new systems. As stated before, the proposed solution of this thesis could be used on the new platform even with some modifications on its implementation. AXI4 bus protocol consists of five different channels:

- Read Address Channel.
- Write Address Channel.
- Read Data Channel.
- Write Data Channel.
- Write Response Channel.

Data can be transferred at both directions between masters and slaves and the size of the transfer can vary. The burst size is limited to 256 data transfers in AXI4 while this number is 1 data transfer per transaction for AXI4-lite. The concept of read and write channels are depicted in Figures 4.4 and 4.5 respectively. As it can be seen, AXI4 allows separated connections for the address and data which simply means simultaneous bidirectional data transfers for reads and writes. AXI4 bus protocol has so many selectable options that make the interface suitable for high throughput systems. Options like bursting, data up-sizing and downsizing, multiple outstanding addresses, and out-of-order transaction processing. AXI bus has a feature to reorder memory requests that have been assigned different ID tags. This is for performance improvement throughout the whole system. Normally, the requests originated from a master have the same ID tag and will be completed in order. Two very important features of AXI4 interface at least to our systems are *exclusive access* and *acknowledgment* in the response channels.

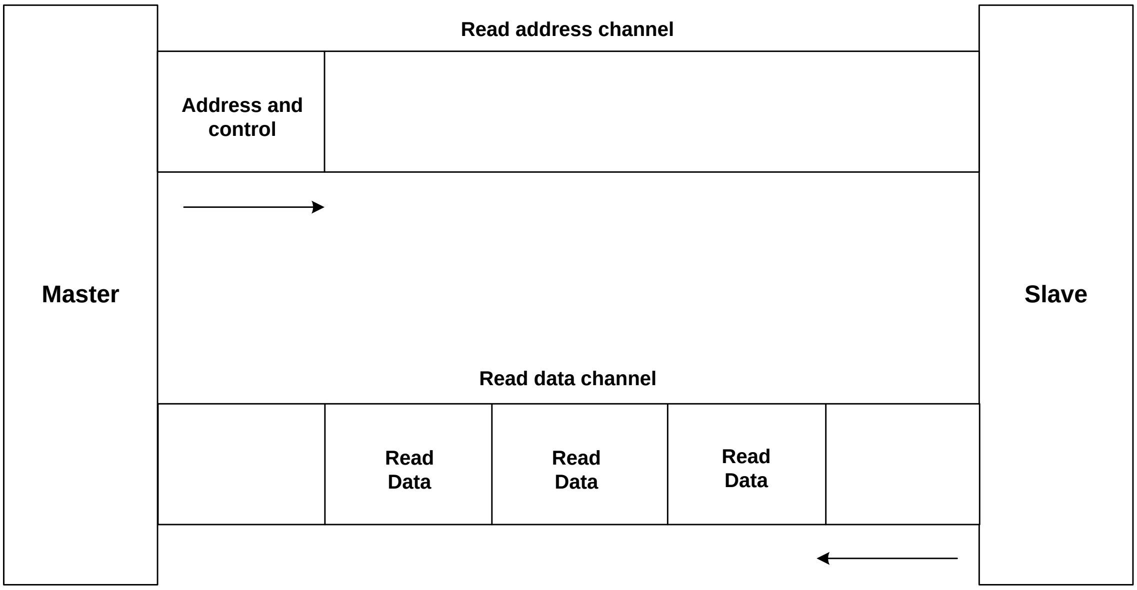

Figure 4.4: AXI read channel

Exclusive access

This feature can be used to establish an exclusive access to an addressed range or location that could have been mapped to a part of memory or a shared resource. Exclusive access is a very decisive feature that can be used in some critical parts of parallel programs to implement shared resource management or reliable standard mutual exclusion. In order to acquire an exclusive access, a procedure needs to be started. The access is terminated by releasing it. Unfortunately, Xilinx has not yet started to fully support this nice feature of AXI standard [64]. This is part of the original features of AXI4 that are dropped by current simplifications of Xilinx.

1. A master performs an exclusive read from an address location.

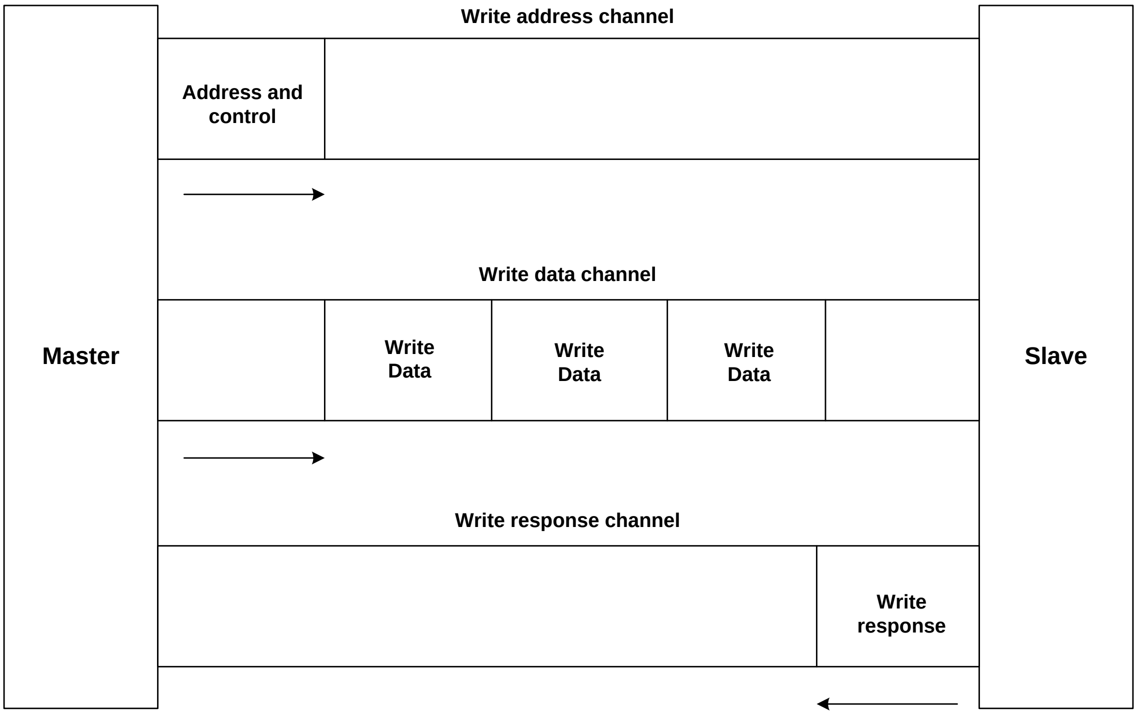

Figure 4.5: AXI write channel

2. At some later time, the master attempts to complete the exclusive operation by performing an exclusive write to the same address location.

3. The exclusive write access of the master is signaled as:

    - *Successful:* if no other master has written to that location between the read and the write accesses.
    - *Failed:* if another master has written to that location between the read and write the accesses. In this case the address location is not updated.

A test-and-set, read-modify-write or load-link/store-conditional facility can be implemented using this feature of AXI interface. This could be very useful in resolving consistency issues in some situations. We will see how this can be used in chapter 6. In Microblaze processor, the exclusive *LWX* and *SWX* instructions operate taking the exclusive access status on the global AXI bus into account. However, since this feature is not yet supported in Xilinx FPGAs, a exclusive access request eventually leads to a failure and it is not possible to establish an exclusive access on a multi-processor system. SWX and LWX can still be used for making exclusive accesses in a multi-thread environment on one Microblaze processor.

Response channels

Having independent response channels offer a very useful feature to parallel programs. It is possible for a writing node in the system to make sure if its data has been received by the receiving node. If the receiving node is an AXI-compatible memory controller, then it simply means that the writer can have the confidence that the data is at least buffered in the memory queue. This feature can be used if the producer processor wants to signal to the consumer(s) that the data is updated. We will see in the following chapters, how this feature of AXI bus can be employed in resolving consistency issues of our system.

4.4 Accelerators

Accelerators in VF embedded systems are dedicated hardware unit implemented in Verilog to increase the overall throughput of the system. They are inserted into the embedded system in Xilinx embedded environment as user defined peripherals. Accelerators are not general purpose processing units, they are only developed to run a specific high-throughput task and deliver the result within a shorter period of time. Nevertheless, at system level, we can look at accelerators as processing nodes in the same position with respect to the bus infrastructure and shared memory as CPUs. They access the external memory, they can have their own local memories and they communicate with other nodes whether it is another accelerator or CPU via shared memory or even a dedicated FSL or AXI-stream link. The local memories of accelerators are normally private memories connected to them on LMB bus. However, these memories can be put on the shared global bus to be accessible by other cores via *PLB bus slave* interface that would lead to a distributed shared memory multi-core system. This kind of local memories are not used as the main systematic data communication channel between the cores, which is the main focus of this project. In addition, their contents are not cached by the cores anyway as they are on-chip memories. So technically, they are not any concern to the main issues addressed in this project. Accelerators are generated by VF tools using high level synthesis techniques. They can be anything like a FFT engine, advanced arithmetic unit, video processing or signal processing module. They can have their own cache controllers. Part of this thesis project was designing and implementing a cache controller for accelerators (covered in chapter 5). The interface to accelerators is also defined and implemented, however, designing a specific accelerator module itself depends on the application and was not one of the goals of this project. Moreover, accelerators will be automatically generated by VF tools anyhow.

4.5 Processor

The general purpose processor used in our embedded system is Microblaze. Microblaze is a highly configurable softcore processor from Xilinx. It is very widely used and well documented by Xilinx. Microblaze has a Harvard architecture. Since it is an embedded processor, it does not provide advanced optimization techniques like dynamic execution or superscalar architecture. The processor has a three/five stage pipeline. Behavioral and simulation models are available for Microblaze and this makes it easy to simulate the whole embedded system in a HDL simulation environment for signal level debugging. Microblaze provides a wide range of configurable features. Microblaze reference manual could be consulted for the full description. The main features are:

- 32-Bit general purpose registers.
- 32-Bit address bus.
- Single Issue pipeline.
- Optional one level of data cache.
- Optional one level of instruction cache.
- Branch target address with branch prediction scheme.
- Virtual memory management.
- Three or five stage pipeline.
- Floating point Unit.
- Multiple bus interface supporting.
  - FSL - Fast Simplex Link.
  - PLB - Processor Local Bus.
  - AXI - Advanced Extensible Interface.
  - XCL - Xilinx Cache Link.
  - LMB - Local Memory Bus.

Microblaze hardware does not reorder instructions. Regarding the compiler, it does not reorder the instructions by default and it does not really matter even in case it does, if we use a *weak/release consistency* model and the synchronization is made sure to take effect and be seen by the other cores after the data access is complete. Consequently, we will not be worried about the destructive effect of compilers or CPU architecture optimizations on the consistency of our parallel programs. Microblaze processor is a very versatile and optimized processor and a very good choice for embedded solutions due to its high degree of availability, productivity and flexibility and also a full support in terms of tooling for design and documentation. As stated before, Vector Fabrics is also going to support ARM processors in FPGA-based embedded systems in near future. It means that in some applications, Microblaze will be replaced with ARM cores in the embedded system. It is obvious that many details will be different, but the original idea of this thesis was to come up with the design pattern and implement and test it on the FPGA only with this idea in mind that ARM-based architectures will accommodate it too. The main reason of studying AXI bus infrastructure was because of this fact, because the ARM processor itself provides all the functionality that is used from Microblaze in this project. Having a comprehensive review of the ARM processor however, was not intended.

4.6 The architecture of our embedded systems

We can now see how our embedded systems generally look like. Figure 4.6 illustrates the overall architecture of our embedded systems implemented on Xilinx FPGAs. A global PLB bus connects all the components. In principle, multiple processors and accelerators can be in the system. In this thesis project, two systems are used and tested to implement and demonstrate the proposed solution. One Dual-Microblaze system and one Accelerator-Microblaze system. The solution can be further extended in Vector Fabrics tools to embedded systems with more cores. Memory accesses from different cores shown in Figure 4.6 with colored dash-lines through their local caches and all the way down through the bus to external memory cause the potential consistency/coherence issues in our parallel programs. These issues are restricted and eliminated by having the proposed WFIFO-based design pattern on a system like Figure 4.6.

