Technical University of Crete
Electronic and Computer Engineering Department

Development of a Parallel Network Processor in Reconfigurable Logic

Diploma Thesis of Raptis Grigoris

Committee:
Papaefstathiou Ioannis, Supervisor
Dollas Apostolos
Pneumatikatos Dionisios

July 2010

Contents

List of Figures

List of Tables

Abstract

Although chip speed has increased exponentially over the past years, nowadays the trend of increasing a processor's speed to get performance speedup seems to be a way of the past. The new direction for achieving faster processing is using multiple cores in one single chip, where the cores share the work load and work in parallel.

The purpose of this diploma thesis was the development and implementation of a 16-Core Parallel Network Processor, in Reconfigurable Logic (FPGA). It was based on a single-core network processor, developed on a previous diploma thesis [1]. The processor operates at 10, 100, 1000 Mbps Ethernet speeds. The design was implemented on a Xilinx Virtex 5 board.

Chapter 1

Introduction

A network processor (NP) is an integrated circuit (IC) which has a feature set specifically targeted at the networking application domain. Usually NPs have generic characteristics similar to general purpose central processing units and they are commonly used in many different types of equipment and products. For more information about NPs see [1], as from now on we will focus on multicore processors, as creating one, was the main purpose of this diploma thesis.

The first microprocessor was introduced in the early 1970's. Since then each generation of processors grew faster, smaller, dissipated more heat and consumed more power. This performance speedup was based on an increase in frequency as higher processor frequency meant a faster, more capable computer. This model broke when the high frequencies caused processors to run at speeds that caused increased power consumption and heat dissipation at detrimental levels.

That's when the idea for the Multicore Processor (MCP) was born. Adding multiple cores within a processor gave the solution. MCPs are running at lower frequencies so that the levels of heat dissipation and power consumption stay in reasonable limits. At the same time the multiple cores work in parallel, processing data faster than a Single Core Processor (SCP).

Of course nothing comes for free. Developing programs that use efficiently MCPs and developing new memory architectures so that the cores have always available the data they need, are questions need to be answered. Despite these rising challenges, MCPs are an important innovation in the microprocessor timeline and they sure are the new trend in processor developing.

Thesis Organization

The thesis is separated into chapters. There are six chapters each covering a different subject and leading naturally to the next one. Information given varies from general theoretical to the details of our design. Now we will give a brief description of each chapter.

In the second chapter we provide basic theoretical information. We start by giving the definition of a multicore processor. Next we explain basic terms and we provide information about core types and memory and memory interconnection. Last we analyze some of the most important issues we have to solve when designing a multicore architecture.

In the third chapter we refer to the related work done in the field of MCPs. We give information about network, general purpose and special purpose multicore processors developed in the commercial area by companies like Intel or AMD. Next we analyse the architecture of the CELL processor and last we refer to the academic research done.

In the fourth chapter we take a deep look in the architecture of our design and we describe the work done for this thesis. We analyse in deep the operation of each component of our processor starting from inner parts and moving on to the top module. We provide block diagrams, Finite State Machines (FSM) transitions and the interface of each module of our processor.

In the fifth chapter we describe the verification process and the performance of our processor. We analyse the software and hardware used during development and verification and we explain the way we tested our processor. Last we give the results about the performance of our design.

Last in the sixth chapter we propose some ways to make the design more efficient in the future.

Chapter 2

Theoretical Background

2.1 Multicore Basics

A multi-core processor is a processing system composed of two or more independent cores. One can describe it as an integrated circuit to which two or more individual processors (called cores in this sense) have been attached. The cores are typically integrated onto a single integrated circuit die (known as a chip multiprocessor (CMP)), or they may be integrated onto multiple dies in a single chip package. A many-core processor is one in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient - this threshold is somewhere in the range of several tens of cores - and probably requires a network on chip.

A dual-core processor contains two cores, a quad-core processor contains four cores, and a hexa-core processor contains six cores. A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device together tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, 2-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores, unlike heterogeneous multi-core systems. Just as with single-processor systems, cores in multi-core systems may implement architectures like:

- Superscalar
- VLIW
- Vector Processing
- SIMD
- Multithreading

Multi-core processors are widely used across many application domains including:

- general-purpose
- embedded
- network
- digital signal processing (DSP)
- graphics

The amount of performance gained by the use of a multi-core processor depends very much on the software algorithms and implementation. In particular, the possible gains are limited by the fraction of the software that can be parallelized to run on multiple cores simultaneously; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores. Many typical applications, however, do not realize such large speedup factors. The parallelization of software is a significant on-going topic of research.

2.1.1 Core Types

There are two types of cores used in a multicore environment. Heterogeneous and Homegeneous. Homogeneous cores are all exactly the same : equivalent functions, frequencies, cache sizes, etc. On the other hand heterogeneous cores run in different frequencies, have different functions etc. There is no answer about which system is the best, as for each there is a trade-off between processor complexity and customization. Each core in a heterogeneous system can have a special function supporting a certain instruction set, while homogeneous core processors are much simpler and easier to be produced.

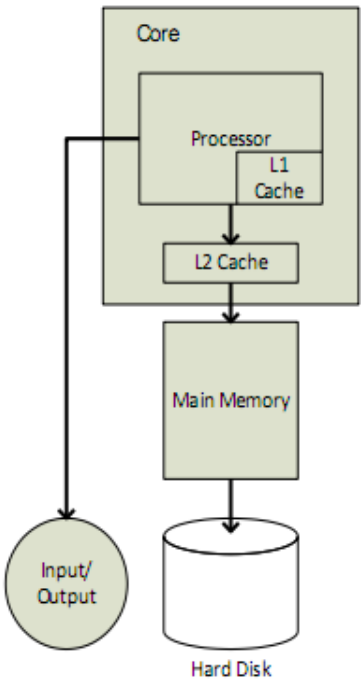
2.2 Memory System

2.2.1 Basic Memory Architecture

The following isn't specific to any multicore design, but it's a basic overview of multicore architecture. Although manufacturers have developed many different designs, architectures need to adhere to certain aspects. The basic configuration of a microprocessor is shown below:

]

Figure 2.1: MicroProcessor Basic Memory Architecture



A general memory rule is the closer to the processor the faster and smaller memory we have. Closest to the processor is level 1 (L1) cache. That is the fastest and smallest memory of the design used to store data frequently used by the processor. Next we have Level 2 (L2) cache which is off chip. It is slower and bigger than L1 cache but still a lot faster and smaller than the main memory. Next we have the main memory. Finally if the data needed are not found in cache or main memory the system must retrieve it from the hard disk.

2.2.2 Memory Size

With numerous cores on a single chip there is an enormous need for increased memory. Multicore processors must have more main memory and larger caches. Of course extra memory will be useless if the amount of time required for memory requests is not improved as well. Interconnection network between the cores are redesigned as faster network means a lower latency in inter-core communication and memory transactions.

2.2.3 Memory Interconnection

If we set two cores side by side there is a need for communication between the cores, and to the main memory. This is achieved with two basic techniques. Either using a single communication bus or an interconnection network. The basic architecture of the two techniques is shown below:

Figure 2.2: Single Communication Bus Architecture

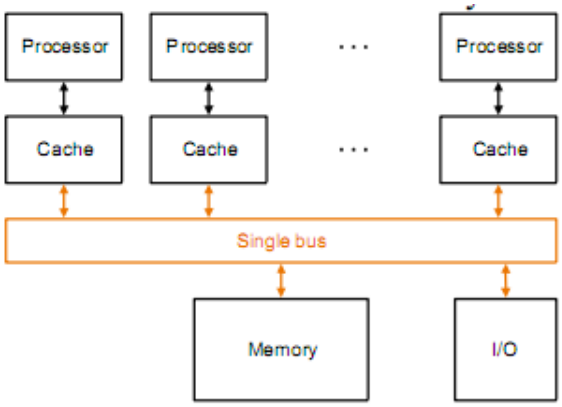
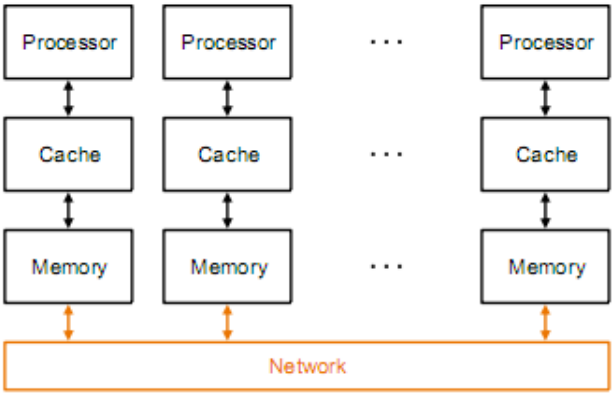


Figure 2.3: Interconnection Network Architecture



The bus approach is used with a shared memory model, whereas the interconnection network model is used with distributed memory. The scalability of a communication bus is limited as for more than 32 cores the bus is overloaded with the amount of data, something that leads to diminished performance.

2.3 Multicore Challenges

Having many cores on a single chip has many advantages but creates some important issues too. The most important one is the memory bottleneck, as providing data to all the cores is a challenging task. Power and temperature management is another problem. Finally applications must take advantage of multiple cores otherwise there is no gain and in some cases there is a loss of performance.

2.3.1 Memory Issues

Increased number of cores means increased need for data. Memory or memories must be fast enough to feed the cores with data in order to keep them busy. Cores must work in parallel so that workload is processed faster. When this is not possible cores will stay unused, waiting for data to be received instead of processing work load and causing a critical delay in the processor performance.

Another issue is memory coherence. Since each core can process and alternate data in memory when another core needs the same piece of data there is a risk that the data might not be the most up-to-date version.

2.3.2 Power and Temperature

If two cores were placed on a single chip, without modifications the chip would theoretically consume twice as much power. A way to prevent this problem is using power control units so that unused cores are powered off and so not consuming energy. Also cores run at a lower frequency to reduce power consumption.

Finally for avoiding the generation of an extreme amount of heat caused by multiple cores, the chip is architected so that the number of hot spots doesnt grow too large, and the heat is spread out across the chip.

2.3.3 Multithreading

The last issue is using parallel processing techniques to get the most performance out of the multicore processor. Programmers have to write applications with subroutines able to run in different cores. These applications must also be written in a way that the usage of cores is balanced too. Cores should be used as equally as possible achieving the parallel working of all cores. With multicore applications multicore efficiency could be increased dramatically.

Chapter 3

Related Work

We will now take a look on some companies and their related work concerning multicore processors for network, general and special use processing. We will describe the most successful example of multicore processing, the CELL processor. Finally we will present solutions given by academic research.

3.1 Commercial Work

3.1.1 Network Processors

Network processors need to become faster and faster in order to deal with high-data rates. In network processing, it is now mainstream for devices to be multi-core. The most important companies developing MCPs are :

- Freescale Semiconductor
- Cavium Networks
- Wintegra
- Broadcom

All these companies are manufacturing processors with eight cores.

3.1.2 General purpose processors

Intel and AMD are the mainstream manufacturers of microprocessors. Intel produces many different types of multicore processors:

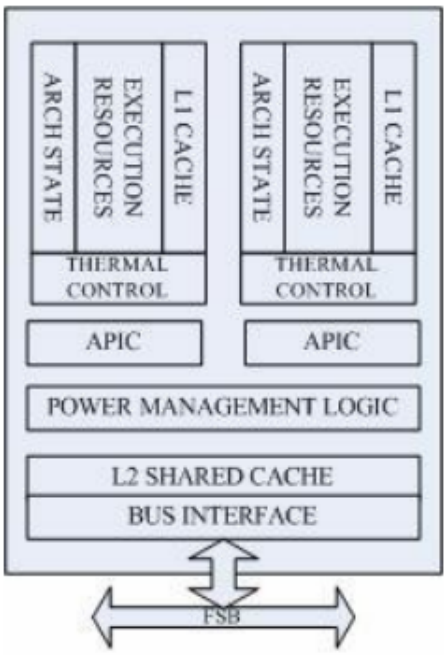
- Pentium D
- Core 2 Duo
- Xeon processor

Similarly AMD presents a full line of processors too :

- Athlon
- Turion
- Opteron

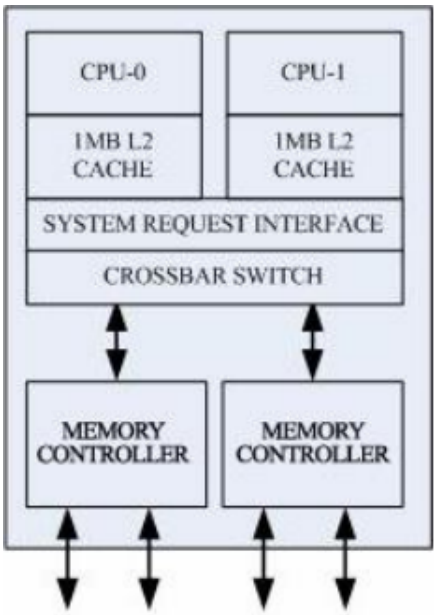
Pentium D, and Athlon are used for desktop computers, Core 2 Duo for both laptop and desktop and finally Xeon and Opteron for servers. Comparing two of the most popular models of each company - Core 2 Duo by Intel and Athlon 64 X2 by AMD - we notice that even though they run on the same platforms their architectures differ greatly. Figures show block diagrams for the Core 2 Duo and Athlon 64 X2, respectively. Both architectures are homogenous dual-core processors.

Figure 3.1: Intel Core 2 Duo Top Level Block Diagram



The Core 2 Duo adheres to a shared memory model with private L1 caches and a shared L2 cache which "provides a peak transfer rate of 96 GB/sec. If a L1 cache miss occurs both the L2 cache and the second core's L1 cache are traversed in parallel before sending a request to main memory.

