PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE ENGENHARIA/ FACULDADE DE INFORMÁTICA

COMPUTER ENGINEERING

DEVELOPMENT OF MPSOCS MANAGEMENT SYSTEMS IN MODERN FPGAS

Felipe Todeschini Bortolon

Final Project Proposal

Prof. Fernando Gehm Moraes, Ph.D.

Porto Alegre

2015

Felipe Todeschini Bortolon

DEVELOPMENT OF MPSOCS MANAGEMENT SYSTEMS IN MODERN FPGAS

Final project work is presented as part of the activities to obtain the Bachelors degree of Computer Engineering at the Faculty of Engineering at the Pontifical Catholic University of Rio Grande do Sul.

Prof. Fernando Gehm Moraes, Ph.D.

ADVISOR

FELIPE TODESCHINI BORTOLON

DEVELOPMENT OF MPSOCS MANAGEMENT SYSTEMS IN MODERN FPGAS

Trabalho de conclusão de curso de graduação apresentado nas Faculdades de Engenharia e de Informática da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial para obtenção do grau de Engenheiro de Computação.

Aprovada em _____ de _____ de _____.

BANCA EXAMINADORA:

Fernando Gehm Moraes

_____

Matheus Trevisan Moreira

_____

Sérgio Johann Filho

_____

To Sandra Todeschini Bortolon, Antônio Francisco Bortolon, Catherine Bortolon and Taís Oliveira Junges for supporting me.

ACKNOWLEDGEMENTS

“If I have seen further than others, it is by standing upon the shoulders of giants.”
Sir Isaac Newton

I would like to express the deepest appreciation to those that contributed for this final project.

To my colleagues and friends from GAPH (*Grupo de Apoio ao Projeto de Hardware*) for the excellent work environment and friendship throughout all this years. Specially, Arthur Posser Trentin, Carlos Henrique, Guilherme Madalozzo and Marcelo Ruaro for the technical discussions and all help.

To Prof. Dr. César Augusto Missio Marcon, for giving my first scholarship and introducing me to academic life.

Specially to my advisor, Prof. Dr. Fernando Gehm Moraes for his patience, dedication, unceasing help, revision of this document, and mainly his honesty. I thank him for trusting on me and in my work.

“Computers are incredibly fast, accurate, and stupid. Human beings are inaccurate, incredibly slow, and brilliant. Together they are powerful beyond imagination”.

Albert Einstein

RESUMO

Utilizar múltiplos processadores em um único chip, i.e. Sistema em Chip Multiprocessados (MPSoC), podem prover o paralelismo necessário para processar as altas taxas de dados que algumas aplicações têm, e.g. processamento de video em tempo real. Projetos que não estão preocupados com o consumo de energia podem se beneficiar do baixo custo de *Field-Programmable Gate Arrays* (FPGAs) para prototipar uma MPSoC. Além disso, os FPGAs modernos são embarcadas com processadores *hard-core* que podem ser utilizados para facilitar a gerência e análise das MPSoCs. Este trabalho utiliza o dispositivo Zynq-7020, o qual é composto de dois microprocessador ARM Cortex-A9 e uma lógica programável Artix Série 7 da Xilinx. Este microprocessador é disponibilizado por diversos vendedores, e.g. NXP e Xilinx, em diferentes combinações de recursos dentro do chip e periféricos. A combinação da alta variedade de dispositivos que utilizam processadores ARM e sua popularidade tornam possível migrar este projeto para dispositivos similares de acordo com os recursos que forem necessários. Este trabalho desenvolveu uma plataforma que utiliza a lógica programável destes dispositivos para prototipar uma MPSoC, a Hermes MPSoC (HeMPS), e os processadores embarcados para gerenciar todo sistema. A HeMPS MPSoC foi modificada para utilizar os processadores ARM como os gerentes globais ao invés do processador Plasma. Isto prove uma interface para acessar os periféricos, além da capacidade de gerência da MPSoC. Foi criado uma interface de comandos, utilizando o protocolo Telnet, que possibilita emulação em tempo real. Desta maneira, um computador remoto pode transmitir aplicações para o microprocessador ARM e este ficará responsável por aloca-las nos processadores disponíveis da MPSoC. Para estabelecer a conexão entre o padrão de comunicação AXI da ARM e o utilizado pela MPSoC, um novo módulo de hardware foi desenvolvido, chamado de *Network Interface ARM-Hermes* (NIAH). Este módulo é reponsável por converter o protocolo ARM para o protocolo da NoC, e vice versa. O projeto resultante utilizou 17% dos recursos totais do dispositivo e atingiu uma frequência de 51 MHz. O *software* trata requisições tanto do computador remoto como dos processadores da MPSoC. O kernel original do IP do Plasma foi reescrito para os *cores* do ARM para que estes executassem uma parte das funcionalidades originais.