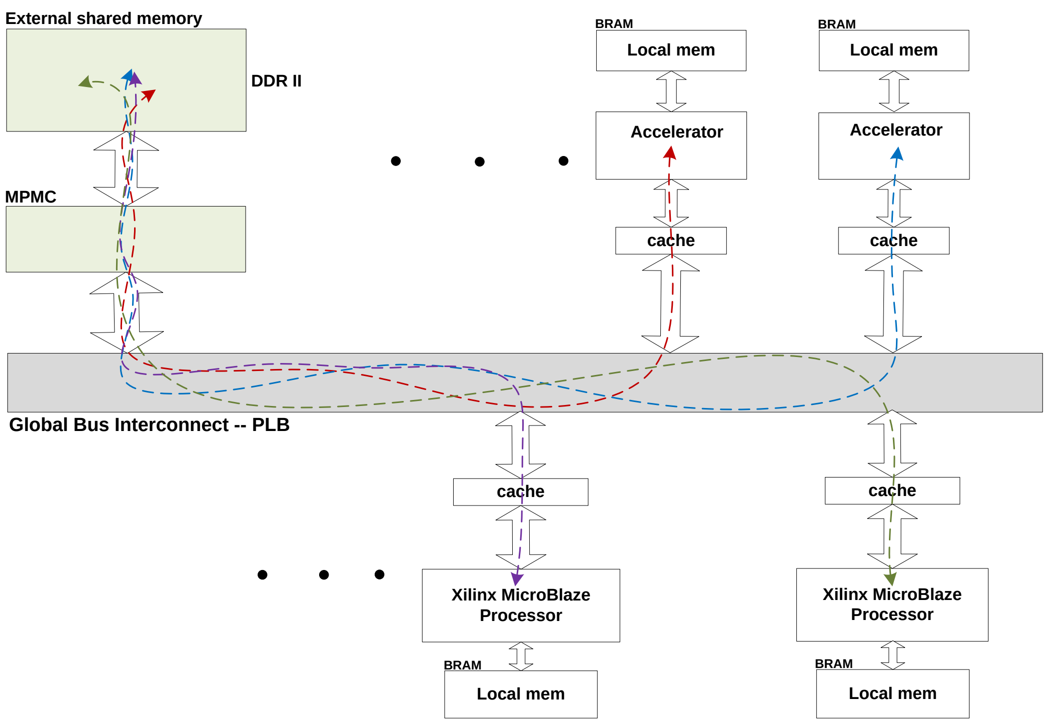

Figure 4.6: General architecture of our FPGA-based embedded systems

5 Accelerator cache memory

In this chapter, Section 5.1 gives a brief introduction to cache memories and their role in the design of computer systems, Section 5.2 specifically focuses on the need of cache memory for hardware accelerators in our target embedded systems, Sections 5.3 to 5.5 that explain the details of the micro-architecture of the cache memory are removed from this public version of the report due to Vector Fabrics company confidentiality.

5.1 Brief introduction to caches

There are lots of books and papers written about memory hierarchy and cache design. Caches are small amounts of high speed memory [65]. Caches have long been a mechanism for speeding up the memory access and are popular in embedded hardware architectures from Microcontrollers to core-based ASIC designs [66]. They were basically designed to fill the performance gap between the CPUs and main memories and they sit between them. Over the years, there have been a huge revolution in both architectural and fabrication properties of CPUs that has led in more and more performance and speed, however, main memories like SRAMS, DDR II or DDR III have never been able to catch up with this growth and are lagging behind CPUs in terms of speed and throughput. On the other hand, the need for larger main memory would not let engineers to put all the memory on the chip as it is too expensive. Therefore, memory hierarchy was invented to logically fill the gap and make it faster for the CPU to access its desired data. Figure 5.1 shows the memory hierarchy concept. As it is clear in the figure, the further the address leaves the CPU to access a data item, the slower it becomes. In this context, the first level of memory hierarchy is smaller than the next one but it is way faster and can service the processor more efficiently if it contains the requested data. Different design policies and architectural tricks are used nowadays to have a cache containing the most frequently used parts of the main memory that are needed the most by CPU. Creative techniques are used to decrease the miss rate of the cache and increase the probability of having the data needed by CPU. It is not intended to go into all the details about cache policies and their characteristics in this chapter. [17, 18] could be referred to for more information. [67] can be used for more in-depth knowledge about caches and memory hierarchy.

By definition, caches are actually the first level of the memory hierarchy that the address encounters once it leaves the processor [17]. It means that traditionally, CPU does not know anything about the cache existence and just simply requests what it needs. It is up to the cache controller to set up the interface between the CPU and the next level of memory hierarchy and send back the proper and usually faster response to the CPU. As stated before, cache is meant to hold the CPU's most desirable data, so

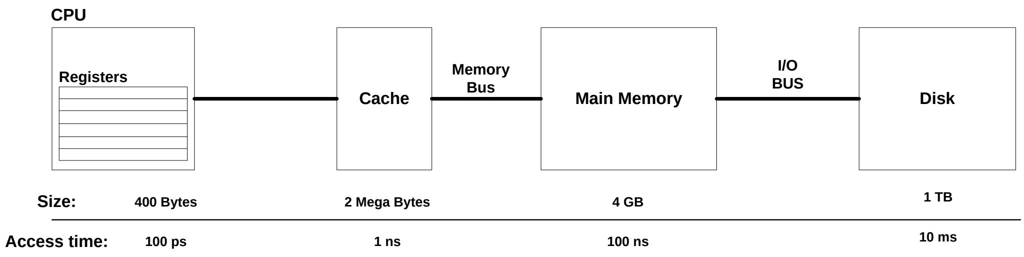

Figure 5.1: Memory Hierarchy

there is a need to figure out how different locations of the main memory are going to be mapped into the cache and how they are going to be accessed and later on replaced with newer data. The simple question to be asked here is that: where is one word or block of the main memory exactly going to be mapped in the cache? Answering this question infer different types of caches. There are in general three main types: *Fully associative* caches are the ones that can store the desired block anywhere and there is not any specific place for the data. In *direct map* caches, the block of data read from main memory has only one place to go in the cache, it means that there are several data blocks from main memory targeting the same place in the cache. *Set associative* caches are something in between, they let the memory block to be placed in any one of the n places inside one *set* in a *n-way set associative* cache. The example in Figure 5.3 illustrates the concept. More complicated systems may use a combination of these three basic types.

To the cache controller, the address generated by CPU seems like what is shown in Figure 5.4. This address is in fact targeting a data item in the main memory. It is interpreted by the cache controller like Figure 5.4 to locate the requested data in the cache.

- **Block offset** is the address for the desired word/byte of data inside the data block.
- **Block address** is the address of the data block. Block address can be further divided into two fields:
  - **Tag** which is the high portion of the data used by cache controller to be compared against the tag field of the data block stored in the cache to see if we have a *cache hit*.
  - **Index** that selects the appropriate cache set or cache line in the cache. The size of the cache or more accurately the number of cache blocks stored in total in the cache determines the value of index. Figure 5.2 depicts a simple example of a direct mapped cache read procedure. If the high portion of the address of the requested data block matches the tag field of the data block already stored in the cache, then there is a cache hit and we can read out the corresponding data out of the cache, otherwise, there is a *cache miss* and the data should be fetched from main memory.

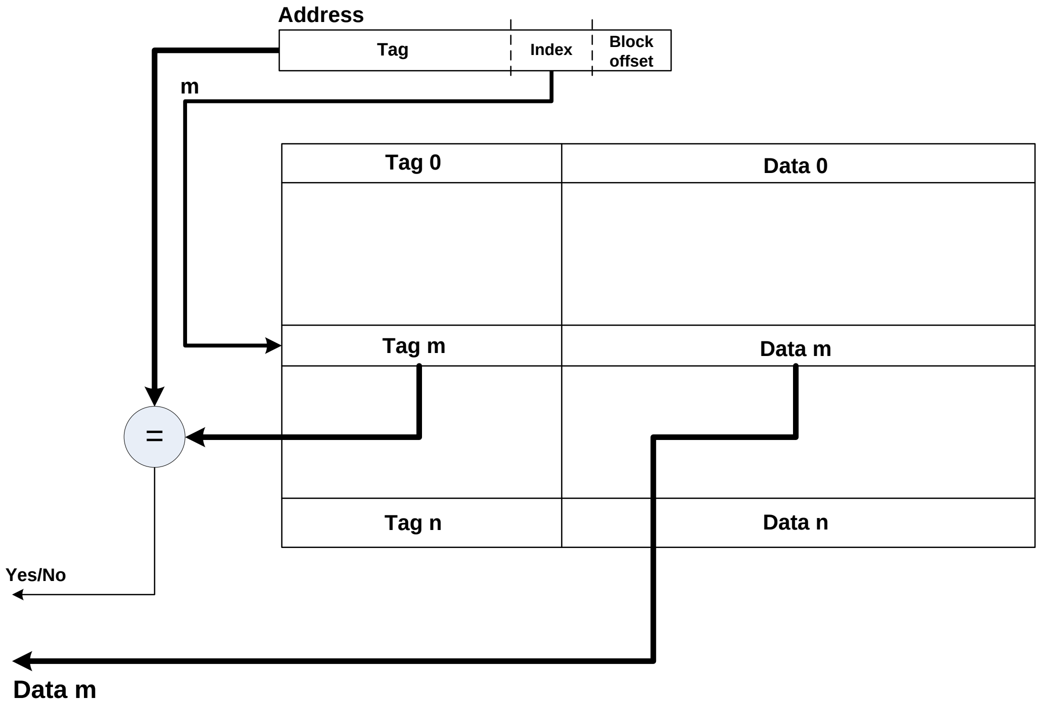

Figure 5.2: Direct map cache access

5.2 Cache in VF embedded systems

In this section, we will take a closer look to the position of the cache controller with respect to the accelerator and the whole embedded system. Figure 5.5 shows how the cache memory fits in a VF embedded system with one accelerator and one Microblaze processor. The access time to external memory for VF accelerator is not trivial in this architecture and that is exactly where the cache controller comes in. The developed cache controller module is going to sit between VF accelerator and PLB bus master interface to speed up the data traffic to/from external memory. VF accelerator still initiates its load and store requests and can have no idea about the intervening cache controller who receives these requests and acknowledges the accelerator with proper data and control signals. The interface of PLB Bus master is a simple synchronous interface called the *ld/st* interface. Both sides of the cache controller should work based on this interface. This leads to the situation that the cache controller is transparent to the accelerator.