Figure 3.2: Athlon 64 X2 Top Level Block Diagram



In contrast, the Athlon follows a distributed memory model with discrete L2 caches. These L2 caches share a system request interface, eliminating the need for a bus. The system request interface also connects the cores with an on-chip memory controller and an interconnect called HyperTransport. HyperTransport effectively reduces the number of buses required in a system, reducing bottlenecks and increasing bandwidth. The Core 2 Duo instead uses a bus interface. The Core 2 Duo also has explicit thermal and power control units on-chip. Thereis no definitive performance advantage of a bus vs. an interconnect, and the Core 2 Duo and Athlon 64 X2 achieve similar performance measures, each using a different communication protocol.

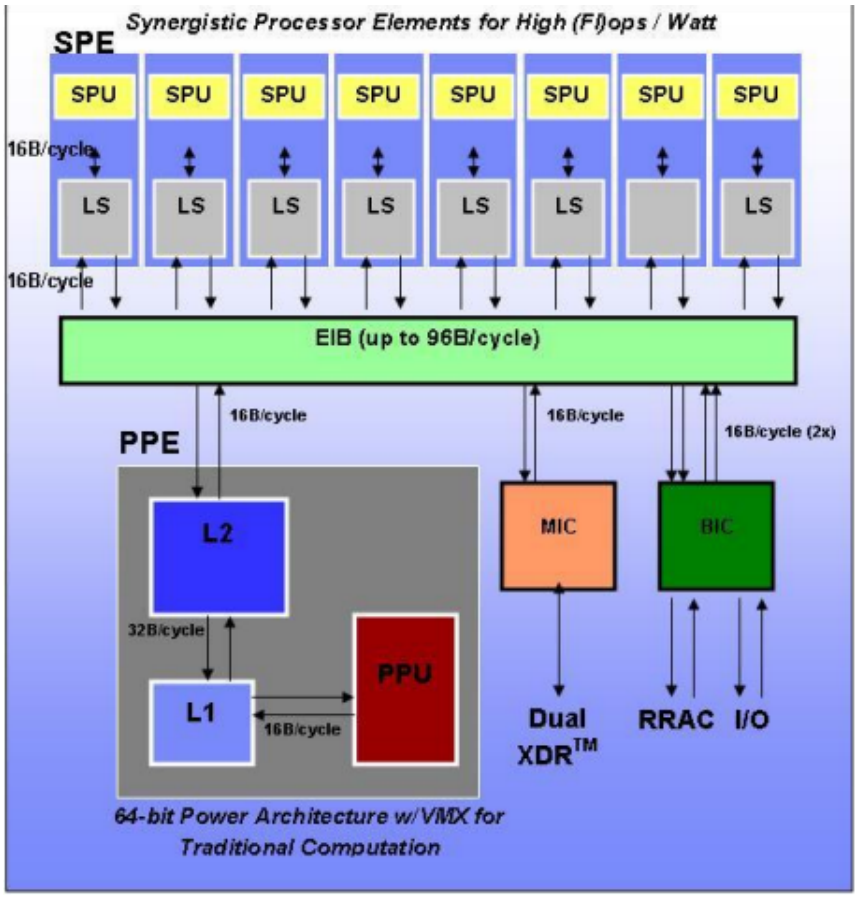
3.1.3 Special Purpose Processors - The CELL

A Sony-Toshiba-IBM partnership (STI) built the CELL processor for use in Sony?s PlayStation 3 therefore, CELL is highly customized for gaming/graphics rendering which means superior processing power for gaming applications.

The CELL is a heterogeneous multicore processor consisting of nine cores, one Power Processing Element (PPE) and eight Synergistic Elements. It was designed to bridge the gap between conventional desktop processors (such as the Athlon 64, and Core 2 families) and more specialized high-performance processors, such as the NVIDIA and ATI graphics-processors (GPUs). It may be utilized in high-definition displays and recording equipment, as well as computer entertainment systems for the HDTV era. Additionally the processor may be suited to digital imaging systems (medical, scientific, etc.) as well as physical simulation (e.g., scientific and structural engineering modeling).

In a simple analysis, the Cell processor can be split into four components: External input and output structures, the main processor called the Power Processing Element (PPE) (a two-way simultaneous multithreaded Power ISA v.2.03 compliant core), eight fully-functional co-processors called the Synergistic Processing Elements, or SPEs, and a specialized high-bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus or EIB. The structure of these four basic elements is shown at the block diagram below:

Figure 3.3: The Cell Architecture



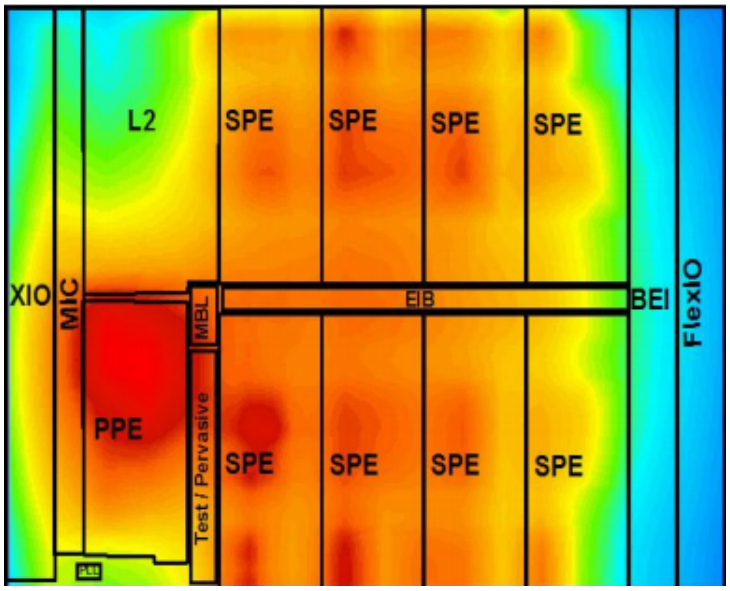
To achieve the high performance needed for mathematically intensive tasks, such as decoding/encoding MPEG streams, generating or transforming three-dimensional data, or undertaking Fourier analysis of data, the Cell processor marries the SPEs and the PPE via EIB to give access, via fully cache coherent DMA (direct memory access), to both main memory and to other external data storage. To make the best of EIB, and to overlap computation and data transfer, each of the nine processing elements (PPE and SPEs) is equipped with a DMA engine.

Since the SPE's load/store instructions can only access its own local memory, each SPE entirely depends on DMAs to transfer data to and from the main memory and other SPEs' local memories. A DMA operation can transfer either a single block area of size up to 16KB, or a list of 2 to 2048 such blocks. One of the major design decisions in the architecture of Cell is the use of DMAs as a central means of intra-chip data transfer, with a

view to enabling maximal asynchrony and concurrency in data processing inside a chip.[28] The PPE, which is capable of running a conventional operating system, has control over the SPEs and can start, stop, interrupt, and schedule processes running on the SPEs. To this end the PPE has additional instructions relating to control of the SPEs. Unlike SPEs, the PPE can read and write the main memory and the local memories of SPEs through the standard load/store instructions. Despite having Turing complete architectures, the SPEs are not fully autonomous and require the PPE to prime them before they can do any useful work. Though most of the "horsepower" of the system comes from the synergistic processing elements, the use of DMA as a method of data transfer and the limited local memory footprint of each SPE pose a major challenge to software developers who wish to make the most of this horsepower, demanding careful hand-tuning of programs to extract maximal performance from this CPU.

The PPE and bus architecture includes various modes of operation giving different levels of memory protection, allowing areas of memory to be protected from access by specific processes running on the SPEs or the PPE. Both the PPE and SPE are RISC architectures with a fixed-width 32-bit instruction format. The PPE contains a 64-bit general purpose register set (GPR), a 64-bit floating point register set (FPR), and a 128-bit Altivec register set. The SPE contains 128-bit registers only. These can be used for scalar data types ranging from 8-bits to 128-bits in size or for SIMD computations on a variety of integer and floating point formats. System memory addresses for both the PPE and SPE are expressed as 64-bit values for a theoretic address range of 264 bytes (16 exabytes or 16,777,216 terabytes). In practice, not all of these bits are implemented in hardware. Local store addresses internal to the SPU processor are expressed as a 32-bit word. In documentation relating to Cell a word is always taken to mean 32 bits, a doubleword means 64 bits, and a quadword means 128 bits. Other interesting features of the CELL are the Power Management Unit and Thermal Management Unit. Power and temperature are fundamental concerns in microprocessor design. The PMU allows for power reduction in the form of slowing, pausing, or completely stopping a unit.

Figure 3.4: Cell Heat Dissipation Overview



The TMU consists of one linear sensor and ten digital thermal sensors used to monitor temperature throughout the chip and provide an early warning if temperatures are rising in a certain area of the chip. The ability to measure and account for power and temperature changes has a great advantage in that the processor should never draw too much power or overheat as the heat is distributed equally along the surface of the chip.

3.2 Academic Research

3.2.1 Asychronous Array of Simple Processors

The Asynchronous Array of simple Processors (AsAP) is maybe the most important effort done in the academic field about multicore processors. It's architecture comprises a 2-D array of reduced complexity programmable processors with small memories interconnected by a reconfigurable mesh network. AsAP was developed by researchers in the VLSI Computation Laboratory (VCL) at the University of California, Davis and achieves high performance and high energy-efficiency, while requiring a relatively small circuit area.

AsAP processors are well suited for implementation in future fabrication technologies, and are clocked in a Globally Asynchronous Locally Synchronous (GALS) fashion. Individual oscillators fully halt (leakage only) in 9 cycles when there is no work to do, and restart at full speed in less than one cycle after work is available.

The multi-processor architecture efficiently makes use of task-level parallelism in many complex DSP applications, and also efficiently computes many large tasks utilizing fine-grain parallelism.

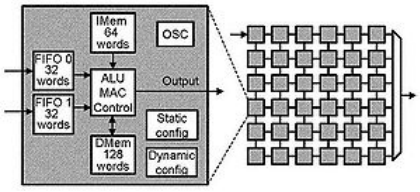
3.2.1.1 Key Features

- Chip multi-processor (CMP) architecture designed to achieve high performance and low power for many DSP applications.
- Small memories and a simple architecture in each processor to achieve high energy efficiency.
- Globally Asynchronous Locally Synchronous (GALS) clocking simplifies the clock design, greatly increases ease of scalability, and can be used to further reduce power dissipation.
- Inter-processor communication is accomplished using a nearest neighbor network to avoid long global wires and increase scalability to large arrays and in advanced fabrication technologies. Each processor can receive data from any two neighbors and send data to any combination of its four neighbors.

3.2.1.2 AsAP 1 - 36 Processors

A chip containing 36 (6x6) programmable processors was taped-out in May 2005 in 0.18m CMOS using a synthesized standard cell technology and is fully functional. Processors on the chip operate at clock rates from 520MHz to 540MHz at 1.8V and each processor dissipates 32mW on average while executing applications at 475MHz. Most processors run at clock rates over 600MHz at 2.0V, which makes AsAP among the highest known clock rate fabricated processors (programmable or non-programmable) ever designed in a university. It is the second highest known in published research papers. At 0.9V, the average application power per processor is 2.4mW at 116MHz. Each processor occupies only 0.66mm. The block diagram of the processor is given below :

Figure 3.5: AsAP 1 Block Diagram



3.2.1.3 AsAP 2 - 167 Processors

A second generation 65 nm CMOS design contains 167 processors with dedicated FFT, Viterbi, and video motion estimation processors. It contains 16 KB shared memories while using long-distance inter-processor interconnect. The programmable processors can individually and dynamically change their supply voltage and clock frequency. The chip is fully-functional while the processors operate up to 1.2 GHz at 1.3 V which is believed to be the highest clock rate fabricated processor designed in any university.

At 1.2 V, they operate at 1.07 GHz and 47 mW when fully active. At 0.675 V, they operate at 66 MHz and 608 W when fully active. This operating point enables 1 trillion MAC or ALU ops/sec with a power dissipation of only 9.2 watts. Due to its MIMD architecture and fine-grain clock oscillator stalling, this energy efficiency per operation is almost perfectly constant across widely varying workloads–which is not the case for many architectures.

3.2.1.4 Applications

The coding of many DSP and general tasks for AsAP has been completed. Mapped tasks include: filters, convolutional coders, interleavers, sorting, square root, CORDIC sin/cos/arcsin/arccos, matrix multiplication, pseudo random number generators, Fast Fourier Transforms (FFT) of lengths 32-1024, a complete k=7 viterbi decoder, a JPEG encoder, a complete fully compliant baseband processor for an IEEE 802.11a/g wireless LAN transmitter and receiver, and a complete CAVLC compression block for an H.264 encoder. Blocks plug directly together with no required modifications whatsoever. Power, throughput, and area results are typically many times better compared to existing programmable DSP processors. The architecture enables a clean separation between programming and inter-processor timing handled entirely by hardware. A recently completed C compiler and automatic mapping tool further simplify programming.

