.

TECHNICAL UNIVERSITY OF CRETE

DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING
ELECTRONICS AND COMPUTER ARCHITECTURE DIVISION

Senior Thesis

"Parallel Computing System Implemented on a Field Programmable Gate Array"

Georgios-Grigorios G. Mplemenos

Committee:

Assistant Professor Ioannis Papaefstathiou (supervisor)
Professor Apostolos Dollas
Associate Professor Dionisios Pnevmatikatos

Chania 2007

Prologue

This senior thesis was elaborated at the Microprocessor and Hardware Laboratory (MHL), for the acquiring of a degree from the department of Electronic and Computer Engineering (ECE) of Technical University of Crete (TUC), under the supervision of the Associate Professor Mr. Ioannis Papaefstathiou, during the academic year 2006-2007.

For the elaboration and completion of this senior thesis, I would firstly like to thank my supervisor professor Mr. Ioannis Papaefstathiou, not only for his contribution and support for the completion of this thesis, but also because he gave me the opportunity to gain knowledge and experience of his research area.

I would also like to thank the Professors Mr. Apostolos Dollas and Mr. Dionisios Pnevmatikatos for their valuable contribution in this thesis, and Mr. Markos Kimionis, EEDIP member and MHL manager, for his support in technical matters.

Furthermore, I would like to acknowledge the support of PhD students Mr. Euripides Sotiriades, Mr. Kyprianos Papademitriou and Mr. Dimitrios Meintanis for their valuable advice and ideas in difficult parts of this thesis.

Also, I would like to thank all my friends and of course all my collaborators, either graduate or undergraduate students, of MHL, for their support and advice.

Last but not least, I would like to thank my family for their constant support and solidarity, not only during my thesis but also during the entire period as undergraduate student, and also because they provided me with the means to fulfill my studies.

Devoted to my family

Summary

Recent advantages in Field Programmable Gate Array (FPGA) technology have made it possible to create soft-core Multiprocessor Embedded Systems (MES), which emphasize in reducing the amount of the dedicated hardware needed to design a system, while increasing the flexibility of the design with the use of software.

A soft-core multiprocessor system is a network of programmable processors crafted out of processing elements, logic blocks and memories on an FPGA. They allow the user to customize the number of programmable processors, interconnect schemes, memory layout and peripheral support to meet the application needs.

In this thesis, we implemented and evaluated a Multiprocessor Embedded System (MES) that can be used for different applications, built from Xilinx Microblaze soft-processors. In order to demonstrate the design flow and the interconnections between the processors and the different peripherals, we build up a MES on a Xilinx Virtex-II Pro FPGA that solves the BLAST-n algorithm. All processors at the same time compare the same query with a different part of the database and report the results.

The main goal is to create a system that utilizes the advantages of FPGAs and soft-core processors, and with a proper parallelism of data, to increase the runtime performance and the throughput of BLAST software compared to the software runs on a common PC.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Description of Soft-Core Multiprocessor Embedded Systems

Parallel computing systems have long been used as means of accelerating program execution in the context of increasing problem size (i.e. data, computational complexity or both). These systems have been traditionally implemented either on high-end multiprocessor computing systems available from companies like IBM, HP and Sun, or on Linux clusters built from commercial of-the-self (CTOS) computers (i.e. Sony Playstation) Both implementation styles for parallel computers have until recently been limited to a single processor per silicon die. New trends in integrated circuit fabrication allow for multiple processing cores to be implemented on the same die. This in turn allows for the characteristics of parallel computing systems to be ported into the embedded computing space.

Multiprocessor embedded systems (MES) provide designers a greater flexibility in systems specification and shift more of the development complexity from hardware to software. This approach does not rule out the use of dedicated hardware acceleration units, which are common in single processor systems. The key characteristic of an MES is the emphasis in reducing the amount of dedicated hardware needed to satisfy design constraints, while increasing the applicability of the design through the software. Efficient use of data and instruction parallelism allow multiprocessor systems to avoid the impact of the underlying hardware and increase the system throughput. The key to this improvement is the correct partitioning and decomposition of the application in terms of both software and hardware.

A soft-core multiprocessor system is a network of programmable processors crafted out of processing elements, logic blocks and memories on a Field Programmable Gate Array (FPGA). They allow the user to customize the number of programmable processors, interconnect schemes, memory layout and peripheral support to meet the application needs. Deploying an application on the FPGA is tantamount to writing software for this multiprocessor system. Results in [8] show that soft multiprocessor systems are viable alternatives for high performance applications. They avoid risks due to high silicon development costs and design turnaround times, while providing a software abstraction to enable a

quick implementation on the FPGA. They also open FPGAs to the larger world of software designers.

Modern FPGAs provide the processing capacity to build a variety of micro-architectural configurations. Today, we can build multiprocessors composed of 10-50 processors (and growing with Moore's law), complex memory hierarchies, heterogeneous interconnection schemes and custom co-processors for performance critical operations. Future projections forecast that embedded systems will be soon composed of over 100 processors on a single chip to guarantee acceptable performance [9]. However, the diversity in the architectural design space makes the task of determining an efficient multiprocessor configuration tuned for a target application challenging. Currently, the designer must manually explore the large and complex design space of micro-architecture to achieve the full performance potential of FPGA multiprocessors.

Xilinx provides tools and libraries for developing soft multiprocessor system in the Virtex family of FPGAs [16]. This environment gives user the opportunity to integrate IBM PowerPC 405 cores, Microblaze soft processors, peripherals and customized hardware onto an FPGA chip.

To sum up, embedded systems are no longer used as simple controllers. Embedded systems need more computational power to satisfy today's applications' needs like audio/video encoding/decoding, image processing, bioinformatics applications, etc. and MES are an option to deal with this increasing computational needs.[10], [11]

In the Microprocessor and Hardware Lab (MHL) together with assistant Professor Ioannis Papaefstathiou, we use state-of-art FPGA technology in order to develop a general purpose soft-core multiprocessor platform. The purpose of this thesis is the implementation and evaluation of a MES that can be used for different applications, built from Xilinx Microblaze soft-processors. In order to demonstrate the design flow and the interconnections between the processors and the different peripherals, we build up a MES on a Xilinx Virtex-II Pro FPGA that solves the BLAST-n algorithm [1]. All processors at the same time compare the same query with a different part of the database and report the results. The main goal is to create a system that utilizes the advantages of FPGAs and soft-core processors,

and with a proper parallelism of data, to increase the runtime performance and the throughput of BLAST software compared to the software runs on a common PC.

1.2 Contribution of the current thesis

The contribution of this thesis is the following:

- Implementation of a soft-core multiprocessor platform with its own common shared external SRAM, with the use of Xilinx tools (ISE 7.1, EDK 7.1) on a Xilinx Virtex-II Pro FPGA.
- Software implementation of BLAST-n machine for better understanding of the algorithm. This implementation is also used as the software of the multiprocessor platform and, of course, serves as the verification and the profiling tool of the multiprocessor platform.
- Embedment of the BLAST-n machine on the multiprocessor platform with proper parallelism of data, verification and behavioral simulation with ModelSim 6.0.
- Evaluation and comparison of the BLAST-n platform results with those from a common PC.

1.3 Thesis Overview

In chapter 2, some of the approaches that have been introduced until today in the implementation of soft-core multiprocessor platforms, are briefly described. In particular, in this chapter those platforms that use the Xilinx Microblaze processor and of course the different approaches for the solution of this problem (number of processors, interconnections between processors and between processors and peripherals) are described. Finally in this chapter the results of each platform that described above are presented, in association with the specific applications that they implement.

Chapter 3 contains a brief overview of the different parts that are used to develop the soft-core multiprocessor platform of this project. In particular, this chapter

describes the structure and functionality of the Microblaze, the model of the memory that is used, the structure of the custom peripherals that were created to support specific operations of the platform and finally the functionality of the necessary Intellectual Properties (IPs) that are available from Xilinx libraries (buses, block RAMS, bridges), so as the implementation to become feasible. Also the contents of chapter 3 deal with the architecture of the multiprocessor platform in details. In this chapter, the way that the platform is implemented and the reason for this specific architecture is described. At this stage, we use simple software so as an initial simulation and verification of the functionality of the platform to be made. At a later stage, a specific application will be developed so as the extraction of useful results to be possible. Finally, in this chapter, the different architectures, that described previously, are compared to this one and the differences are discussed.

Chapter 4 contains a brief description for the verification of the MPLEM system. To be more specific, in this chapter all the steps that were made in order to verify the correct functionality of our multiprocessor platform are described. A verification process is followed not only for every peripheral that is connected to the system, but also for the whole platform. All the verification process was made with the Modelsim 6.0a with behavioral simulation.

Chapter 5 describes a brief overview of the BLAST algorithm and its use in molecular biology. Besides that, the contents of chapter 5 deal with BLAST software. After distinguishing the different software programs according to the form of the processed data, the BLAST-n program of the popular NCBI software is presented. Also, we describe the implementation of a software system we developed from scratch, running the main BLAST-n machine and the verification we made by comparing the output of this machine to the output of the NCBI tool. Last but not least, it is described the reason for choosing this particular algorithm to be applied on a multiprocessor platform and the way this software is embedded in it.

Chapter 6 includes all the results and the performance comparisons between the multiprocessor platform and the software machine that is developed, while chapter 7 proposes future work to further evolve this system in order to increase its performance and its reliability.

2. Soft-core Multiprocessor Platforms

The approaches of soft-core multiprocessor systems that have been introduced until today are quite a few. In this chapter some of these approaches will be described. Each one of them uses a different architecture and a different test application. On the other hand, the soft-processor that is used in all these approaches is the Xilinx Microblaze.

2.1 Soft Multiprocessor System for JPEG Compression

In [12] the design flows of an FPGA-based multiprocessor system for high performance multimedia application is demonstrated and are explored different on-chip interconnects for multiprocessor system. In this approach, a JPEG encoder is constructed on a Xilinx Virtex-II Pro FPGA. This design can compress a BMP into JPG image in high speed. Also in this approach different interconnections between processors, including bus, dual-port memory, FIFO and DMA controller are implemented so as the trade-off between them to be explored.

A typical single-core Microblaze system is as follows and a JPEG encoder has been mapped onto it. A cache can be put between processor and external SDRAM. It's not shown on the following diagram because cache is considered as part of the Microblaze processor component in EDK.

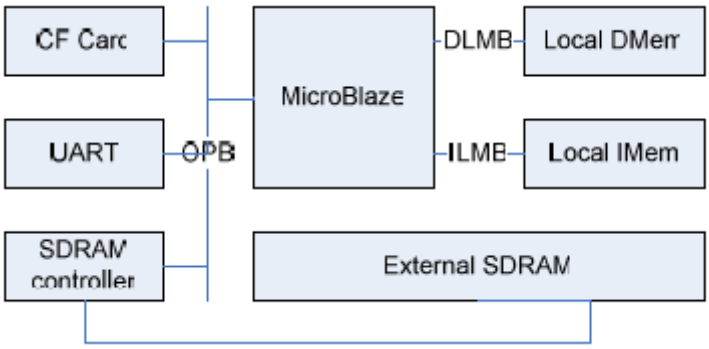

Figure 2.1 Typical single–core Microblaze systems

2.1.1 Soft Multiprocessor Systems on Xilinx FPGA

The JPEG encoder is implemented on a Xilinx Virtex-II Pro 2VP30 FPGA with Xilinx Embedded Development Kit (EDK). For the entire system, including I/O, a

Xilinx XUP2Pro board, with Compact Flash (CF) card interface and external memory is used.

The soft multiprocessor system consists of four Microblaze processors, BRAMs, peripherals, external memory and interconnections as shown below. Besides FIFO interconnection, three other types of interconnections, OP bus, dual port memory and DMA, are evaluated later.

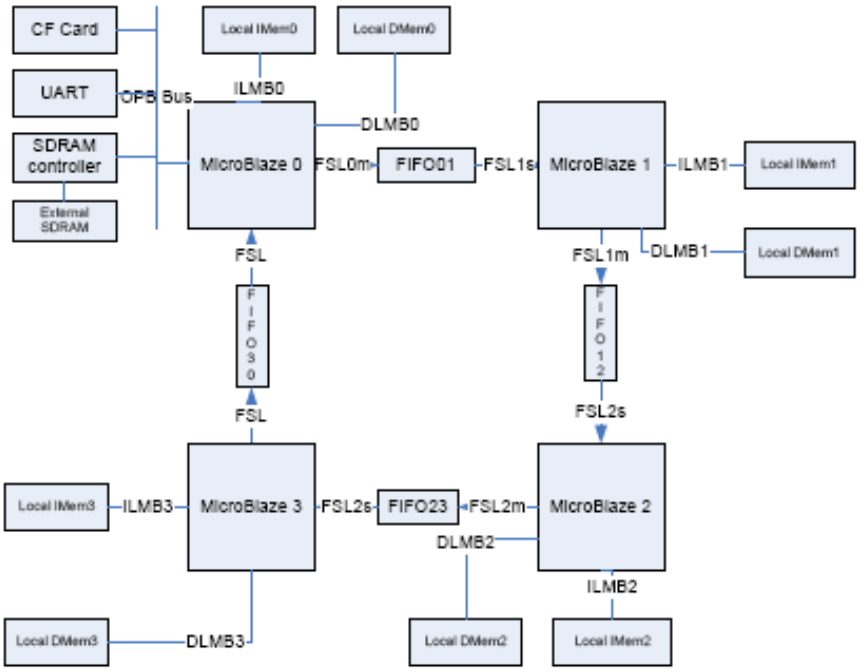

Figure 2.2 A soft multiprocessor system

Microblaze 0 in the system is used for I/O, external memory access and debugging while the rest three processors do the computation. External DDR memory is used as image buffer because CF card access is slow. The system runs at 100MHz, due to the limitation of OPB bus.

2.1.2 JPEG Encoder Application

Regarding the application, a baseline JPEG encoder with color conversion and sub sampling is implemented on the multiprocessor platform. Except for the I/O and bootstrap, the JPEG encoder algorithm includes BMP and JPG header parsing, color conversion, DCT, zigzag scan, quantization and variable-length encoding. Following is the data flow of JPEG encoder.

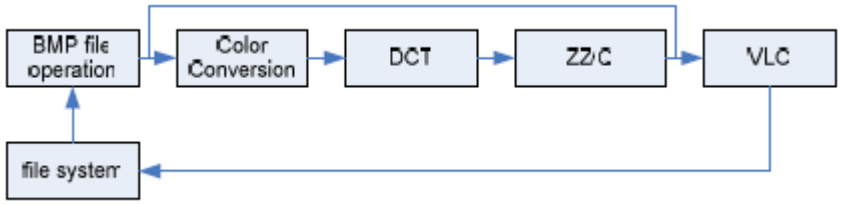

Figure 2.3 JPEG encoder data flow

These tasks are partitioned onto four processors, for instance the FIFO interconnection scheme is as follows.

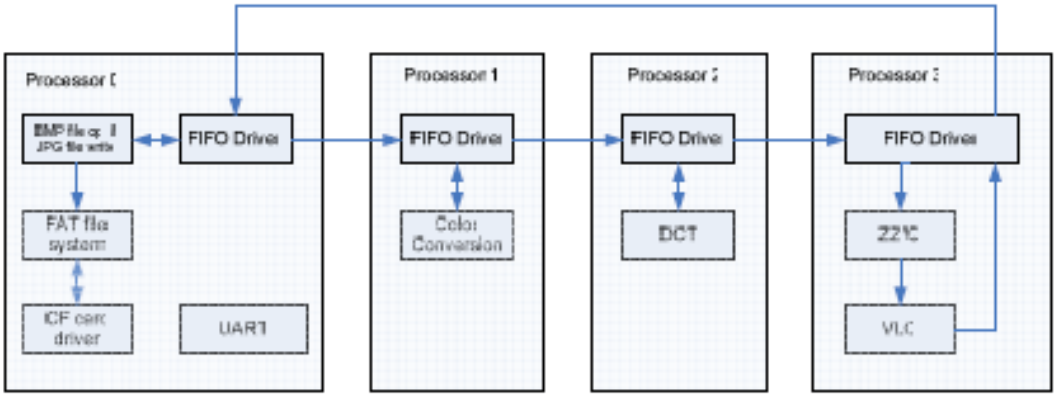

Figure 2.4 JPEG task partitioning

The table is a detailed description including input and output of every processor.

P#	Function	Input	Output
0	dedicated I/O	JPG bitstream	BMP macro block (RGB) + image size + end of image indication
1	color conversion	BMP macro block (RGB)	BMP macro block (YUV) + image size + end of image indication
2	DCT	BMP macro block (YUV)	f-domain macro block+ image size + end of image indication
3	ZZ/Q + VLC	f-domain macro block	JPG bitstream

Table 2.1 Detailed task partitioning with input and output

The advantages of this partitioning are:

- Low memory requirement. Actually Microblaze 1, 2 and 3 needs to store only a few macro blocks which is 16x16 pixels each.
- Easy to improve performance by dedicated hardware accelerators because every processor is dedicated to a well-defined task.

2.1.3 Streaming Programming Model

With regards to the programming model, this is modified from a shared memory model to a streaming model. All tasks share the same address space and communicate via shared memory. However, in order to maximize the throughput, this four processors need to run in parallel and therefore a streaming model is better. The inter-process communication is adapted to a message-oriented model as well. Compared to shared memory, explicit message passing is easier to deploy, monitor and debug.

2.1.4 Interconnection Exploration: Bus Interconnection

Besides that, in [12] different types of interconnections for evaluation and comparison purposes are also presented four. The first type of interconnection is the “Bus interconnection”. This is an easy way to connect four processors via a bus. Xilinx provides OPB bus with arbitration. All processors, external memory and peripherals can just be connected to the OPB bus and it works. The hardware architecture of four processor system connected by bus is as follows.

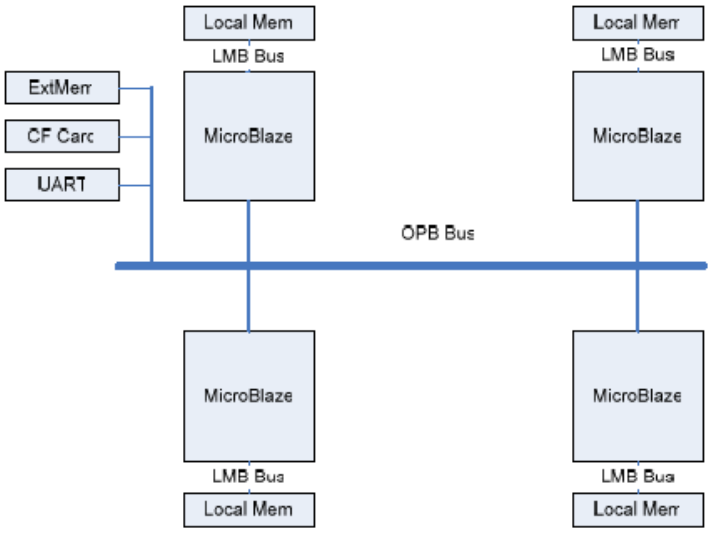

Figure 2.5 Hardware architecture of four-processor system connected by bus

The bus is shared by four processors, peripherals and external memory. Therefore it's a bottleneck of the system. It's very difficult for four processors to archive full-parallel running with bus interconnection. It may be used for a starting point for multiprocessor platform design.

2.1.5 Interconnection Exploration: Dual Port Memory Interconnection

The second type of interconnection is the "Dual Port Memory Interconnection". Because all on-chip memory blocks on Xilinx FPGA are dual port memories, it's easy and efficient to employ dual port memory as communication channel between processors. The hardware and software architecture of four processor system connected by dual port memory blocks is as follows.