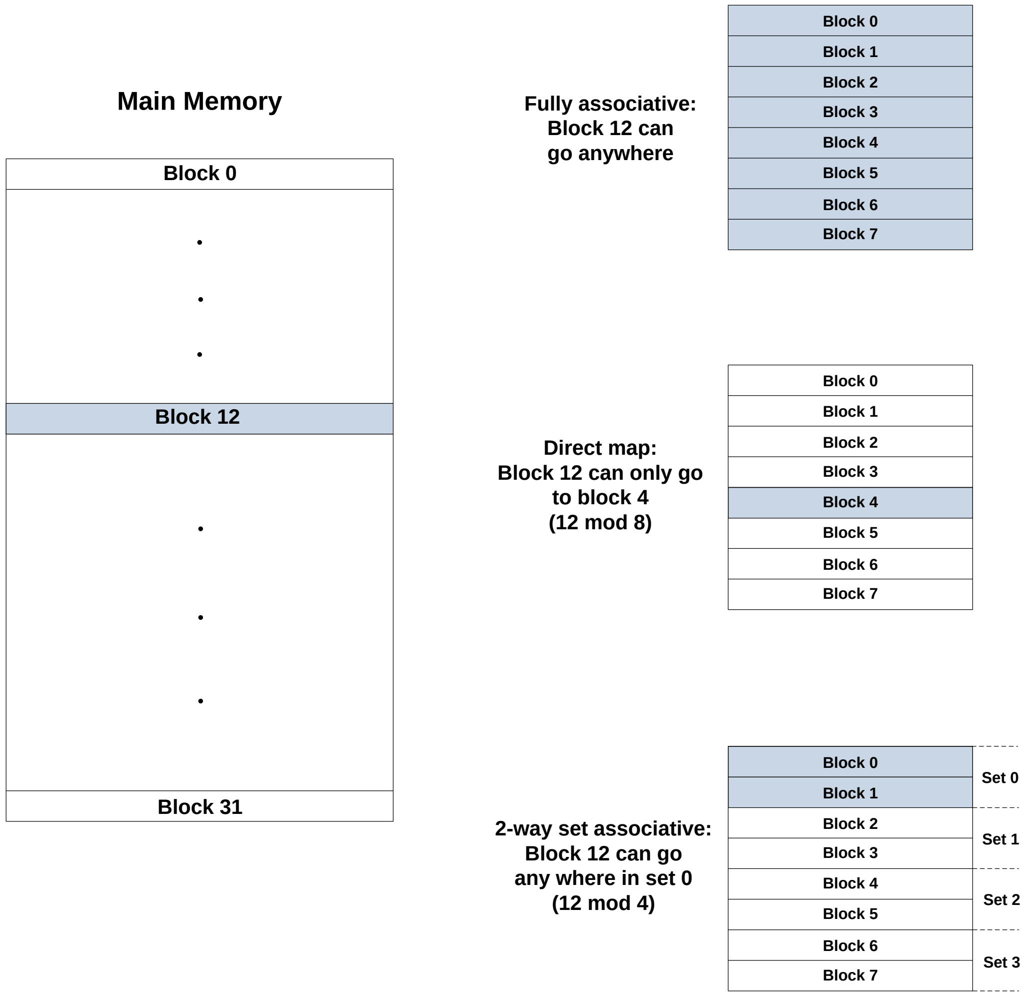

Figure 5.3: Basic types of cache organizations

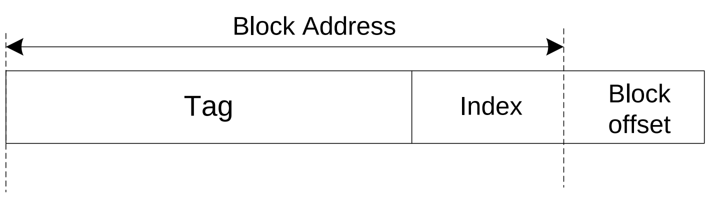

Figure 5.4: Address bus mapping

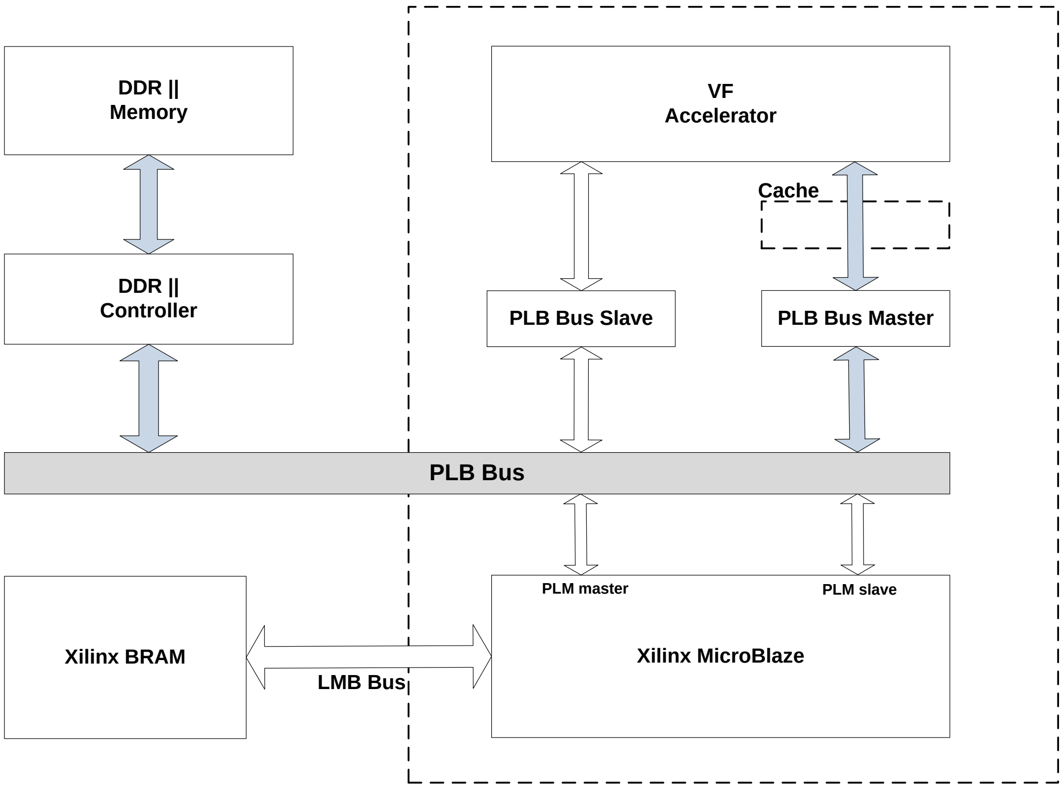

Figure 5.5: Cache memory of the accelerator in VF embedded systems

6 Software lib and APIs

This chapter reviews the principles and features of the implemented WFIFO communication channel and its main APIs. Section 6.1 reviews the idea of software-based cache coherence control and explains the basic design pattern and the way the its cache coherence is maintained, Section 6.2 presents the details of the streaming library and its main software APIs, the main concepts and definitions are discussed in this section for the user and finally Section 6.3 explains in details how a SW-SW embedded system consisting of two Microblaze processors is designed on the FPGA platform and how the software APIs and the streaming library are used to establish the WFIFO communiucation channel.

6.1 Basic design pattern

We first introduce the concept of the software-based cache coherence control in embedded systems. Assume that we have a simple design pattern in a Dual-Microblaze system. One processor is processing a block of data stored in the main memory and after the processing is done, the block ownership is transferred to the other processor. This ownership transfer is actually the synchronization that can happen through any hardware or software channel between these two cores. Figure 6.1 illustrates the concept. The data block is sequentially stored in the shared memory. Naturally, both CPUs would cache the data block content as long as they are doing any kind of processing or access on/to its contents. Our approach is not to transfer the whole data block via a hardware FIFO channel or FSL link as it is not scalable nor vendor-independent. As a result, the data transfer is nothing except the transfer of the ownership of the data block. In such a scheme, CPU 2 will be notified by some means that it can now start accessing the data block. It is obvious that the data block is only being accessed by one core at a given time and so the domain of the coherence issues is limited. However, the synchronization itself is still something in common between both cores and the data block itself has been naturally updated in the cache of CPU 1 before the synchronization takes place. We must therefore make sure that after CPU 2 sees the synchronization, it will only access the latest values of data in the main memory updated by CPU 1. This can be achieved by flushing out the data block to the main memory before the synchronization point I. After CPU 2 is done reading the data block, it has to invalidate its local copy of the data block in its cache before synchronization point II, because CPU 1 is going to modify the block again. This simple model is the basic data structure of the proposed WFIFO design pattern in terms of caching and synchronization. The data block could be elements or windows of the WFIFO. Extra control structs and address management fields are needed to establish the desired FIFO operation between the cores. The synchronization also needs to be imported into the FIFO control structs in combination with high level cache control commands to make sure about the coherent data exchange. If our parallel application works based on the basic pattern of figure 6.1, the cache coherence control becomes simple: we need to flush out all the data items in the range of the data block from the cache of CPU 1 to external memory before the synchronization point. The flushing out can be achieved using the standard *cache control APIs* of the processor or the *cache control interface* of the accelerator's cache controller.

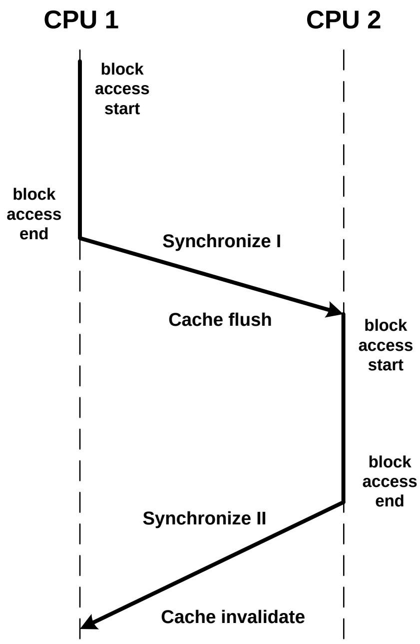

Figure 6.1: Block level software-based cache coherence

6.1.1 Microblaze cache handling

Most of the newer processors provide at least some basic cache control software APIs. ARM and Microblaze also do so. In this project, the cache handling APIs of Microblaze were used [68]. Microblaze processor has configurable optional caches for data and instructions. Both caches are restricted to be only direct mapped. So they are not optimum choices for our design pattern. It is possible to configure the data cache as *write through* or *write back*. The cache size, tag size and block size are also configurable. In this project, the data cache was configured as write-back with the cache block size of 8 words. Microblaze provides cache control APIs for both instruction cache and data cache. We do not interfere with the operation of the instruction cache and only manage the data cache. The data cache control APIs of Microblaze [68] are as follow:

- **Microblaze_enable_dcache()**
  Used for enabling and disabling the cache in the program. The cache must have been already enabled in the hardware during the initial configuration of the pro-

cessor in XPS environment. This function must be always used in the beginning of all programs.

- **void Microblaze_flush_dcache()**
  This function flushes out the whole content of the data cache to external memory. It also invalidates the cache. This API is actually used when we are using a write-back policy in the cache.

- **void Microblaze_flush_dcache_range(unsigned int cache_addr, unsigned int cache_len)**
  The same functionality as the previous function but with more flexibility. It will flush out the cache content within the specified address range *cache_addr* to *cache_addre + cache_len -1.*

- **void Microblaze_invalidate_dcache()**
  Invalidates the entire cache content.

- **void Microblaze_invalidate_dcache_range(unsigned int cache_addr, unsigned int cache_size)**
  Invalidates the specified address range in the cache. It can be used to invalidate part of or all the data cache content.

Using the invalidation APIs are more dangerous to the data content of the cache as the cache controller does not take care of the dirty data contents. In this thesis, we used the *Microblaze_flush_dcache_range* API in the software implementation of the WFIFO. As discussed in chapter 5, the designed cache controller of the accelerator also provides the same feature as these Microblaze APIs. This feature is used when using the WFIFO interface between the CPU and the accelerator. It must be noted that in all of these APIs, the invalidation or flush out starts from the cache block that the *cache_addr* belongs to and ends at the cache block that *(cache_addr + cache_len - 1)* belongs to. Many tests were done to make sure about the functionality of these APIs on a Dual-Microblaze system on the Spartan3dsp Development board. The C programs were executed on each processor independently to exchange some blocks of data based on the structure of figure 6.1. All the situations of the data channel were programmed and tested on the FPGA board using different data traffics.

6.2 The stream library

In this section we will review the properties and details of our WFIFO streaming library and its APIs. *VFStream* [69] is a streaming library used for data exchange and communication between POSIX threads. It is originally developed at *Vector Fabrics* for WFIFO data communication in multi-thread Linux-based environments. It relies on pthread libraries. The main feature of this library is its token-based implementation. The channel buffer for data communication is constructed in the shared memory. In this thesis project, the WFIFO was developed based on the structure of *VFStream* library that provides an API set according to our design pattern. The original implementation of VFStream was completely modified and synchronization points and cache control APIs were added to it appropriately. Some features of the original library were removed and some new features were added. The new VFStream library can be used to establish a *coherent and consistent* WFIFO communication channel on a multi-processor system on Xilinx FPGAs. It can be also used for communication between hardware and software. It is not possible to explain all the details of the implementation in this report. However, the main definitions and the most important APIs are discussed in the following section. From this point on, VFStream means the version that was implemented and tested for this thesis project. In the following section, we will review the principles and concepts of the token-based WFIFO structure. These concepts are mainly from the original VFstream library from Vector Fabrics that have been ported to our embedded system.

6.2.1 Basic definitions and concepts

Figure 6.2 shows the architecture of the WFIFO core. It is a circular FIFO channel buffer working in a first-in-first-out manner with the length of n.

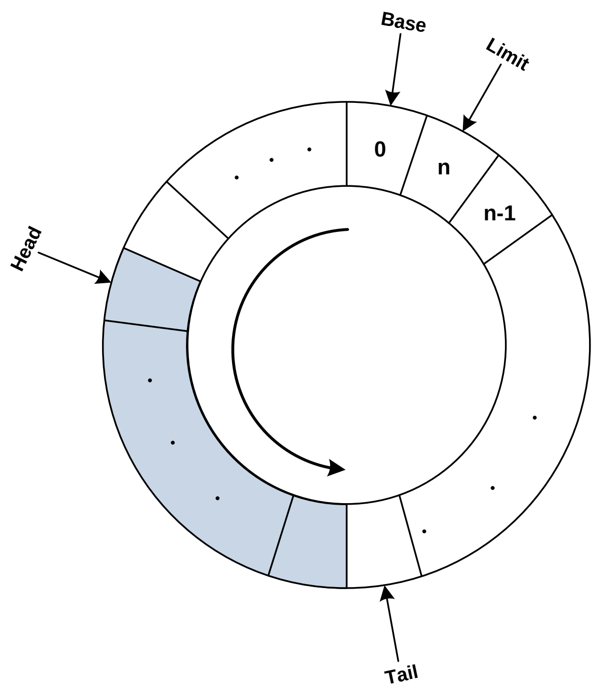

Figure 6.2: Circular FIFO channel structure

- **Tokens**
  Tokens are the basic units of data being transferred in VFStream FIFOs. Tokens

have starting address and length parameters.

- **Base pointer**
  This is the address of the first data token of the FIFO channel in the memory. It is set when the channel is created.

- **Limit pointer**
  This is the first address beyond the the address of the last data token of the FIFO channel. It is set when the channel is created.

- **Head pointer**
  This is the address of the latest data token available in the FIFO to be read. It is set to *base* when the channel is created.

- **Tail pointer**
  This is the address of the next available room token for writing a data. It is set to *base* when the channel is created.