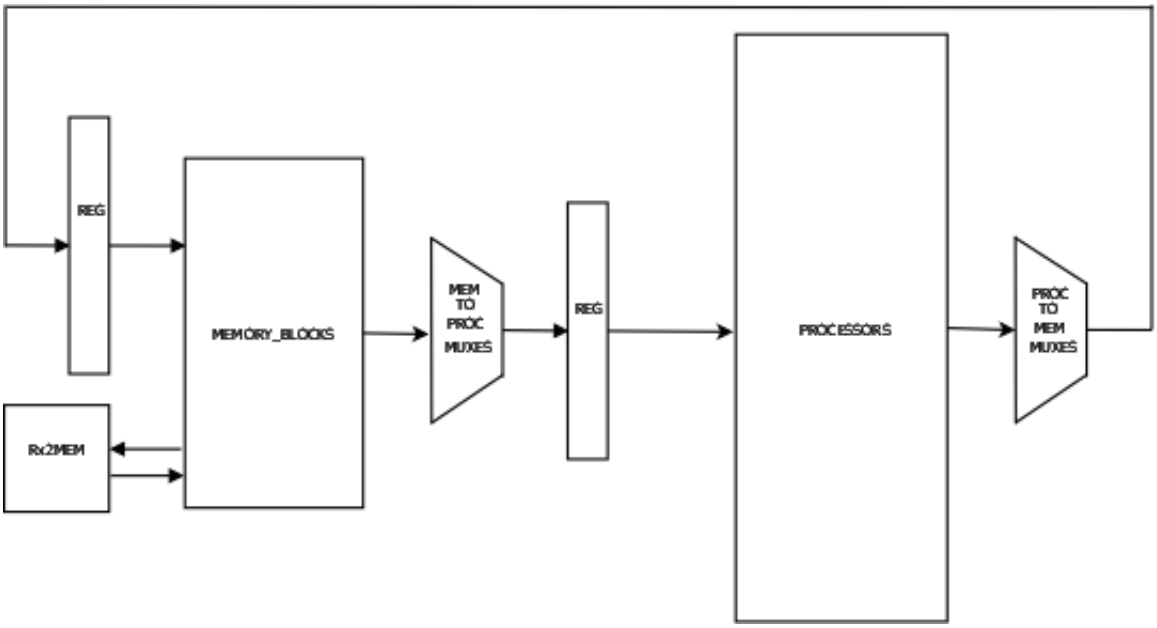
Chapter 4

Architecture

4.1 Architecture Overview

The parallel processor we designed consists of modules, each one responsible for the execution of a specific task. Our design can be seperated into 3 basic components. The first module is the Rx2Mem responsible for passing data, from the LocalLink (LL) to the Memory system. Next we have the Memory Blocks module which is the data centre of our design as it contains the memories needed for our design to work and also the logic for data transferring, from and to the cores. Last we have the Processors module which contains the 16 cores of our processor and the logic needed for their synchronisation and communication with the Memories. The Top Level Diagram of the processor is given below, showing the 3 basic components and the way they connect to each other.

Figure 4.1: Top Level Diagram



4.2 Architecture-In depth

4.2.1 Rx2Mem

The Rx2Mem module is responsible for data transferring from the LocalLink to the memories contained in the Memory Blocks module and specifically to the two main blocks of memory called Control 1 and Control 2. It also provides information about the length and the number of the received frames.

In order to achieve parallel processing these two memory blocks which we will describe later - must be independent from each other and able to receive and pass data at the same time. In order to achieve this, Rx2Mem passes the same information to both Control units so that they both have available all the data that they might be requested to pass to the the cores as the processing of frames occurs and so keep the cores busy with data processing. This is achieved by connecting the output signals of the Rx2Mem module as inputs, to both Control 1 and control 2 units inside the memory blocks module. The Rx2Mem module interface is given below :

Table 4.1: Rx2Mem Interface

Name	Type	Length
clk	in	1
reset	in	1
sof_n_i	in	1
eof_n_i	in	1
data_i	in	8
src_rdy_n_i	in	1
dst_rdy_n_o	out	1
dst_rdy_n_i	in	1
start_addr	out	1
data_o	out	8
frame_num	out	10
frame_len	out	11
frame_done	out	1
addr_inc	out	1

4.2.2 Memory Blocks

The next basic component of our design is the memory blocks component. Memory blocks is the memory centre of our processor as it contains all the needed memories for our design to work. The interface and the block diagram of the module are given below :

Table 4.2: Memory Blocks Interface

Name	Type	Length
mem_blocks_p2m_frame_done_1	in	1
mem_blocks_req_frame_1	in	1
mem_blocks_req_frame_number_1	in	10
mem_blocks_req_byte_1	in	1
mem_blocks_req_byte_number_1	in	11
mem_blocks_wr_req_byte_now_1	in	1
mem_blocks_frame_check_ok_1	in	1
mem_blocks_select_memory_block_1	in	1
mem_blocks_just_need$_byte$_1	in	1
mem_blocks_instr_code_1	in	3
mem_blocks_transfer_1	in	1
mem_blocks_increase_frame_len_1	in	1
mem_blocks_increase_processed_frame_num_1	in	1
mem_blocks_p2m_wr_addr_inc_1	in	1
mem_blocks_data_from_proc_1	in	8
mem_blocks_p2m_frame_done_2	in	1
mem_blocks_req_frame_2	in	1
mem_blocks_req_frame_number_2	in	10
mem_blocks_req_byte_2	in	1
mem_blocks_req_byte_number_2	in	11
mem_blocks_wr_req_byte_now_2	in	1
mem_blocks_frame_check_ok_2	in	1
mem_blocks_select_memory_block_2	in	1
mem_blocks_just_need$_byte$_2	in	1
mem_blocks_instr_code_2	in	3
mem_blocks_transfer_2	in	1
mem_blocks_increase_frame_len_2	in	1
mem_blocks_increase_processed_frame_num_2	in	1
mem_blocks_p2m_wr_addr_inc_2	in	1
mem_blocks_data_from_proc_2	in	8
mem_blocks_r2m_starting_addr_1	in	1
mem_blocks_r2m_wr_addr_inc_1	in	1
mem_blocks_data_from_fifo_1	in	8
mem_blocks_r2m_frame_done_1	in	1
mem_blocks_frame_num_1	in	10
mem_blocks_frame_len_1	in	11
mem_blocks_r2m_starting_addr_2	in	1

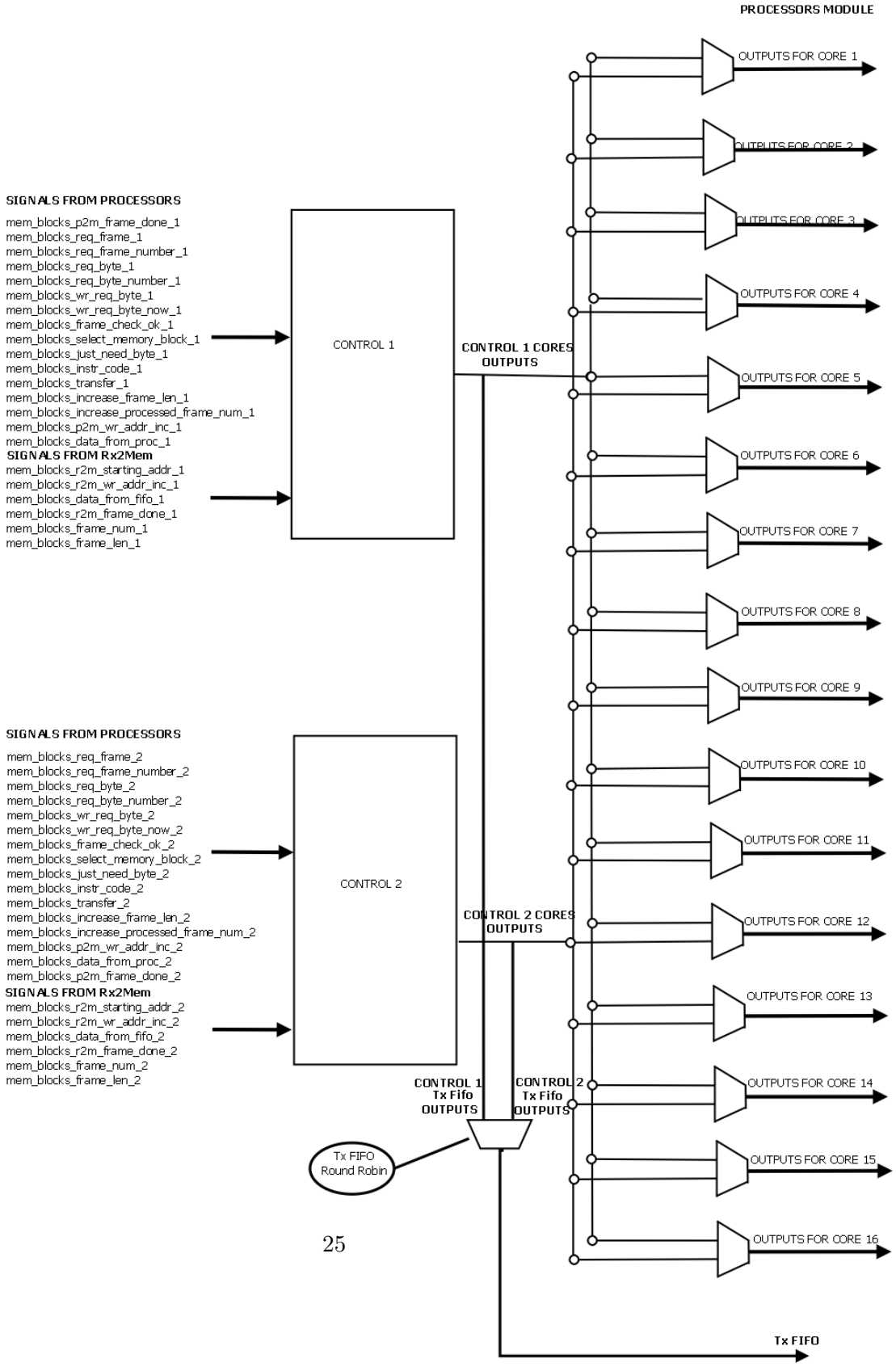
mem_blocks_r2m_wr_addr_inc_2	in	1
mem_blocks_data_from_fifo_2	in	8
mem_blocks_r2m_frame_done_2	in	1
mem_blocks_frame_num_2	in	10
mem_blocks_frame_len_2	in	11
mem_blocks_m2p_frame_done_for_proc1	out	1
mem_blocks_data_for_proc1	out	8
mem_blocks_last_frame_received_for_proc1	out	10
mem_blocks_begin_transmission_for_proc1	out	1
mem_blocks_cur_frame_len_valid_for_proc1	out	1
mem_blocks_cur_frame_len_for_proc1	out	11
mem_blocks_cur_byte_num_for_proc1	out	11
mem_blocks_m2p_frame_done_for_proc2	out	1
mem_blocks_data_for_proc2	out	8
mem_blocks_last_frame_received_for_proc2	out	10
mem_blocks_begin_transmission_for_proc2	out	1
mem_blocks_cur_frame_len_valid_for_proc2	out	1
mem_blocks_cur_frame_len_for_proc2	out	11
mem_blocks_cur_byte_num_for_proc2	out	11
mem_blocks_m2p_frame_done_for_proc3	out	1
mem_blocks_data_for_proc3	out	8
mem_blocks_last_frame_received_for_proc3	out	10
mem_blocks_begin_transmission_for_proc3	out	1
mem_blocks_cur_frame_len_valid_for_proc3	out	1
mem_blocks_cur_frame_len_for_proc3	out	11
mem_blocks_cur_byte_num_for_proc3	out	11
mem_blocks_m2p_frame_done_for_proc4	out	1
mem_blocks_data_for_proc4	out	8
mem_blocks_last_frame_received_for_proc4	out	10
mem_blocks_begin_transmission_for_proc4	out	1
mem_blocks_cur_frame_len_valid_for_proc4	out	1
mem_blocks_cur_frame_len_for_proc4	out	11
mem_blocks_cur_byte_num_for_proc4	out	11
mem_blocks_m2p_frame_done_for_proc5	out	1
mem_blocks_data_for_proc5	out	8
mem_blocks_last_frame_received_for_proc5	out	10
mem_blocks_begin_transmission_for_proc5	out	1
mem_blocks_cur_frame_len_valid_for_proc5	out	1

mem_blocks_cur_frame_len_for_proc5	out	11
mem_blocks_cur_byte_num_for_proc5	out	11
mem_blocks_m2p_frame_done_for_proc6	out	1
mem_blocks_data_for_proc6	out	8
mem_blocks_last_frame_received_for_proc6	out	10
mem_blocks_begin_transmission_for_proc6	out	1
mem_blocks_cur_frame_len_valid_for_proc6	out	1
mem_blocks_cur_frame_len_for_proc6	out	11
mem_blocks_cur_byte_num_for_proc6	out	11
mem_blocks_m2p_frame_done_for_proc7	out	1
mem_blocks_data_for_proc7	out	8
mem_blocks_last_frame_received_for_proc7	out	10
mem_blocks_begin_transmission_for_proc7	out	1
mem_blocks_cur_frame_len_valid_for_proc7	out	1
mem_blocks_cur_frame_len_for_proc7	out	11
mem_blocks_cur_byte_num_for_proc7	out	11
mem_blocks_m2p_frame_done_for_proc8	out	1
mem_blocks_data_for_proc8	out	8
mem_blocks_last_frame_received_for_proc8	out	10
mem_blocks_begin_transmission_for_proc8	out	1
mem_blocks_cur_frame_len_valid_for_proc8	out	1
mem_blocks_cur_frame_len_for_proc8	out	11
mem_blocks_cur_byte_num_for_proc8	out	11
mem_blocks_m2p_frame_done_for_proc9	out	1
mem_blocks_data_for_proc9	out	8
mem_blocks_last_frame_received_for_proc9	out	10
mem_blocks_begin_transmission_for_proc9	out	1
mem_blocks_cur_frame_len_valid_for_proc9	out	1
mem_blocks_cur_frame_len_for_proc9	out	11
mem_blocks_cur_byte_num_for_proc9	out	11
mem_blocks_m2p_frame_done_for_proc10	out	1
mem_blocks_data_for_proc10	out	8
mem_blocks_last_frame_received_for_proc10	out	10
mem_blocks_begin_transmission_for_proc10	out	1
mem_blocks_cur_frame_len_valid_for_proc10	out	1
mem_blocks_cur_frame_len_for_proc10	out	11
mem_blocks_cur_byte_num_for_proc10	out	11
mem_blocks_m2p_frame_done_for_proc11	out	1
mem_blocks_data_for_proc11	out	8
mem_blocks_last_frame_received_for_proc11	out	10
mem_blocks_begin_transmission_for_proc11	out	1
mem_blocks_cur_frame_len_valid_for_proc11	out	1
mem_blocks_cur_frame_len_for_proc11	out	11
mem_blocks_cur_byte_num_for_proc11	out	11