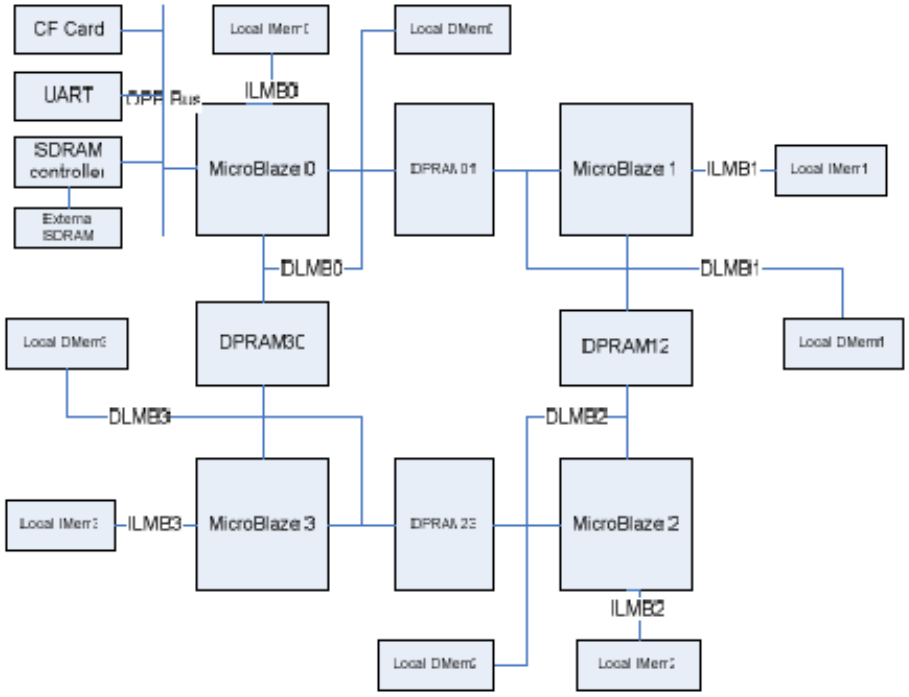

Figure 2.6 Hardware architecture of four-processor system connected by dual port memory blocks

Similar to the general architecture, every processor has two LMB buses, I-LMB bus and D-LMB bus. However, the data LMB bus here is connected to two dual port memory blocks in addition to data memory block. Each port of every dual memory block is connected to the data LMB bus of two different processors and .

therefore constitutes a communication channel. Every dual port memory is assigned to its dedicated address space as well. Processors can access dual port memory via normal memory access. The access is one-cycle-access and

predictable because it's connected to LMB bus. There is no inter-process synchronization directly supported by dual port memory interconnection. It needs to be implemented through additional code or additional hardware.

2.1.6 Interconnection Exploration: FIFO Interconnection

Another often-used communication channel in a multiprocessor system is a "FIFO interconnection". Compared to the last implementation, dual port memory blocks are replaced by FIFOs. The hardware architecture of four-processor system is as follows.

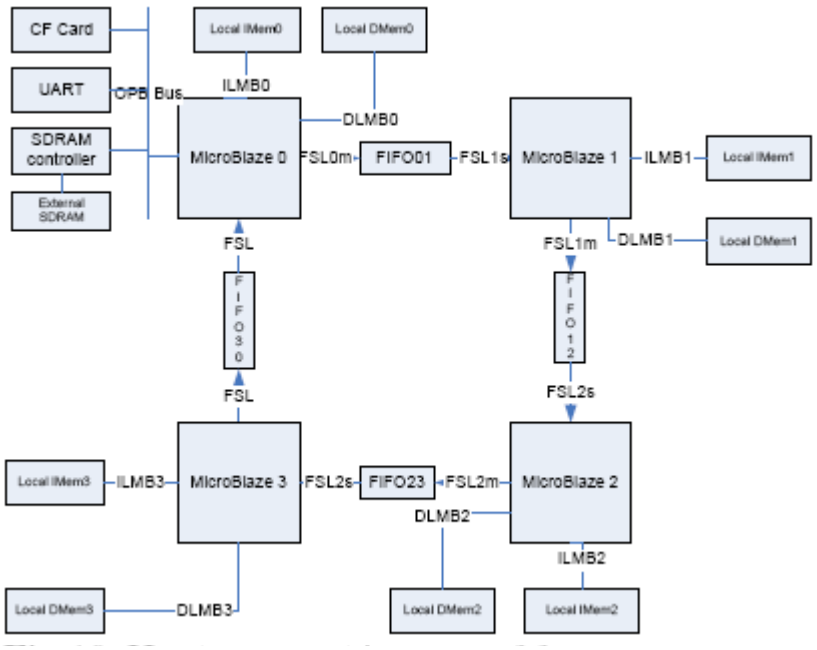

Figure 2.7 Hardware architecture of four-processor system connected by FIFO

FIFO is connected to processor via FSL bus. So there are two more buses for every processor, FSL master and FSL slave. FSL has built-in FIFO capacity. It's an ideal solution for FIFO implementation. Furthermore, there is hardware synchronization mechanism built in which is easy and efficient.

2.1.7 Interconnection Exploration: DMA Interconnection

The last interconnection that is presented in [12] is the "DMA interconnection". DMA has its advantage in multiprocessor systems and is getting more and more deployed. Compared to dual port memory and FIFO, it's an active component. So it can move data in parallel to processors without any attention from processor.

The hardware architecture of four Microblaze system connected by FIFO and DMA is as follows.

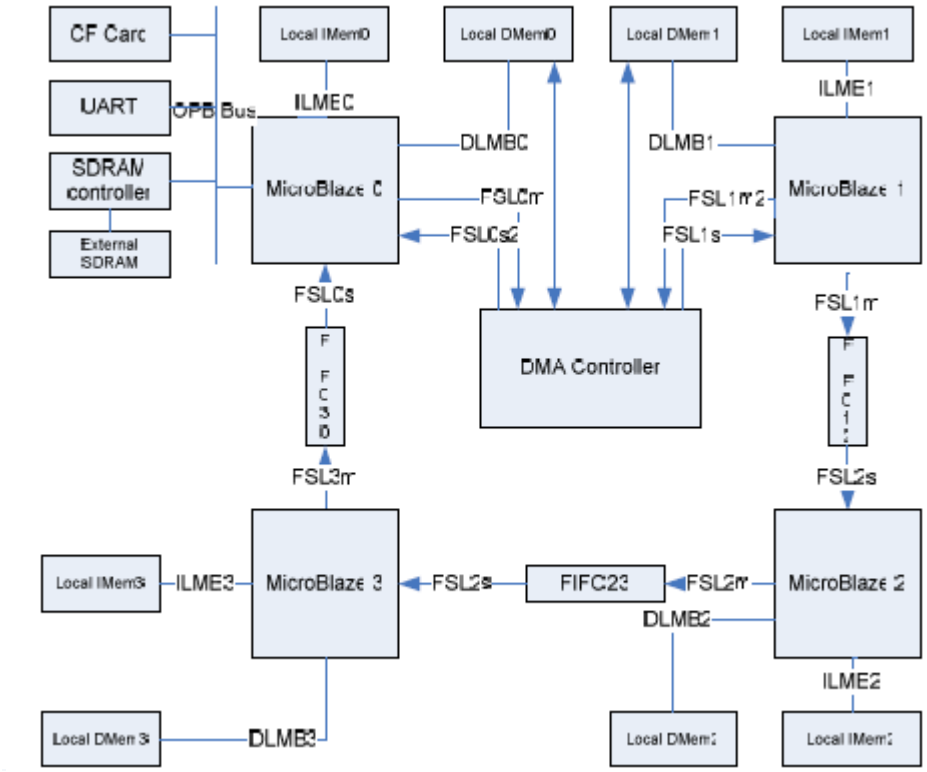

Figure 2.8 Hardware architecture of four-processor system connected by FIFO and DMA

Compared to the previous system, the FIFO between processor 0 and processor 1 is replaced by DMA controller. The DMA controller has two sets of interfaces, to processor 0 and processor 1 respectively. For each interface, there is a memory bus connected directly to local data memory of the processor. It reads directly from the local data memory or writes directly to it. Besides that, the processor can configure and read back status via FSL master and slave bus. There is one channel inside the DMA controller. The processor only needs to set starting address, ending address, size of data block and go. No CPU intervention need. Synchronization is also provided by controller and if no data moved by DMA, the processor can stop as well.

2.1.8 Interconnection Exploration: Conclusions

Regarding the results that can be conducted there is a trade-off between different types of on-chip interconnections and therefore they should be deployed depending on the application. Bus is easier to implement but poor in performance.

Dual port memory is easy to implement and efficient but it's resource consuming, inflexible due to the fixed topology and it needs synchronization mechanism. FIFO is easy to implement and uses built-in synchronization mechanism but it's inflexible due to its fixed topology and less efficient because it requires the processor to copy data into the FIFO. Finally, DMA controller is flexible, scalable and efficient. The disadvantage is the complexity of the controller.

Because of the after-manufacturing programmability of FPGA, the best interconnection is a combination of these interconnection types in a topology that targets for the application.

2.2 Soft Multiprocessor System for IPV4 Packet Forwarding

In [13] is presented IPv4 packet forwarding on a multiprocessor on a Xilinx Virtex-II Pro FPGA. This particular design achieves a 1.8 Gbps throughput and loses only 2.6x in performance compared to an implementation on the Intel IXP-2800 network processor.

2.2.1 Soft Multiprocessor Systems on Xilinx FPGAs

The packet forwarder is implemented on a Xilinx Virtex-II Pro 2VP50 FPGA, using the Xilinx EDK [16]. The building block of the multiprocessor system is the Xilinx Microblaze soft processor. The soft multiprocessor is a network composed of the multiple soft Microblaze cores, the peripherals in the fabric, the dual IBM PowerPC 405 cores, and the distributed BRAMs on the chip. The multiprocessor network is supported by two communication links: the IBM CoreConnect buses and the point-to-point FIFOs. The multiprocessor is clocked at 100 MHz due to restrictions on the clock rate of the OPB.

2.2.2 IPv4 Packet Forwarding Application

The IPv4 packet forwarding application runs at the core of network routers and forwards packets to their final destination. As a result, a soft-multiprocessor is designed for the data plane of the IPv4 packet forwarding application.

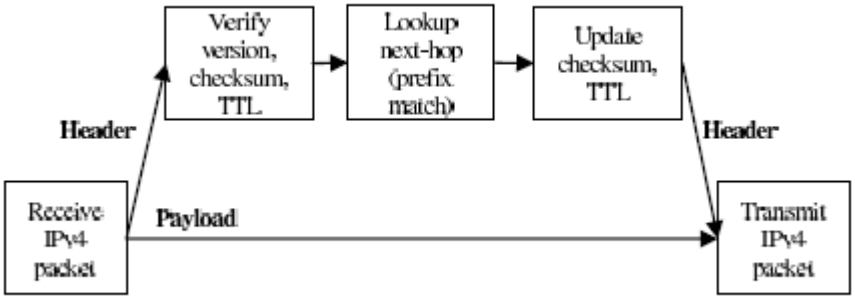

Figure 2.9 Data Plane of the IPv4 packet forwarding application

The design objective is to maximize router throughput.

2.2.3 Soft Multiprocessor Design for Header Processing

The forwarding data plane (Figure 2.9) has two components: IPv4 header processing and the packet payload transfer. In [13] first the construction of a soft multiprocessor system for header processing is described. Figure 2.10 shows the final multiprocessor design. The micro-architecture consists of multiple arrays of pipelined Microblaze processors.

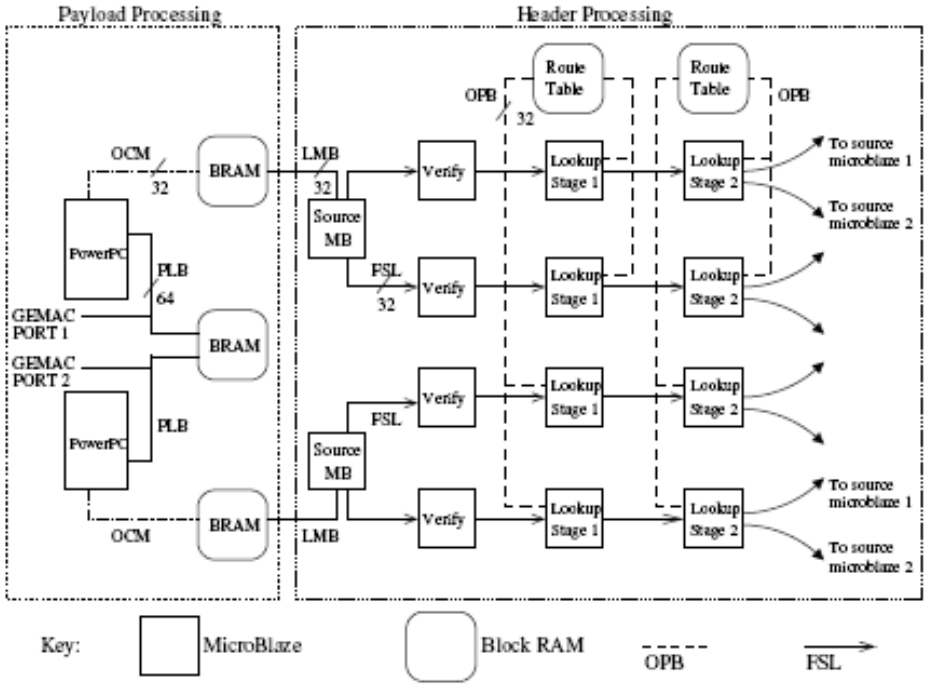

Figure 2.10 soft multiprocessor systems for the data plane of the IPv4 packet forwarding application

A staring reference for baseline performance is a single processor solution, where the entire header processing runs on a Microblaze. The route table is stored in

BRAM and accessed over the OPB. Under this scenario the IPv4 forwarding requires 270 cycles per packet. The maximum throughput that can be achieved by this single processor design operating at 100MHz is 0.17 Gbps.

At a first step towards multiprocessor design, the header processing is pipelined. Each branch of the header processing micro-architecture in Figure 2.10 is a pipelined array of three Microblaze processors along which a single header is processed. FSL links transfer the entire header among processors. The first pipeline stage performs IP header verification. The 6 lookup memory accesses of the trie lookup algorithm are partitioned equally between the second and the third pipeline stages, and hence can be performed in parallel. The third pipeline stage performs an additional memory access to determine the egress port. The trie table is also divided among multiple BRAM modules, and each processor accesses route table over a separate OPB bus. For the application decomposition in Figure 2.10, the throughput of a single array is around 0.5 Gbps.

Pipelining is a means to parallelize the application temporally. The next degree of parallelism comes from replicating the pipeline arrays in space. Each header constitutes a logically independent control flow. Hence, multiple branches can process different headers in parallel. Each branch executes the same decomposition of the header processing application. Two factors restrict the number of branches in the design:

- FPGA BRAM cells bound the number of processors (with a 300KB route table and 8KB local memory per processor, a Virtex-II Pro 2VP50 FPGA can allow only 15-20 processors)
- Branch executions are not independent due to concurrent memory accesses to the route table over a shared bus.

Taking area and arbitration constraints into account, the final multiprocessor design for header processing (Figure 2.10) replicates the single pipeline array into 4 branches. All processors in lookup stages 1 and 2 access the same part of the route table in shared memory over the OPB bus. From experiments, there is a significant drop in OPB performance if more than 2 processors share the same bus. The BRAM memory is dual-ported. Hence, the same route table memory can be serviced by 2 OPB buses. Thus, the choice of 4 branches is optimum for multiprocessor designs where shared resources are accessed over the OPB. The

measured throughput of the header processing multiprocessor in Figure 2.10 is 1.8 Gbps.

2.2.4 Performance Characteristics

Regarding the performance characteristics of the soft multiprocessor for header processing, the breakup of the number if instructions and cycles executed by each pipeline stage of the multiprocessor for header processing in Figure 2.10 is shown in Table 2.2. The two IP lookup stages are bottlenecks in the design. Table 2.3 summarizes area, memory and performance of the multiprocessor for header processing in Figure 2.10. Area utilization is less than 50% but memory is a tighter constraint. The local memories occupy 14x8 =112 KB, and the routing table occupies 300KB. The throughput of the router in Figure 2.10 is 1.8 Gbps.

Stage	# Instructions	# Execution Cycles
Verify	64	97
Lookup Stage 1	57	110
Lookup Stage 2	56	114

Table 2.2 Execution times for processing one packet header

# Processors	14 (MicroBlaze)
Area	11,250 slices (out of 23616 on 2VP50) 48% utilization
Memory (on-chip BRAM)	454 KB (out of 522 KB), 87% utilization (major components are 300 KB route table, 8 KB instruction+data memory per processor)
Throughput	1.8 Gbps

Table 2.3 Design characteristics of the soft multiprocessor for header processing on the Xilinx Virtex-II Pro 2VP50

2.2.5 Payload Transfer in the Multiprocessor Design

Later in [13] is described the payload transfer in the multiprocessor design. The multiprocessor in Figure 2.10 shows the payload transfer component and its interface to the multiprocessor for header processing for a 2-port 2 Gbps router. A Gigabit Ethernet MAC (GEMAC) for each port handles packet reception and transmission under the control of the PowerPC processors. The GEMACs transfer the packet header and payload to BRAM memory over the PLB. The header and a

pointer to the payload location are then transferred over the On-Chip Memory (OCM) bus into memory that is shared between the PowerPC and the header processing multiprocessor. There is one source Microblaze processor per router port, which reads the header from the OCM, transfers the header to the Microblaze array, and writes back the processed header back into the OCM. Each packet is transferred over the PLB twice, once during the reception and once during transmission. The PLB has simultaneous read and write data paths with a total bandwidth of 12.8 Gbps. This is sufficient to buffer and transfer the packet payload at 2 Gbps line rates.

2.2.6 Evaluation of Soft Multiprocessor Solutions

Finally, in [13] the performance of the soft multiprocessor system which presented above is compared to the performance of a software solution of IPv4 forwarding application, on the Intel IXP2800 network processor. The IXP2800 is a state-of-the-art multiprocessor specialized for packet forwarding applications. It has 16 RISC micro-engines clocked at 1.4 GHz for data plane operations and an Intel XScale processor for control and management plane operations. Meng, et al, report a throughput of 10 Gbps on the IXP2800 for the packet forwarding application for different packet sizes.

Table 2.4 shows the relative performance of the IXP2800 and soft multiprocessor solutions for IPv4 packet forwarding. The IXP28000 performs about 2.6x better that the soft multiprocessor for packet forwarding in terms of normalized throughput. This is because the IXP28000 was specifically designed to target forwarding applications.

	Soft Multiprocessor	IXP2800
Technology ($\lambda\,\mu m$)	0.13	0.13
Clock Frequency (MHz)	100	1400
Area ($A\ mm^2$)	130	280
Throughput ($T\ Gbps$)	1.8	10
Norm. Throughput ($T/\frac{A}{\lambda^2}$)	1	2.6

Table 2.4 Performance results of the data of the IPv4 packet forwarding application

However, the advantage of soft multiprocessors is the low development cost for deploying an application on a target platform.

2.3 A Microblaze Based Multiprocessor Soc

[14] presents a study of the viability of making a multiprocessor system in a chip using the Microblaze soft- core processor of Xilinx. Performance of data communication is studied, and also some parallel applications are used for testing speedup and efficiency of the system.