Controlling the status of the FIFO is done using the above basic pointers. In the beginning, the channel and the read and write ports are created. Read and write ports have *data* and *room* pointers that are used with *head* and *tail* pointers to check if the FIFO is full or empty. Data pointer of the read port is the address of the token that can be read and the room pointer of the write port is the address of the token that can be written to. Initially, the *head*, *tail*, *room* and *data* pointers are all set to *base* as the FIFO is empty. If head and tail are equal then the FIFO is empty. The address just before head is always unused so that tail would never bump into head again after it wraps around. This is for distinguishing between a full FIFO and an empty FIFO. The FIFO is first created and its data and control space is allocated in the memory. Then the read port and write port are created to be connected to the created FIFO. Data write and read are done via these ports. It is possible to define one read port and one write port attached to each FIFO channel. However, multiple readers or writers can exits in the system to read and write data from/to the FIFO. After defining the FIFO and its ports, the following standard procedure should be followed to communicate data:

1. A room token must be acquired.

2. Data is written into the room token.

3. After the Writer is done with the room token, the room is released.

4. After this point, the room token is considered occupied until a reader acquires and reads it.

The process above is executed using the provided APIs. The pointers of the FIFO and the cache coherence controls are handled automatically. After at least one room token is released in the FIFO, a reader processor can start accessing that token. The procedure is as follows:

1. A data token must be acquired.

2. Data is read out from the data token.

3. After the reader is done with the data token, the data is released.

4. After this point, the data token is considered empty until a writer again acquires and writes into it.

In the write procedure, we refer to the token as *room token* and in the read procedure, we refer to it as *data token.* Using this method of memory access we have a great flexibility in data communication between processing units. The flexible features of this WFIFO system are:

1. It is possible to acquire more than one room token. The room release will always release the oldest acquired room.

2. Because the acquired room token is accessed using its address, a writer CPU or accelerator has the possibility to do any kind of processing on the token space in the memory. The room token can be temporarily read or written during the processing and after the final values are written, the room will be released.

3. It is possible to acquire more than one data token. The data release will always release the oldest acquired data token.

4. It is possible to define a token of any size. At the highest level, APIs are defined for reading and writing the standard C-compatible data objects to the FIFO. However, a layer lower and using the provided *put* and *get* primitives, it is also possible to access each byte, half-word, word or double-word inside a wide token with an offset parameter. This offsetting adds even more flexibility to the FIFO system and freedom to processors. The Windowed-FIFO is in fact a FIFO constructed this way, with wide tokens (windows) and the flexibility to access any byte in the window by means of *offset.* Using this method, in a video application, we can define a token as large as one video frame. After the first processing unit is done with its work on this frame, it will be released so that the next processing unit can access it for the next phase of the processing. In another application, we may define a small FIFO only to send tiny data items to other cores.

5. It is possible to skip a data token without reading it. In many applications, this improves the performance.

6. It is not necessary to completely fill a wide room token before releasing it or to completely read a wide data token before releasing it.

7. Several FIFO channels can be defined in the main memory without any extra overhead.

Figure 6.3 and 6.4 illustrate the standard object access and the wide token access concepts respectively.

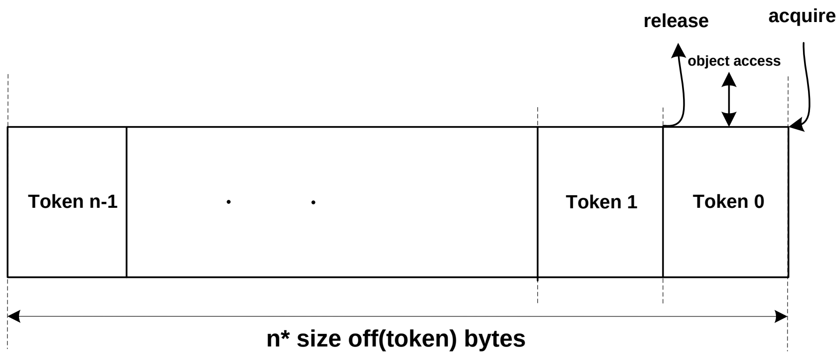

Figure 6.3: Standard object access

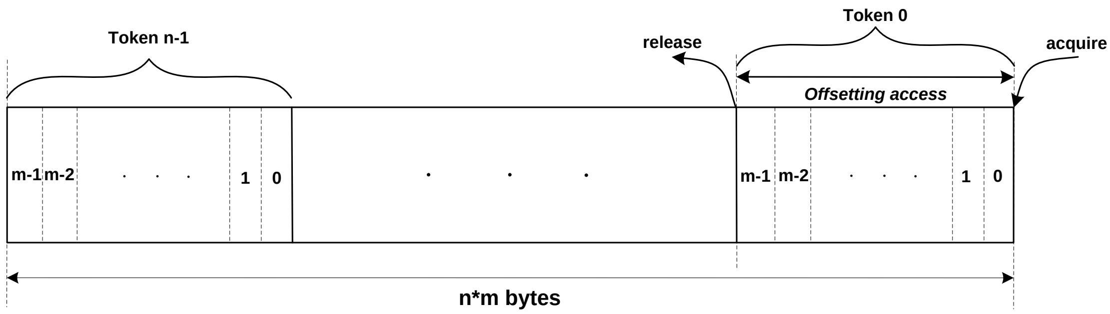

Figure 6.4: Wide token access with offsetting

Read port and write port are structs defined in the FIFO for handling the FIFO administration. They have several fields that contain the parameters of the FIFO channel and pointers to the next available token for read or write. So many data types and functions are defined in the system that are not possible to be completely explained here. However, in the following section, the main APIs are described in more details.

6.2.2 VFStream main APIs

- **vfstream_chan_t* vfstream_create_chan(**
  int num_tokens,
  size_t token_size,
  vfstream_malloc_t* ctl_space,
  vfstream_malloc_t* buf_space)
  This API is used to create a FIFO channel and initialize its control structs. The

length of the FIFO is *num_tokens* and the size of each token is *token_size* bytes. *ctl_space* is the pointer to the memory management implementation that allocates memory for storing the control structs of the FIFO. In our Microblaze platform, this is a function that statically allocates some space in the shared memory on the *uncached* area. *buf_space* is the pointer to the memory management implementation that allocates memory for storing the channel buffer itself. In our Microblaze platform, this is the standard *malloc* function of C that dynamically allocates memory from *heap* section that is mapped to the shared *cached* area. The address of the FIFO channel control struct is returned.

- **vfstream_wport_t* vfstream_create_write_port(**
vfstream_chan_t* chan,
vfstream_malloc_t* port_space)
This API is used to create a write port and initialize its control structs. It will be connected to the FIFO channel pointed by *chan*. The port_space points to the memory management implementation. In our Microblaze platform, this is a function that statically allocates some space from the shared memory in the *uncached* area. The address of the write port control struct is returned.

- **vfstream_rport_t * vfstream_create_read_port(**
vfstream_chan_t *chan,
vfstream_malloc_t *port_space)
This API is used to create a read port and initialize its control structs. It will be connected to the FIFO channel pointed by *chan*. The *port_space* points to the memory management implementation. In our Microblaze platform, this is a function that statically allocates some space from the shared-memory in the *uncached* area. The address of the read port control struct is returned.

- **bool_t vfstream_room_available(vfstream_wport_t* wport)**
This API is used to check if there is any *room token* available in the FIFO connected to the write port pointed to by *wport*. It returns *true* on success.

- **bool_t vfstream_data_available(vfstream_rport_t* rport)**
This API is used to check if there is any *data token* available in FIFO connected to the read port pointed to by *rport*. It returns *true* on success.

- **vfstream_token_t* vfstream_acquire_room(vfstream_wport_t* wport)**
This API is used to acquire a room token of the FIFO channel connected to the write port pointed to by *wport*. It is *blocking* and on success returns the address of the acquired room token. There is also a *non-blocking* version of this API available. It will use the head and tail pointers and succeeds if the FIFO is not full.

- **void vfstream_release_data(vfstream_wport_t* wport, vfstream_token_t* token)**
This API is used to release the token (pointed to by *token*) of the FIFO connected to the write port (pointed to by *wport*) after writing it. It will take care of properly flushing out the data to main memory, invalidating the cache content corresponding

to the data token and updating the tail pointer of the FIFO. It is very important to understand that this API writes all the latest updated values of the token to the shared memory and invalidates the cache copies. This is done so that the reader can have access to the latest values of the token through shared memory. The invalidation is necessary because the next time that the writer processor wants to write to this token, it must be allocated in the cache again for new updates.

- **vfstream_token_t* vfstream_acquire_data(vfstream_rport_t* rport)**
  This API is used to acquire a data token of the FIFO channel connected to the read port pointed to by *rport*. It is *blocking* and on success returns the address of the acquired data token. There is also a *non-blocking* version of this API available. It will use the head and tail pointers and succeeds if the FIFO is not empty.

- **void vfstream_release_room(vfstream_rport_t* rport, vfstream_token_t* token)**
  This API is used to release the data token (pointed to by *token*) of the FIFO connected to the read port (pointed to by *rport*) after reading it. It will take care of properly invalidating the cache range corresponding to the data token and updating the head pointer of the FIFO. It is very important to understand that this API only invalidates the values of the read token in the cache so that the next time that this token is read again, its contents will be brought into the cache from the shared memory as the old value in the cache has been already invalidated, otherwise, we would have gotten the old values in the local cache of the reader processor.

- **void vfstream_skip_data(vfstream_rport_t* rport)**
  This API is used to skip a data token in the FIFO connected to the read port(pointed to by *rport*).

- **void vfstream_write_int8(vfstream_wport_t *wport, int8_t data)**
  This API is used to write one *int8* object into the FIFO channel via its write port pointed to by *wport*. This is a high level API that simply handles a standard data object. It acquires the room, puts the data and releases the data token. Similar APIs are implemented for writing all the other standard objects.

- **int8_t vfstream_read_int8(vfstream_rport_t* rport)**
  This API is used to read one *int8* object from the FIFO channel via its read port pointed to by *rport*. This is a high level API that simply handles a standard data object. It acquires the data, gets the data and releases the room token. Similar APIs are implemented for reading all the other standard objects.

- **vfstream_put_int8(vfstream_token_t* token, size_t offset, int8_t data)**
  This is a lower-level API or primitive used to put one signed character to the token pointed to by *token* and at the address *offset* from the token start address, This API can be best used when using wide tokens. The token must have been already acquired and should be released in the end. Several similar APIs are implemented to *put* singed and unsigned data objects to the room token.

- **int8_t data vfstream_get_int8(vfstream_token_t* token, size_t offset)**
  This is a lower-level API or ptomitive used to get one signed character from the token pointed to by *token* and at the address *offset* from the token start address, This API can be best used when using wide tokens. The token must have been already acquired and should be released in the end. Several similar APIs are implemented to *get* singed and unsigned data objects from the data token.

Based on the descriptions above, a basic data exchange starts with a *vfstream_acquire_room* to acquire a room token. Then the writer processing unit has the freedom to do any kind of processing on the token area and in the end, it has to release it using *vfstream_release_data* and the FIFO automatically goes to the next token. The reader processing unit will be waiting until its *blocking vfstream_acquire_data* call returns an acquired data token, then the reader can do any kind of read on the token area and after it is done, it has to release the data token using *vfstream_release_room*.

6.3 VFStream on a Dual-Microblaze platform

In order to have a working VFStream library on the Dual-Microblaze platform, multiple settings and configurations needed to be made. Reading the manuals, analyzing the concepts and running several test programs on the FPGA board was the methodology to prepare the needed basis on the Dual-Microblaze environment to properly accommodate the VFStream FIFO system. The most important points and assumptions to establish the coherent WFIFO system are as follow:

- A customized linker script was defined per processor. The linker scripts determine the memory sections that the linker uses to allocate memory for the data and instructions of the compiled programs. Since the EDK tool does not have the facility to compile and build a program for a multi-processor system at once, special attention was needed to properly take care of the data sharing and memory sections on the *shared* memory from each processor's point of view in such a way that in the end, when both processors start working having no idea about the existence of the other processor, we will have our desired parallel program and inter-core WFIFO-based communication properly established in the system. Later on, the whole settings and compilation can be done at once in VF tools.

- There should be always a master processor in the system that creates the channel, the read port and the write port in a shared *un-cached* area in the memory so that the other core(s) can also have access to it.

- The address to the read port of the channel is written to a specific shared location for the reader processor(s). This location can be a hardware FIFO or in the case of this thesis project, a specific address in the shared *un-cached* area.

- Using the address of the read port struct, the reader processor has the possibility to access the FIFO and communicate with the writer processor.

- The synchronization is done per token and using the FIFO pointers and control structs that are located in the *un-cached* area. This way, both processors are always seeing the valid values of this control information.
- The definition of the *cacheable* and *un-cacheable* memory areas is done in XPS when making the embedded system hardware. The CPUs are both configured to have the same cacheable and un-cacheable memory ranges on the global address space.
- The instructions and other variables that are not shared between the processors are stored in the main memory in sections that are only accessed by that processor. This is done by defining different memory sections in the linker script of each processor.
- All the requests to the shared un-cached memory area are processed on the PLB bus of the processor. As we have seen before, the requests are completed in order on the PLB bus and eventually on the input port of MPMC. So even if the requests on two ports of the MPMC get passed each other, the consistency of the whole system is still maitained, because the FIFO control structs are updated on the same PLB bus and after the last write to the token is done, they will be seen after the token is updated in the memory.
- A potential consistency issue might still exist in this system that needs more precise attention. After the token is released, Microblaze is flushing out the cache contents to the cached area of the external memory. As we had seen before, this access is done on the *XCL* interface. It is only after the execution of the cache flush APIs that the FIFO pointers are updated in the un-cached area. This second memory access is done on the PLB bus. When the cache flush API returns in the C program and the execution of the simple writes to the pointers get started, there is actually no firm guarantee that there is not any remaining transaction on the XCL bus. There could be still some data on the XCL bus and there is a minor chance that the FIFO pointers are updated in the memory and even seen by the other core, and while the FIFO contents are not yet updated in the memory via XCL, the reader processor starts reading at least some old bytes of the token. This is a very rare scenario that never showed up in any of the multiple tests and simulations were run for this project. Nevertheless, it had to be analyzed and reviewed. To make sure that this problem never happens in our system, the MPMC was configured in such a way that the port connected to the XCL interface of the writer processor had the highest priority in the arbitration and the XCL interface of the reader processor had the next lower priority. Below this level, other ports connected to the PLB buses of the processors can have any priority order. In case of using the AXI bus, this problem can be solved more efficiently. AXI-based Microblaze systems can be configured in such a way that all the accesses to both the cached and un-cached memory areas take place on the AXI bus. Since the bus can be configured to keep the order, there will be no issue like the one described above. We can even further use the *acknowledgment* feature of the read response channel to be absolutely sure of data arrival in the memory controller before updating the FIFO pointers.