Palavras-chave: MPSoC, FPGA, Sistemas de gerência, Zynq-7000.

ABSTRACT

Using multiple cores on a single chip, i.e. Multiprocessor System-on-Chip (MPSoC), can provide the necessary parallelism to process the high incoming data rate that some applications have, e.g. real-time video processing. Designs not constrained by power can benefit from Field-Programmable Gate Arrays' (FPGAs) low-cost to prototype MPSoCs. Moreover, modern FPGAs are embedded with hard-core processors that can be further explored to ease MPSoC management and analysis. This work uses a Zynq-7020 device, which contains two ARM®Cortex-A9 cores and an Artix-7 Series programmable logic. Many vendors offer this microprocessor, e.g. NXP and Xilinx, with different combinations of on-chip resources and peripherals. The combination of this wide offering and its popularity makes it possible to migrate this design to other similar devices according to project needs, i.e. different resources. This work creates a platform that uses the programmable logic to prototype an MPSoC, the Hermes MPSoC (HeMPS), and the embedded processors to manage the complete system. The HeMPS MPSoC was modified to use the ARM processors as a global manager element instead of original Plasma IP. This provides an interface to access peripherals combined with MPSoC management capabilities. It was created a command interface using Telnet enabling real-time emulation. Thus, a host computer can transmit applications to the ARM processor that will be responsible for allocating them in available MPSoC processors. To establish the connection between the ARM AXI and the target MPSoC standards a new hardware module was created, the Network Interface ARM-Hermes (NIAH). This hardware converts the ARM protocol to the NoC protocol, and vice-versa. Resulting prototyped design occupies 17% of total device resources and achieved a 51 MHz clock frequency. The software handles both the host computer and MPSoC processors requests. Original Plasma IP master kernel was rewritten for ARM cores to perform a subset of its features.

Keywords: MPSoC, FPGA, Management Systems, Zynq-7000.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AMBA	Advanced Microcontroller Bus Architecture
APSoC	All-Programmable System-on-chip
API	Application Programming Interface
APU	Application Processing Unit
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ASMBL	Advanced Silicon Modular Block
AXI	Advanced Extensible Interface
BRAM	Block Random Access Memory
CLB	Configuration Logical Block
CPU	Central Processing Unit
DMA	Direct Memory Access
EDA	Electronic Design Automation
ELF	Executable and Linkable Format
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
IC	Integrated Circuit
IP	Intellectual Property
LUT	Look- Up Tables
MPE	Media Processing Engine
MPSoC	Multiprocessor systems-on-chips
NI	Network Interface
PL	Programmable Logic
PS	Processing System
TFTP	Trivial File Transfer Protocol
RAM	Random Access Memory
ROM	Read Only Memory
SoC	System-on-chip
SSI	Stacked Silicon Interconnection
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuits

CONTENTS

1 INTRODUCTION

In 1985, Xilinx shipped its first Field Programmable Gate Array (FPGA), the XC2064; offering 800 gates, sold for $55, and was produced in a 2.0µm process [ROE99]. It was the beginning of a new era for circuit designers. Before programmable logic, design changes, after the specification was finished, were often unfeasible mostly due to cost increase and manufacturing delays. After FPGAs arrival, new features could be added up to fabrication time without the same impacts. Furthermore, project duration decreases while complexity increases thanks to the fast-growing assortment of intellectual property (*cores*) use [COL99].