2.3.1 Communication test

For testing and measuring the capabilities of the FSL links (its architecture will be described later in this thesis) transmitting data between processors, a simple system has been built. This system consists of 2 Microblaze cores interconnected via 2 FSL links, BRAMs for data and instruction memory, for each processor connected to the local memory buses, and a timer for measuring data transfer times connected to MB_0. A diagram of this system can be seen in the next Figure:

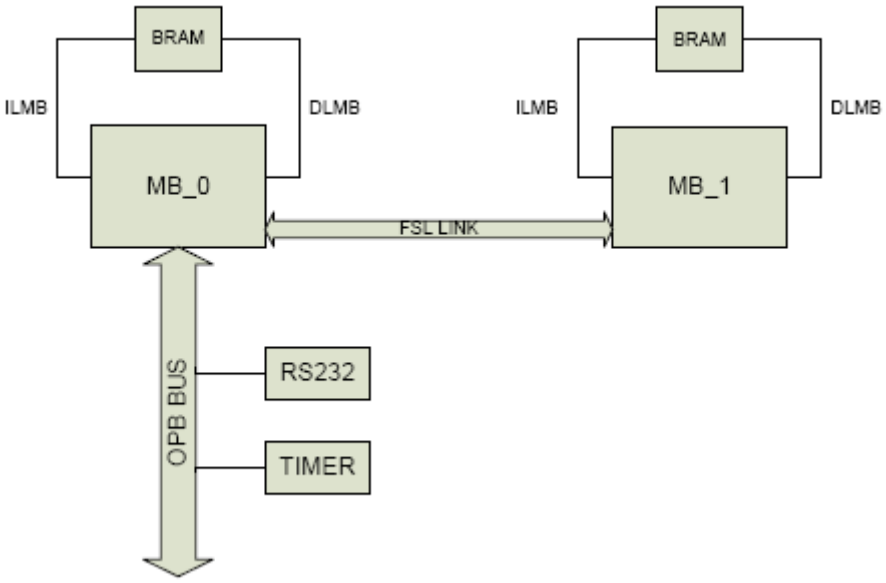

Figure 2.11 Microblaze System for testing communication

This system was used to test the viability of transmitting data over the FSL links and measure the time consumed in this task.

For testing the speed of the links, a program,, for data transfers has been developed. This program was used for transferring different sizes of data, from a simple 32 bit word to a matrix of 32x32 unsigned integers (4KB). In the next graph we can see the different speeds obtained with these tests.

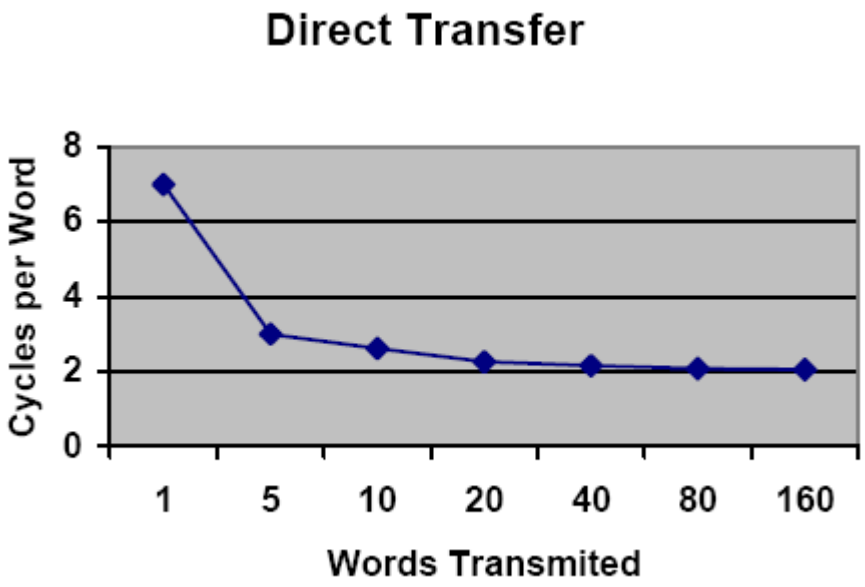

Table 2.5 Direct transfer speeds over FSL link

In this graph the cycles per word consumed for different sizes of data transmitted are represented. With this kind of transmission and large data sizes, speeds of 2 cycles per word transmitted are reached. This configuration is good if a low volume of data is transferred.

2.3.2 Network Topology

There are many network topologies that can be materialized with point to point links. The pros and cons of three of them will be described and the viability of using them in a multi Microblaze design using FSL links for point to point data transfer:

- Completely Meshed: a completely meshed network is a network in which each node is connected to every other node in the network. It is a good way to reduce the travelling time of packets over a network, because data goes directly from sender to receiver, but its main disadvantage is that the number of links grows extremely quickly when the number of nodes is increased. A completely meshed network topology with Microblaze and FSL links will only be possible for just 9 Microblazes, because of the limitation of 8 FSL links for each Microblaze processor.

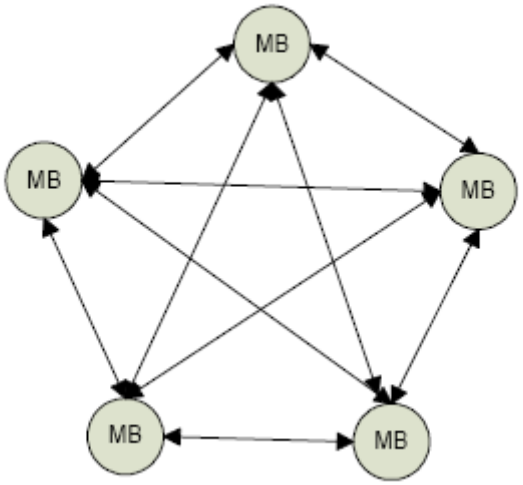

Figure 2.12 Completely meshed network

- Ring Network: a ring network is a network in which each node in the network is connected to the following and the preceding node in the network, forming a ring. Data is passed from node to node until it reaches the destination node. With Microblaze and FSL links there will not be a size limit for the network, because each Microblaze would just use 2 FSL links. The main problem of this topology is that data transfers from two nodes are far from each other and very consuming

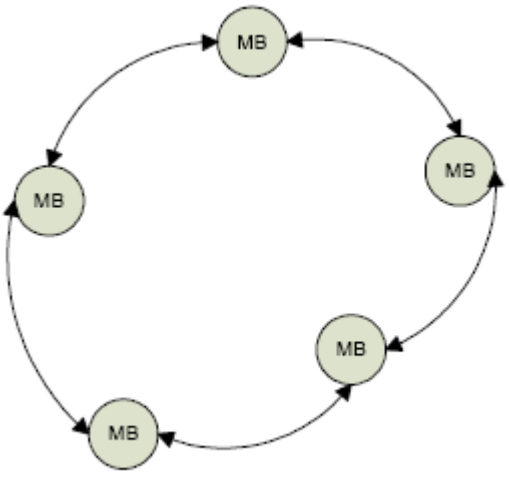

Figure 2.12 Ring network

- Star Network: a star network is a network in which each node is connected to a central node. The weak point of this topology is that if the central node fails, the whole system fails. This weakness is not very important in an embedded system, where all the nodes are in the same chip. Another

weakness is that all communications go through the central node, so if the application run is communication intensive an important bottleneck will be presented in the central node. Using this topology it is possible to build systems with 1 Microblaze as a central node, and up to 8 Microblazes as general nodes. Also bigger systems can be built by linking various subsystems together. This is the choice taken for developing this particular multi Microblaze system. With this architecture, the central node will be the one that decides which fragments of the work are assigned to each general node, and will also be responsible for grouping the results given by the general nodes.

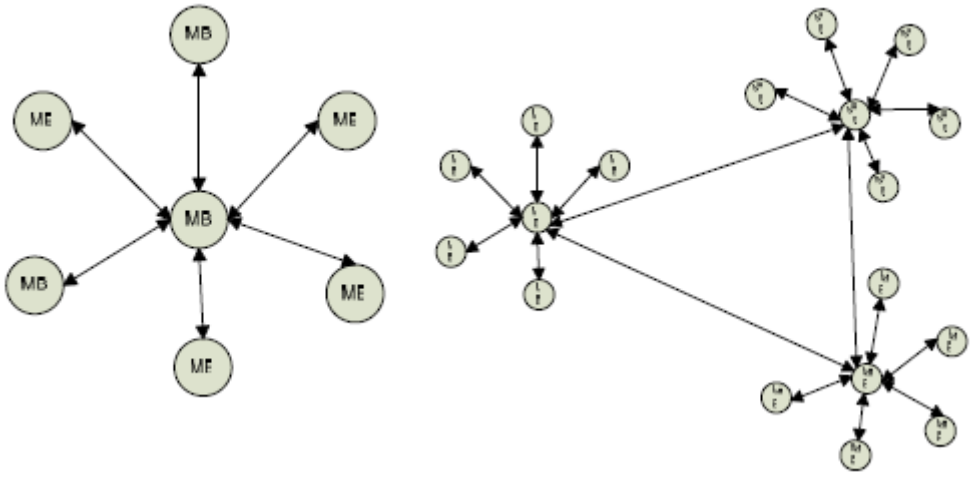

Figure 2.13 Star network (left), and linked star networks (right)

2.3.3 Complete System

Once the communication method (FSL links) and the network topology (star network) were decided, 4 systems were built: 1, 2, 4 and 8 Microblazes each. The system with 1 Microblaze consisted of just the central node, and was built with: 1 Microblaze, 16KB of BRAM for instruction and data memory, an uartlite and a timer attached to the OPB bus. The general nodes were built with one Microblaze and 16KB of BRAM for instruction and data memory. With this system some parallelizable applications for testing the speedup obtained due to the use of many processors instead of one were used.

2.3.4 Speedup and Efficiency

The speedup of a parallel algorithm is defined as the time needed to solve the problem using just one processor divided by the time needed to solve the problem with p processors. In an ideal system and with ideal parallel algorithm, the speedup would be equal to the number of processors p. In real world, it is always lower due to communications overhead.

$$speedup = \frac{t_s}{t_p}$$

Another measure of the performance of a parallel system is efficiency. Efficiency is defined as the speedup per processor.

$$efficiency = \frac{speedup}{p}$$

In an ideal system with an ideal parallel algorithm, the efficiency would be equal to 1.

2.3.5 First Application: Matrix Multiplication

The first application used to test the system that is described in [14] was an application that performed matrix multiplications. The parallelization of the matrix multiplication algorithm implemented consists of sending one or more rows of the first matrix, and the whole second matrix to each processor. So each processor obtains one or more rows of the resulting matrix and returns it to the central processor, which is the responsible for merging the results.

The results obtained for speedup and performance are shown in the following tables:

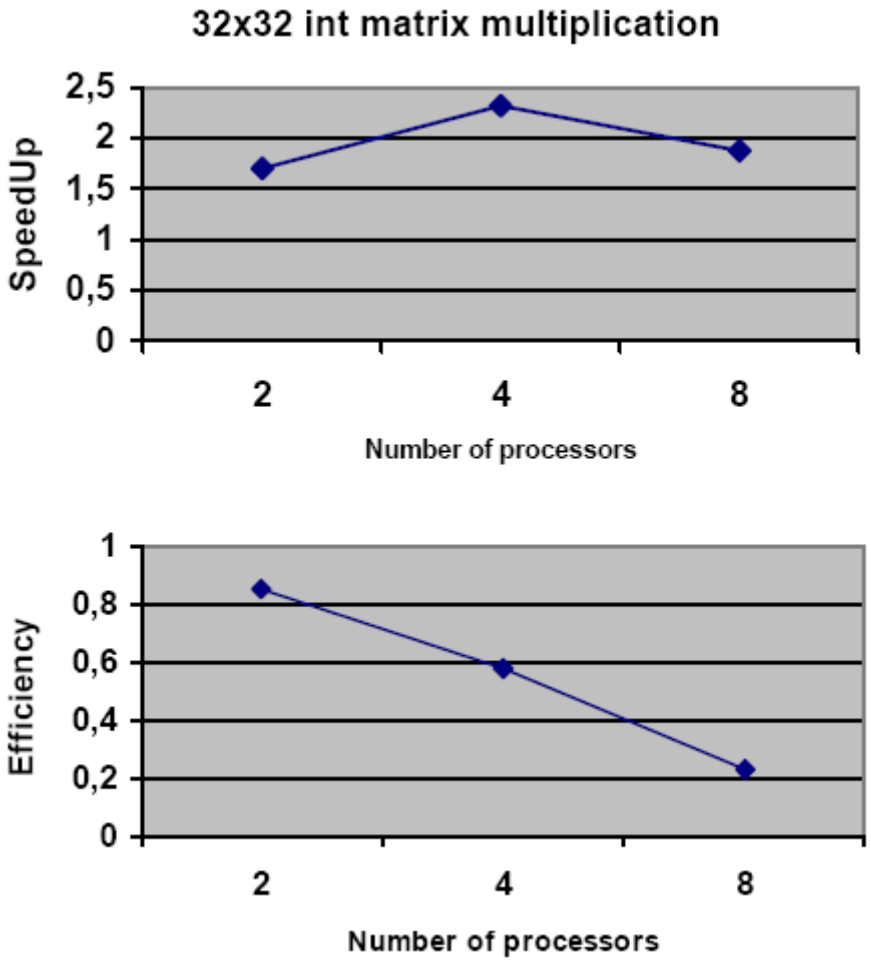

Table 2.6 Speedup and efficiency for 32x32 integer matrix multiplication parallel program

As it can be seen in Table 2.6, results are not as good as it could be expected. It can be seen that the efficiency falls quickly, when the number of processors grows, reaching an efficiency of 20% with 8 processors. This is because when the number of processors is increased, more time is spent in communications; consequently the time saved with parallelism is spent in data communication.

2.3.6 Second Application: Cryptographic Application

Another application used to test the efficiency of the system in [14] was a cryptographic application using the AES (Rijndael) algorithm. The AES takes an input of 128 bits and a key of 128, 196 or 256 bits and generates an output of encrypted 128 bits. The decryption is made with the same key used in the encryption process with a similar process. The implemented version of AES used to test the system is the 128 bits key version. The test implemented consists on encrypted or decrypting a test of 1024 bytes in size. The text is split in 16 bytes

blocks, and each processor is always ready to process a 16 byte block. When the central processor receives a text file to be encrypted / decrypted, it starts to send blocks to each processor. It also has to send the encryption key to be used. The results obtained, are shown in the next table:

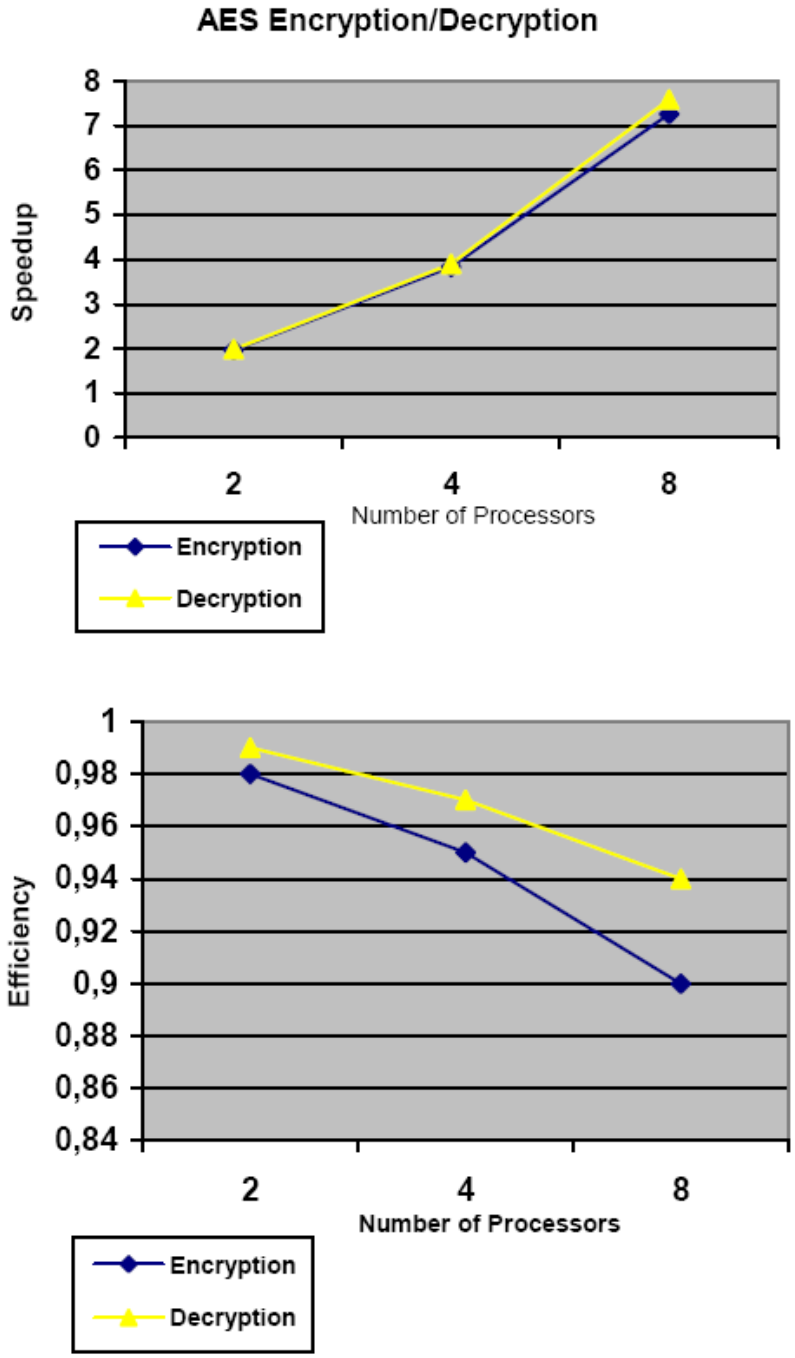

Table 2.7 Speedup and efficiency fro AES encryption/decryption

As it can be seen, the system offers a good performance, with efficiencies higher than 90%, quite better than the matrix multiplication test. The efficiency when

doing decryption is higher than with encryption, because the decryption algorithm is a little more time consuming so the relationship between computing time against communications time is higher.

2.3.7 Conclusions

From [14] can be concluded that the FSL links are an ideal choice for exchanging data between processors due to their high speed data transfer rates. This system is very appropriate for parallel algorithms in which the data transfer time is substantially lower than the computing time and that soft-core processors are appropriate for building multiprocessors systems on a chip.

3. MPLEM Architecture

This chapter describes the structure and functionality of the Microblaze, the model of the memory that is used, the structure of the custom peripherals that were created to support specific operations of the platform and finally the functionality of the necessary Intellectual Properties (IPs) that are available from Xilinx libraries (buses, block RAMs, bridges), so as the implementation to become feasible. Also here the architecture of the multiprocessor platform we implemented is described. More specifically, not only the way all the interconnections are made between the peripherals, the memory and the processors is described, but also the reason for this specific architecture. Besides that, an early simulation for this platform in this chapter is described. Simple software is implemented, to test the correct functionality, not only of the platform, but also of all peripherals that are used. The name is chosen for this platform is Multiprocessor Platform for Embedded systems or M.PL.EM.