mem_blocks_m2p_frame_done_for_proc12	out	1
mem_blocks_data_for_proc12	out	8
mem_blocks_last_frame_received_for_proc12	out	10
mem_blocks_begin_transmission_for_proc12	out	1
mem_blocks_cur_frame_len_valid_for_proc12	out	1
mem_blocks_cur_frame_len_for_proc12	out	11
mem_blocks_cur_byte_num_for_proc12	out	11
mem_blocks_m2p_frame_done_for_proc13	out	1
mem_blocks_data_for_proc13	out	8
mem_blocks_last_frame_received_for_proc13	out	10
mem_blocks_begin_transmission_for_proc13	out	1
mem_blocks_cur_frame_len_valid_for_proc13	out	1
mem_blocks_cur_frame_len_for_proc13	out	11
mem_blocks_cur_byte_num_for_proc13	out	11
mem_blocks_m2p_frame_done_for_proc14	out	1
mem_blocks_data_for_proc14	out	8
mem_blocks_last_frame_received_for_proc14	out	10
mem_blocks_begin_transmission_for_proc14	out	1
mem_blocks_cur_frame_len_valid_for_proc14	out	1
mem_blocks_cur_frame_len_for_proc14	out	11
mem_blocks_cur_byte_num_for_proc14	out	11
mem_blocks_m2p_frame_done_for_proc15	out	1
mem_blocks_data_for_proc15	out	8
mem_blocks_last_frame_received_for_proc15	out	10
mem_blocks_begin_transmission_for_proc15	out	1
mem_blocks_cur_frame_len_valid_for_proc15	out	1
mem_blocks_cur_frame_len_for_proc15	out	11
mem_blocks_cur_byte_num_for_proc15	out	11
mem_blocks_m2p_frame_done_for_proc16	out	1
mem_blocks_data_for_proc16	out	8
mem_blocks_last_frame_received_for_proc16	out	10
mem_blocks_begin_transmission_for_proc16	out	1
mem_blocks_cur_frame_len_valid_for_proc16	out	1
mem_blocks_cur_frame_len_for_proc16	out	11
mem_blocks_cur_byte_num_for_proc16	out	11

mem_blocks_dst_rdy_n_o	out	1
mem_blocks_data_to_fifo	out	8
mem_blocks_src_rdy_n_o	out	1
mem_blocks_sof_n_o	out	1
mem_blocks_eof_n_o	out	1
mem_blocks_processed_frame_info_for_processed1	out	11
mem_blocks_processed_frame_info_for_processed2	out	11
mem_blocks_processed_frame_done_for_processed1	out	1
mem_blocks_processed_frame_done_for_processed2	out	1
general_start_process	out	1

Figure 4.2: Memory Blocks Block Diagram



As we can see in the block diagram of the module (fig 4.2) inside memory blocks there are two main memory systems called Control 1 and Control 2. Each one is capable of working independently from the other thus allowing parallel processing. The set of multiplexers chooses the correct signals for the processors module and the Tx Fifo. We will now take a deeper look in the operation of the component.

The two memory blocks called Control 1 and Control 2 are two identical components containing the same number of memories and logic needed for their operation. Each one of the Control modules contains three Block RAMs (BRAMs) of 64 kB each that hold frame data, two BRAMs of 27 kB each that hold information about the received and processed frames and finally a FIFO used to queue outgoing frame information. The first BRAM is called data ram Rx and is used to store the incoming frames from the Rx2Mem module. Next we have the data ram TX and data ram processed memories which store the processed frames. The two information brams store information about the received and processed frames. They have a word size of 27 bits, keeping information about the starting address of the frame in the data BRAM and the frame"s length. Last we have the FIFO to queue the frames we have to transmit to the MAC. The FIFO stores the starting address and the length of the frame so thst this is found in the data BRAM and then transmitted.

The way Control modules feed the cores with data is based on an odd-even policy. Control 1 works with the even numbered cores while control 2 works with the odd numbered cores. That means that Control 1 works with cores number 1, 3, 5, 7, 9, 11, 13, and 15 while Control 2 is responsible for cores number 2, 4, 6, 8, 10, 12, 14, 16. The decision of using two memory components was taken by the fact that the Virtex 5 board which was the target board of our design, has a limited memory capacity and so we used the maximum amount of memories that could fit in this board. With the seperation of the cores into odd and even groups of 8, we achieve a balanced operation and a better distribution of the work load between the cores. Finally we have a clearer view of the operation of our design avoiding malfunctions.

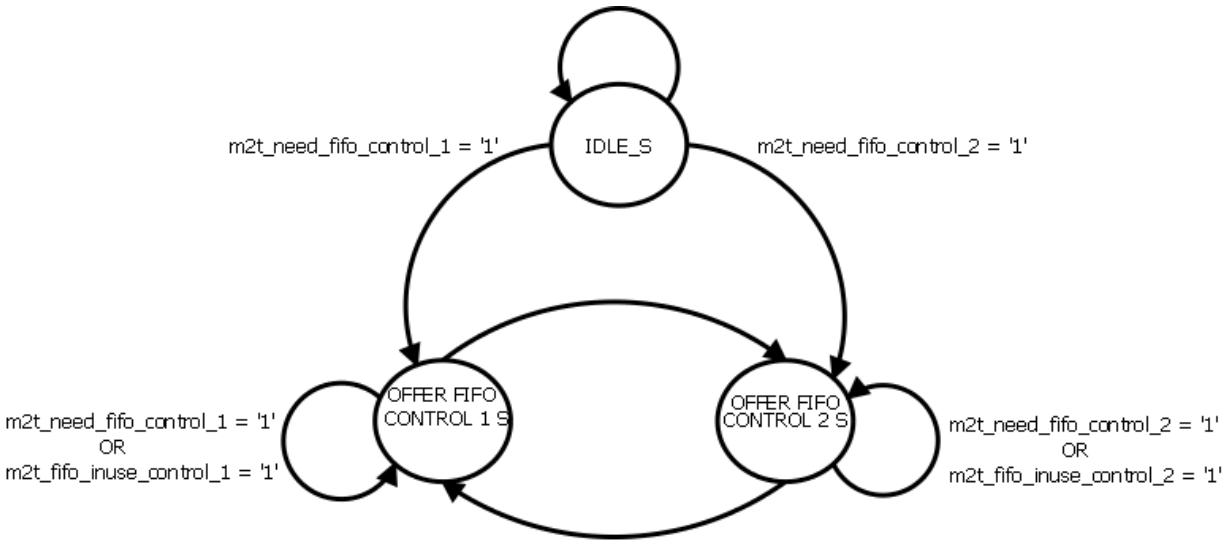
The operation of memory blocks is based on a system of multiplexers that each moment provide data to the working processors. As soon as the first frame is written in the received frames memory of the first or the second memory block, signal general start process is asserted high. That ignites the operation of the first core with the rest of the cores to follow, in a way that we will analyze in deep in chapter 4.3. Once a core starts its operation processors module will send signals to Control Units shown in block diagram as "SIGNALS FROM PROCESSORS". Once the Control units have all the information provided by the incoming signals they can commence the data forwarding.

In order to achieve parallel working all outgoing signals of the component must be connected to all the processors so that each one of them can work independently. This is the point where the multiplexers system takes action. The large number of signals doesnt allow us to present them in the block diagram and so we have put them in groups called "CONTROL 1 CORES OUTPUTS" and "CONTROL 2 CORES OUTPUTS". Control 1 output signals are m2p frame done 1, data to proc 1, last frame received 1, begin transmission, cur frame len valid, cur frame len, cur byte num 1. Similarly for Control 2. We insert these signals by pairs in the multiplexers. To make this clearer for example signals data to proc 1 form Control 1 and data to proc 2 from Control 2 are inserted to one multiplexer and so on for all the signals. According to the core that works we choose the correct one. Each one of the multiplexers shown in the component block diagram represents 7 2to1 multiplexers each one for each pair of signals from the two Control Units. For the 16 cores that gives us a total of 112 multiplexers. We have a common select signal for the odd cores and another common one for the even cores decided by two fsms which we will analyse later. At this point we must make clear that the reason for connecting signals from both Control units to each one of the cores -while we have mentioned the odd-even policy- is to make our design more complete and make it easy for changes in the future as the scalability for the use of more memories is necessary.

When an odd core works, select signal becomes 0 so that the signals of control 1 pass the multiplexers. The data is passed to all the odd cores as we have already mentioned the outputs of all the multiplexers are connected to all the cores but it will be used only by the core that actually asked for them, and needs them. In the same way when an even core asks for data the select signal of the cores multiplexers is set to 1 and so the data from control 2 are passed. With this way we make the cores independent from each other as they can have available all the data they need at any time, no matter of the operation or not of the rest of the cores.

Finally inside the memory blocks component we have a Round Robin fsm, scheduling the transmission of frames to the Tx FIFO. The use of a Round Robin logic was necessary to avoid errors in data transmission as collisions could happen in the scenario that both Control 1 and Control 2 transmitted frames at the same time. The logic of the Round Robin fsm is quite straightforward. Scheduler offers the Tx FIFO to both control units one after another. When one needs it for passing data to the TX FIFO, occupies it until the transmission is finished while the other Control Unit cannot have access at the same time. Each Control Unit can transmit at maximum one frame before passing the access to the next one, so that balanced transmission is achieved between both Control Units. With this way we write the correct data to the Tx FIFO avoiding collisions and errors. The Tx Fifo Round Robin fsm diagram is shown below:

Figure 4.3: Tx FIFO Round Robin



The fsm stays in IDLE state until one of the Control modules requests the Tx Fifo. If Control 1 requests memory by asserting the m2t need fifo control 1 signal the fsm moves to state OFFER FIFO CONTROL 1 S. Similarly moves to OFFER FIFO CONTROL 2 when Control 2 requests Tx FIFO. The fsm will remain in this state as long as the Control needs the Tx Fifo and so signal m2t fifo in use signal is '1'. When Control finishes passing data to Tx FIFO both signals m2t fifo in use and m2t need fifo will become '0' and so the fsm will transit to the next state offering the Tx FIFO to the next Control Unit. If the Control Unit needs it signal m2t need control will be '1' and so the fsm will offer the memory. If not the fsm will keep offerinf the Tx Fifo until a Control Unit occupies it.

According to the state of the fsm and so which one of the Control Unit transmits to the Tx FIFO, we set the select signal of the multiplexers that pass the correct signals to Tx FIFO, shown in the block diagram (fig 4,2) as "CONTROL 1 Tx FIFO OUTPUTS " and "CONTROL 2 Tx FIFO OUTPUTS". These signals are mem blocks data to fifo, mem blocks src rdy n o, mem blocks sof n o, mem blocks eof n o. When we are in state OFFER FIFO CONTROL 1 S signal muxess select fifo becomes '0' and so data from Control 1 pass. Similarly when in state OFFER FIFO CONTROL 2 S select signal becomes '1' and so data from Control 2 are passed to the Tx Fifo.

4.3 Processors Module

Processors component is the container for the 16 cores of our design. Except the 16 cores, inside the component there is the needed logic for the correct signal connections between the cores and the rest of the architecture. We have a series of fsms responsible for the start up of the cores, two Round Robin modules responsible for memory access and logic needed for controlling the frames each core occupies. Last, Processors module contains two mirrored instruction memories and two memories called processed frames info.

The reason we have two pairs of each memory is because as we mentioned before our architecture is divided into two independent memory blocks capable of working at the same time. That creates the need for two mirrored instruction memories and two processed frame info memories, one containing information about the processed frames of the first memory block, and the other containing information about the frames processed in the second memory block. Each command ROM has a capacity of 1024 commands, each command 35 bits long, making it a total size of 35kB. The interface and block diagram of the processors module is shown below:

Table 4.3: Processors Interface

Name	**Type**	**Length**
clk	in	1
reset	in	1
mem_blcks_data_for_proc1	in	8
mem_blcks_last_frame_received_for_proc1	in	10
mem_blcks_begin_transmission_for_proc1	in	1
mem_blcks_cur_frame_len_valid_for_proc1	in	1
mem_blcks_cur_frame_len_for_proc1	in	11
mem_blcks_cur_byte_num_for_proc1	in	11
mem_blcks_receiving_done_for_proc1	in	1
mem_blcks_data_for_proc2	in	8
mem_blcks_last_frame_received_for_proc2	in	10
mem_blcks_begin_transmission_for_proc2	in	1
mem_blcks_cur_frame_len_valid_for_proc2	in	1
mem_blcks_cur_frame_len_for_proc2	in	11
mem_blcks_cur_byte_num_for_proc2	in	11
mem_blcks_receiving_done_for_proc2	in	1
mem_blcks_data_for_proc3	in	8
mem_blcks_last_frame_received_for_proc3	in	10
mem_blcks_begin_transmission_for_proc3	in	1
mem_blcks_cur_frame_len_valid_for_proc3	in	1
mem_blcks_cur_frame_len_for_proc3	in	11
mem_blcks_cur_byte_num_for_proc3	in	11
mem_blcks_receiving_done_for_proc3	in	1
mem_blcks_data_for_proc4	in	8
mem_blcks_last_frame_received_for_proc4	in	10
mem_blcks_begin_transmission_for_proc4	in	1
mem_blcks_cur_frame_len_valid_for_proc4	in	1