The component density per integrated circuit (IC) had also been doubling at regular intervals – accordingly to Gordon E. Moore [SCA97], and thus allowing Systems-on-Chip (SoC) to emerge. A SoC is an integrated circuit that implements most or all of the functions of a complete electronic system [JER05]. Classically, they were projected as Application Specific Circuits (ASIC) until FPGAs, which comprise programmable logic arrays, started to arise. Note that FPGAs may further embed central processing units (CPUs) and peripherals creating a heterogeneous device. There are several advantages of FPGAs over ASIC as, for example, higher flexibility, reduced development time, and design cost. On the other hand, FPGAs have significant power consumption and thus might not suit low-power systems.

Systems with a single processor may not provide enough performance for some applications, e.g. real time video processing. Therefore, using multiple processors may provide the necessary computational concurrency required to handle incoming data rates. This architecture is called Multiprocessor System-on-Chip (MPSoC). Moreover, the cost pressures inherent in large markets, as well as the fact that many of the applications have strict performance, power and energy requirements makes MPSoCs a sensible solution [WOL04].

Xilinx provides a flexible platform for implementing MPSoCs: The Zynq ‘All-Programmable SoC’ (APSoC). It comprises two main parts: a Processing System (PS) formed around a dual-core ARM Cortex-A9 processor, and a Programmable Logic (PL), which corresponds to reprogrammable logic fabric. Both sections are linked by standard Advanced eXtensible Interface (AXI) connections. While the PL section is appropriate for implementing parallel logic, the PS supports software routines and even an operating system. Thus, the overall functionality of an embedded system can be appropriately partitioned between hardware and software [CRO14].

ARM processors are largely used nowadays, e.g. cellphones, due to their performance and availability. Several manufacturers provide their version with different hardware combinations that might be used in different applications. Moreover, the high availability of ARM based systems makes it a low cost solution.

The *goal* of this work is to exploit Zynq’s ARM processors to manage an embedded MPSoC in the PL portion. The processor creates an interface between a host computer and the MPSoC. Therefore, it is possible to inject data into the system and return the results to the computer.

1.1 Motivation

Embedded systems designs become every generation a more viable solution for medical, military, mobile and other daily applications. The reason for that is the increasing demand for devices capable of performing complex operations. Although the field of work might not be novel, it includes several main areas covered throughout the Computer Engineering Bachelor's degree and future technologies trends. The target Xilinx platform is a brand new solution that embraces hardware and software, hence permitting a thorough study about current design challenges, concerns, and advantages. For those reasons, investigate those devices and apply concepts learned throughout the course are the focus of this final project.

1.2 Objectives

This final project has as its primary goal to apply concepts learned throughout the Computer Engineering course and to acquire knowledge about systems design. To fulfill the main goal, the specific objectives include:

- Understand FPGA technology with embedded ARM processors;
- Understand the communication techniques between software and reconfigurable components of modern FPGAs;
- Understand an industrial design flow using Electronic Design Automation (EDA) tools, such as Vivado, to quickly built complex designs;

This work intends to fulfill those milestones creating a system abstractly depicted in Figure 1. The Target Device consists of an MPSoC prototyped in the Programmable Logic and an ARM processor that manages this system. The processor is an interface between the Programmable Logic and a Host computer. Initially, the host transfers applications to the processor, which are going to be stored in the memory (1). Afterwards, the host issues a request (2) to execute any of the available applications. Then, ARM processor reads the memory and transfer requested application to the Programmable Logic (3). After the operation is complete, results are transferred back to ARM processor (4) and sequentially to the host machine (5).