3.1 Description of Hardware

3.1.1 Microblaze Architecture

Microblaze is a soft, 32-bit reduced instruction set computer (RISC) processor designed by Xilinx for their FPGAs. Compared to other general purpose processors, it's quite flexible with a few configurable parts and capable of being extended by customized co-processors. There are a number of on-chip communication strategies available including a variety of memory interfaces. Following is the core block diagram of Microblaze processor. [19]

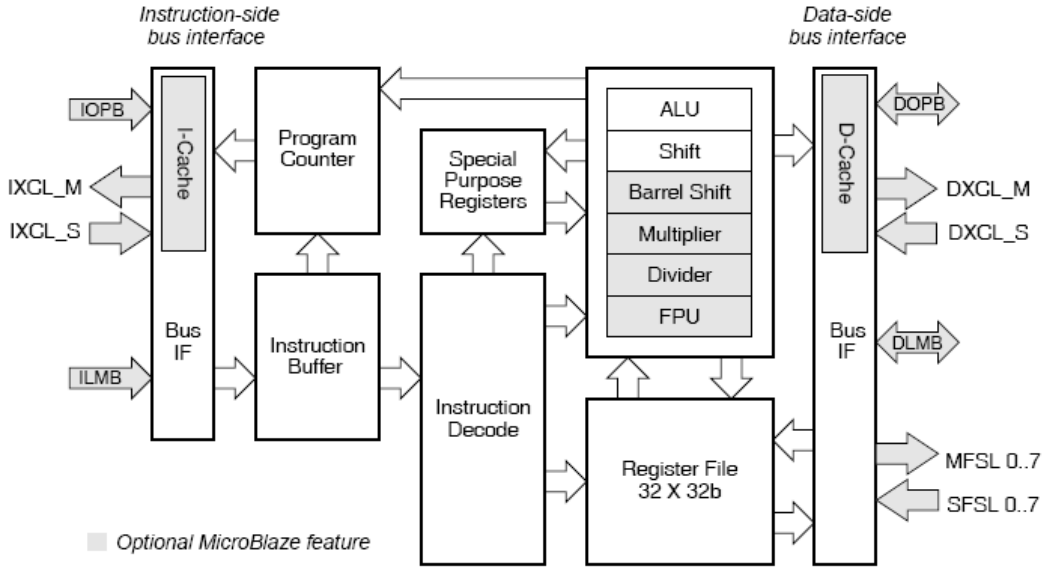

Figure 3.1 Microblaze core block diagram

Similar to most of RISC processors, Microblaze processor has an instruction decoding unit, 32x32b general purpose register file, arithmetic unit and special purpose registers. In addition, it has an instruction prefetch buffer. The arithmetic unit is configurable, as shown in Figure 3.1. The Barrel Shift, Multiplier, Divider and Floating Point Unit (FPU) are optional features. Microblaze processor has a three-stage pipeline: fetch, decode and execute. For most of instructions, each stage takes one clock cycle. There is no branch prediction logic. Branch with delay slot is supported to reduce the branch penalty.

Microblaze is a Harvard-architecture processor, with both 32-bit Instruction-bus and Data-bus. Cache is also an optional feature. Three types of buses, FSL, LMB, and OPB are available. All three types of buses will be described later.

3.1.2 On-Chip Peripheral Bus V2.0 (OPB) with OPB Arbiter

The OPB Bus with OPB Arbiter module is used as the OPB interconnect for Xilinx FPGA based embedded processor systems. The bus interconnect in the OPB V2.0 specification is a distributed multiplexer implemented as an AND function in the master or slave driving the bus, and an OR function to combine the drivers into a single bus. [18]

The features of OPB V2.0 with OPB arbiter are the following:

- Includes parameterized OPB Arbiter

- Includes parameterized I/O signals to support up to 16 masters and any number of slaves
- The OR structure can be implemented using only LUTs or can use a combination or LUTs and fast carry adder to reduce the number of LUTs in the OR interconnect
- Includes a 16-clock Power-on OPB bus Reset and parameter for high or low external bus reset.
- Includes input for reset from Watchdog Timer.

The Xilinx OPB V20 bus core allows the designer to tailor the OPB bus arbiter to suit the application by setting certain parameters to enable/disable features.

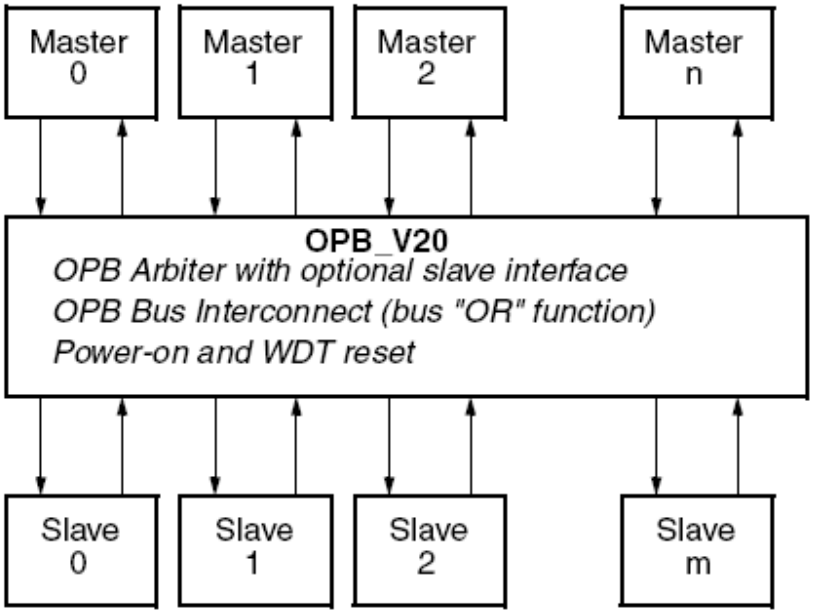

Figure 3.2 OPB System Using OPB_V20

3.1.3 Fast Simplex Link Channel (FSL)

Microblaze contains eight input and eight output FSL interfaces. The FSL channels are dedicated unidirectional point-to-point data streaming interfaces. The FSL interfaces in Microblaze are 32 bits wide. Further, the same FSL channels can be used to transmit or receive either control or data words. The performance of the FSL interface can reach up to 30 MB/sec. This throughput depends on the target device itself. The FSL bus system is ideal for Microblaze-to-Microblaze or streaming I/O communications. [18]

The main features of the FSL interface are:

- Unidirectional point-to-point communication
- Unshared non-arbitrated communication mechanism
- Control and Data communication support
- FIFO-based communication
- Configurable data size
- 600 MHz standalone operation

The FSL bus is driven by one Master and drives one Slave. The next figure shows the principle of the FSL bus system and the available signals.

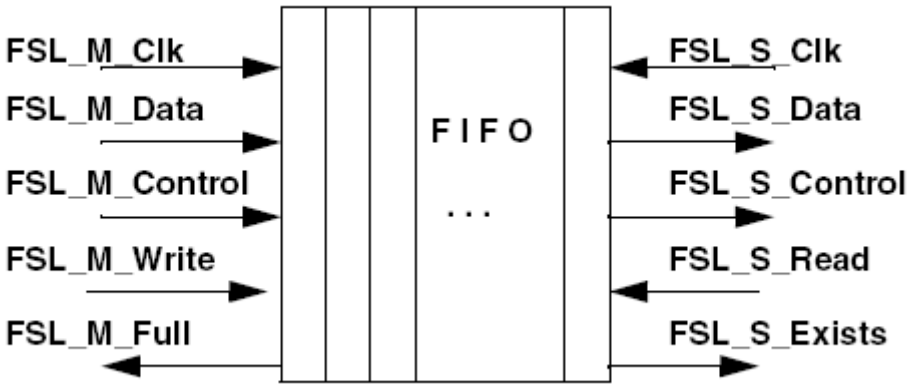

Figure 3.3 FSL Interface Signals

Xilinx EDK provides a set of macros for reading and writing to or from an FSL link. There are two ways of reading/writing on an FSL link: blocking or not blocking, and also there are different instructions for reading/writing data or control words.

3.1.4 Local Memory Bus (LMB)

The LMB module is used as the LMB interconnect for Xilinx FPGA-based embedded processor systems. The LMB is a fast, local bus for connecting Microblaze instruction and data ports to high-speed peripherals, primarily on-chip block RAM (BRAM). [18]

The main features of LMB are:

- Efficient, single master bus (requires no arbiter)

- Separate read and write data buses7Low FPGA resource utilization
- 125 MHz operation

A typical Microblaze system using two LMBs is shown in figure 3.4. This system illustrates the use of both Instruction and Data side LMB buses connecting to a dual-ported BRAM Block via separate LMB BRAM interface controllers.

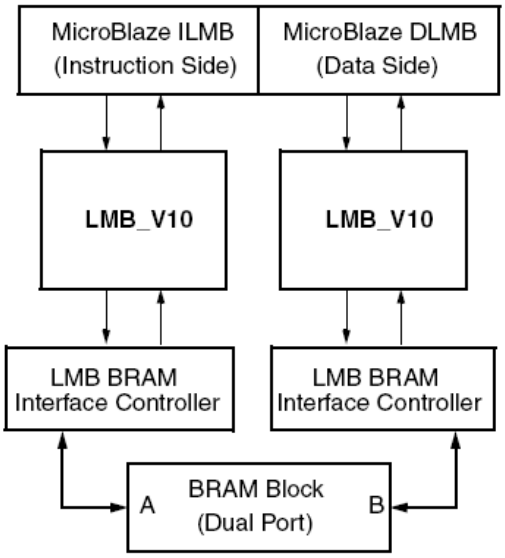

Figure 3.4 Typical Microblaze System using Two LMBs

3.1.5 LMB Block RAM Interface Controller

The LMB BRAM interface controller is the interface between the LMB and the bram_block peripheral. A BRAM memory subsystem consists of the controller along with the actual BRAM components that are included in the bram_block peripheral. [18]

The main features of LMB BRAM Controller are:

- LMB bus interfaces with byte enable support
- Used in conjunction with bram_block peripheral to provide fast BRAM memory solution for Microblaze ILMB and DLMB ports
- Supports byte, half-word, and word transfers

3.1.6 Block RAM (BRAM)

The BRAM block is a configurable module that attaches to a variety of BRAM interface Controllers. The BRAM Block structural HDL is generated by the EDK

design tools based in the configuration of the BRAM interface controller IP. All BRAM Block parameters are automatically calculated and assigned by the EDK tools Platgen and Simgen. [18]

The features of BRAM are:

- Fully automated generation and configuration of HDL through EDK Platgen/Simgen tools
- Number of BRAM primitives utilized is a function of the configuration parameters for: memory, address range, number of byte-write enables, the data width, and the targeted architecture
- Both Port A and Port B of the memory block can be connected to independent BRAM Interface Controllers: LMB, OPB, Processor Local Bus (PLB), and On-Chip Memory (OCM)
- Supports byte, half-word, word and doubleword transfers provided to the correct number of byte-write enables have been configured

3.1.7 OPB to OPB Bridge (Lite Version)

The OPB to OPB Lite Bridge is used to connect two OPB buses. The bridge has one master port and one slave port. Two bridges may be used together to support full bus mastership in both directions. [18]

The features of OPB to OPB Bridge are:

- Provides a bridge between two OPB buses
- Connections for one master-side bus and one slave-side bus
- Parameterized data bus widths
- Simple transaction forwarding reduces LUT count
- Requires the two OPB buses to be on the same clock and the same size
- No support for data buffering or posted writes

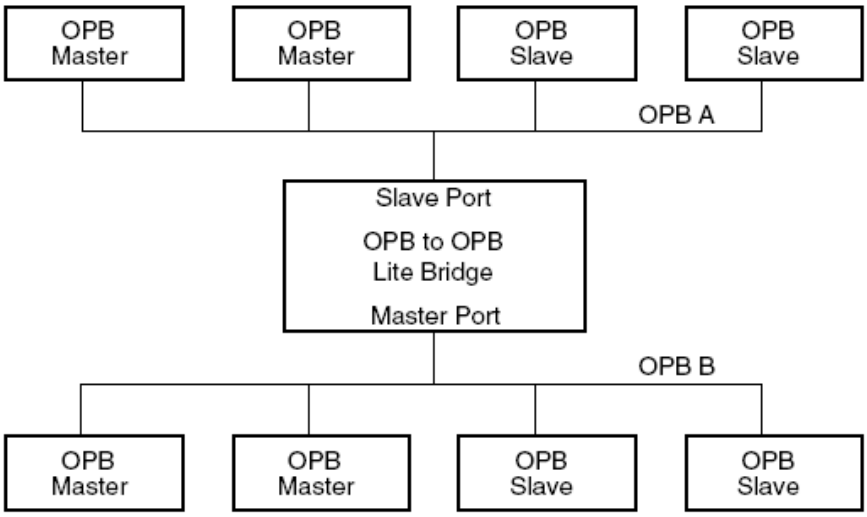

Figure 3.5 OPB System with Bridge

Figure 3.5 shows a typical system with two OPB buses interconnected with a bridge. Any OPB master on OPB B can initiate a transaction (read or write) on OPB A. Note that masters on OPB A cannot get to OPB B unless another bridge is used in the opposite direction. Figure 3.6 shows a system with two bridges. In this system, masters on OPB B can initiate reads and writes to slaves on OPB A, and masters on OPB A can initiate reads and writes to slaves on OPB B.

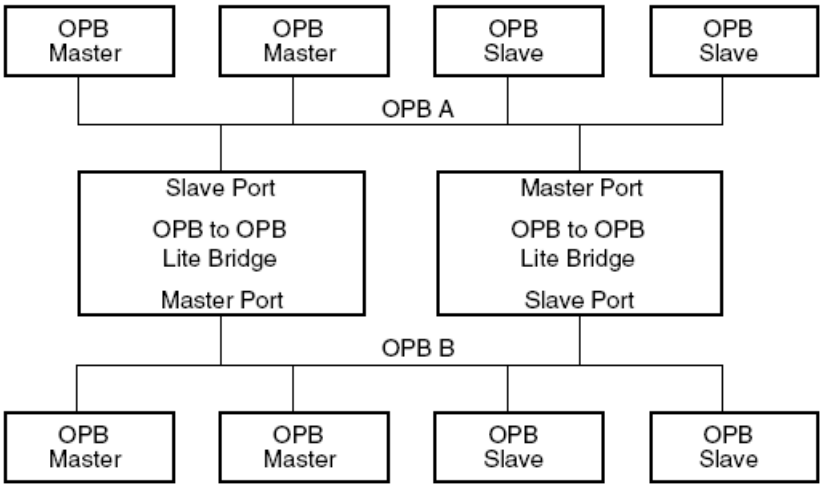

Figure 3.6 OPB System with Two Lite Bridges

3.1.8 Cypress CY7C1041 256Kx16 Static RAM

The behavioral model of external memory that is used to store the input data of the multiprocessor platform is the CY7C1041 provided from Cypress. The CY7C1041 is a high-performance CMOS static RAM organized as 262,144 words by 16 bits.

Writing to the device is accomplished by taking Chip Enable and Write Enable inputs LOW. If Byte Low Enable is LOW, then data from I/O pins (I/O_0 through I/O_7) is written into the location specified on the address pins (A_0 through A_{17}). If Byte High Enable is LOW, then data from I/O pins (I/O8 through I/O_{15}) is written into the location specified on the address pins (A_0 through A_{17}).

Reading from the device is accomplished by taking Chip Enable and Output Enable LOW while forcing the Write Enable HIGH. If Byte Low Enable is LOW, then data from the memory location specified by the address pins will appear on I/O0 to I/O7. If Byte High Enable is LOW, then data from memory will appear in I/O_8 to I/O_{15}. The input/output pins are placed in high impedance state when the device is deselected, the outputs are disabled, the Byte High Enable and Byte Low Enable are disabled, or during a write operation. [20]

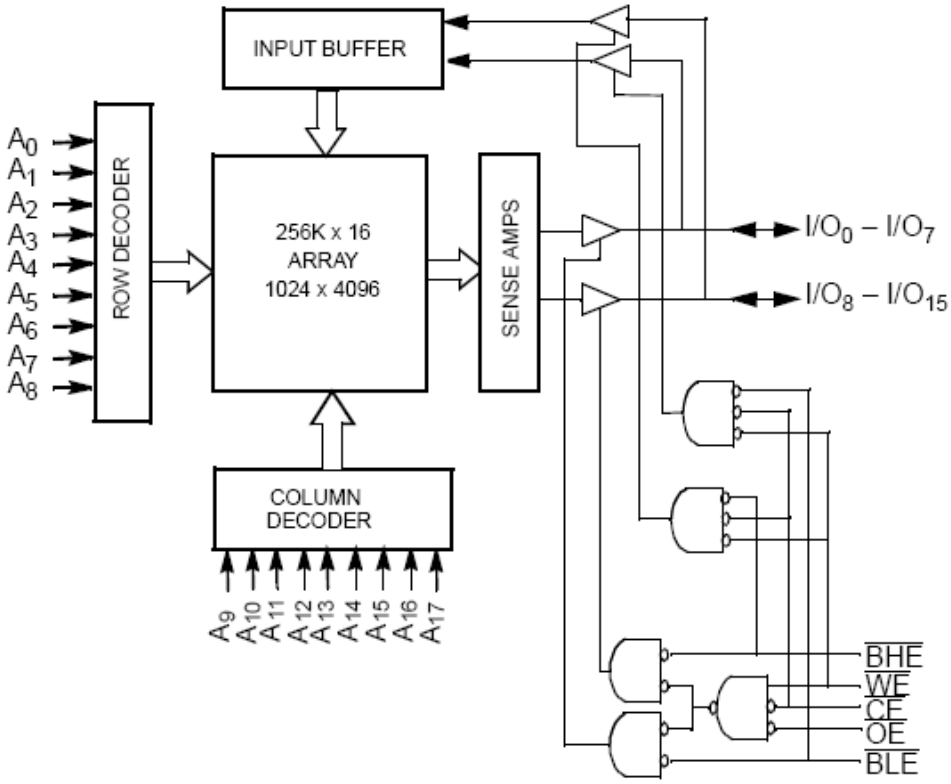

Figure 3.7 SRAM Logic Block Diagram

$\overline{CE}$	$\overline{OE}$	$\overline{WE}$	$\overline{BLE}$	$\overline{BHE}$	I/O_0–I/O_7	I/O_8–I/O_{15}	Mode	Power
H	X	X	X	X	High Z	High Z	Power Down	Standby (I_{SB})
L	L	H	L	L	Data Out	Data Out	Read All Bits	Active (I_{CC})
L	L	H	L	H	Data Out	High Z	Read Lower Bits Only	Active (I_{CC})
L	L	H	H	L	High Z	Data Out	Read Upper Bits Only	Active (I_{CC})
L	X	L	L	L	Data In	Data In	Write All Bits	Active (I_{CC})
L	X	L	L	H	Data In	High Z	Write Lower Bits Only	Active (I_{CC})
L	X	L	H	L	High Z	Data In	Write Upper Bits Only	Active (I_{CC})
L	H	H	X	X	High Z	High Z	Selected, Outputs Disabled	Active (I_{CC})

Figure 3.8 SRAM Truth Table

Figure 3.7 shows the logic block diagram of the memory, while figure 3.8 shows its truth table.

3.1.9 FSL Peripheral

The FSL Peripheral is a custom peripheral built to support the multiplatform functionality. This peripheral is used to store the data from the external SRAM that described previously, and send this data to a Microblaze processor. The FSL peripheral is connected with the Microblaze via an FSL bus, and is connected with the SRAM with custom logic (there is no bus available from the Xilinx libraries that connects this peripheral with this specific model of SRAM).

The FSL peripheral contains a Dual Port BRAM organized as 4096 words by 32 bits that stores the data which come from the external memory, and a controller used to arbitrate the communication of this peripheral with the processor and the SRAM.

Figure 3.9 presents the logic block diagram of this peripheral, while figure 3.10 presents the Finite State Machine (FSM) that acts as the controller of the FSL peripheral.