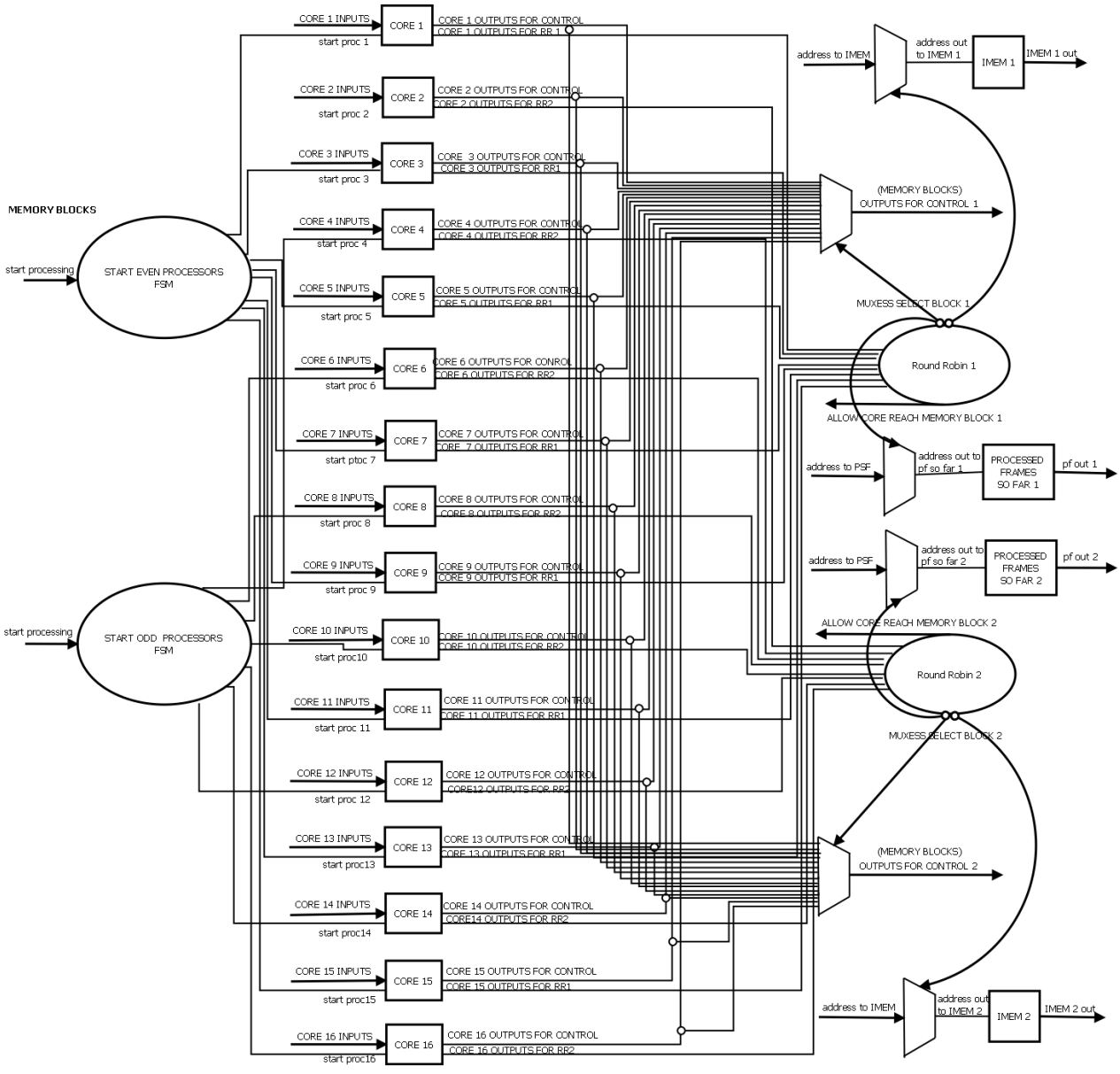
mem_blcks_cur_frame_len_for_proc4	in	11
mem_blcks_cur_byte_num_for_proc4	in	11
mem_blcks_receiving_done_for_proc4	in	1
mem_blcks_data_for_proc5	in	8
mem_blcks_last_frame_received_for_proc5	in	10
mem_blcks_begin_transmission_for_proc5	in	1
mem_blcks_cur_frame_len_valid_for_proc5	in	1
mem_blcks_cur_frame_len_for_proc5	in	11
mem_blcks_cur_byte_num_for_proc5	in	11
mem_blcks_receiving_done_for_proc5	in	1
mem_blcks_data_for_proc6	in	8
mem_blcks_last_frame_received_for_proc6	in	10
mem_blcks_begin_transmission_for_proc6	in	1
mem_blcks_cur_frame_len_valid_for_proc6	in	1
mem_blcks_cur_frame_len_for_proc6	in	11
mem_blcks_cur_byte_num_for_proc6	in	11
mem_blcks_receiving_done_for_proc6	in	1
mem_blcks_data_for_proc7	in	8
mem_blcks_last_frame_received_for_proc7	in	10
mem_blcks_begin_transmission_for_proc7	in	1
mem_blcks_cur_frame_len_valid_for_proc7	in	1
mem_blcks_cur_frame_len_for_proc7	in	11
mem_blcks_cur_byte_num_for_proc7	in	11
mem_blcks_receiving_done_for_proc7	in	1
mem_blcks_data_for_proc8	in	8
mem_blcks_last_frame_received_for_proc8	in	10
mem_blcks_begin_transmission_for_proc8	in	1

mem_blcks_cur_frame_len_valid_for_proc8	in	1
mem_blcks_cur_frame_len_for_proc8	in	11
mem_blcks_cur_byte_num_for_proc8	in	11
mem_blcks_receiving_done_for_proc8	in	1
mem_blcks_data_for_proc9	in	8
mem_blcks_last_frame_received_for_proc9	in	10
mem_blcks_begin_transmission_for_proc9	in	1
mem_blcks_cur_frame_len_valid_for_proc9	in	1
mem_blcks_cur_frame_len_for_proc9	in	11
mem_blcks_cur_byte_num_for_proc9	in	11
mem_blcks_receiving_done_for_proc9	in	1
mem_blcks_data_for_proc10	in	8
mem_blcks_last_frame_received_for_proc10	in	10
mem_blcks_begin_transmission_for_proc10	in	1
mem_blcks_cur_frame_len_valid_for_proc10	in	1
mem_blcks_cur_frame_len_for_proc10	in	11
mem_blcks_cur_byte_num_for_proc10	in	11
mem_blcks_receiving_done_for_proc10	in	1
mem_blcks_data_for_proc11	in	8
mem_blcks_last_frame_received_for_proc11	in	10
mem_blcks_begin_transmission_for_proc11	in	1
mem_blcks_cur_frame_len_valid_for_proc11	in	1
mem_blcks_cur_frame_len_for_proc11	in	11
mem_blcks_cur_byte_num_for_proc11	in	11
mem_blcks_receiving_done_for_proc11	in	1
mem_blcks_data_for_proc12	in	8
mem_blcks_last_frame_received_for_proc12	in	10

mem_blcks_begin_transmission_for_proc12	in	1
mem_blcks_cur_frame_len_valid_for_proc12	in	1
mem_blcks_cur_frame_len_for_proc12	in	11
mem_blcks_cur_byte_num_for_proc12	in	11
mem_blcks_receiving_done_for_proc12	in	1
mem_blcks_data_for_proc13	in	8
mem_blcks_last_frame_received_for_proc13	in	10
mem_blcks_begin_transmission_for_proc13	in	1
mem_blcks_cur_frame_len_valid_for_proc13	in	1
mem_blcks_cur_frame_len_for_proc13	in	11
mem_blcks_cur_byte_num_for_proc13	in	11
mem_blcks_receiving_done_for_proc13	in	1
mem_blcks_data_for_proc14	in	8
mem_blcks_last_frame_received_for_proc14	in	10
mem_blcks_begin_transmission_for_proc14	in	1
mem_blcks_cur_frame_len_valid_for_proc14	in	1
mem_blcks_cur_frame_len_for_proc14	in	11
mem_blcks_cur_byte_num_for_proc14	in	11
mem_blcks_receiving_done_for_proc14	in	1
mem_blcks_data_for_proc15	in	8
mem_blcks_last_frame_received_for_proc15	in	10
mem_blcks_begin_transmission_for_proc15	in	1
mem_blcks_cur_frame_len_valid_for_proc15	in	1
mem_blcks_cur_frame_len_for_proc15	in	11
mem_blcks_cur_byte_num_for_proc15	in	11
mem_blcks_receiving_done_for_proc15	in	1
mem_blcks_data_for_proc16	in	8
mem_blcks_last_frame_received_for_proc16	in	10
mem_blcks_begin_transmission_for_proc16	in	1
mem_blcks_cur_frame_len_valid_for_proc16	in	1
mem_blcks_cur_frame_len_for_proc16	in	11
mem_blcks_cur_byte_num_for_proc16	in	11
mem_blcks_receiving_done_for_proc16	in	1
data$_{f}or_{p}rocessed1$	in	11
data$_{f}or_{p}rocessed2$	in	11
processed_frame_done_for_processed1	in	1
processed_frame_done_for_processed2	in	1
start_processing	in	1

req_frame_p_for_control1	out	1
req_frame_number_p_for_control1	out	10
req_byte_p_for_control1	out	1
req_byte_number_p_for_control1	out	11
wr_req_byte_p_for_control1	out	1
wr_req_byte_now_p_for$_c ontrol$1	out	1
frame_check_ok_p_for_control1	out	1
increase_frame_len_p_for_control1	out	1
p2m_wr_addr_inc_p_for_control1	out	1
increase_processed_frame_num_p_for_control1	out	1
select_memory_block_p_for_control1	out	1
just_need_byte_p_for_control1	out	1
instr_code_p_for_control1	out	3
transfer_p_for_control1	out	1
pause_receive_for_control1	out	1
data_o_p_for_control1	out	8
frame_sent_done_p_for_control1	out	11
req_frame_p_for_control2	out	1
req_frame_number_p_for_control2	out	10
req_byte_p_for_control2	out	1
req_byte_number_p_for_control2	out	11
wr_req_byte_p_for_control2	out	1
wr_req_byte_now_p_for$_c ontrol$2	out	1
frame_check_ok_p_for_control2	out	1
increase_frame_len_p_for_control2	out	1
p2m_wr_addr_inc_p_for_control2	out	1
increase_processed_frame_num_p_for_control2	out	1
select_memory_block_p_for_control2	out	1
just_need_byte_p_for_control2	out	1
instr_code_p_for_control2	out	3
transfer_p_for_control2	out	1
pause_receive_for_control2	out	1
data_o_p_for_control2	out	8
frame_sent_done_p_for_control2	out	11

Figure 4.4: Processors module block diagram



4.3.1 Signals from/to the Cores

Taking a closer look to the Processors module block diagram (see 4.4) we should mention that the very large amount of inputs-outputs and logic consisted of muxes doesnt allow us to represent them all as they have.For this reasong we represent the signals and muxes logic in groups. CORE 1 INPUTS, CORE 2 INPUTS, etc, represent the input signals of the cores coming from the Memory Blocks module. For each one of the cores these signals are: "mem blcks data for proc", "mem blcks receiving done for proc", "mem blcks last frame received for proc", "mem blcks begin transmission for proc", "mem blcks cur frame len valid for proc", "mem blcks cur frame len for proc", "mem blcks cur byte num for proc" (followed by the core number). These inputs to the cores are decided by the logic inside the Memory blocks module described in chapter 4.2.

Next we have the outputs of the cores to the Memory blocks module shown in the block diagram (see 4.4) as CORE 1 OUTPUTS FOR CONTROL, CORE 2 OUTPUTS FOR CONTROL, etc. For each core these outputs are "req frame p for control", "req frame number p for control", "req byte p for control", "req byte number p for control", "wr req byte p for control", "wr req byte now p for control", "frame check ok p for control", "increase frame len p for control", "p2m wr addr inc p for control", "increase processed frame num p for control", "select memory block p for control", "just need byte p for control", "instr code p for control", "transfer p for control", "pause receive for control", "data o p for control", "frame sent done p for control" (followed by the core number). Seperating the same signals from each core and inserting them into 16to1 muxes gives us the final outputs for the two Control modules shown in the block diagram as OUTPUTS FOR CONTROL 1 and OUTPUTS FOR CONTROL 2. With this said we must mention that each one of the muxes shown in block diagram represents a set of muxes for each one of the aformentioned signals for each one of the two Control modules inside the Memory blocks component.

Last, CORE OUTPUTS for RR1 and RR2 are the outputs of the cores sent to Round Robin modules. These are: "processor(core number) use memory block(memory block number)", processor(core number) i need mem (memory block number).