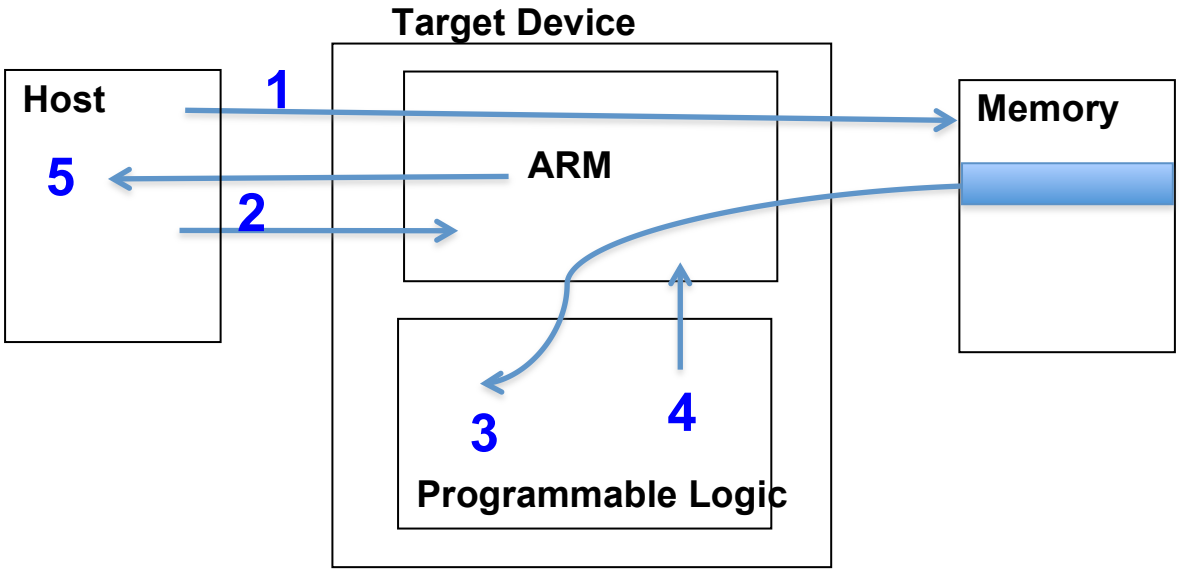

Figure 1 – Final Project abstract design.

1.3 Document Structure

After the introduction and contextualization are set forth by this first Chapter, the second Chapter provides (*i*) a description of conventional FPGA structure, (*ii*) presentation of HeMPS MPSoC and (iii) Vivado EDA. The third Chapter discusses the target board – Zedboard Kit, which resources were used and how they were configured. The fourth Chapter describes ARM AXI and Hermes interfaces, and presents NIAH hardware module, which was developed in this work. Chapter five describes the complete hardware and software set up created. The design is validated and analyzed in Chapter six, followed by Chapter seven that presents the conclusions.

2 THEORETICAL REFERENCE

This Chapter presents FPGA related concepts, MPSoC architecture details and the Vivado toolset used throughout this work.

2.1 Field Programmable Gate Array (FPGA)

FPGA introduction in the market changed electronic circuits design workflow. Reprogramming a device in the field merely to correct faulty behavior or to add a new feature became possible. Despite this capability, a hardware developer should understand how FPGA works, down to its essence, for efficient utilization of available resources. This section provides a brief comparison of programmable devices over ASIC and an overview of its basic building blocks [DEH08].

One of the FPGA's advantage is programmability. The same reason that enables this feature, i.e. configurable logic circuit, hinders its final area, timing and performance gain to achieve ASICs characteristics. Those are optimized to specific applications and thus save design area, consume less power, reduce timing delays and enhance computational performance. Even if FPGA is not as efficient as ASIC, its flexibility and reduced development cost are beneficial for many areas, e.g. scientific research. Moreover, with FPGA industry can release cheaper versions of a new product and, if successful, design a new ASIC version. To reuse circuit logic, i.e. FPGA fabric logic, grants less time to finish products and reduces costs.