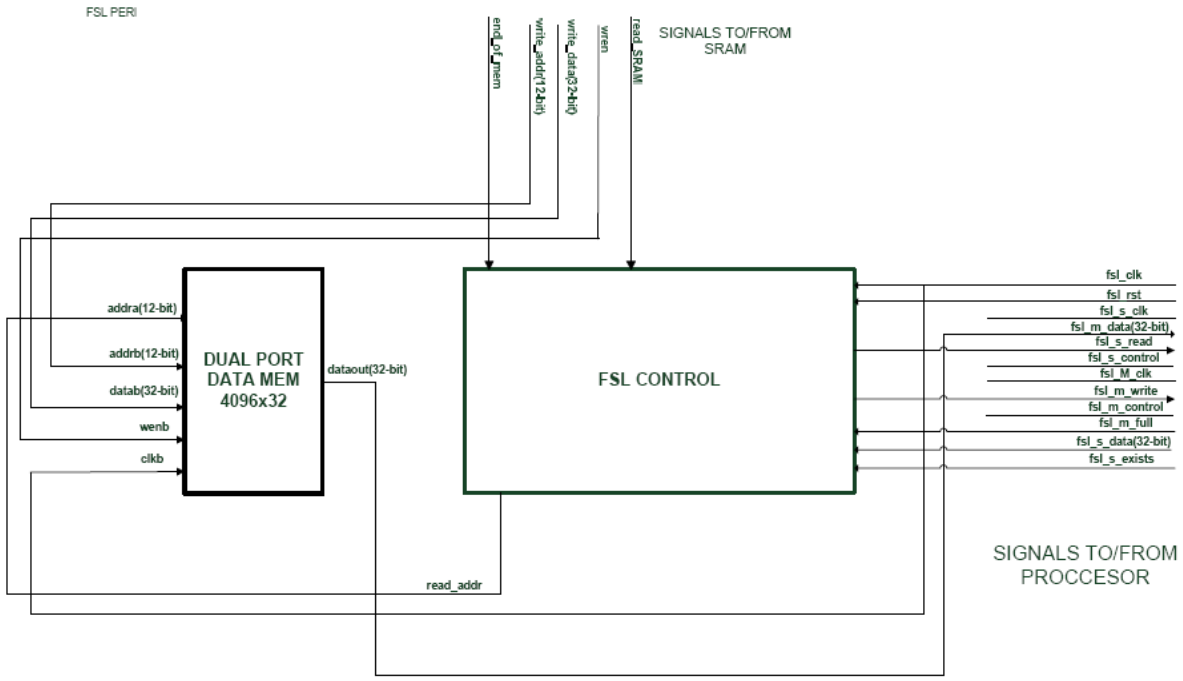

Figure 3.9 FSL Peripheral Block Diagram

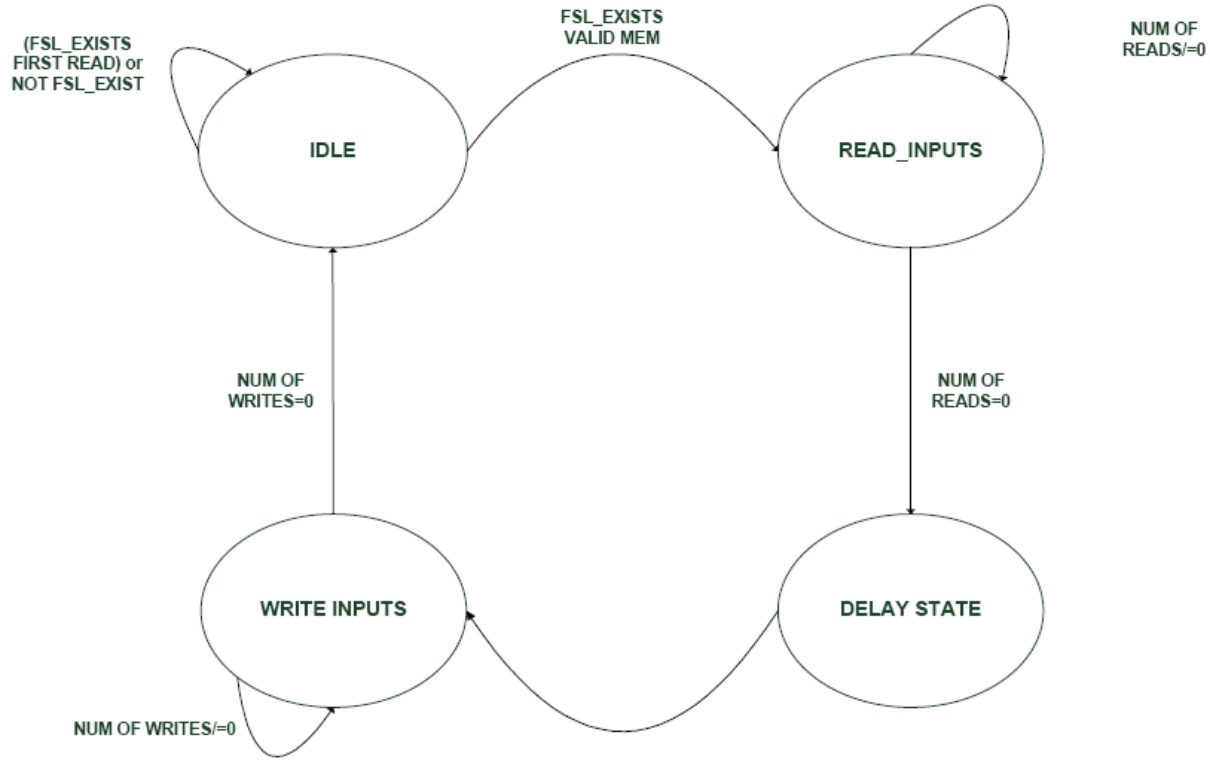

Figure 3.10 FSM of the FSL Peripheral Control

In figure 3.9 is presented the connection of the custom peripheral we built with the processor and the external SRAM. The pin out of the peripheral is explained in table 3.1.

Signal Name	I/O	Description
fsl_clk	in	system clock
fsl_rst	in	system reset
fsl_s_clk	in	slave clock (not used)
fsl_m_data	out	data from BRAM to processor
fsl_s_read	out	read data from FSL link
fsl_s_control	in	slave control (not used)
fsl_m_clk	out	master clock (not used)
fsl_m_write	out	write enable to FSL link
fsl_m_control	out	master control data (not used)
fsl_m_full	in	FSL link full
fsl_s_data	in	address from processor
fsl_s_exists	in	indicates if the processor tries to read from the peripheral
end_of_mem	in	indicates if the reading from the SRAM ends
write_addr	in	address to FSL BRAM
write_data	in	data from SRAM to FSL BRAM
wren	in	write enable to BRAM
read_sram	out	read data from SRAM

Table 3.1 Pin out of FSL Peripheral

Regarding the size of the BRAM of this peripheral, it is chosen as follows: the total number of the BRAM on XUP2VP30 (the FPGA that is used for this thesis) is 136. Each BRAM has 18 Kbit size. The remaining number of BRAMs after the placement of the 14 Microblaze processors is 72. We split this number of BRAMs to 14 peripherals (one for each processor), so for each peripheral correspond 5 BRAMs. So the total number of Kbits of each BRAM is 5*18 or 90 Kbits. We also need each position of the memory to be 32 bits wide, because this is the most convenient as each register of a Microblaze processor is 32 bits wide. As a result we need 90 Kbits/ 32 bits positions or 2812 positions. But the Xilinx tools cannot commit BRAMs of this size. The Xilinx tools commit memories of 4096 positions of 32 bits. So it is necessary to reduce the number of the BRAMs that is committed from the processors, so as to fulfill this constraint. The exact number of BRAMs that is used from the processors and the way we use them will be described in the next chapter of this thesis.

Regarding the ports of the BRAM, the port A is used for asynchronous read of the stored data, while Port B is used for synchronous write from the SRAM.

The FSM of the FSL controller that is described in figure 3.10 works as follows: Before the processor asks for a read from the FSL peripheral, it is in idle state. If the processor asks for data from the FSL peripheral, then the FSL controller asks the external SRAM to initialize the FSL BRAM. When the BRAM is initialized from the external SRAM (this process will be described later in this thesis) and the processor asks for a read, the peripheral enters in the read mode. Then each address that is asked from the processor is passed to the BRAM where the data are stored, until the number of reads goes to 0. The number of reads that are made from the processor are 4096, equal to the positions of the BRAM. When the number of reads is 0 the peripheral enters in the delay state. This state is necessary, because the dual port BRAM needs two cycles to read the address and put out the data from an address. During this state, the controller also initializes the counter of the number of reads from the processor, and asks from the SRAM to re-initialize the BRAM of the FSL peripheral. Then the peripheral enters the Write state. The data read from the BRAM are written on the FIFO of the FSL link and as a result the processor can read, and consequently use them.

In this point, it is necessary to mention that if the BRAM of the peripheral is not initialized the processor doesn't need to resend the addresses, but it stalls until the memory is ready to be read. So no stall mechanism is needed to be build, thus the FSL supports this function with the use of the FIFO in the FSL link.

3.1.10 SRAM controller

In order to use the SRAM which is described previously, a controller is designed. This controller arbitrates the accesses to the SRAM from the FSL peripherals by defining the way that the data is sent to them. The SRAM controller is connected to the SRAM from the one side and with all the FSL peripherals from the other side.

In figure 3.12 the FSM of the controller of the SRAM is presented and in Table 3.2 the pin out of the SRAM controller is presented.

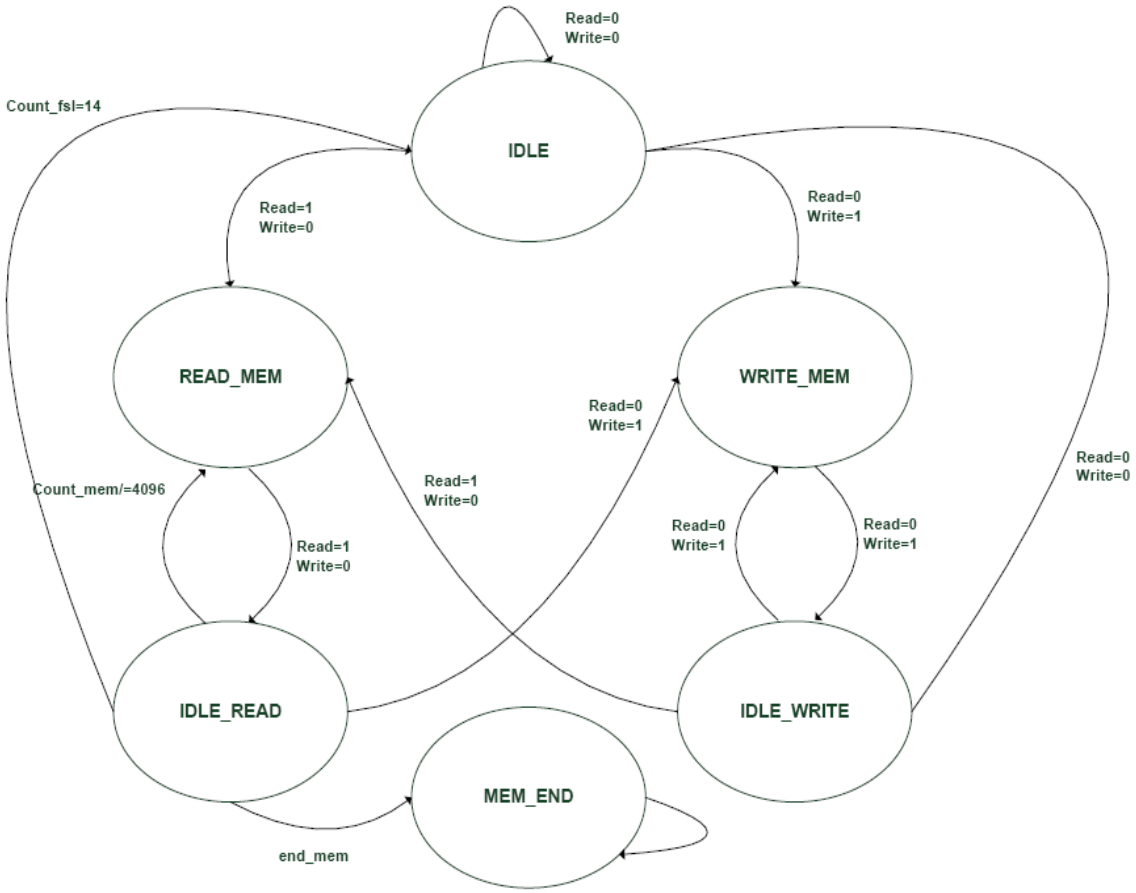

Figure 3.11 FSM of the SRAM controller

Signal Name	I/O	Description
clk	in	system clock
reset	in	system reset
addrin	in	input addrsss (18-bit)
datain	in	input data (32-bit)
write_mem	in	write mem if read_mem=0
read_mem	in	read_mem if write_mem=0
addrout	out	addrout to FSL peripheral (12-bit)
dataout	out	data to fsl peripheral (32-bit)
wren0-13	out	write enable BRAM of the FSL peripheral

Table 3.2 Pin out of the SRAM controller.

The SRAM controller that is presented in figure 3.11, implements a round robin algorithm for the initialization of the BRAMS of the FSL peripherals. The initialization of the memory is supposed to have been done with a “magic” way. Despite this fact, the SRAM controller supports writing to the memory, but in this thesis it is never used.

To be more specific, the SRAM controller works as follows: Initially, the SRAM controller is at the idle state. At this phase, the SRAM controller waits for a read or a write request, while it initializes all the signals and the necessary counters for the implementation of the Round Robin algorithm. When a read request comes from the FSL peripheral, a transition takes place from the idle to the read state. Then the controller starts to read data from the SRAM and initializes the FSL peripherals serially. There are three counters, one that counts the number of the peripherals to be initialized, one that counts the address to be send to the FSL peripheral, and one that counts the address to be send to the SRAM. After that the controller sets the control signals of the SRAM as follows: CE_b=0, OE_n=0, WE_n=1, BLE_n=0, BHE_n=0 [20]. Also, the controller sets the appropriate write enable signal for the FSL peripheral to be initialized, and sets the addresses signals to the SRAM and to the FSL peripheral BRAM. After that, a transition takes place from the read to the idle state. This state is obligatory only due to the SRAM functionality (a transition of the previously described control signals of the SRAM has to be made).

Then a continuous transition from idle read to read state is made for 4096 times, so as the whole BRAM of the FSL peripheral to be initialized with data. After 4096 times the control goes to the idle state and the whole process is repeated for all the 14 peripherals. After 14 repetitions the control goes to the memory-end state where it stays for ever because, in this thesis, it is not predicted the SRAM to be re-initialized externally (as it is mentioned above the SRAM is initialized externally with a “magic” way which will be described later in this thesis).

3.2 MPLEM Platform Topology and Interconnection

In figure 3.12 is presented the block diagram of the multiprocessor platform. At this block diagram all the interconnections that have been made for the implementation can be seen and will be explained later at this chapter.

For the implementation of this platform were used:

- 14 Xilinx Microblaze soft-core processors, 14 custom FSL peripherals, and 14 BRAMS, one for each processor.
- A 4 MB external Cypress SRAM
- 2 OPB buses
- 2 OPB to OPB bridges
- 2 shared BRAMS
- An RS232 peripheral for I/O purposes

In [14] different network topologies for a Microblaze multiprocessor platform are presented. At these topologies, all Microblaze processors communicate directly each other through the FSL bus. In this thesis a different topology is chosen. All processors communicate each other, not directly, but with the use of the OPB buses. The reason this topology is used, is that the processors work almost independently and there is no need for fast transfer of data from one processor to the other. Furthermore, the main goal for the implementation of this platform is to fit in one single FPGA the maximum number of processors that can

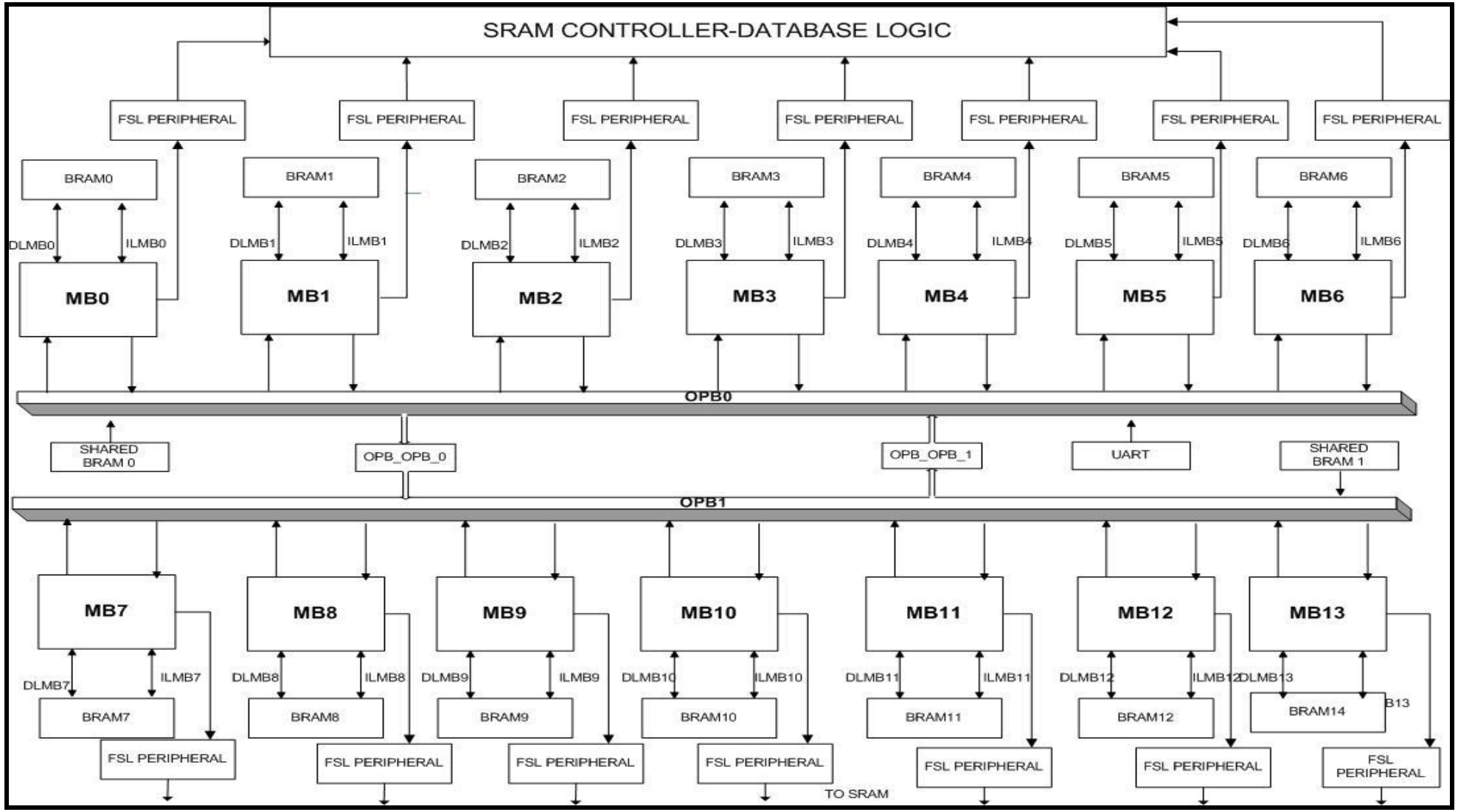

Figure 3.12 Multiprocessor Platform Block Diagram

share simple messages and work in parallel with different data that come from the external SRAM. So the best way for this is to connect all the Microblaze processors on the OPB buses.