4.3.2 Start Processing

As soon as we are ready for processing to begin, two fsms are responsible for starting up the operation of the cores. As we have already mentioned, cores are separated into odd-even groups, so we have two fsms, one responsible for the odd numbered cores and one responsible for the even numbered shown in the block diagram (fig 4.3) as "START EVEN PROCESSORS FSM" and "START ODD PROCESSORS FSM". The FSM diagrams are given below:

Figure 4.5: Starting Cores FSM 1 - Even Cores

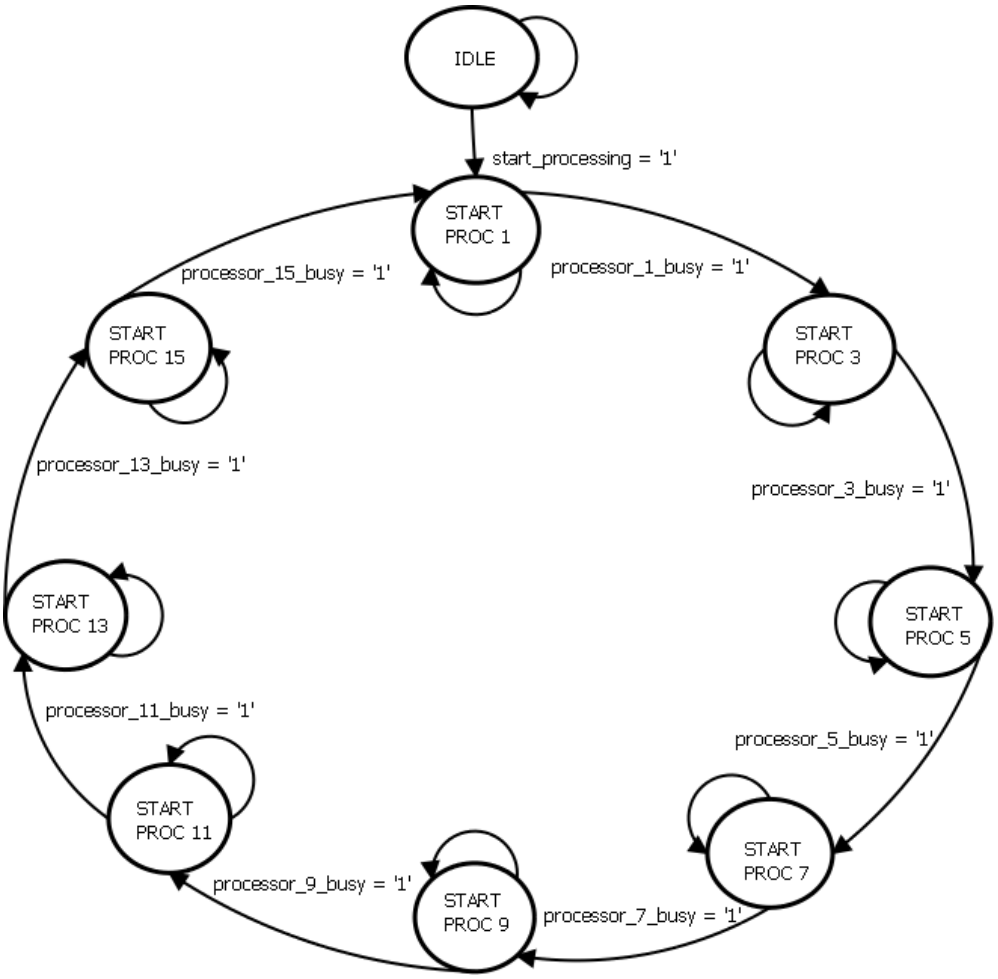
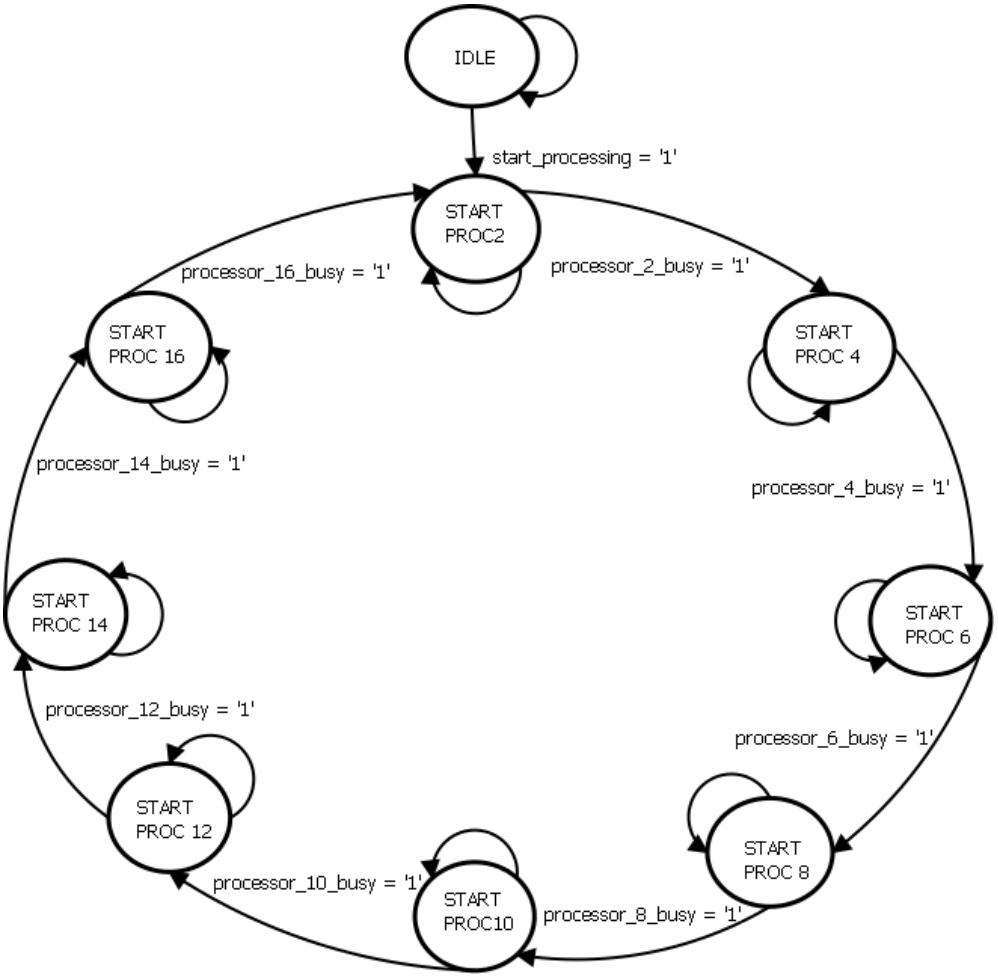


Figure 4.6: Starting Cores FSM 2 - Odd Cores



As shown in diagrams 4.4 and 4.5 we have two fsms working with the same way. The fsm will wait in IDLE state until the start processing signal - coming from the memory blocks module as general start process- is asserted high. Once this happens the fsm will move on to the START PROC 1 state. This will ignite the operation of core number 1 by asserting the start processor signal high. If this core is already working - so its busy signal is asserted high - the fsm will proceed to the next state which is the START PROC 3 state. Similarly the fsm will continue to the next states starting up the cores one after another while passing from the cores that are already busy. With this way we commence the operation of the cores and we make the first step for the instructions execution. At this point, we must make clear what starting up a core means.

When the fsms we described start up the cores, they make them move on from IDLE state to a state that they ask for memory access. This is where they will wait for other parts of logic to interfere - specifically the Round Robin modules - and make them proceed with the actual execution of commands. We will describe the way this happens in the next paragraph. The second fsm works with the exact same way being responsible for the odd numbered cores. Similarly it will commence the operation of odd cores and send them to wait for memory access.

4.3.3 Round Robin - Memory Access

Our design uses Round Robin logic in order to control the access of the cores to the memories. Inside the Processors module we use two Round Robin modules that work independently and in parallel, one controlling the access of even numbered cores to control 1 and the other controlling the access of odd numbered cores to control 2. The Round Robin fsm diagrams are given below:

Figure 4.7: Round Robin FSM 1 - Even Cores

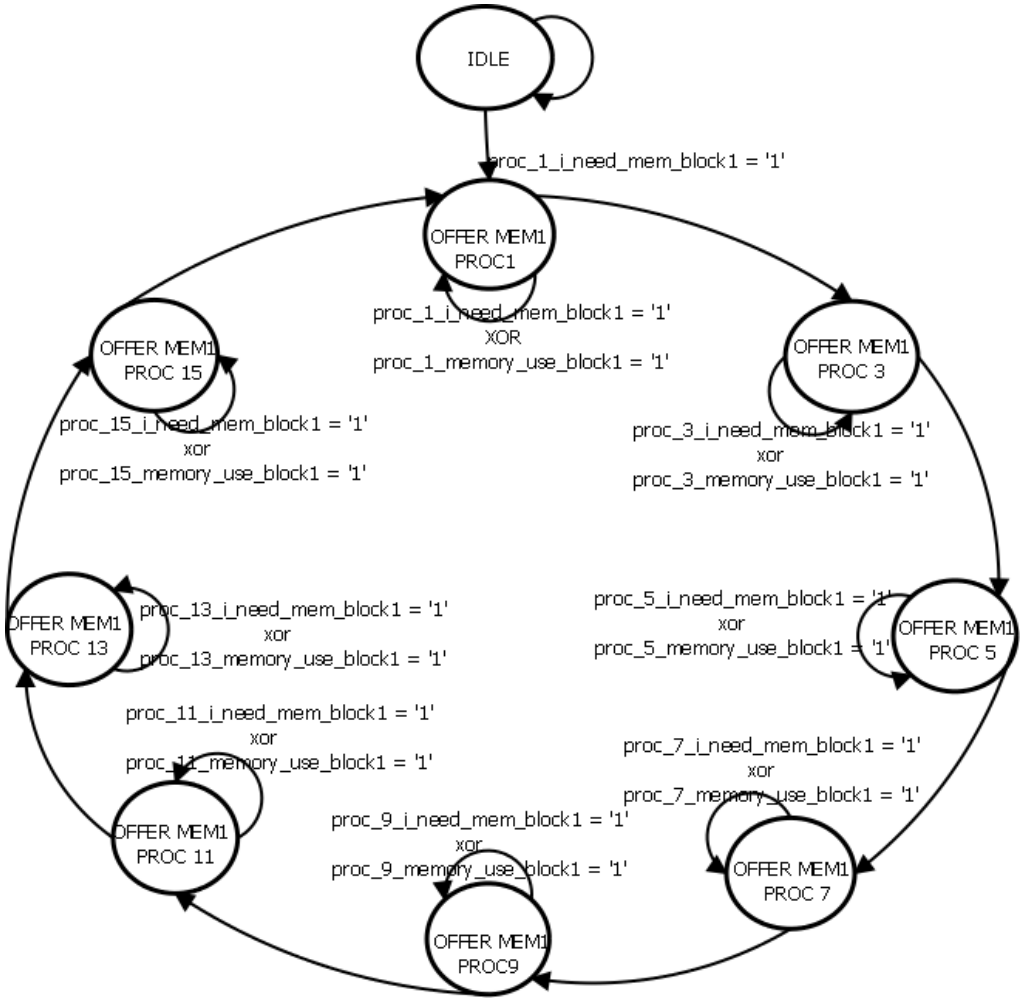
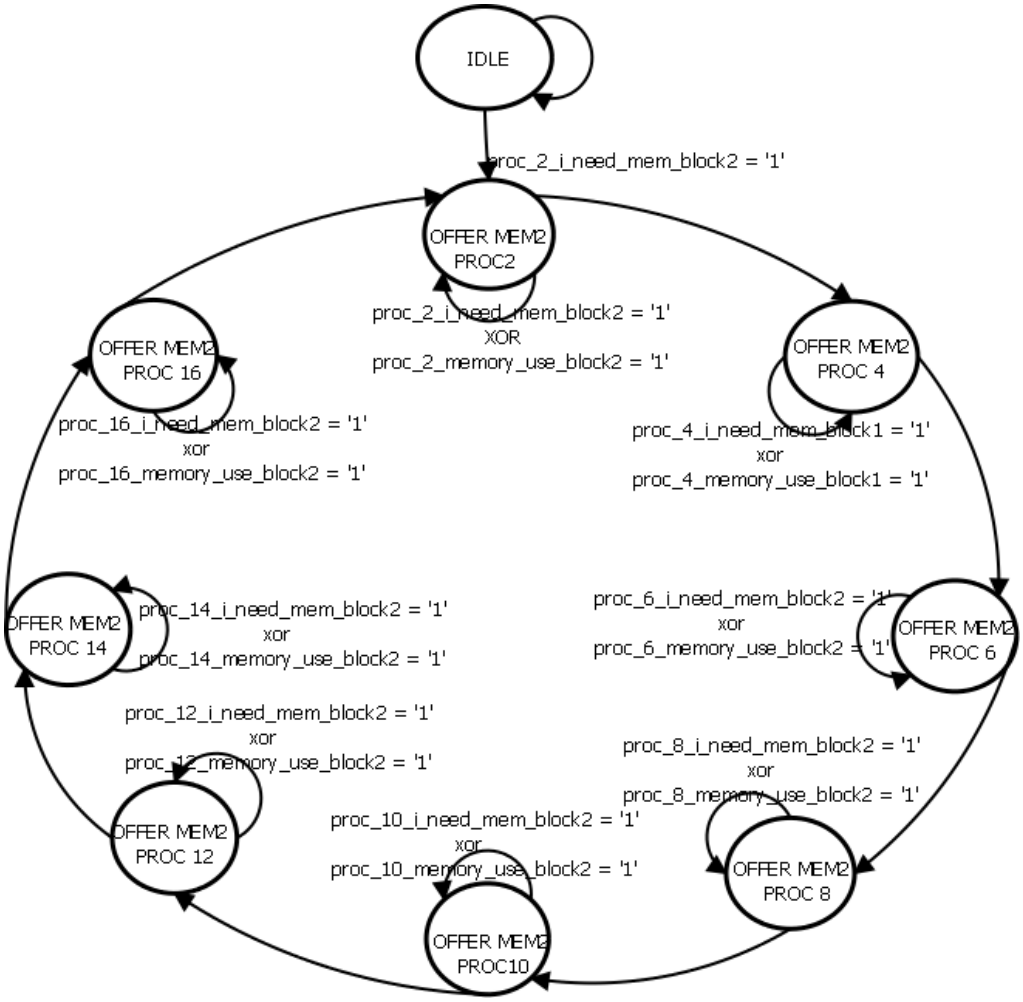


Figure 4.8: Round Robin FSM 2 - Odd Cores



robin2.png

The logic used is quite simple. The fsm will offer the memory to every core, one after another. If the core needs the memory, and no other core uses the memory at the same time, it will be allowed to occupy it. The fsm will wait for the core to finish and then it will offer the memory to the next one. If the core doesnt need the memory the fsm will continue and offer it to the next core. With this way we share memory access time in a balanced way and also we avoid latencies caused by cores that dont need the memory.

In a deeper look the fsm will wait in IDLE state until the "proc 1 i need memory"" signal is asserted high, meaning that core number 1 makes a request to access the memory. The fsm will move on to state OFFER MEM PROC 1 generating the signal "allow proc 1 reach memory 1". Once this signal is asserted the core can access the memory and commence its actual operation. While in this state, the fsm also generates the signal "muxess select block 1 giving to it the value 0000. This signal is the select signal for the set of multiplexers inside the Memory block and Processor components. With this way we insure that the correct signals will pass from and to core number 1. The fsm will remain in this state until core 1 finishes the execution of its first instruction. When this happens signal proc 1 memory use block 1 will become 0, meaning that the core no longer uses the memory.