Memory section	**Origin**	**Length**	**Description**
ilmb_dlmb	0x00000050	0x00001FB0	4 KB of on-chip RAM that is usually used to store data and instructions of the boot-up image.
shared_bram_contrl_0	0x8A208000	0x00001000	4 KB of shared on-chip memory, can be used to store any shared data between the cores.
DDR2_SDRAM	0x90000000	0x00FFFFF8	16 MB space in the main memory only used for this processor to store the instructions and the unshared variables.
MALLOC_SECTION	0x91FFFFF8	0x04000008	64 MB space in the main memory. The heap section is mapped to this memory section. It will be used to hold the FIFO channels.
DDR2_data	0x96000000	0x02000000	32 MB space in the main memory to be used for any cached shared data. This may contain any data outside the FIFO channel.
external_uncached	0x98000000	0x08000000	128 MB, the rest of the main memory is un-cacheable and is used for storing the FIFO control structures and read and write ports.

Table 6.1: Memory sections for the writer processor

- It is possible to put the synchronization in any other shared place between the processors. It must be noted that the synchronization flags or variables can not be cached themselves as there are the basis on which the data transfer and cache coherence are working. That is the reason that the FIFO control structs are located on the *un-cached* are of the main memory. If we store them in a shared BRAM that is accessible by both processors on their PLB buses, then there is a higher chance that the consistency is broken. Because the BRAM is on chip and it might be updated and read before the data is updated in the main memory.

Table 6.1 shows the memory sections defined in the linker script of the writer processor. The total size of the DDR II memory is 256 MB. The first half is configured as cacheable area and the second half is configured as un-cacheable area. The amount of un-cacheable area was significantly less than this, however, Microblaze only allows defining an un-cacheable memory area with a length of *power of two*. This can be changed if this limitation is removed. The definition of the memory sections of the reader processor is mostly the same except for the 16 MB section defined for the instructions and data that are *unshared*. This is definitely not the only possible linker script. The size of the memory sections can be changed based on the application or size of the available DDR II memory on the FPGA board. Figure 6.5 depicts the memory map of the shared main memory and its sections defined for the entire Dual-Microblaze system. We can now see how the whole solution conforms to the original requirements of cache coherence and memory consistency. The *write serialization* is guaranteed in the system using the *sequential* PLB bus, the PLB bus master and priority settings on the ports of MPMC. In the case of using the AXI bus, it is also guaranteed as it was explained before. The *atomicity requirement* is also guaranteed, because the design pattern and FIFO control take care of it. The consistency model used in the design pattern is *release consistency* and the only memory interactions that actually need to be atomic are at token *acquire* and *release* points. All the memory accesses before releasing the token are taking place on the XCL interface, either to the cache if there is hit, or to the external memory if there is miss. Even if the memory accesses were reordered, it would not affect the correct functionality as first, the other processor has not yet started to access them and second, the token address range will be all flushed out to the external memory in order at release time and before the synchronization point. Figure 6.6 shows the hardware architecture of the Dual-Microblaze embedded system that was designed to develop and test the final WFIFO solution on the Spartan3dsp Xilinx FPGA. Modules like RS232 UART or MDM (Microblaze debug module) are not shown in the figure for simplicity.

Figure 6.5: External shared memory sections

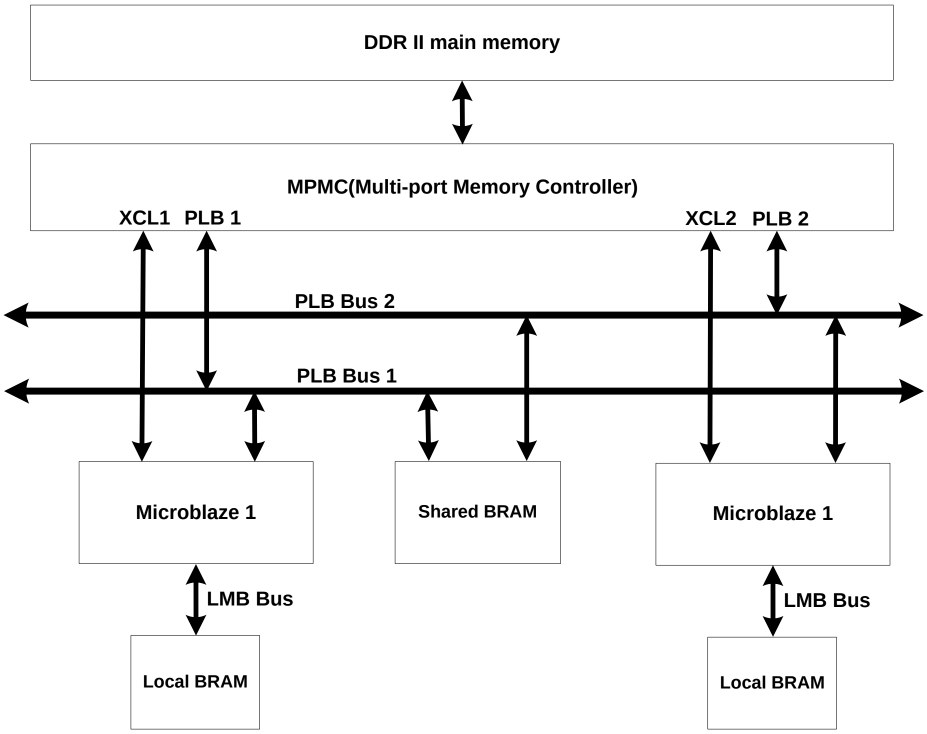

Figure 6.6: Dual-Microblaze hardware architecture

7 SW-HW platform

This chapter presents a SW-HW embedded system and the way that our WFIFO communication channel works on it. Section 7.1 introduces the main concepts and features of the SW-SW system, Section 7.2 illustrates the architecture of a SW-HW embedded system and how it hosts the WFIFO channel and finally Section 7.3 introduces the FIFO Interface Module(FIM) and explains how it is used by the accelerator to communicate with the other core via WFIFO channel.

7.1 Basic concepts

A unique feature of the proposed solution in this thesis is that it can be established between software and hardware too. The result is having a WFIFO communication between the accelerator and the CPU. In principle, most of the functionality described in chapter 6 can be accomplished on a SW-HW embedded system. To this end, the most important requirements are a *controllable* cache controller for the accelerator and a *special hardware interface* that can handle the FIFO administration in the shared memory. The *FIFO interface module (FIM)* was designed and implemented for this purpose. The accelerator can use this module to handle the WFIFO communication with the other party. However, there are some requirements for this whole system to work:

- The address of the read port and write port structs must be already known to the accelerator and passed to FIM. This normally takes place at the initial phase of setting up the whole embedded system.
- The FIM can not be used to create a FIFO channel and its read and write ports. There must be a *master processor* to do these at system set up.
- The accelerator does not need to have a prior knowledge about the kind and size of the tokens of the FIFO channel. It can receive them from FIM module.
- The FIM is a controller that already has the knowledge of how the FIFO control structs look like and by receiving their address from accelerator, it will take care of the FIFO control steps.
- One FIM module can be used to handle one read port and one write port of one or two FIFOs in the memory.
- It is still the job of the accelerator to execute the main procedure to make use of the FIFO. Steps like acquire and release are to be taken care of by the accelerator using the FIM.

- The accelerator can acquire multiple tokens. Releasing, will release the oldest acquired token.
- The accelerator has two main ports: a *VF ld/st* port for connecting to the cache controller and a *FIM port*.
- The accelerator must be able to distinguish between the cached and the un-cached memory areas of the main memory. All the accesses to the cached area including reading and writing from/to the tokens take place on the cached port via ld/st interface.
- Before the token access, the accelerator should acquire it using the FIM interface and after the token access is done the accelerator should take care of the cache flush/invalidation using the *cache control interface* of the cache memory. The last step will be releasing the token on the FIM interface so that the FIFO administration gets done properly. This is the main difference between a SW-SW system and a SW-HW system. Contrary to the SW-SW system, the *cache control* is actually separated from the *FIFO synchronization* phase.

7.2 SW-HW system architecture

Vector Fabrics tools are responsible to create the whole system and configure it. So, all the above steps and also the steps explained in section 6.3 are parts of the system set up that are integrated in and managed by Vector Fabrics tools, user will not be directly involved in these steps. Figure 7.1 shows the hardware architecture of the Accelerator-Microblaze system for WFIFO communication. Modules like RS232 UART or MDM (Microblaze debug module) are not shown in the figure.

It is worthwhile now to review how the accelerator's cache controller micro-architecture fits in our communication model. In the proposed WFIFO communication, the accelerator can read from one FIFO and write to another FIFO through two independent ports. The main feature of the cache controller was minimizing the possibility of conflict misses in the cache. Since the normal situation is that the memory area allocated to the FIFO elements being read is different from the memory area of the FIFO elements being written, the cache controller works perfectly decreasing the misses in the cache. In other words, the reader (CPU or accelerator) is normally reading from a FIFO and writing to another FIFO that will be read by another reader later on. There would have been a higher rate of conflict misses, if both the read FIFO and write FIFO memory areas had been mapped into the same cache memory and so probably the same cache block. This is in fact the *separate* read and write streams that was discussed in chapter 5. The cache controller consistency protocol is still capable to resolve any other situation in which the same data is shared between reads and writes, even thought it normally is not suposed to happen on the same token based on the *release consistency* model being used in the design pattern. In a system like the one shown in Figure 7.1, we can have a FIFO system with wide tokens even bigger than the size of the cache memory itself. This is another flexibility of this system. Because we know that in fact, having a bigger cache memory in a stream-based memory access does not considerably decrease the miss rate unless the size of the cache can be as large as the size of the whole FIFO which is not possible in many applications on a FPGA. The proposed cache controller has the ability to decrease the cache misses in a FIFO-based communication due to its separated read and write caches.

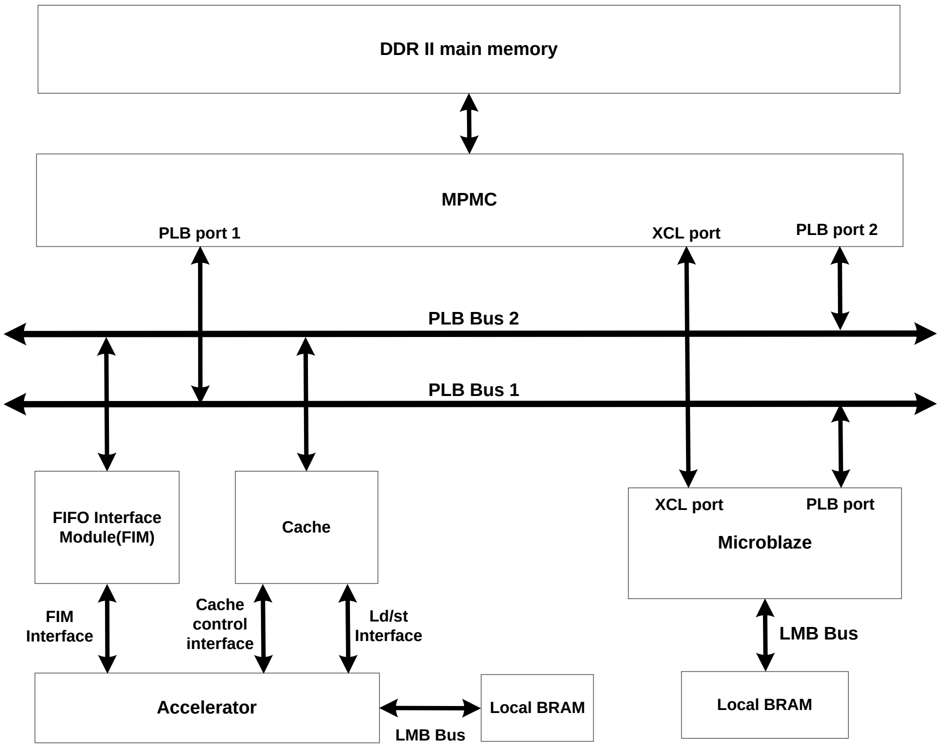

Figure 7.1: Hardware architecture of the Accelerator-Microblaze system

7.3 FIFO Interface Module

The FIM interface that was designed and implemented for this project is explained in this section. Basically, it is a simple interface in terms of timing and sequence of events in two main phases.