3.2.1 Microblaze to OPB interconnection

Every Microblaze processor is connected directly to the OPB bus with the use of two links, one for data and one for instructions. Besides that, every Microblaze processor is connected on the OPB bus as a Master, so it has the right to write onto the bus and to send data to other slave peripherals or other processors. The interconnection of one Microblaze processor on the OPB bus is presented in figure 3.13.

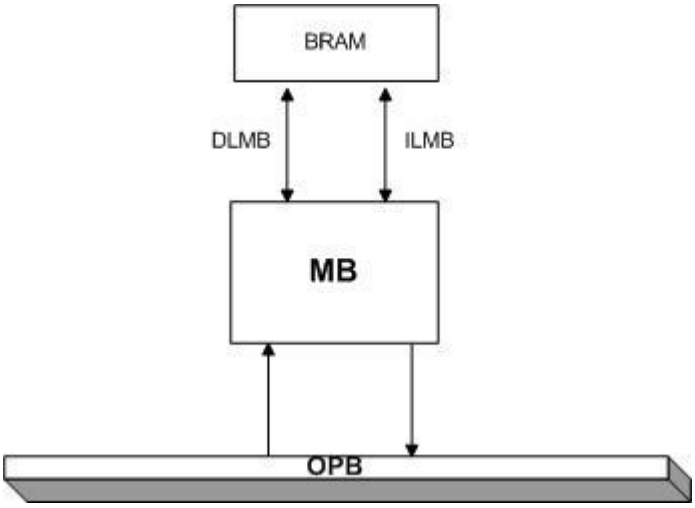

Figure 3.13 Single Microblaze interconnection

3.2.2 Microblaze to Local BRAM interconnection

In figure 3.13 the connection of the Microblaze processor with its local memory is also presented. The connection is made through two buses: the data local memory bus (DLMB) and the instruction local memory bus (ILMB). The DLMB is used to transfer data to and from the processor, while the ILMB is used to transfer the instructions to and from the processor. At this point it should be mentioned that the software and the data of the processor are stored in this local memory. Also, the size of the local BRAM should be enough so as the software of the Microblaze to fit in.

3.2.3 Microblaze to FSL peripheral interconnection

Every Microblaze processor is also connected with an FSL peripheral (its functionality was described in the previous chapter) with the use of the FSL link. This peripheral is not connected with the Microblaze processor through the OPB bus, because a high transfer speed is needed between this peripheral and the processor. The FSL link can work up to 600 MHz clock rate while the OPB bus is much slower if many processors and peripherals are connected onto this.

The Microblaze is the master peripheral of the FSL bus while the FSL peripheral is the slave. This means that the processor can ask any time data from the FSL peripheral and the peripheral can write them on the FSL bus. If the bus is full, then the processor and the FSL peripheral wait until the FIFO of the FSL bus empties.

The communication between the processor and the FSL bus is made with the use of some macros, which are provided from the Xilinx library. The macros are used are 2:

- *microblaze_bread_datafsl (val, id):* This macro performs a blocking data get function on an input FSL of Microblaze; id is the FSL identifier and can range from 0 to 7.
- *microblaze_bwrite_datafsl (val, id):* This macro performs a blocking data put function on an output FSL of Microblaze; id is the FSL identifier and can range from 0 to 7. [21]

These two macros are used together, the one after the other. The first macro is used to write to the FSL bus the addresses in the BRAM of the FSL peripheral that contain the data the Microblaze wants, while the second macro is used by the Microblaze so as to read the data that the FSL peripheral writes in the FSL bus.

Both macros use block reads and write because the processor needs to wait for the data to be ready so as to continue its execution. On the contrary if the non-block macros were used then the data the processor would receive would be wrong.

3.2.4 OPB to OPB interconnection

As it was previously described, the main goal of this work is the creation of a multiprocessor soft core system by utilizing all the resources that a specific FPGA

chip provides to us. So after some experimentation, it was found that on a Virtex II Pro FPGA (XC2VP30) chip fit 14 Xilinx Microblaze processors. But all these processors cannot be connected on a single OPB bus. An OPB bus can hold only up to 8 Master peripherals and an infinite number of Slave peripherals. So, for the implementation of a system that has 14 processors, 2 OPB buses are needed. In figure 4.1 is presented the way the 2 OPB buses are connected together to hold 14 Microblaze processors. The connection of 2 OPB buses is succeeded with the use of the OPB to OPB bridges. In this thesis, 2 bridges were used; in order every processor in each bus to have the right to access as Master to a processor that exists on the other bus. (The functionality of OPB to OPB bridges is described in the previous chapter).

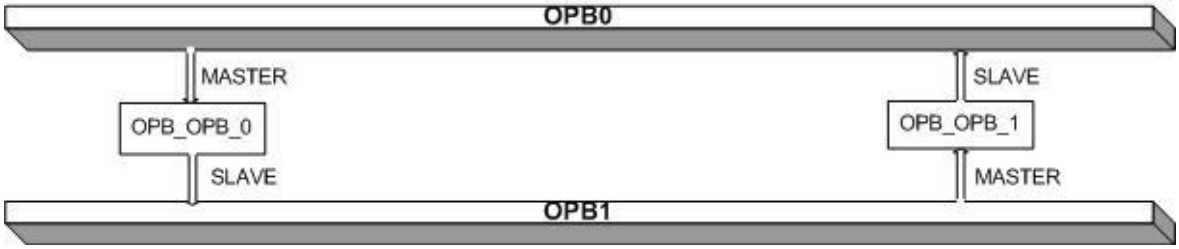

Figure 3.14 OPB to OPB Bridge Interconnection

In Figure 3.14 presented in details the interconnection of the bridges with the OPB buses. The OPB_OPB_0 bridge acts as master in OPB0 and as a slave in OPB1, while OPB_OPB_1 acts as master in OPB1 and as a slave peripheral in OPB0. In this way a processor which is connected in OPB1 can write data in OPB0 through the OPB_OPB_1. At this point it should be mentioned that as the OPB to OPB bridge acts a master in the one side of a bus, the number of processors, or other master peripherals in general, that can be connected on a bus is reduced to 7. This is the reason why in this project every bus holds only 7 processors and not 8.

The communication between two processors that are on a different bus, is not made with the use of a specific macro, but with message sharing with the use of the shared BRAM 0, that is connected in OPB0, and the shared BRAM 1, that is connected in OPB1. For example, if Microblaze 0 wants to send a message to Microblaze 7 (these processors are on a different bus), has only to write in shared BRAM 1 and then the Microblaze 7 reads the data from shared BRAM 1. The same stands for the opposite.

3.2.5 Slave Peripherals on OPB

As it was previously mentioned, on the OPB are also connected two shared BRAMS, one on each bus. These BRAMS (Shared BRAM 0, Shared BRAM 1) are connected as slaves on the buses, and they act as shared memories for the processors in order message passing to be possible. Every processor can access any time to every BRAM and to write or read data. The priority for accessing to these memories is controlled by the OPB arbiter that is embedded in the OPB bus.

Also, a serial port peripheral is connected on the OPB bus. This peripheral is used only for test reasons (if the design would be downloaded on the XUP board).

3.2.6 FSL peripheral to SRAM controller interconnection

Every FSL peripheral, as it is presented in figure 3.15, is connected to the SRAM controller, which is inside in the FPGA and is connected to the external SRAM.

In Figure 3.15 the interconnection of an FSL peripheral with the SRAM controller is presented.

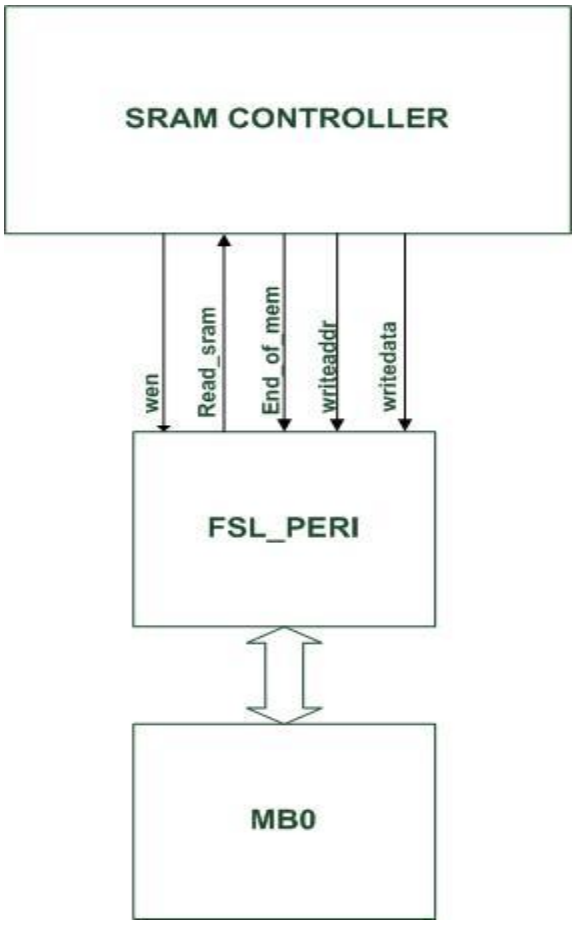

Figure 3.15 FSL Peripheral with SRAM Controller Interconnection

Every FSL Peripheral is connected with the SRAM controller via 5 signals, as there was no library bus provided from Xilinx, because the SRAM controller is custom. The functionality of these signals is described in the previous chapter of this thesis.

As it can be conducted from here, the processors are not depended on the SRAM controller and thus they can process data while the FSL peripheral communicates with the SRAM for its initialization.

3.2.7 SRAM controller to SRAM interconnection

As it was previously mentioned, the external SRAM is used must be supported from an SRAM controller that is inside the FPGA chip. So, we implemented an SRAM controller, which is inside the chip. The interconnection of this controller with the external SRAM is presented in figure 3.16

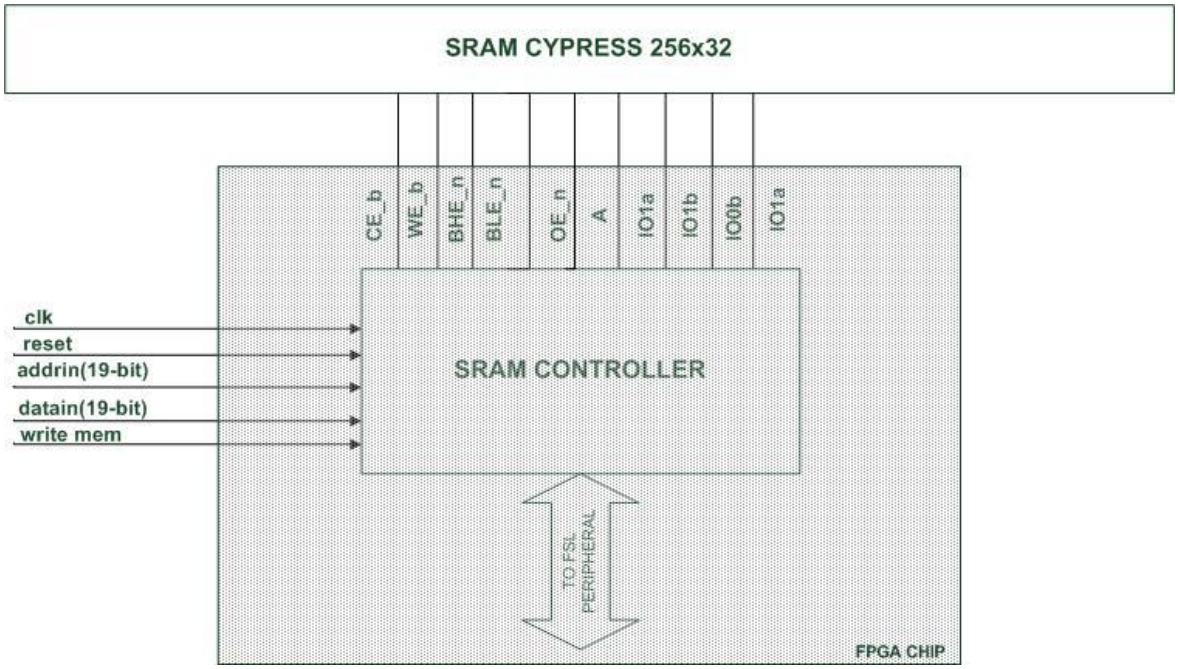

Figure 3.16 SRAM controller to SRAM interconnection

The clock and the reset signals of the SRAM controller are the same as for the rest of the platform. The signals “addrin”, “datain” and “write_mem” are external inputs of the controller and are used for initializing the SRAM. In this thesis, the SRAM is never initialized manually but is initialized in the testbench for the simulation of the platform.

The functionality of the SRAM and the SRAM controller was described in the previous chapter. The SRAM controller defines the way the SRAM communicates with the rest of the system. The reason an SRAM was chosen for this platform and not a DDR memory is that the implementation of a DDR controller is very complicated (which is not included in the goals of this thesis). In the future a DDR controller may be implemented to support fast transfer of the data to and from the rest of the platform. The only reason an SRAM was chosen is that it's easy to implement an SRAM controller and the I/O communication is much faster that any other way of communication (serial port for example).

3.3 MPLEM Functionality

The multiprocessor platform that previously described works as follows: Initially, the system is at a reset state. After some cycles, the software of the processors, which is the same for every processor, begins to execute. The SRAM controller will begin to work, only when all the processors ask for data from the memory. This happens, because we want the data from the external SRAM to be divided uniformly among the processors. Then the SRAM will initialize all the memories of the FSL peripherals. When every memory of the FSL peripherals would have been initialized, then the corresponding processor starts to read the data. When it finishes the initialization, the FSL peripheral is ready to be reinitialized. When all the peripherals are ready, the SRAM starts to send new data to these memories. At the same time, the processors execute the rest of the software. This process is repeated until all data of the SRAM are sent to the FSL peripherals. In this way, all the processors work in parallel by processing different data at the same time.

The SRAM initialization can be done externally, in case the platform is downloaded on a board. In this thesis on the contrary, the memory initialization is done only in simulation mode, since the model of the memory is available to us only in behavioral mode. So, as we need 32-bit data long, we use 2 instances of the model we have, connected in parallel. As a result, the total amount of memory that is available is 262144 positions by 32 bits long. For the initialization of the memory in behavioral mode, a procedure is created which reads the data from a memory initialization file (MIF) and stores those data in the array which stands for the memory. This file is read by the memory, when a read operation is made. Finally, in order the SRAM model to work properly, the switching characteristics of

this behavioral model are set, according to those described in [20]. The switching characteristics, which are chosen for the behavioral simulation of this platform, are those presented in table 4.1:

Parameter	**Description**	**Time (ns)**
Trc	Read Cycle Time	20
Taa	Address to Data Valid	1
Toha	Data Hold From Address Change	1
Tace	(not CE) LOW to DATA Valid	1
Tdoe	(not OE) LOW to DATA Valid	0
Tdbe	Byte Enable to Data Valid	3
Thzbe	Byte Disable to High Z	6
Thzoe	(not OE) HIGH to High Z	3
Thzce	(not CE) HIGH to High Z	3
Twc	Write Cycle Time	20
Tsce	(not CE) LOW to Write End	20
Taw	Address Set-Up to Write End	20
Tha	Address Hold from Write End	0
Tsa	Address Set-Up to Write Start	0
Tpwe	WE Pulse Width	20
Tsd	Data Set-Up to Write End	15
Thd	Data Hold from Write End	0
Tbw	Byte Enable to End of Write	20

Table 3.3 SRAM Switching characteristics

3.4 Parallel Interconnection

The number of processors is limited for this implementation due to the size of the target FPGA (we used the Virtex-II Pro xc2vp30 FPGA). In order to increase the number of processors, and consequently increase the throughput and the runtime performance of the system, we can do two things:

- Use a larger FPGA
- Connect many FPGAs in parallel.

Regarding the first idea, the topology of the network will not change significantly. We will have to use more OPB buses, since, as it was previously mentioned, the number of master peripherals on every bus is limited to 8. In figure 3.16 is presented the interconnection when using a larger FPGA:

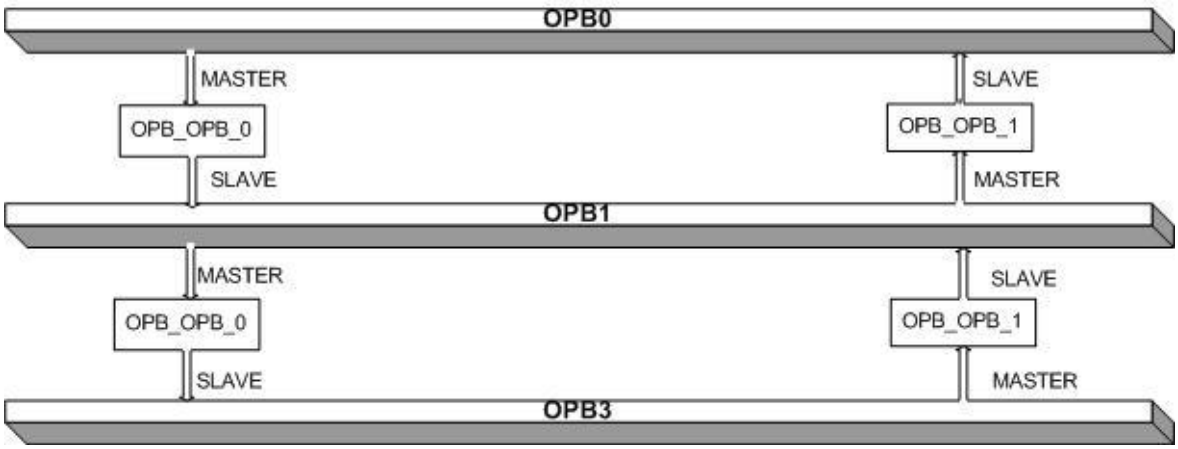

Figure 3.17 Multiple buses interconnections

At this case, the number of processors in OPB0 will remain the same (7 processors). On the contrary, the number of processors in OPB1 will be reduced to 6, since one master position should be occupied by the OPB_OPB_0 bridge. The number of processors in OPB3 will be the same as in OPB0 (7 Microblaze processors). So, the total number of processors in this case is 20. Likewise, if the size of the FPGA does not limit us, we can add more buses and, consequently, more processors.

Another way to increase the total number of processors is to connect many FPGAs in parallel. In figure 3.18 is presented one way to connect many FPGAs in parallel.

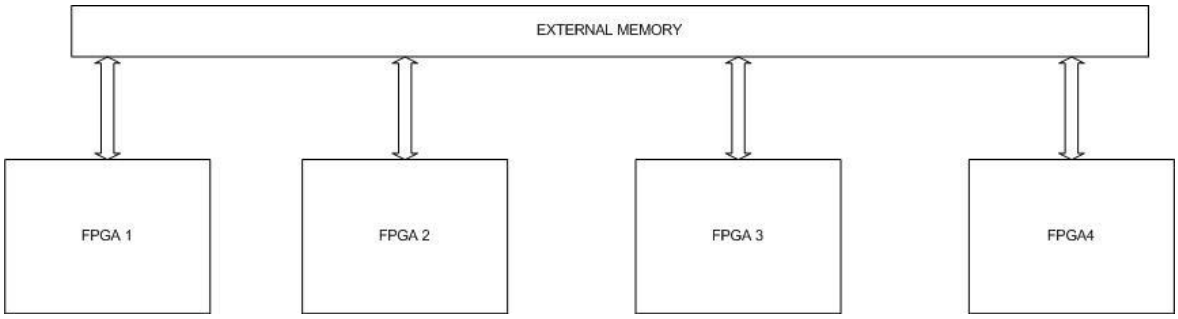

Figure 3.18 Parallel FPGA Interconnection

The main idea in order to connect many FPGAs, which run MPLEM on them, is to connect them with the use of the external memory. As the memory is outside the FPGA, we can connect many of them with the memory. The arbitration is done by the memory, which divides the data among the processors.