Once this happens the fsm will move on to the next state offering the memory to core number 3. At this point we must remind that the first Round Robin controls the access to memory block 1 so it communicates with odd numbered cores only. If core number 3 needs the memory and no other core uses it at the same time, the fsm will allow access generating the correct signals for the memory block component. If core 3 doesnt need the memory the fsm will move on with the next core and so on. Last if core 3 needs the memory but another core is using it, it will wait until the fsm reaches the OFFER MEM PROC 3 state. While Round Robin 1 works for memory block 1 at the same time Round Robin 2 works controlling access of the odd numbered cores to memory block 2. The operation of the second fsm is exactly the same with the one we described earlier but this time it communicates with odd numbered cores and their access to memory block 2.

4.3.4 Data Dependencies and Solution

As we have already mentioned data dependencies are a major issue for multi-core designs. The solution given by our design is rather simple and effective.

Our design follows a policy where commands are loaded in the instruction memory in a way that the frame numbers, they refer to, are sequential. Taking advantage of this characteristic we decided that as soon as a core starts executing a command for a certain frame, it is responsible for executing all the commands referring to this frame, while no other cores can execute an instruction referring to the already occupied frame. For example

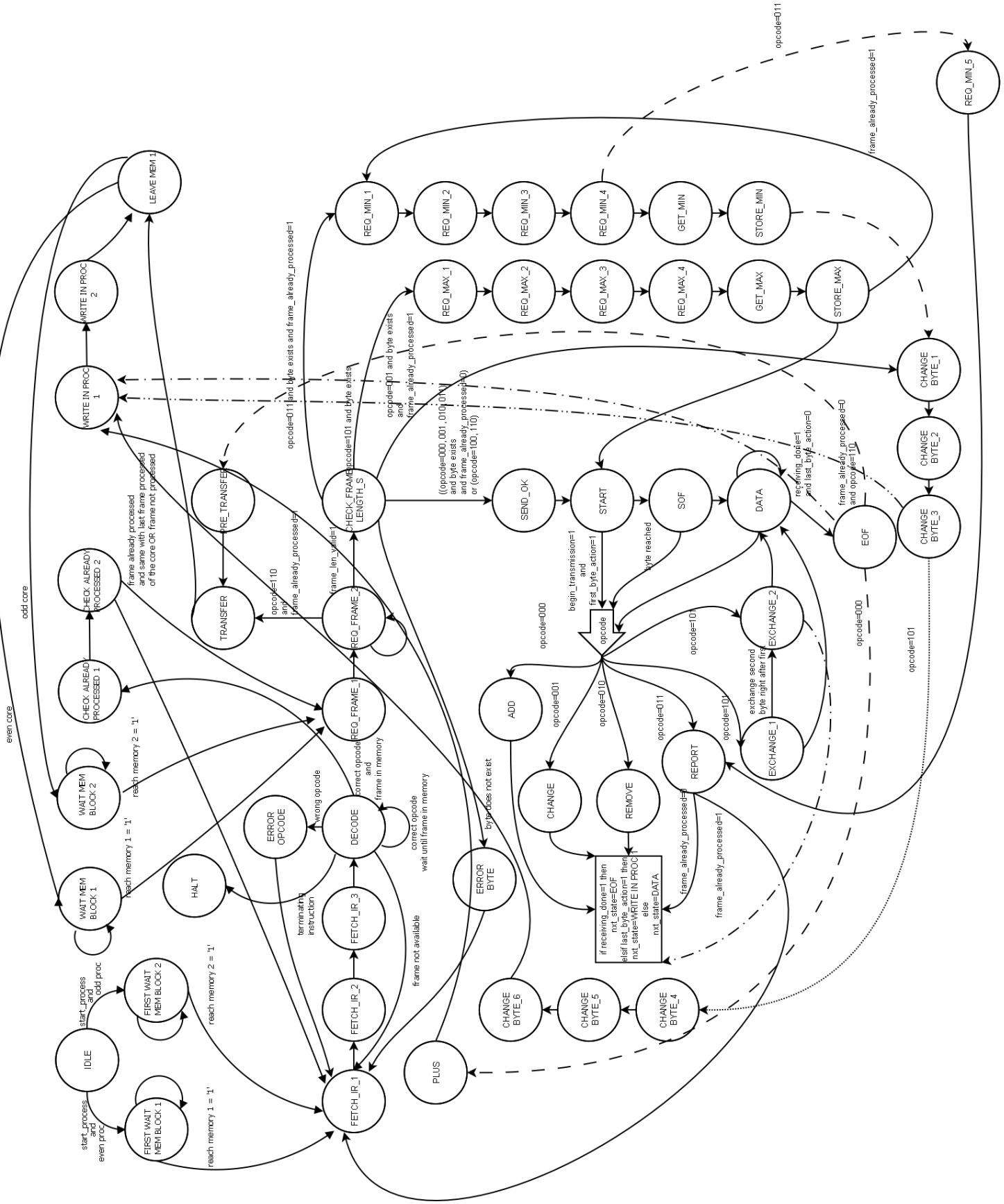
if core number 5 fetches an instruction (this will be the first instruction) referring to frame number 9 it will be the only processor that will fetch instructions for frame 9 and is responsible for executing all the batch of commands referring to this frame.

In order to achieve this, we have to make a series of checks. The first check point is inside the processors module. Once we commence the operation of a core with the fsms we described previously, we have to decide if the core will execute the instruction or it will move on to the next instruction of the instruction memory. Each core has an out signal called "frame occupied" indicating the frame number that the core is currently processing. Also the 31 to 22 bits of the command indicate the frame number that the command refers to. When a core fetches a command we compare the frame number of the command with the frame occupied signals of the remaining 15 cores. With this way we generate 15 signals having the value of comparing the occupied frames with the frame that the command refers to. Passing these 15 signals through OR gates creates the signal "allow processor to fetch". If this is 1 then the first part of the check is passed and the core continues with more checks that we will describe later. If not, the core continues with the next command -with a way described later- until it finds an instruction referring to an unoccupied frame.

4.3.5 Process Unit

Our multicore processor uses a homogeneous core policy, so all the 16 cores are the same, working at the same frequency supporting the same instruction set and operating with the exact same way. We will now take a look at the operation of the core. The fsm of the process unit is shown below:

Figure 4.9: Process Unit FSM



Once signal "general start process" from the memory blocks component is asserted high signalling that the first frame is written in memory the fsms we described (see 4.3.1) will start up the cores by asserting the "start process" signal. When this happens the cores fsm will leave the IDLE state. At this point an important check is made. Each core has an output signal called "odd or even proc" which declares if the core is odd or even numbered and is generated simply by choosing the least significant bit of the core number. We check this signal and if its 1 the core is even, so the fsm moves on to the FIRST WAIT MEM BLOCK 1 state and asserts the signal "i need mem block 1" high, otherwise the core is odd and so the fsm moves on to state FIRST WAIT MEM BLOCK 2 and asserts the signal "i need mem block 2". With this way we send the core to a state where it asks for memory access from the Round Robin modules we described (see 4.3.2). The core will wait in this state until the Round Robin modules allow it to access the memory.

Looking closer the fsm states we can see there are two different states where the cores wait for memory access. For even numbered cores we have FIRST WAIT MEM BLOCK 1 and WAIT MEM BLOCK 1 and for odd numbered cores we have FIRST WAIT MEM BLOCK 2 and WAIT MEM BLOCK 2. The difference is that the core will visit the FIRST states only the first time it will wait for memory just after it has left the IDLE state. From this point and on every time the core finishes the execution of a command it returns to the "simple" wait for memory states. The reason for doing this is not obvious but we will try to make it clearer with an example. Lets assume that the instruction memory is loaded with one instruction for each frame as shown below :

Instruction for frame 1

Instruction for frame 2

.

.

.

.

Instruction for frame 16

According to the policy we follow in order to avoid data dependencies core 1 will handle the first instruction. Core 2 will check the first instruction but will not execute it and so it will handle instruction 2. As we continue with the same way core 16 must check 15 previous commands that we already know that refer to unavailable frames as the cores before core 16 have already

occupied them. If we think that we need 3 clock cycles just to increase the address of the instruction memory and fetch the next command, the total latency caused by unnecessary checks is extremely high. At this point we can avoid the latency with a simple idea. Each core can use the checks of its previous core, and so continue checking from the address the previous core stopped and not from the beginning of the instruction memory. Back to our example with this policy followed core 16 will continue from where core 14 stopped (odd cores are based to checks of odd cores and even cores to checks of even cores) plus 1. The technique we described works fine but hides a trap. It can be used only for the first time a core searches the instruction memory.

To make this clearer imagine that core 3 handles a batch of 2 commands referring to frame 7 and core 5 handles a batch of 5 commands referring to frame 11. Lets say that core 3 found the first command in address position 22 of the instruction memory. Using this as a mark point core 5 will start checking the instruction memory from address position 23. The first time this scenario will work fine. As the processing will continue core 3 will finish the 2 commands (while core 5 will still work on frame 11) and will continue searching the instruction memory for the next available frame. This can be found anywhere but certainly many address positions later. If core 5 keeps basing its search on the address position that core 3 stopped, it will leave its batch of commands unfinished as it will start checking many positions later. This is why we can use the technique only for the first time of searching.

With that being explained and back to our architecture that is the reason we have the states FIRST WAIT MEM BLOCK 1 and 2. When the fsm moves on from them, to FETCH IR 1 state we assert signal first time" high, as a flag indicating that this is the first time the core searches for an available instruction. Finally we use this signal to set the initial address to the address counter inside the core. If this signal is 1 and so the core searches for the first time the starting address is set to be the address that the previous core stopped plus 1. If not, the starting address is just the previous address of the core itself. Back to our example when core 5 searches for the first time its address counter will be initialised to position 23 (where core 3 stopped plus 1). For the next times the address will not be initialised according to core 3 and so core 5 will continue with positions 24, 25, 26, 27 finishing its batch of commands correctly.

From this point and on the core will never go back to the FIRST wait for memory states but it will use the normal wait for memory states every time it needs to occupy the memory. As soon as the core gets permission to occupy the memory the fsm will move on to FETCH IR 1, FETCH IR 2 and FETCH IR 3 states. Here the core will make another check. We have decided that the terminating command is the 11111111111111111111111111111111. The core checks if the command is this, and if so it moves on to HALT state. With this way we terminate the operation of a core in order to lower the

energy consumption of our design as the cores that dont have instructions to execute, are moved to IDLE state.

Back to the norm, the fsm moves on to DECODE IR S state. It will now check the value of the signal "frame available" we described in paragraph 4.1.3.2. If this is 1 - indicating that the frame is available - the fsm will make two more checks. First it will check the frame number of the command. If this is 1111111 the fsm will go to state ERROR BYTE S. If not, the fsm will check if the frame that the command refers to is written in memory. If it is, it will move on to state CHECK ALREADY PROC 1 S otherwise we will wait to DECODE IR S state until the frame is written in memory. Back to the first check, if the frame is already occupied from another core the fsm will return to FETCH IR 1 state in order to check the next command of the instruction memory.

When in CHECK ALREADY PROC 1 S the fsm moves on to CHECK ALREADY PROC 2 S. At this point we just consumed clock cycles in order to have available the signals that we will use on the check we will describe next. Frame availability is not covered totally by the first check we performed, see [243] as there is a rare possibility that check one does not cover. Lets make this clearer using an example. Imagine that core number 1 handles a batch of five commands referring to frame number 5 while core number 3 handles one command referring to frame number 9. While core 1 executes its commands, core 3 will have processed the one command for frame 9 and will move on occupying a new frame. When core 1 searches for a new instruction it will fetch the instruction that core 3 has already executed. Right at this point is the hole of the first check. Core 3 will have now occupied a new frame and so frame 9 will appear unoccupied. Check one will allow to core 1 to execute the instruction and finally we will have executed the same command two times leaving frame 9 with wrong data. We will now present the solution to this problem.

As we have already described each time a new command is fetched, the frame number provided is set as an address in the frame substitution BRAM. The least significant bit of the output is the signal "frame already processed". If this is 1 the frame is already processed otherwise the frame is processed for the first time. Taking advantage of this signal we can give the solution to our problem. If the frame is processed for the first time then the core can execute the instruction. If the frame is already processed but the frame number is not the same with the last frame the core processed then the core cant execute the command as this means that the frame is already processed by another core. Last if the frame is already processed and the frame number is the same with the last frame number the core processed, the core can execute the instruction as the frame is processed by the same core earlier. Back to our example when core 1 finds the command for frame 9 it will proceed to the second check. The check will show that frame number 9 is already processed while cores 1 last frame processed is frame 5. This

will show that frame 9 is already processed by another core and so core 3 will reject the command.

Back to CHECK ALREADY PROC 2 S we proceed with the second check we just analysed. The fsm will move on to state REQ FRAME S. If signal "frame already processed" is 0 or signal "frame already processed" is 1 and signals "frame num" and "last frame processed" have the same value. In all the other cases the fsm will return back to FETCH IR 1 S state so that the core will look for a new instruction to execute. Once all the aforementioned checks are passed the core is finally ready to execute the command and commence the actual processing of the frame. For detailed information about the processing itself see [1]. After the processing of the frame is finished, the fsm will move on to state WRITE IN PROC 1 and then to WRITE PROC 2. With this way we consume two clock cycles in order to write the correct data to processed frames and processed frames info memories. Next the fsm moves on to state LEAVE MEM 1 S state.

The reason we always move on to this state is for releasing the memory the core has occupied. To make this clearer only at this state signals "need memory" and "memory use" the transition signals of the Round Robin fsms - are 0 at the same time. With this way we make the Round Robin fsm to move on to the next state and so offer the memory to the next cores. Finally the core will return to WAIT MEM BLOCK 1 or WAIT MEM BLOCK 2 depending if it is an odd or even core. At this state it will wait to regain access to the memory and repeat the whole process we described.

Chapter 5

Verification and Performance

5.1 Hardware Used

All the needed equipment for the implementation of our design was provided by the Microprocessors and Hardware Lab (MHL) of the Technical University of Crete (TUC). We will now a take a look on the hardware used.