1. First, the accelerator should configure the FIM with the address of at least one read or write port.

2. After the address(es) are properly received by FIM, the actual work of FIFO management can get started.

Figure 7.2 depicts the timing diagram of the first phase of the FIM interface. The accelerator needs to use the *port_valid* twice to program the read port address and the write port address if it wants to use the FIM for both ports. *X* means *don't care*.

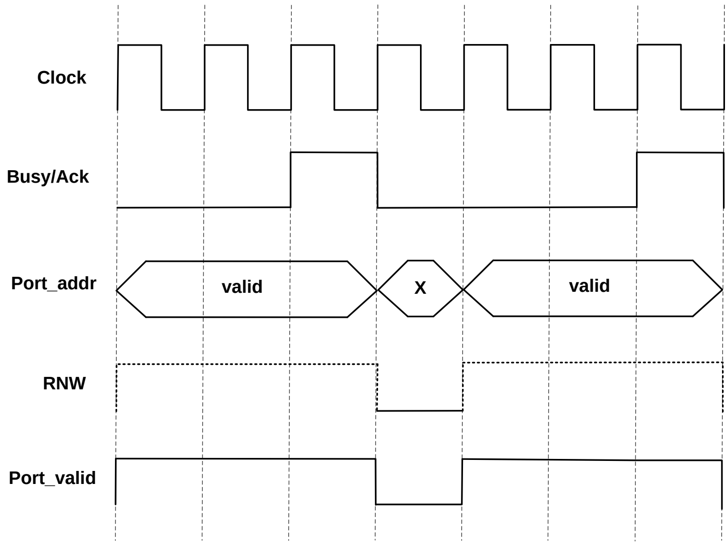

Figure 7.2: First phase of FIM interface

After each port address configuration, the FIM needs some time to do some initial checks. *port_addr*, *port_valid* and *RNW* must be kept valid until *busy_ack goes* high for one cycle indicating that the port address configuration is complete. Figure 7.3 shows the timing diagram of the second phase of the FIM interface. The latency from a request to its acknowledgment is not precisely determinable and depends on the traffic on the BUS and memory controller. The number of cycles shown in Figure 7.3 is a hypothetical latency used for simplicity in the timing diagram. In reality, it could be a lot more.

FIM can receive three types of requests:

1. **Acquire (acq):** means that accelerator wants to acquire a token from the FIFO.
2. **Release(rel):** means that accelerator wants to release a token.
3. **Availability(ava):** means that accelerator wants to check if there is a token available.

Regardless of the request type, *RNW* determines if it is for the read port or for the write port. *1* means read and *0* means write. After an acquire request, the *busy/ack* signal goes high as long as FIM is busy processing. After the result is ready *busy/ack* goes down meaning an acknowledgment to the request and the *address* and *length* of the acquired token will be made available on *token_addr* and *token_length* respectively. These two ports will be valid until another request is activated. The encoding of the

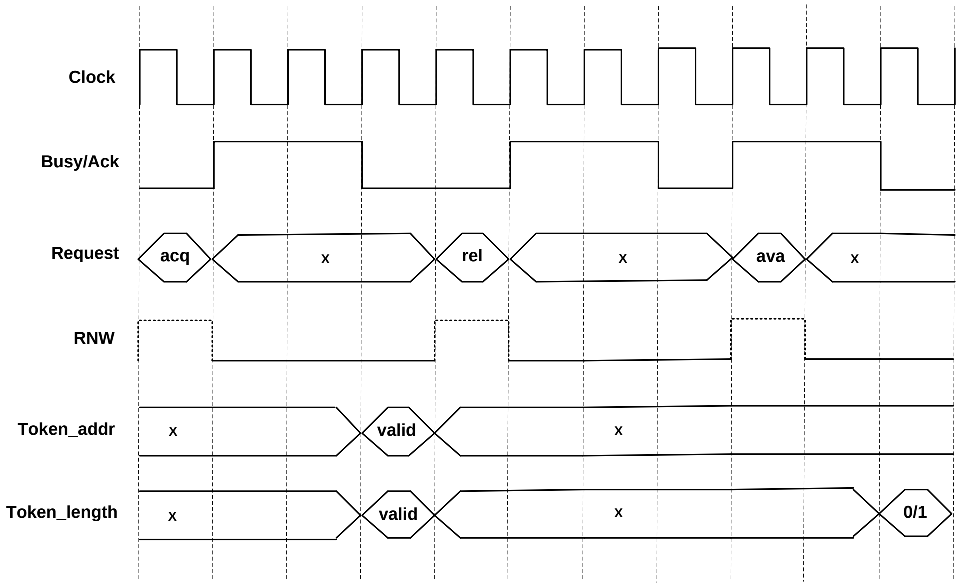

Figure 7.3: Second phase of FIM interface

Request	Meaning
00	No request - the default value
01	Acquire
10	Release
11	Availability

Table 7.1: Encoding of request on FIM interface

request is shown in Table 2.3. FIM must be first configured with the port addresses(es) based on the timing in Figure 7.2. If the write port address is not initially configured, then a *room acquire* can not be requested or the FIM might go busy for ever until it is *explicitely* stopped.

If the request is availability, the *lsb* bit of *token_length* can be used by accelerator to check the result. If this bit is 0 at the fall time of *busy_ack*, it means that no token is available. If this bit is 1 at the fall time of *busy_ack*, then there is at least one token available. FIM works in a blocking manner until it can acquire the token. If for some reason, it is desired to stop the procedure, the *stop* input needs to become 1 for enough number of cycles after the acquire is requested until *busy_ack* goes down. This will reset the procedure of the current running acquire process and the FIM will be ready to receive a new request again, this is not shown in Figure 7.3. The *request* and *RNW* must be kept valid at least until the clock edge that *busy_ack* goes high. At this clock edge the value of request and RNW are don't care to FIM. Using FIM, the accelerator can handle reading from one port and writing to another port with a different token length. The accelerator can use the three standard provided *requests* to *skip* a token by simply *acquiring* and then immediately *releasing* the token. It is always possible to change the address of any of the read and write ports as long as FIM is not busy. RTL simulations and on-board tests were used to verify the functionality of the interface. A simple accelerator could be used to read and write data through the channels just to verify the functionality of the communication model.

8 Tests and results

This chapter presents the test methodologies used in the project and reviews the results. Section 8.1 explains the procedure used to verify the functionality of the developed cache memory, Section 8.2 reviews the results of the synthesis of the cache memory for two different cache configuration on two different FPGAs and finally Section 8.3 explains the test methodology and example programs developed to test the overall functionality of the WFIFO channel on the FPGA board.

8.1 Functional verification of the cache memory

The test of the system had two main parts: testing the cache memory and testing the WFIFO channel in both SW-SW and SW-HW systems. In general, there were two different ways to verify the cache controller before actually using it in a system on the FPGA board:

8.1.1 System level test

A complete HDL simulation environment was already set up and tested for one sample Xilinx embedded system in ISE simulator. One way to simulate the cache controller could be integrating it into this system as a component and verify the functionality of the whole system with the cache controller as one of its sub-modules. This way was not done in this project for several reasons:

1. Simulating/debugging a HDL model of the whole system is a very slow and time consuming process.
2. It is very hard to create all the possible situations for the cache controller. Because we should be able to have different systems running different C codes to eventually result in what we want at the input of the cache controller.
3. We did not have any simulation model for the DDR II memory.

All the above reasons made it very hard and even unreasonable to verify the cache controller integrated in the whole embedded system. The second alternative was so faster and more efficient with better result:

8.1.2 Module level test

Since the input and output ld/st interface to/from the cache controller is simple in terms of timing and functionality, it was reasonable to have a Verilog test-bench environment to simulate and verify the cache controller. In this scheme, we can simply design and develop any possible test scenario for the cache controller and apply them to the module in the test-bench and then monitor the output. All the verification procedure is happening at module level and we can independently verify the cache controller's functionality in different situations. This way, we will be sure that it will integrate perfectly fine into the embedded system. To this end, five major steps were done:

1. Several test scenarios and input stimulus similar to VF accelerator requests were designed.
2. An abstract model for external memory system was developed in Verilog with two ld/st interface ports to be each used by read and write caches. This module is some behavioral always and initial blocks to understand the ld/st interface, interact with read and write caches and execute their requests on a two dimensional array used as external memory storage model.
3. An abstract behavioral Verilog block was designed and developed to play the role of the accelerator and issue load and store requests.
4. Input test vectors and load/store requests were read from a file and appended to the cache controller and its behavior was monitored to make sure about the functionality.
5. In case of any inproper behavior, waveforms were analyzed and checked to find the root of the bug.

Figure 8.1 illustrates the concept of the module level verification methodology employed for the cache controller.

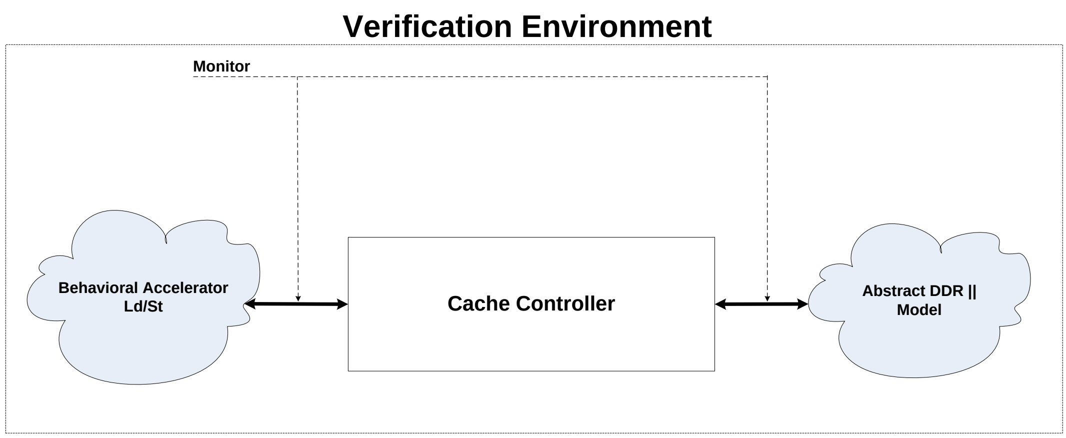

Figure 8.1: Module level verification of the cache controller

As an example of the cache controller functionality, the simulation output of one scenario is depicted in Figure 8.2. In this scenario, first a read is requested starting on address zero with the size of 8 bytes which has led into a miss (*iR_CACHE_HIT* is zero at rising edge of *oCOMMAND_VALID*). So main controller has issued a *oR_CACHE_REPLACE* as the address has also missed on the write cache. From this moment on, read cache controller has retrieved the whole block from external memory and droped its *busy* signal after it is done. And then, the main controller issued a *reload* to redo the cache read and this time it has led to a hit on the read cache (*iR_CACHE_HIT* is asserted) and the cache is able to consecutively serve eight read requests from VF accelerator as they all targeted to the same read data block. The read data from the cache is transferred to VF accelerator on the ld/st interface *RDAT* port (iRDAT) in eight clock cycles in a row with *oVFACK[1]* as read data valid.