In order to implement the parallel interconnection, is just to put out the signals from each FPGA that are connected to the external memory.

With this implementation, we increase significantly the number of processors. Despite that, the number of FPGAs that are connected in parallel is unlimited. The only disadvantage is the bottleneck from the external memory. It is very slow to initialize all the processors so if we connect many FPGAs in parallel, the throughput will remain the same, the runtime performance too.

4. System Verification and Synthesis

In order to verify the correct functionality of the platform, initially all the custom peripherals were simulated and verified. After that, the whole platform functionality was verified, by implementing simple software for the platform. All the verification was made with behavioral simulation at ModelSim 6.0a.

4.1 SRAM Behavioral Model Verification

The model of the SRAM is used in this thesis is provided by Cypress. As a consequence, it should have been verified for its proper functionality. Despite that, in order to ensure the correct functionality of the memory and, as a result, the correct functionality of the system, the model of the SRAM is verified in behavioral simulation with ModelSim 6.0a. Many testbenches were created, which examine the most cases for reading from and writing to the memory. A testbench is created with no values, in order to examine whether the SRAM works properly and reads no values or not. Also a testbench with values in different positions was created so as to examine if we can read or write at random places in the memory. Finally, a testbench was created in order to examine whether or not we could read all the positions in this specific model of memory.

Also, at this point the correct use of the memory initialization file is verified. As it was previously mentioned, a memory initialization file was created with some initial values for the memory. What was tested is if the memory was initialized properly and, of course, if theses values appear in the output when a read operation is tried.

4.2 SRAM controller Verification

The SRAM controller that is implemented to support the functionality of the SRAM was also verified through behavioral simulation. A testbench was created, which examines all the cases for the SRAM controller. The behavioral simulation was done with the use of ModelSim 6.0a.

Besides the standalone simulation of the controller, it was also tested with the SRAM, in order to verify the correct communication between the two peripherals. For this reason a test bench was created that does many read and write

operations to and from the memory. In order to examined the correct functionality of the SRAM controller with the memory, we examine the output from the Modelsim in order to verify that the SRAM returns the correct values and also to verify if we write in some positions in the memory we can later read them.

4.3 FSL Peripheral Verification

Another peripheral that was verified is the FSL peripheral. In order to simulate this peripheral, the local BRAM is initialized with some random values with the use of a memory initialization file (MIF). The testbench that was created tries to replace the communication with the Microblaze processor. Some requests are made to the FSL peripheral, and this returns the values that are stored in the local BRAM. The cases that were tested are for many successive reads and for full or not full FSL.

4.4 Multiprocessor network Verification

The Microblaze network, without the external SRAM and the SRAM controller was also verified for its correct functionality. In order to simulate this network, simple software was developed. This software asks for some data from the FSL peripheral, reads the returned values (communication between Microblaze and FSL peripheral testing), and sends some of them through the OPB bus to other processors and to other slave peripherals (communication between processors testing).

4.5 MPLEM Verification

Finally, it was made the verification of the whole system. Initially we examined the correct communication between the processors. So testing software was created, that runs on all processors, and its main work is to exchange values between the processors. The first processor sends a value to every other processor and waits for their response. When a processor receives this value adds a number to this value according to its position in the network and sends it back to the first processor. The output from the simulation tool (Modelsim) is the same with the one expected. We can see that every processor replies to the first through the OPB bus.

Besides that, the communication with the FSL peripheral and the SRAM controller should be verified. So testing software was created, that runs on every Microblaze that asks for some values from its own FSL peripheral. The way these requests are made is with the use of some functions that are provided from the Xilinx software library. So, we have to examine in the simulation tool if the processor reads correctly the values from the FSL peripheral. This data come from the external memory with the help of the SRAM controller.

In every case, the system works properly, without an algorithm running on it. In chapter 5, after embedding the BLAST software in MPLEM, is described also a further verification for MPLEM. Many different simulations were made in order to examine not only the correct functionality of the BLAST software on the MPLEM but also to ensure the correct functionality of the platform itself on a parallel algorithm.

Another way for verifying the correct functionality of the platform is to download the design on an FPGA chip, but now this is impossible since the model of the memory is not synthesizable.

4.6 Synthesis Results

As it was previously mentioned, the whole platform cannot be synthesized, since the model of the memory we used in this thesis, is behavioral. Nevertheless, the part of the design that consists of the processors (without the memory and the memory controller) can be synthesized. As a result, we can predict the overall speed of the design, without taking into account the speed of the external memory. This is feasible, since the memory is outside the FPGA chip (it's a separate chip).

The processors network has been implemented (until synthesis phase) with Xilinx EDK 7.1 and Xilinx ISE 7.1 tool, using as target FPGA the xc2vp30 of the Virtex-II Pro family, package ff896 with speed grade -7. Table 4.1 shows the design summary of the multiprocessor network system.

Device Utilization Summary		
Number of 4 Input LUTs	23010 out of 27392	84%
Number of BRAMs	120 out of 136	88%
Number of MULT18X18s	42 out of 136	30%

Table 4.1 Synthesis Results

The speed of the system is 96.483 MHz according to the synthesis report obtained from the Xilinx Tools.

As the Table 3.4 shows the critical aspects of this design are the number of LUTs and the number of BRAMs. As a result, the maximum number of processor that can fit into a Virtex 2 Pro FPGA is 14 as this number is determined by the number of the available BRAMs and the number of the available LUTs.

On a new technology and larger FPGA the number of processors can be increased significantly since the amount of the available resources would be increased.

5. BLAST Implementation

In this chapter is described a brief overview of the BLAST algorithm and its use in molecular biology. At this point it should be mentioned that in this thesis all the terminologies and definitions of DNA sequence matching problems and BLAST algorithm are not described in details, since this is not the main goal of this thesis. All the terminology and definition details that are used in this thesis are described in detail in [2].

Besides that, the contents of this chapter deal with BLAST software. After distinguishing the different software programs according to the form of the processed data, the BLAST-n program of the popular NCBI software is presented. Also, we describe the implementation of a software system we developed from scratch, running the main BLAST-n machine and the verification we made by comparing the output of this machine to the output of the NCBI tool. Last but not least, it is described the reason for choosing this particular algorithm to be applied on a multiprocessor platform and the way this software is embedded in it.

5.1 Brief Description of BLAST Algorithm

BLAST is the acronym of Basic Local Alignment Tool and it was firstly presented in [2] by S.F Altchul et. al. in 1990. In order to remain consistent with the terminology found in the original paper, it is important for us to describe the basic terms.

A segment is a substring of a sequence. Given two sequences, a segment pair is a pair of substrings of the same length, one of each sequence. The subsequences of a segment pair can be gaplsessly aligned as there are of the same length.

Given a scoring scheme for DNA sequences, a + 5 for every match and a penalty of –4 for every mismatch, a Maximal Segment Pair (MSP) is defined to be the highest scoring pair of identical length segments chosen from two sequences. An MSP may be of any length as its boundaries are chosen to maximize its score. This score provides a measure of local similarity for any pair of sequences. However, as a *molecular biologist may be interested in all conserved regions shared by two proteins, not only in their highest scoring pair,* a segment pair is defined to be locally maximal if its score cannot be improved either by extending or

by shortening both segments. BLAST can seek all locally maximal segment pairs with scores above some cutoff [2].

We now have all the necessary information to describe precisely the performance of BLAST. Given a query sequence, BLAST returns all the segment pairs between the query and the database sequence with scores above a certain score S. Most severs running the BLAST software provide a default value of S, but also a user may also define a value for S.

Blast algorithm consists of 3 steps which implementation depends on the form of the data processed, nucleotide sequences or amino acid sequences. In the following discussion, BLAST dealing with nucleotide data will be discussed in details, whereas shorter explanations will be given regarding the manipulation of amino acid data.

The first step of the algorithm involves the compiling of the list of high scoring words. For DNA sequences this list contains all contiguous w-mers, i.e words of length w, in the query sequence. For nucleotide sequences, the value of w is usually 12 and a typical range of this value is between 11 and 15. Obviously this list will contain n-w+1 w-mers where n is the length of the query sequence. To better illustrate the algorithm steps for DNA sequences, we will use a smaller value for w in our examples. Let *ACGTAAATGCAG* be the query sequence of length 12 and let w be equal to 3. The word list will contain 10 w-mers. As it is shown in figure 5.1, *ACG* will be the first one, *CGT* the second, *GTA* the third etc. and *CAG* will be the last one.

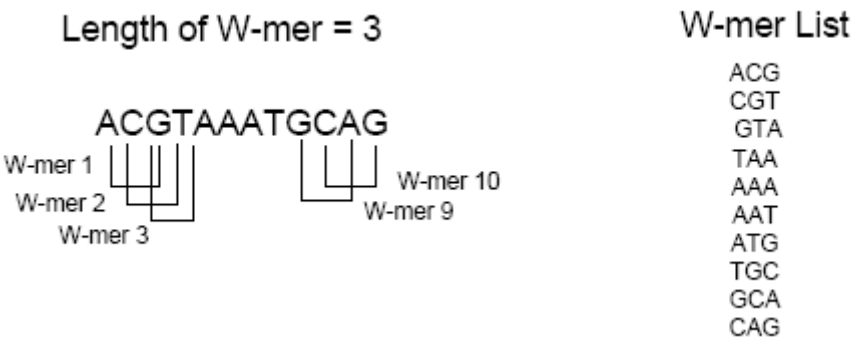

Figure 5.1 Step 1 of BLAST (from [15])

For queries with protein sequences containing all words which score at least T when compared with some word in the query sequence. So, a query may be represented by no words in the list or by many.

The Second Step is the search of the database for “hits”. After the word list generation, the database sequences are searched for an exact match between any substring of the w-mer list and the database sequence. Every word list found in the database is called hit and it is possible to be part of a High Scoring Pair (HSP). Figure 5.2 shows an instance of the execution of step 2, when the incoming database stream matches with a word of the lit list.

Figure 5.2 Step 2 of BLAST

As soon as a hit is identified, in a straightforward process, not differing in case of nucleotide or amino acid data, it is extended by the step3 of BLAST for finding a locally MSP. In the original BLAST paper it is stated that for timing reasons the process of extending in one direction is terminated when a segment pair which score is below a certain distance below the best score found for shorter extensions is reached. According to this paper, the added inaccuracy is negligible.

Figure 5.3 shows step by step the extension of the hit found in figure 5.2. In the first iteration of the extension process there are matches in both extension directions, so the score increases by 10. In the second and third iterations there is a match only in the one extension direction so the score in both iterations is increased by one, as each match yields 5 and each mismatch is penalized with -4. In the fourth iteration, there are mismatches in both directions and the score should be decreased by 8. As the score decreases in this iteration, the extension process stops without taking into account the last iteration.

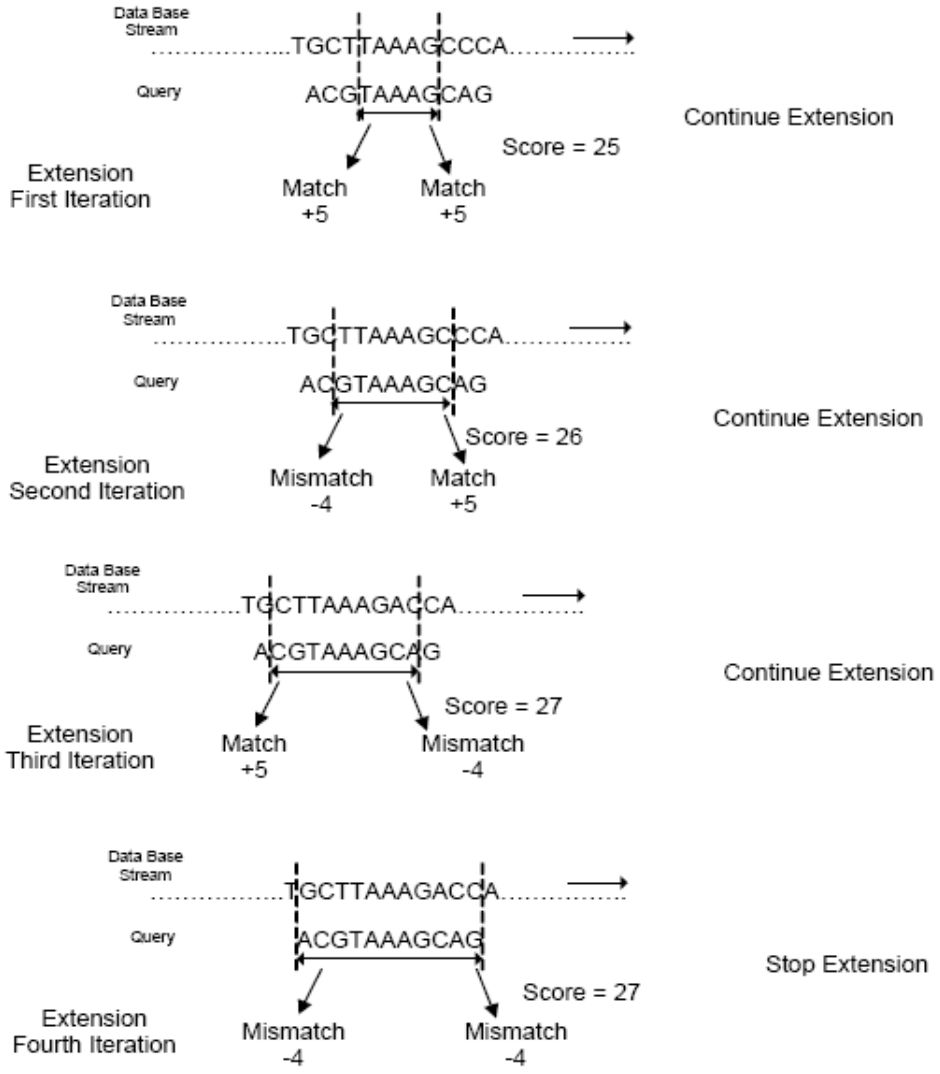

Figure 5.3 Step 3 of the BLAST Algorithm

5.2 The Different BLAST Programs

The BLAST algorithm is employed by the programs blastp, blastx, blastn, tblastn, tblastx. Their differenced are summarized in table 5.1.

Program	Description
blastp	Query: amino acid, Database: amino acid
blastn	Query: nucleotide, Database: nucleotide
blastx	Query: translated nucleotide sequence, Database: amino acid
tblastn	Query: amino acid, database: translated nucleotide sequence

tblastx	Query: translated nucleotide sequence, database: translated nucleotide sequence

Table 5.1 The different BLAST programs

In the current thesis for simplicity, as it is explained later, it is the blastn program that is used and all the other programs are disregarded.

5.3 The NCBI Implementation

Since 1988 the National Center for Biotechnology Information (NCBI) [3], created public databases, conducts research in computational biology, develops software tools for analyzing genome data, and disseminates biomedical information- all for the better understanding of molecular processes affecting human health and disease.

In this website there is an open source implementation of the BLAST algorithm while from the ftp pages of the Center there is available the genetic database which consists of numerous files containing biological data. In addition, there is available a web application which performs biological search in a certain application. For benchmarking reasons we downloaded the executable files of version 2.2.10 of the implementation. The details of the NCBI tool are described in details in [24].

5.4 Dimensioning

Before continuing with any implementation, it is essential to discuss the different sizes of the queries and the databases. In [24] there is a concise description of the available sizes of the queries and the databases. As it is described in this paper, databases are classified by looking the size of a particular database, either in terms of characters or in terms of megabytes.

There are three different cases; small, medium and large. Small consist of 400 sequences or 4.7 MB, medium is between 400 and 6000 sequences or 5 MB and 200 MB and large is between 6000 and 200000 sequences or 200 MB and 4 GB.

On the other hand, the type of query is classified by the number of sequences (single or multiple) and by the total number of characters involved in the query (small, medium and large). In the case of a single sequence, small corresponds to

less than or equal to 2000 characters, medium is between 2000 and 50000 characters, and large is between 50000 and 200000 characters. Multiple sequence queries were classified by the number of characters and by the total number of sequences per query. For multiple sequences; small corresponds to less than or equal to 2000 characters and a total of 20 sequences or less per query; medium is between 2000 and 50000 characters and between 20 and 200 sequences query; large is between 50000 and 200000 characters and between 200 and 2000 sequences per query.

Last but not least is the size of the w-mers we will produce. According to NCBI manual, the most frequent values for the blastn program vary between 11 and 15, while the default value for this implementation is the 11.

In this thesis, we use w-mer length 12, query length 1000 and small database sizes, since the size of the memory that is available to us is small.

5.5 Software Implementation

As it was previously mentioned, we have chosen to implement in software the blastn algorithm, for simplicity reasons. This software is implemented in C, compared with the NCBI tools for validation reasons, and later is used as the application for our multiprocessor platform. In figure 5.4 is presented the flowchart of our software implementation.

The database for this implementation, for our convenience must be not only translated according to the table 5.2 but also all the comments should be removed in order to be in readable format. In table 5.2 is presented the mapping of the database letters.

Nucleotide Character	Number Conversion
A	1
T	2
C	3
G	4

Table 5.2 Mapping of the DNA letters with numbers

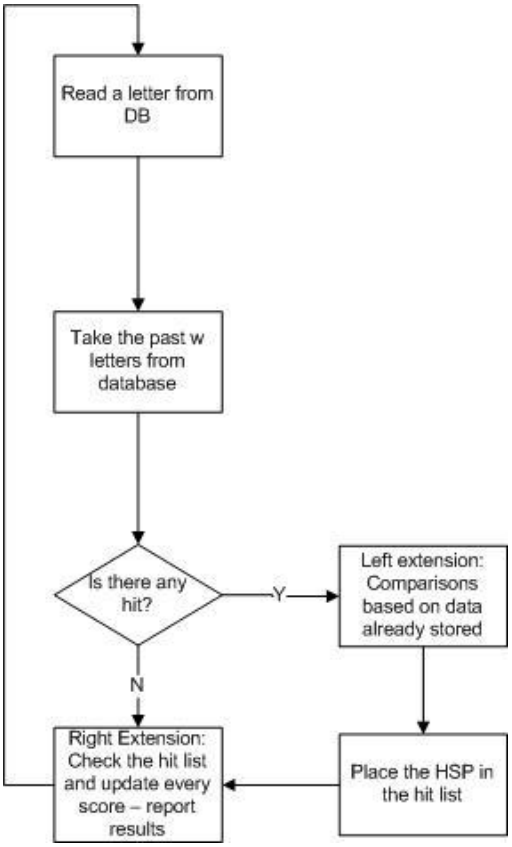

Figure 5.4 Flowchart of blastn software implementation

The software application we created in C language takes input a query file and a database file in readable format. The output is a list of all HSPs produced by the BLAST process and their scores without considering any statistical, pre-filtering or overlapping issues. The scoring matrix used is presented in table 5.3.

	A	**T**	**C**	**G**
A	5	-4	-4	-4
T	-4	5	-4	-4
C	-4	-4	5	-4
G	-4	-4	-4	5

Table 5.3 The scoring matrix of our software implementation

According to table 5.3 the score for each match is 5 while the score for each mismatch is -4. The size w of the w-mers is fixed and equal to 12 while the threshold value T is 50. Obviously a seed (i.e. a portion of a database which

produces a hit) is scored with 60, as there are 12 letters matching with a w-mer and scored with 5.