5.1.1 PC Workstation

The PC we used was a standard desktop computer equipped with an Intel Core 2 Duo processor and 4 GB of RAM, features that were enough for making it keep up with the requirements of our work with no problem. The PC was also equipped with a network card able to operate at Gbit / sec Ethernet speed.

5.1.2 Xilinx Virtex 5 Board

For the implementation of our design we used a Xilinx Virtex 5 fpga board and specifically the model Virtex 5 - 5XC5VLX110T of the LX family. Some features of the board are:

- RS232 Serial Port
- 16 character x 2 Line LCD Display
- USB interface chip with host and peripheral ports
- DVI video connector
- PS / 2 mouse and keyboard connectors
- Programmable System Clock Generator Chip
- General purpose DIP switches, LEDs and Pushbuttons

- Ethernet Port

5.1.3 Ethernet cable

In order to connect the board with the PC, we used an Ethernet category 5e crossover cable supporting Gigabit Ethernet speeds. We connected the one end to the network card of the PC and the other to the Ethernet port of the board.

5.1.4 Xilinx Programmer Cable

In order to download our design on the FPGA board we used the Xilinx Programmer cable coming with the board. We connected the one end to a USB port of the PC and the other to the Programmer cable port of the board.

5.2 Software Used

5.2.1 Software used during development

5.2.1.1 Xilinx ISE

Xilinx ISE was the tool we used most during the development of our design. The ISE Design Suite features include design entry and synthesis supporting Verilog or VHDL, place-and-route (PAR), completed verification and debug using ChipScope Pro tools, and creation of the bit files that are used to configure the chip.

Using ISE, we wrote the whole code of our design in VHDL. We also used it to simulate the behaviour of the design in behavioural and PAR simulation. Through hundreds of simulations and checks, we came up with the desired functionality of our parallel processor.

5.2.1.2 Xilinx EDK

After finishing our work with ISE we downloaded our design to the FPGA using Xilinx EDK. EDK is a tool used for the customization and download of hardware designs in a variety of target boards. It synthesises, maps and places the design on the board. Finally it creates the bit stream - we transmit through the programmer cable- in order to program the FPGA.

5.2.1.3 Xilinx Core Generator

The CORE Generator System is a design tool that delivers parameterized cores optimized for Xilinx FPGAs. It provides a catalog of ready-made functions ranging in complexity from simple arithmetic operators such as adders, accumulators, and multipliers, to system-level building blocks such

as filters, transforms, FIFOs, and memories. We used Core Generator to instantiate the Ethernet MAC, BRAMs, ROM and FIFOs we use in our design.

5.2.2 Software used during Verification

5.2.2.1 Xilinx Chipscope Pro

ChipScope Pro inserts logic analyzer, bus analyzer, and Virtual I/O low-profile software cores into the design. These cores allow us to view all the internal signals and nodes within the FPGA. We used it after the download of our design on the FPGA, in order to see the behaviour of our circuit under real world conditions and not just simulation.

5.2.2.2 Wireshark

Wireshark is a free and open-source packet analyser. It is used for network troubleshooting, analysis, software and communications protocol development, and education. We used it in order to monitor the packets, our design received and sent, and make sure that frames were altered according to the commands on them. Some of the most important features of the tool are:

- Capture live packet data from a network interface
- Display packets with very detailed protocol information
- Open and Save packet data captured
- Import and Export packet data from and to a lot of other capture programs
- Filter packets on many criteria
- Create various statistics
- Standard three-pane packet browser
- Multi-platform: Runs on Windows, Linux, OS X, Solaris
- Colorize packet display based on filters
- Live capture and offline analysis

5.2.2.3 PackEth

Packeth is a GUI packet generator tool for Ethernet. It allows you to create and send any possible packet or sequence of packets via Ethernet. It originally ran on Linux but later a Windows version was released. We used packEth in order to create the type of packets we wanted and send them to our design through the Ethernet cable. Some of its key features are:

- Ethernet II, Ethernet 802.3, 802.1q, QinQ
- ARP, IPv4, user defined network layer payload
- UDP, TCP, ICMP, IGMP, user defined transport layer payload
- RTP (payload with options to send sin wave of any frequency for G.711)
- Sending sequence of packets
- Delay between packets, number of packets to send
- Sending with max speed, approaching the theoretical boundary
- Saving configuration to a text file and load from it

5.3 Verification Process

In order to verify the correct functionality of our design we first downloaded it to the FPGA board through the Xilinx Programmer Cable. Next we connected the PC's network adapter with the Ethernet port of the board using the Ethernet cable. We started Wireshark and set its interface to the network card of our PC. Then we created specific frames with packeth and sent them to our design by groups. At this point we should mention that Windows send packets as soon as there is connectivity between the board and the PC, but creating and sending our own packets made the verification process clearer as we had better control of the frames and the data changes we expected to see. Through Wireshark we could see that our design received the frames and sent them back to the PC altered, according to the commands we loaded in the instruction memory. With this way we could verify the correct operation of the parallel network processor. The list of commands we loaded in instruction memories during the verification process is given below:

Add 1,15,05
Add 1,16,07
Change 1,1,0A
Report 1,1
Add 1,1,09
Add 1,3,09
Transfer 1

Add 2,31,0A
Add 2 ,31,0B
Add 2 ,31,0C
Report 2, 6
Transfer 2

Add 3,31,0F
Add 3,31,11
Report 3,1
Add 3,28,13
Change 3,1,FF
Transfer 3

Add 4,13,0A
Add 4 ,45,0B
Exchange 4,10,14
Add 4,31,0E
Add 4 ,45,0B
Transfer 4

Add 5,22,0C
Transfer 5

Add 6,31,0A
Change 6,1,0A
Report 6,1
Add 6,1,09
Transfer 6

Add 7,31,0A
Add 7 ,31,0B
Add 7 ,31,0C
Add 7,31,0D
Add 7,31,0E
Transfer 7

Add 8,31,0A
Add 8 ,31,0B
Report 8,8
Add 8,2,13
Change 8,1,FF
Exchange 8,1,88
Transfer 8

Report 9,8
Add 9,2,13
Change 9,1,EE
Exchange 9,1,4
Add 9,31,0A
Add 9 ,31,0B
Transfer 9

Add 10,15,05
Add 10,16,07
Change 10,1,0A
Report 10,1
Add 10,1,09
Add 10,3,09
Transfer 10

Add 11,31,0A
Add 11,31,0B
Add 11 ,31,0C
Report 11, 6
Transfer 11

Add 12,31,0F
Add 12,31,11
Report 12,1
Add 12,28,13
Change 12,1,FF
Transfer 12

Add 13,13,0A
Add 13 ,45,0B
Exchange 13,10,14
Add 13,31,0E
Add 13 ,45,0B
Transfer 13

Add 14,22,0C
Add 14,13,0A
Add 14 ,45,0B
Exchange 14,7,2
Transfer 14

Add 15,31,0A
Change 15,1,0A
Report 15,1
Add 15,1,09
Transfer 15

Add 16,31,0A
Add 16 ,31,0B
Add 16 ,31,0C
Add 16,31,0D
Add 16,31,0E
Transfer 16

Add 17,31,0A
Add 17 ,31,0B
Report 17,8
Add 17,2,13
Change 17,1,FF
Exchange 17,1,88
Transfer 17

Report 18,8
Add 18,2,13
Change 18,1,EE
Exchange 18,1,4
Add 18,31,0A
Add 18,31,0B
Transfer 18

Report 19,8
Add 19,2,13
Change 19,1,EE
Exchange 19,1,4
Transfer 19

5.4 Perfomance

Analysing the perfomance of our parallel processor, we must mention that like with other execution techniques like pipelining, parallel processing achieved by our processor does not reduce the execution time of a single instruction. In fact the core of our design adds a latency of 3 clock cycles for every instruction, compared to the single core processor we were based on, because of the states WRITE IN PROC 1, WRITE IN PROC 2 and LEAVE MEM 1 that were added in the Process unit fsm (These are the extra states after the core is allowed to access memory). With our parallel processor we increase the throughput, being able to finish the processing of a large batch of commands faster. Throughput is the critical unit of measurement as in real programs, the commands to be executed are hundreds.

The speed-up of our design, as it was developed for download and testing on the Xilinx Virtex 5 5XC5VLX110T reaches the factor of 2 as the limited space for memory (148 Block RAM) causes a bottleneck that reduces the total speed up. Confirming Amdahl's law our processor using 16 cores and 2 main memory systems executes the same number of commands, two times faster than the single core processor we were based on. Besides calculating the speed-up in theory, we also confirmed it during development, through simulation with Xilinx ISE, comparing the total time from the fetch of the first instruction to the execution of the last. The measurements were made for the same batch of commands for both the multicore and single core processors.

Placing the design on bigger boards of the Xilinx Virtex family offers an impressing speed-up. Virtex XC5VLX330T with 324 RAM blocks available will allow 4 times faster processing, while XC5VSX240T board has 516 blocks allowing 7 times faster processing. Finally a speed-up reaching the factor of 16 could be achieved using the Virtex XC6VSX475T with 1064 RAM blocks thus allowing the fully parallel operation of all the 16 cores of our processor making it about 15 to 16 times faster than a single core processor.

Chapter 6

Future Work

The parallel network processor developed in this diploma thesis works efficiently but there is always space for improvement.

The most important improvement that could be done is the increase of memory components. Unfortunately our design was limited by the memory capacity of the Virtex board we had available. Nevertheless our architecture is designed in a way that is fully scalable. Extra memory components can be put quite easily with only a few adjustments and modifications. Using bigger boards of the Xilinx family could allow the use of even 15 - 16 memory blocks, that would make our design having a really impressing processing speedup.

Another improvement could be the separation of each core in different parts responsible for the execution of a specific task, thus allowing pipeline execution of commands. Another way to achieve pipelined processing would be the separation of the cores into groups of cores that would still work in parallel but would perform specific tasks and communicate with each other.

Taking advantage of the faster processing ability we could support a bigger instruction set. This would make the parallel network processor more complete and able to execute a variety of tasks and programs. Last we could alter the commands in the instruction memories in a dynamic way during the processing, since now the instruction memories are filled with instructions using coe files, before the processing begins.

Bibliography

[1] "Development of a Network Processor in Reconfigurable Logic"
Costas Stefanatos, Diploma Thesis - Technical University of Crete (TUC) [Diploma Thesis]

[2] "Computer Organization and Design : The Hardware/Software Interface"
David A. Patterson, John L. Hennessy - Third Edition [Book]

[3] "Computer Architecture: A Quantitative Approach"
David A. Patterson, John L. Hennessy - Third Edition [Book]

[4] Wikipedia, the free encyclopedia, Multi-core processor, article
http://en.wikipedia.org/wiki/Multi-core processor [Web site]

[5] Wikipedia, the free encyclopedia, Network processor, article
http://en.wikipedia.org/wiki/Network processor [Web site]

[6] Wikipedia, the free encyclopedia, Parallel Processing, article
http://en.wikipedia.org/wiki/Parallel processing [Web site]

[7] Wikipedia, the free encyclopedia, Parallel Computing, article
http://en.wikipedia.org/wiki/Parallel computing [Web site]

[8] Wikipedia, the free encyclopedia, CELL (microprocessor), article
http://en.wikipedia.org/wiki/CELL [Web site]

[9] IEEE Multicore is bad news for supercomputers, article
http://spectrum.ieee.org/computing/hardware/multicore-is-bad-news-for-supercomputers [Web site]

[10] Intel Multicore technology, article from Intel website
http://www.intel.com/multi-core/ [Web site]

[11] Wikipedia, the free encyclopedia, Asychronous Array of Simple Processors, article
http://en.wikipedia.org/wiki/Asynchronous$_array_of_simple_processors[Website]$

[12] Wikipedia, the free encyclopedia, Intel Corporation, article
http://en.wikipedia.org/wiki/Intel Corporation [Web site]

[13] Wikipedia, the free encyclopedia, Advanced Micro Devices, article
http://en.wikipedia.org/wiki/AMD [Web site]

[14] Wikipedia, the free encyclopedia, Round Robin Scheduling, article
http://en.wikipedia.org/wiki/Round-robin scheduling [Web site]

[15] EE Times, News and Analysis, CPU designers debate multi-core future, article
http://www.eetimes.com/electronics-news/4076123/CPU-designers-debate-multi-core-future [Web site]

[16] Bryan Schauer, Discovery Guides 2008, Multicore Processors - A Necessity
www.csa.com/discoveryguides/multicore/review.pdf [PDF]

[17] IEE Rev. – September 2005 – Volume 51, Issue 9, p.3235 , Two Heads Are Better Than One,
W. Knight, September 2005 [PDF]

[18] IEEE CS, March/April 2007, Multicore Processors for Science and Engineering, P. Frost Gorder
http://ieeexplore.ieee.org/xpl/freeabs$_all.jsp?arnumber = 4100923[Paper]$

[19] D. Geer, Computer, IEEE Computer Society, May 2005, Chip Makers Turn to Multicore Processors
http://ieeexplore.ieee.org/xpl/freeabs$_all.jsp?arnumber = 1430623[Paper]$

[20] Xilinx, Virtex 5 - Family Overview
www.xilinx.com/support/documentation/data$_sheets/ds100.pdf[PDF]$

[21] Xilinx, Virtex 6 - Family Overview
www.xilinx.com/support/documentation/data$_sheets/ds150.pdf[PDF]$

[22] Xilinx, Xilinx DS550 Virtex-5 Embedded Tri-Mode Ethernet Wrapper v1.5, Data Sheet
www.xilinx.com/support/documentation/ip.../v5$_emac_ds550.pdf[PDF]$

[23] Xilinx, Xilinx UG340 Virtex-5 Embedded Tri-Mode Ethernet MAC Wrapper v1.5, Getting Started Guide
www.xilinx.com/support/documentation/.../v5$_emac_gsg340.pdf[PDF]$