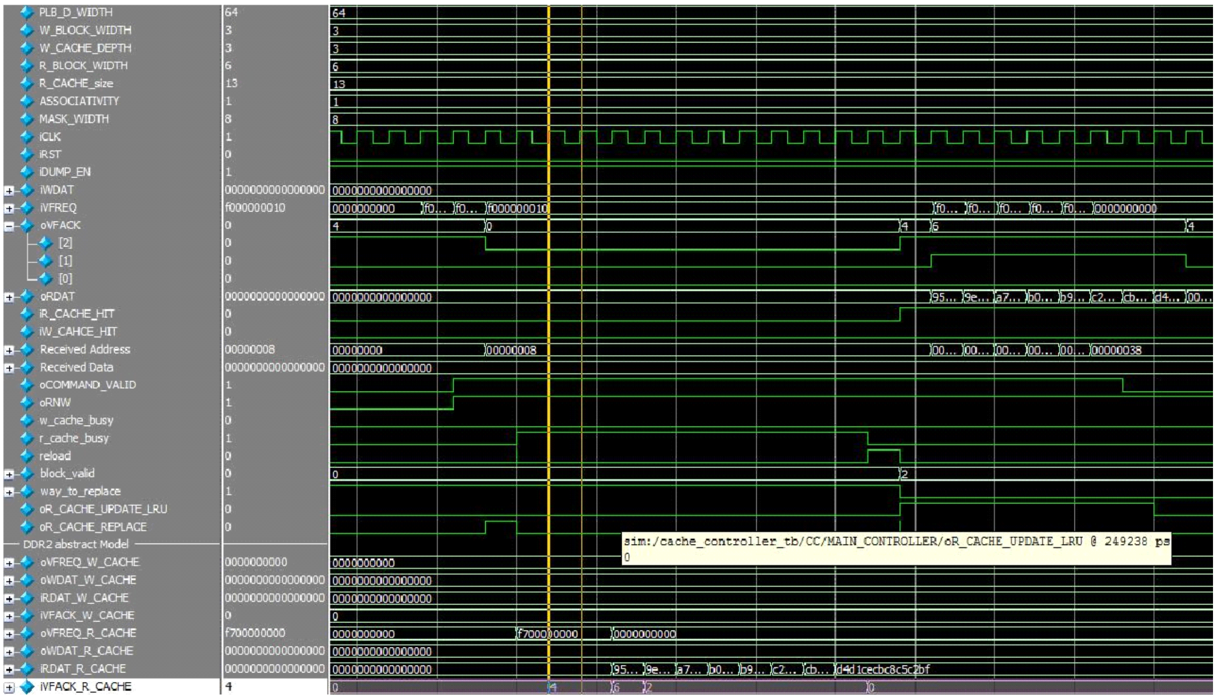

Figure 8.2: Waveform of a sample test scenario

As mentioned before, LRU algorithm is used to determine a cache block for replacement on a miss. The algorithm works as follow: based on the degree of associativity, a counter is considered for each way in the cache set. For example, if ASSOCIATIVITY is 2 then the counter is 2 bits to hold the history. At any hit on a cache way, its associated LRU counter will be reset to 0 and the LRU fields of all the other valid ways with values originally less than the *hit way* will be incremented by 1, the LRU field of other valid ways will remain unchanged and the invalid ways LRU fields will be 0. if a cache miss occurs and the cache set is not full (at least one invalid or empty way exists), the new retrieved cache block will be written into that invalid way, its LRU field will be reset to 0 and all the other valid ways LRU fields will be incremented by 1. If a miss happens and the cache set is full (all the ways of the set have a valid cache block), then there is definitely one way that has its LRU field bits all 1, this way will be selected (as the one who has been accessed least recently) for replacement. The new arrived data block will take its place, its LRU field will be reset to 0 and all the other valid ways LRU fields will be incremented by 1. In this process, among all the ways in one full cache set, always the least recently used one will be selected for replacement. The LRU field of invalid cache ways in the set are always 0.

8.2 Synthesis results

Different synthesis runs were done using Xilinx ISE to make sure that there are no critical synthesis warning/errors available. Post Synthesis simulation also showed proper functionality of the system. In this section two different cache organizations synthesis results are compared for two different FPGA chips: *Virtex-5(XC5VLX85-1FF1153C)* and *Spartan3dsp(xc3sd3400a-fg676c-4)*.

- **First cache organization**
  - Write cache with 8 entries and 8-bytes cache blocks
  - Read cache with 8KB total size, 64-bytes cache blocks and 2-way set associativity
- **Second cache organization**
  - write cache with 8 entries and 8-bytes cache blocks
  - Read cache with 8KB total size, 64-bytes cache blocks and 4-way set associativity

It can be seen in the synthesis results of Tables 8.1, 8.2, 8.3 and 8.4 that increasing the degree of associativity will make the cache access time longer that leads into a lower clock frequency. The reason is that a cache with a higher associativity degree has wider multiplexers and bigger combinational logic blocks in its critical path so the number of LUTs are also increased with a 4-way associative cache. Another conclusion of the synthesis results is that using *spartan3dsp* gives a slower cache memory comparing to *Virtex-5*. Newer FPGA families are even faster than Virtex-5 used in this project. The final hardware platforms of VF tools will be mostly from newer FPGA families anyhow which means even better timing results.

The synthesis results of the cache memory shows that on the Spartan3dsp FPGA, the cache can only be directly connected to the accelerator in VF embedded systems if the maximum achieved frequency is higher than the global clock speed of the embedded system. However, in case that the accelerator wants to operate with a higher clock frequency, it is essential to bridge the data between the accelerator and the cache memory. For this purpose, asynchronous FIFOs can be used which are not tested in this project. On the Virtex FPGA, the same problem may exist. The clock domain bridging becomes necessary if we want to avoid slowing down the accelerator to the speed of the cache memory. Another restriction of the clock speed is the maximum acceptable frequency of the Microblaze processor. Normally, this is the same clock oscillator that is used as the global clock connected to the accelerators as well. If the accelerator wants to have a different clock than the CPU's, more clock bridging issues will appear in the system that need more attention. Even thought the most efficient design techniques are used in the implementation of the cache memory like using separated combinational and sequential parts or designing the controllers with two different processes, etc, the use of huge multiplexers are the main bottleneck for the maximum achievable clock frequency. Further work and optimizations on the micro-architecture of the cache memory itself might lead to a higher maximum frequency.

Maximum Frequency:	117 MHz
Best achievable clock period:	8.54 ns
Inferred number of RAM blocks	4
Number of Slice Registers:	433 out of 51,840 1%
Number of Slice LUTs	1,037 out of 51,840 2%

Table 8.1: Synthesis results of Virtex-first cache organization

Maximum Frequency:	104 MHz
Best achievable clock period:	9.59 ns
Inferred number of RAM blocks	6
Number of Slice Registers:	432 out of 51,840 1%
Number of Slice LUTs	1,403 out of 51,840 2%

Table 8.2: Synthesis results of Virtex-second cache organization

Maximum Frequency:	51 MHz
Best achievable clock period:	19.53 ns
Inferred number of RAM blocks	4
Number of Slice Registers:	442 out of 47744 1%
Number of 4-input LUTs	1074 out of 51,840 2%

Table 8.3: Synthesis results of Spartan-first cache organization

8.3 Testing the WFIFO

The next step was testing the WFIFO communication on the development board. Using EDK tool-set, embedded systems like Figure 6.6 and 7.1 were built and configured on the board to run the tests. The master CPU was used to communicate with PC, read the input *test files* on the *compact flash* memory on the FPGA board and start the WFIFO

Maximum Frequency:	46 MHz
Best achievable clock period:	21.72ns
Inferred number of RAM blocks	6
Number of Slice Registers:	441 out of 47744 1%
Number of 4-input LUTs	1602 out of 47744 3%

Table 8.4: Synthesis results of Spartan-second cache organization

communication on the system. The final contents of the FIFO were also read by this CPU and stored on another *result file* on the compact flash. The flash memory card could be easily connected to the PC using a commercial USB card reader to see the final data received in the FIFOs. A lot of time was spent on learning how to establish a file access from Microblaze to the compact flash card on the board in C or how to integrate the accelerator as a peripheral in the embedded system and so on. It is not intended to discuss the details of these steps in this report. [68, 54, 55] and Xilinx help pages of EDK were the main references for these tasks. Figure 8.3 shows the test set-up. Xilinx tools are running on the PC and communicating with the development board via the standard *JTAG* port. The FPGA configuration is also done via the JTAG port as well as downloading the *executable software images* and monitoring the status. An extra RS-232 port of the board was used to check the desired hotspots during the operation of the embedded systems. This is a simple method used during debugging by inserting small *printf* instructions in the program to be used as assertions checks.

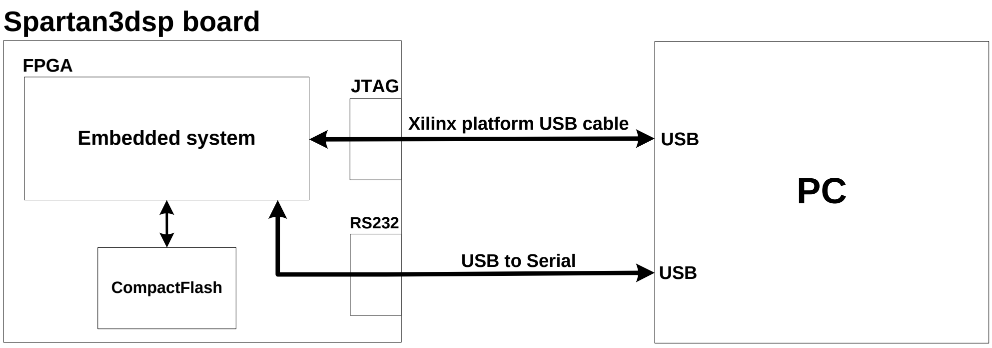

Figure 8.3: Test set-up of the WFIFO

In order to show the correct functionality of the WFIFO system, several different applications were developed and tested on the board. It must be noted that as long as the WFIFO channel is working properly, there is in fact no real difference between the applications being run as any application that is designed and working based on the design pattern is the same in terms of data communication.

- **Data transfer applications**

Multiple data transfer tests were done on the board. Different FIFO sizes and data traffics were tested. All the possible situations on the FIFO were also tested to make sure about the correct operation of the FIFO synchronization and handshake.

- **BTC(Block Turbo coding)**
  Another test was a Block Turbo Coding. It is a matrix-based coding scheme. Hamming codes are applied on the rows of the matrix and then on the columns of the matrix to increase the ability of detecting data corruption that might have happened during transmission. It was tested in such a way that one CPU was responsible for the rows and the other for the columns. The encoded matrices were stored on the flash memory to be compared with the results of the original codes on the desktop PC. The inter-core communication was based on the proposed WFIFO.

- **JPEG decoder**
  As a more complex application, a JPEG decoder program was also tested on the SW-SW system. A functional parallelism was applied to the original JPEG codes so that *Huffmann* decoding was done on the first processor and *IDCT* and *color conversion* were done on the second processor. This test was not done on the HW-HW system as it was not intended to spend time on developing an IDCT or color conversion in Verilog. VF tools will be responsible to do so after the WFIFO system is completely integrated in them.

Conclusion 9

This thesis has reviewd and identified the general issue of memory consistency and cache coherence in multi-core computer systems. Further, it has focused on FPGA-based platforms and narrowed down the range of the problem by targeting multi-core stream-based applications and to resolve their consistency and coherence issues. The project was carried out at Vector Fabrics B.V. Eindhoven, The Netherlands. VF multi-core embedded systems consists of general purpose CPUs and dedicated hardware units called accelerators. This chapter presents a short summary of the work described in this thesis, a brief review of the thesis contributions, the main achievements of the project, the potential issues of the developed solutions and finally some suggestions for future work.

9.1 Summary

Chapter 1 started with an introducion to the project, the company at which the project was carried out was briefly introduced, the main problem addressed in this thesis was described and the state of the art was reviewd and analyzed. This chapter also described the thesis contributions and the organization of the whole thesis report. Chapter 2 presented a detailed description of memory consistency and cache coherence issues in computer systems, several examples were used and discussed to show the principles and concepts of these issues, the difference between memory consistency and cache coherence was also illustrated. Chapter 3 presented the basics of the proposed design pattern, several communication models were studied and compared in this chapter and eventually the details of the proposed communication pattern were described. Chapter 4 presented and reviewd the basics of the Xilinx FPGA platform used for the software and hardware developments in the project, the main components of the FPGA-based embedded systems were studied and their impact on the memory consistency was analyzed and compared in this chapter. Chapter 5 described the architecture and features of the designed and developed cache memory for hardware accelerators, the implementation details of the cache memory and the way it can be used in FPGA-based embedded systems were also described in this chapter. Chapter 6 described the principles of a basic design pattern and presented the basics of software-based cache coherence control, the software streaming library and its main APIs were discussed, a dual-processor embedded system and the way that software APIs are used on it for coherent data communication were also discussed in this chapter. Chapter 7 presented the principles of FPGA-based embedded systems consisting of hardware accelerators and CPUs, it was described in this chapter how the WFIFO communication pattern works on such systems and how the cache memory of the accelerator fits in the streaming communication pattern. Eventually, Chapter 8 presented the verification methodologies and explained the test example embedded systems and applications that were developed and used to show the correct functionality of the developed software APIs and hardware cache memory, the synthesis results of the cache memory were also presented and described in this chapter.

9.2 Contributions and achievements

Contributions

In this thesis project, a consistent/coherent memory sub-system was designed and implemented for FPGA-based multi-core embedded systems running stream-based applications. Memory consistency and cache coherence issues were studied and analyzed in general and then we focused on Xilinx-based embedded systems of Vector Fabrics and reviewed the memory and cache related issues that they have. We considered stream-based applications as the target applications of the system and based on their features and also Vector Fabrics'needs, a suitable communication pattern was proposed that could be used as the main basis of the architecture of the target parallel programs generated by VF tools in terms of data exchange and communication. Several different communication models were reviewed and a flexible windowed-FIFO model was proposed as the final communication pattern. The software APIs of the windowed-FIFO were implemented and tested on both SW-SW and SW-HW systems on the FPGA platform. The next step was designing and developing a cache memory using behavioral synthesizable Verilog based on the proposed WFIFO communication pattern. The final result of the project was a flexible windowed-FIFO communication channel through external shared memory that can be used to coherently communicate and transfer data on a multi-core embedded system on a Xilinx FPGA. The data communication can take place between two Microblaze CPUs or a Microblaze CPU and a dedicated hardware accelerator. The cache coherence issue between the cache memory of the CPU/accelerator and the cache memory of the accelerator/CPU was resolved in the design of the Windowed-FIFO using high level cache control facilities. The consistency model used in the design pattern was *release consistency*. All the tests and simulations were done using Xilinx. ISE/EDK tool-set. A spartan3dsp 3400 development kit was used for on-board tests. During this project, around three months of the total time of the project was spent on learning and understanding all the hardware and software aspects of the platform and their exact influence and running several tests and experiments on the FPGA board. Several manuals and documents were studied to learn the tools. Three months were spent on software and hardware development and two months were spent for testing and documenting the project.