Now we will briefly describe the implementation details of each step of the algorithm.

Step1: Initially the query file is scanned and its content is stored in an array as integers 1, 2, 3, 4. After that, we scan this array and create a linked list of the w-mers. We remind that this list consists of all possible words of length w that lie on buffer. For the implementation of the list, dynamic memory allocation has been used. Dynamic allocation was preferred because it is simpler to implement. As soon as step 1 finishes, we proceed to the database scanning.

Steps 2, 3: This stage of the algorithm is repeated until the entire database file is scanned. For each iteration, one letter is read from the database file. Then for each letter, a word with length w is created and it's compared against the list of the w-mers in order for hits to be found. If a hit is found we immediately proceed to the left extension while the score is greater than a threshold (we selected 50). In order for the left extension being possible, some of the last database letters have to be kept in the main memory. So, a buffer has been introduced with size double as the size o f the query where every new incoming letter is stored. When the left extension of an HSP finishes, we proceed with the right extension. However, as data for this process are not yet available, this HSP is added into a waiting list (called hit list) for the right extension. This hit list contains all the "active" HSPs, i.e. those HSPs whose extension has not finished yet. Obviously, HSPs are characterized by two integer numbers indication the starting positions of the HSP in the query and the database, its length and its score.

The right extension is slightly different. First of all we disregard the score of the left extension and we consider that the score is the one obtained from the seed. Then for each database letter we extend by one letter all of the nodes located in the previously stated hit list. If a score of any node falls under the threshold the right extension procedure terminated for this node and the HSP is reported.

For a better understanding of the procedure described for steps 2 and 3, we add the flowchart in figure 5.4

It should be admitted that this method is not really good, because the extension process is very weal. First we extend left until the threshold is met and afterwards we disregard this score for the right extension.

Comparing the speed of this unit with the NCBI BLAST tool, it is obvious that the latter is much faster for the same inputs is the same computer. This is obvious because we did not care about the performance of the software.

Regarding the outputs of both software tools, there were a lot of incompatibilities. The reason is that the NCBI tool performs a lot of optimizations as well it applies statistical methods in the output of its results. However, the outputs of the NCBI tool were included in the output of our implementation. For this reason we are confident for the functionality of this software.

5.6 BLASTn Software on MPLEM

Each Microblaze processor of our multiprocessor platform executes the custom BLASTn software tool we implemented, in order to compare the performance of this software on two different machines: on the multiprocessor platform and on a Personal Computer with a Pentium Processor. The performance comparison will be analyzed in the next chapter in this thesis.

First of all, we will have to explain the reasons for choosing this particular algorithm for our multiprocessor platform. It's easy to implement in software the custom BLASTn tool so the design time is very small. Besides that, the BLAST algorithm can be easily parallelized, in order every processor to execute the same software at the same time with the same results. To success this, every processor has at the same time, to compare the same query, with a different part of the DNA database. Besides that, this platform was not designed only for BLAST algorithm, so some changes have to be made in order to be possible the BLAST algorithm to be executed.

The only part of the multiprocessor platform that has to be parameterized is the SRAM controller. The SRAM controller has to send to every processor the 1000 next letters of the database to the FSL peripherals, in order the right extension to be possible.

So, the SRAM controller spits the database letters among the processors and at these letters are added 1000 more database letters, which is the worst case for the extension step.

The worst case conducts from the size of the query that the processors examine. If the query is 1000 letters long, and a hit is found in last positions of the database part that a processor examines then right extension must be done for this hit. The worst case happens when for this hit we have the maximum extension. For 1000 letters long query, there can be an extension up to 1000 letters.

Besides that, the software must be parameterized, in a way that the Microblaze processors can execute this software. First of all, the way the letters are read from the database must be changed. In a PC the database is stored in a text file. But a Microblaze processor can not read a file, due to the absence of a hard disk drive. So, the database, after is translated, is stored in the external SRAM. The initialization of the SRAM is done with the use of the MIFs for behavioral simulation. Except for that, the query is no longer stored in a text file, but in the processors memories in an array of integers. Every processor has the same query to compare against a different part of the database.

Regarding the results, those can be sending to the RS232 port. But, for testing reasons, the results are sent to the data BRAMS of each processor, because it's difficult to simulate the functionality of the RS232.

The rest software remained the same as the one that runs on a PC, so now a comparison in performance can be done (it will be described later in this thesis).

Before compiling the C code on the processors, a linker script is necessary to be made, in order a mapping of the memory to be done. This linker script is generated automatically by the Xilinx EDK.

5.7 System Verification

After the embedment of the software on the MPLEM, we are ready to simulate the platform in order to validate the correct functionality of the system. For this reason, we have loaded different databases to the external memory in order to examine all possible cases.

We have examined the following cases:

- No hits: we load in the external SRAM a small random database that consists of the four letters as it was previously described, and a query from which no hits with the sequence database are conducted.
- Query part of the database: we load in the external SRAM a random database. The query is a small part of this database and we examine then the hits that are found
- All the database the same letter: we create a small database which consists of the same letter. This case is not real, since there is no organism that all its genes are the same. We examine this database with a query that consists of the same letters. We expect that hits are found in all the positions of the database.
- Database part of a common true database: we load in the MPLEM a real database in order comparison with the NCBI tool to be feasible.

Those cases are the test benches we created in order to examine the correct functionality not only of the BLAST on the MPLEM, but also to verify the correct functionality of the MPLEM with a real parallel algorithm such BLAST.

The simulator that is used for the verification of our system is the Modelsim 6.0a with behavioral simulation since timing simulation is not feasible cause of the reasons that were previously described.

The outputs of our system for each hit are: the position of the hit in the query and the database, the score of this specific hit and the length of the hit. These outputs are then compared with the results that come up of the custom software we implemented. The test benches that are used as input for this software are the same with those used for the verification of MPLEM in order to be certain for the correct functionality of our platform. At this point we should mention that we can use this software as the validator tool since its functionality was compared with the NCBI tools.

6. MPLEM Performance

In this chapter, a performance comparison will be made between the BLASTn software on the multiprocessor platform, and the BLASTn software that runs on a common Intel Pentium 4 processor.

6.1 BLASTn Performance on PC

Measurements of the BLASTn software, that we implemented, have been made on conventional computers, with identical queries. Runs of BLASTn software were performed on a 3.2 GHz Intel Pentium 4 processor with 512 MB of memory, running Microsoft Windows XP, and the CPU usage was profiled with the Intel VTune Performance Analyzer 9.0. The database and the query are part of ecoli.nt database. Different measurements were made for different sizes of database, whereas the size of the query remained always the same (1000 sequences long). Every measurement was repeated five times. In table 6.1 are presented the results from the measurements on the PC.

Processor	Speed (MHz)	Database size (letters)	Query size (letters)	Execution time (sec)	Throughput (letters/sec)
Intel P4	3200	8192	1000	34.45	237.79
Intel P4	3200	14336	1000	59.86	241.16
Intel P4	3200	28672	1000	127.34	225.16
Intel P4	3200	57344	1000	236.97	241.98

Table 6.1 BLASTn Performance on PC

6.2 BLASTn Performance on MPLEM

Measurements of the BLASTn software have been also made on the multiprocessor platform we implemented. Every processor compares the same query with length 1000 sequences with a different part of the database. The database that was used is the same with the database used for the experiments on the PC. In Table 6.2 are presented the results of the experiments that have been made with the ModelSim 6.0a. Due to the long simulation time, in the

following table is presented only one measurement for a database 57344 letters long.

Speed (MHz)	Query size (letters)	Database size (letters)	Execution time (sec)	Throughput (letters/sec)	Speedup
96.49	1000	57344	12.67	4528.01	19.10

Table 6.2 BLASTn Performance on MPLEM

6.3 Throughput Comparison

In the following section we will present some measurements that show the variation of the system throughput when we change the number of processors of the system. Measurements of performance were made for 2, 4, 8, 10 and 14 processors. The database that was used for these measurements was 8192 letters long, while the query size was 1000 letters long. The measurements were made with behavioral simulation with the Modelsim 6.0a.

Number of processors	Query size (letters)	Database size (letters)	Execution time (sec)	Throughput (letters/sec)	Speedup
2	1000	8192	5.68	1442.10	6.06
4	1000	8192	4.14	1976.92	8.31
8	1000	8192	1.52	5371.61	22.85
10	1000	8192	1.26	6523.07	27.43
14	1000	8192	0.93	8777.34	36.91

Table 6.3 System throughput for different number of processors

In figure 6.1 is presented a diagram which shows how the throughput changes – increases in this case- when the number of the programmable processors is increased.

In figure 6.2 is presented a diagram which shows the estimated speedup increase for different number of Microblaze processors, compared with the performance of blast software on an Intel Pentium 4 processor @ 3.2GHz.

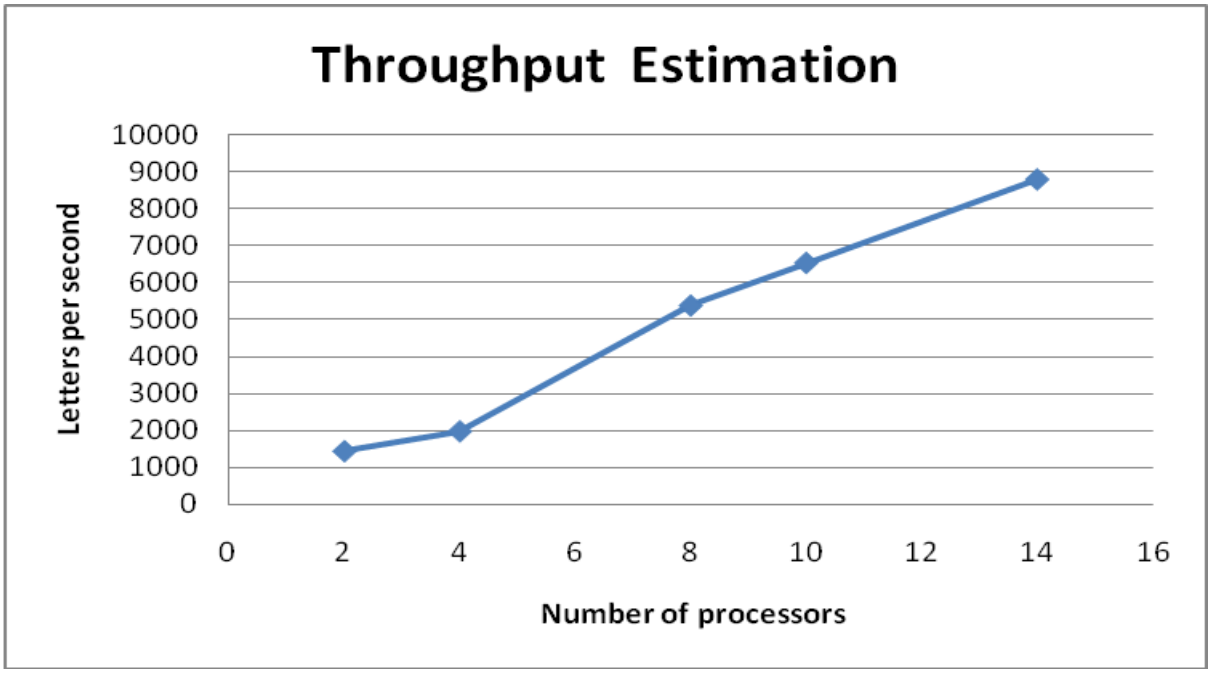

Figure 6.1 System throughputs for different number of processors

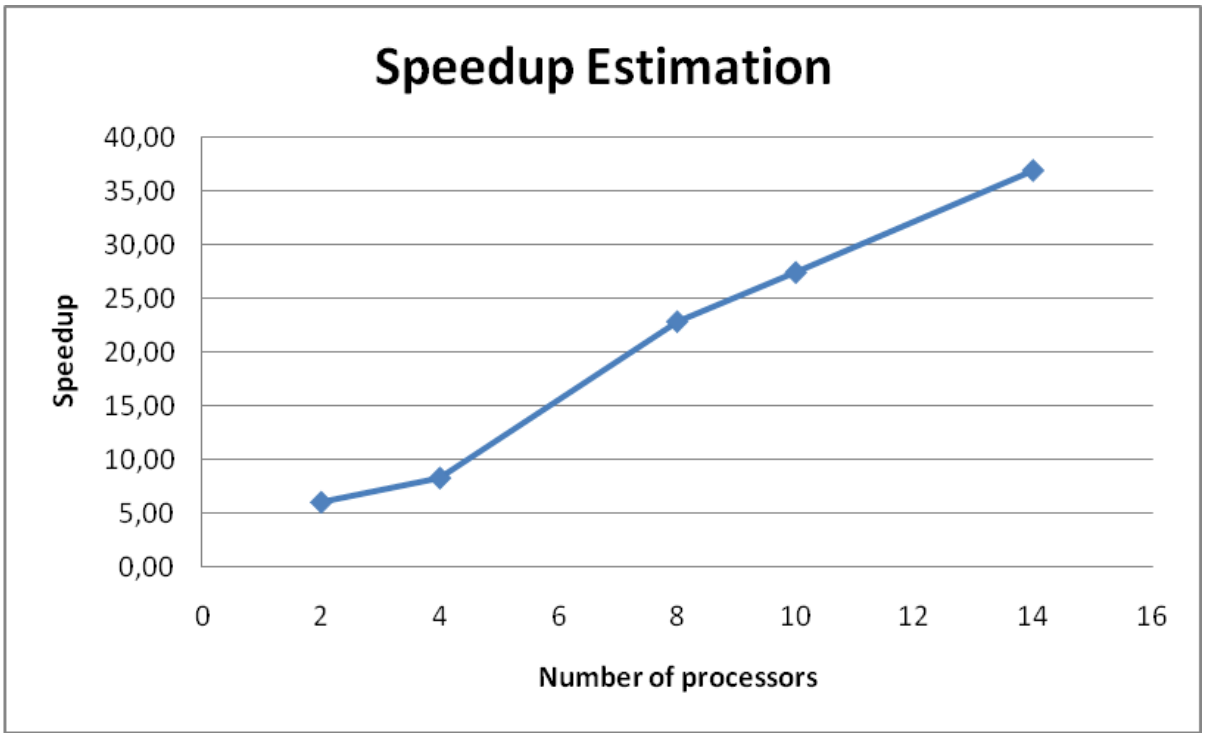

Figure 6.2 Estimated Speedup for different number of processors

6.4 Comparison

From the results that are presented in Tables 6.1 and 6.2, we can see that the BLAST-n machine is about 20 times faster that on a common PC. At this point we should mention that we have to make more measurements in order to be sure about the speedup that is achieved.

Furthermore, from table 6.3 we can see that the system throughput increases when the number of processors is also increased. The system throughput when the system consists of 14 processors is about 6 times bigger for the same database size and for the same query compared to 2 processors. Besides that, the speedup achieved when the system consists of 14 processors is about 37 higher compared to the BLAST software performance on a common pc.

7. Future Work

Many things can be done in order to improve the performance of this multiprocessor platform, since this is its first implementation. First of all, a significant increase of the speed can be achieved, by increasing the number of soft-core processors of the platform. In order to achieve this, a new FPGA chip has to be used, and the topology of the network has to be changed. Besides that, the IBM PowerPC processor that is embedded in many FPGA chips can be used, in order to solve some I/O difficulties.

Furthermore, the use of another SRAM chip can be studied, in order to increase the capacity and the speed of the system. At this point it should be mentioned that we can use a DDR memory instead of an SRAM in order to increase the capacity of the platform. Generally, DDR chips are faster and have more capacity than SRAM chips.

Also, in order to increase the throughput of the MPLEM, we can connect many FPGAs in parallel and as a result to achieve better performance for our system.

Finally, new topologies of the processors have to be studied, in order to find out the best architecture of this system.

Besides the performance issues that are necessary to be studied, we can also improve some other things on this design. First of all, it’s a good idea to download the whole design on a board and compare the results with those that come up form the simulation. Furthermore, performance comparisons with other BLAST machines (software or hardware) can be done, in order to have a better view of the performance of this platform. Finally, other algorithms can be implemented on this platform, in order to explore their performance on it.

8. References

Book Chapters

[1] J. Meidanis and J.C Setubal, "*Introduction to Computational Molecular Biology*", PWS Publishing Company, 1997 ch. 3.

[2] S. Altschul, W. Gish, W. Miller and E. Myers, "*Basic Local Alignment Tool",* Elsevier J. Mol. Biol., vol. 215, pp 403-410, 1990.

Web Locations

[3] **http://www.ncbi.nih.gov**
[4] **http://www.xilinx.com**
[5] **http://en.wikipedia.org**
[6] **http://www.model.com**
[7] **http://www.cypress.com**

Published Papers

[8] K. Ravindran, N, Satish, Y. Jin, and K. Keutzer. "*An FPGA-based Soft Multiprocessor System for IPv4 Packet Forwarding*". In International Conference on Field Programmable Logic and Applications (FPL), August 2005

[9] Chris Rowen, Tensilica Inc. "*Fundamental Change in MPSoCs: A fifteen year outlook. In MPSOC'03 Workshop Proceedings*". International Seminar on Application-Specific Multi-Processor SoC, 2003

[10] Wolf, W: "*The Future of Multiprocessor Systems-on-Chip*". Proceedings of the Design Automation Conference (DAC'04). 2004, pp. 681-685

[11] Wolf, W: "*Multimedia Applications of Multiprocessor Systems-on-Chip*". Proceedings of the Design, Automation and Test in Europe Conference (DATA'05). 2005. Pp. 86-89

[12] Sun Wei: *"A FPGA-based Soft Multiprocessor System for JPEG Compression*". Technical University Eindhoven. The Netherlands

[13] Kaushik Ravindran, Nadathur Satish, Yujia Jin, Kurt Keutzer: "*An FPGA-Based soft multiprocessor system for IPV4 packet forwarding*", University of California at Berkley, CA, USA.

[14] P. Huerta, J. Castillo, J. I. Martinez, V. Lopez: "*A Microblaze Based Multiprocessor Soc*", HW/SW Codesign Group, Universidad Rey Juan Carlos, Madrid, Spain.

[15] E. Sotiriades, C. Kozanitis, and A. Dollas, "*FPGA based Architecture of DNA Sequence Comparison and Database Search*", Reconfigurable Architectures Workshop (RAW 2006), 20th IEEE International Symposioum Parallel and Distributed Processing (IPDPS), 25-29 April 2006.

User Manuals

[16] Xilinx Inc., "*Platform Studio and EDK*",

[17] Xilinx Inc., "Xilinx *XUP Virtex-II Pro Development System*"

[18] Xilinx Inc., "*Processor IP Reference Guide*", Feb. 2005

[19] Xilinx Inc., "*Microblaze microcontroller Reference Design User Guide*", Sep. 2005

[20] Cypress, *CY7C1041 256K x 16 Static RAM*

[21] Xilinx Inc., *"OS and Libraries Document Collection"* July 2005

[22] Xilinx Inc., "*Microblaze Hardware Reference Guide*", Jan. 2002

[23] Xilinx Inc., "*Platform Studio User Guide*", Feb. 2005

Dissertations

[24] C. Kozanitis, "*Study and Implementation with Reconfigurable Logic of BLAST Algorithm for DNA Sequence Matching and Database Search*", Senior Thesis, Dept. of Electronics and Computer Engineering, Technical University of Crete, Chania 2006.