Achievements

Comparing to the existing solutions and based on the test results and simulations, the work in this thesis project has shown improvements in four main directions:

1. **Complexity**: Separated read and write caches and avoiding complex static or dynamic prefetching and locality optimization techniques decreases the level of

complexity of the proposed solution which in turn means easier integration and porting to other platforms. Furthermore, contrary to some other solutions like [6] the communication pattern is not fully implemented in hardware. This would additionally decrease the complexity.

2. **Scalability**: Having the WFIFO channel implemented in the global shared memory has made the solution easier to scale up. Adding and defining multiple FIFO channels and accessing them through software APIs that can be easily run on the general purpose CPU can be easily done with negligible hardware overhead.

3. **Flexibility**: The proposed WFIFO channel offers a wide range of functionality on the FPGA. By only modifying the APIs and the FIM module, it is possible to even extend the functionality in future without significant amount of work.

4. **Portability**: The proposed WFIFO channel can be easily ported to other platforms with minimum modifications as it is not too dependent on specific hardware or software components of the system.

9.3 Open issues

The most important drawback or issue in the system is the potential problem of the PLB-based systems. PLB bus protocol does not provide any feedback or acknowledgment service on the data reception and this might be a problem in synchronization points. This problem was solved by setting a higher priority for the memory port of the external memory controller connected to the writing processor. However, this is not a systematic solution as it is dependent on a specific memory controller module. As it was explained in Chapter 4, AXI-based systems which are the next generation of VF embedded systems can efficiently remove this potential problem.

9.4 Future work

The following extensions can be considered for future work:

- **Cache controller**
  It only handles aligned requests in this version. It can be later extended so that it can handle more complex inputs if needed. Another possible extension to the cache controller could be defining a direct connection between the cache controller and the cache memory of the processor to transfer the data between them. This was avoided in this project due to its extra complexity and area overhead. However, in a larger FPGA chip or in the next generation of Xilinx FPGA chips with integrated ARM cores, it might be a realistic option.

- **Windowed-FIFO porting**
  The implemented Windowed-FIFO solution can be ported to other supported platforms of Vector Fabrics like X86 platform. Based on the concepts discussed in this report and the knowledge acquired, it is not a difficult task as the critical points

of the model were identified and discussed. Minor modifications are needed for the porting.

- **PLB to AXI**
  Another extension could be migrating to AXI-based systems on Xilinx FPGAs. The AXI systems were analyzed and reviewed in this project and are now available on newer FPGAs of Xilinx. The next generation of Vector Fabrics tools will support AXI-based embedded systems too, so, this is a natural extension to this project.

- **Microblaze to ARM**
  In near future, Xilinx will have ARM hardcores in its FPGAs. Microblaze can be replaced with ARM. It is obvious that some modifications might be needed in the implementation of the software APIs. AXI system can be used with both Microblaze and ARM processor. So, an attractive version of the solution could be on such a system consisting of both processors and also hardware accelerators.

Bibliography

[1] J. F. Cantin, M. H. Lipasti, and J. E. Smith, “The complexity of verifying memory coherence and consistency,” *IEEE Trans. Parallel Distrib. Syst.*, pp. 663–671, July 2005.

[2] J. Lee, C. Park, and S. Ha, “Memory access pattern analysis and stream cache design for multimedia applications,” in *Proc. of the ASP-DAC 2003 Design Automation Conf*, p. 2227, 2003.

[3] S. V. Adve and K. Gharachorloo, “Shared memory consistency models:a tutorial,” September 1995.

[4] S. V. Adve, *Designing Memory Consistency Models For Shared-Memory Multiprocessors.* PhD thesis, University of Wisconsin-Madison, 1993.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] K. Huang, D. Grunert, and L. Thiele, “Windowed fifos for fpga-based multiprocessor systems.,” in *ASAP’07*, pp. 36–41, 2007.

[7] ARM, *AMBA protocol specifications and design tools*, 4.0 ed., 2007-2010.

[8] T. fu Chen and J. loup Baer, “A performance study of software and hardware data prefetching schemes,” in *In Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[9] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, “The efficacy of software prefetching and locality optimizations on future memory systems,” *J. Instruction-Level Parallelism*, 2004.

[10] J.-L. Baer and T.-F. Chen, “An effective on-chip preloading scheme to reduce data access penalty,” in *SC’91*, pp. 176–186, 1991.

[11] J.-L. Baer and T.-F. Chen, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Trans. Comput.*, pp. 609–623, May 1995.

[12] C. Zhang and S. A. McKee, “Hardware-only stream prefetching and dynamic access ordering,” in *Proceedings of the 14th international conference on Supercomputing*, ICS ’00, pp. 167–175, ACM, 2000.

[13] P. Grun, N. Dutt, and A. Nicolau, “Apex: access pattern based memory architecture exploration,” in *ISSS-01*, pp. 25–32, 2001.

[14] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, “Design and programming of embedded multiprocessors: an interface-centric approach,”

in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pp. 206–217, ACM, 2004.

[15] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf, "Yapi: Application modeling for signal processing systems," in *In Proc. 37th Design Automation Conference (DAC2000*, pp. 402–405, ACM Press, 2000.

[16] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for fpga-based soft multiprocessor systems," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis CODES+ISSS '05*, pp. 273 – 278, 2005.

[17] J. L. Hennessy and D. A. Patterson, *computer architecture, A Quantitative Approach.* Morgan Kaufmann, 2007.

[18] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design.* Morgan Kaufmann, 2009.

[19] D. J. Lilja, "Cache coherence in large-scale shared memory multiprocessors: Issues and comparisons," *ACM COMPUTING SURVEYS*, vol. 25, pp. 303–338, 1993.

[20] C.-C. Wu, D.-L. Pean, and C. Chen, "Look-ahead memory consistency model.," in *ICPADS'98*, pp. 504–510, 1998.

[21] L. Lamport, "How to make a correct multiprocess program execute correctly on a multiprocessor," *IEEE Trans. Comput.*, pp. 779–782, July 1997.

[22] G. Andrews, *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[23] Xilinx, *MicroBlaze Processor Reference Guide*, 12.0 ed., March 2011.

[24] ARM, *ARM Synchronization Primitives*, August 2009.

[25] MIPS Technologies, Inc., *MIPS R4000 Synchronization Primitives*, 1993.

[26] A. Birrell, J. Guttag, J. J. Horning, and R. Levin, "Synchronization primitives for a multiprocessor: A formal specification," 1987.

[27] M. M. Michael and M. L. Scott, "Implementation of atomic primitives on distributed shared memory multiprocessors," in *Proc. First Symp. on High Performance Computer Architecture*, pp. 222–231, 1995.

[28] P. E. Mckenney, "Memory barriers: a hardware view for software hackers," 2009.

[29] ARM, *ARM Architecture Reference Manual*, 1996-2010.

[30] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, 1990.

[31] A. Krishnamurthy and K. A. Yelick, "Optimizing parallel spmd programs," in *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pp. 331–345, Springer-Verlag, 1995.

[32] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, pp. 282–312, April 1988.

[33] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors," in *Proceedings of the 14th annual international symposium on Computer architecture*, ISCA '87, (New York, NY, USA), pp. 234–243, ACM, 1987.

[34] Y. Afek, G. Brown, and M. Merritt, "Lazy caching," *ACM Trans. Program. Lang. Syst.*, pp. 182–205, January 1993.

[35] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," in *In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 245–257, 1991.

[36] M. Brorsson and P. Stenstrom, "Modelling accesses to migratory and producer-consumer characterised data in a shared memory multiprocessor," in *In Proceedings of Sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 612–619, 1994.

[37] M. R. Marty, *Cache coherence techniques for multicore processors.* PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 2008. AAI3314233.

[38] J. Archibald and J. loup Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, vol. 4, pp. 273–298, 1986.

[39] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, pp. 1112–1118, 1978.

[40] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *In Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 280–289, 1988.

[41] M. A. Heinrich, *The performance and scalability of distributed shared-memory cache coherence protocols.* PhD thesis, Stanford University, 1999.

[42] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. dietrich Weber, "Comparative evaluation of latency reducing and tolerating techniques," in *In Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 254–263, 1991.

[43] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon, "Comparison of hardware and software cache coherence schemes," 1991.

[44] P. Stenström, "A survey of cache coherence schemes for multiprocessors," *Computer*, pp. 12–24, June 1990.

[45] H. Cheong and A. V. Veidenbaum, "A cache coherence scheme with fast selective invalidation," in *ISCA'88*, pp. 299–307, 1988.

[46] I. Tartalja and V. Milutinovi, "Classifying software-based cache coherence solutions," *IEEE Softw.*, pp. 90–101, May 1997.

[47] V. F. B.V., "Vf analyst data sheet," 2010.

[48] V. F. B.V., "Vf embedded brochure," 2011.

[49] M. P. I. Forum, "Mpi: A message-passing interface standard," tech. rep., National Science Foundation Science and Technology Center Cooperative, November 2003.

[50] Xilinx, *LogiCORE IP Fast Simplex Link (FSL) V20 Bus*, 2.11c ed., April 2010.

[51] Xilinx, *LogiCORE IP AXI-Stream FIFO*, 1.0 ed., September 2010.

[52] P. Huerta, J. Castillo, J. I. Mártinez, and V. López, "Multi microblaze system for parallel computing," in *Proceedings of the 9th International Conference on Circuits*, pp. 31:1–31:6, World Scientific and Engineering Academy and Society (WSEAS), 2005.

[53] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, pp. 471–475, North Holland, Amsterdam, August 1974.

[54] Xilinx, *Embedded System Tools Reference Manual*, 13.1 ed., March 2011.

[55] Xilinx, *EDK Concepts, Tools, and Techniques*, 13.1 ed., March 2011.

[56] Xilinx, *Xilinx ISE manual*, 13.1 ed., March 2011.

[57] Xilinx, *XtremeDSP Development Platform: Spartan-3A DSP 3400A Edition User Guide*, 2.2 ed., Novemeber 2008.

[58] Xilinx, *PowerPC Processor Reference Guide*, 1.3 ed., January 2010.

[59] Xilinx, *IP Processor Block RAM (BRAM) Block Data Sheet*, 2.3 ed., March 2011.

[60] Xilinx, *LogiCORE IP Multi-Port Memory Controller (MPMC) (v6.03.a)*, 6.03a ed., March 2011.

[61] Xilinx, *Local Memory Bus (LMB) v1.0 (v1.00a)*, 1.0 ed., April 2005.

[62] Xilinx, *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a)*, 1.05a ed., September 2010.

[63] IBM, *Processor Local Bus (128-bit)*, 4.6 ed., May 2007.

[64] Xilinx, *AXI Reference Guide*, 13.1 ed., March 2011.

[65] E. T. Ososanya and . J. Matthews, D., "Design and performance evaluation of an expendable modular directory scheme for maintaining cache coherency in multiprocessor systems," in *Proceedings of the 27th Southeastern Symposium on System Theory (SSST'95)*, SSST '95, IEEE Computer Society, 1995.

[66] B. Jacob, "Cache design for embedded real-time systems," in *Proceedings of the Embedded Systems Conference, Summer*, 1999.

[67] J. Handy, *Cache Memory Book.* Academic Press, 1998.

[68] Xilinx, *OS and Libraries Document Collection*, 13.1 ed., March 2011.

[69] V. Fabrics, "Vf stream library." http://www.Vector Fabrics.com.

Curriculum Vitae

I was born in Tehran, Iran in 1979. After my graduation from high school I successfully passed the global entrance exmination of Iranian Universities and was ranked 304 among more than a milion of participants all around the country, I started my bachelor in computer harware engineering in Iran University of Science and Technology (IUST) and graduated in 2002 when I started working as a full time digital design engineer at rastafann technologies. I was responsible for design and implementation of digital blocks on FPGAs using VHDL and Verilog on Xilixn and Altera platforms. The company was active in the field of telecommunication products and I was involved in several projects. I joind Kavoshcom R&D group after one and a half year at rastafann and started as an ASIC/FPGA design engineer. We taped out two RFID ICs at kavoshcom and the results were successful. After two and half years at Kavoshcom I left Iran for Malaysia. I joined elecomp technologies and worked on high speed digital protocol analyzers and excersizers. The company was working with Lecroy and the challenge and technical drive of the projects were high enough that put me under pressure and gave me the opportunity to gain more knowledge and insight on HW/SW co-design and verification on FPGAs. After a year at elecomp technologies, It was finally time to pursue my original goal which was going to Europe. I chose Netherlands over my other options for several reasons and started at Altran technologies as a Microelectronic consultant. After almost 6 Months at Altran, I found the chance to go back to university again and continue my education. I started my master at TU Delft in embedded systems. I chose this major as my goal was to expand my low-level software skills and combine them with my hardware experience to become a more versatile and complete digtial design engineer.

Vahid Roostaie
Email: vahid.roostaie@gmail.com
Cell:+31-643176394