When dedicated embedded CPUs are available, as Zynq's ARM Cortex-A9, features must be wisely separated to both exploit the reconfigurable logic circuit and the processor, which is programmed by software. Sequential code often takes advantage of processor's characteristics, while heavy tasks might use hardware's parallel nature to reduce computation time. Software implementation and management takes advantage of available libraries and drivers. Thus, it is possible to develop sophisticated and complex systems to monitor and control the device peripherals. Choosing where logic will be mapped, either software or reconfigurable logic, is the first action to better use devices with this dual nature, i.e. software and hardware design.

A field-programmable gate array (FPGA) is, in a simplified form of view, a set of configurable logic blocks embedded in a general routing structure and input/output cells around the device (Figure 2). Moreover, those blocks are interconnected by configurable interconnections. The complete functionality is partitioned and divided among logic blocks, and the interconnections are properly configured.

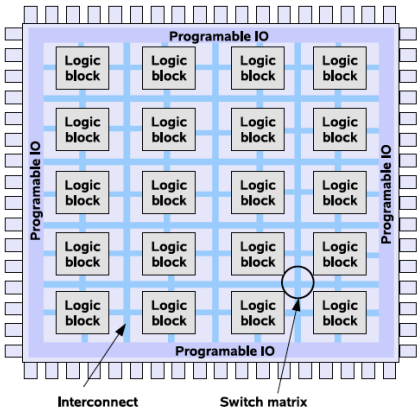

Figure 2 - A simplified view of FPGAs architecture [BOB07].

2.1.1 Configuration technologies

Basically, there are two major hardware configuration technologies: antifuse and memory-based. While antifuse paradigm is restricted to interconnections realization, memory-based is used for both computation and interconnections [BOB07]. The last technology is further categorized into Static Random-Access Memory (SRAM), Electrically Erasable Programmable Read-Only Memory (EEPROM), and Flash.

Antifuse's customizable connections includes a special component that is initially in open state (Figure 3 shows an example of Vialink's antifuse technology). When programming the device, high voltages are applied to melt the dielectric and, thus create a permanent connection. That is why this paradigm is also called one-time programmable FPGAs.

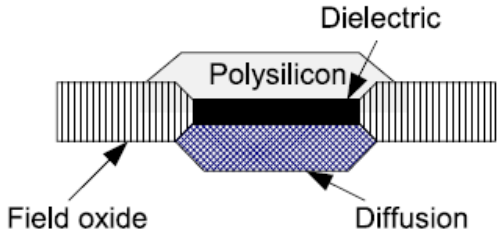

Figure 3 - Q-Logic Vialink antifuse technology [BOB07].

SRAM memory-based technology is the most widely used method for storing configuration in commercially available FPGAs [DEH08]. Differently from antifuse, SRAM memory-based technology enables to configure both interconnections and logic blocks through the output of SRAM cells (Figure 4). The connection between two wires can be made using *pass transistors* (Figure 5(a)), thus allowing, or not, current to pass. Logic blocks, also called function-generators, use the same concept. Functionality is reprogrammed just by changing multiplexers (Figure 5(b)), and look-up tables (LUTs) (Figure 5(c)) stored values.

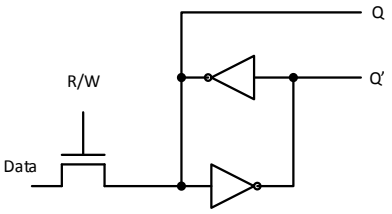

Figure 4 - Xilinx SRAM cell.

One of the SRAM advantages is device's reconfiguration in well-known technology. On the other hand, this approach requires large amounts of area for both connections switches and logics blocks. Furthermore, since the device is volatile, i.e. it does not retain configuration when power is cut off, it is necessary to embed non-volatile storage to reload configuration at power-on.

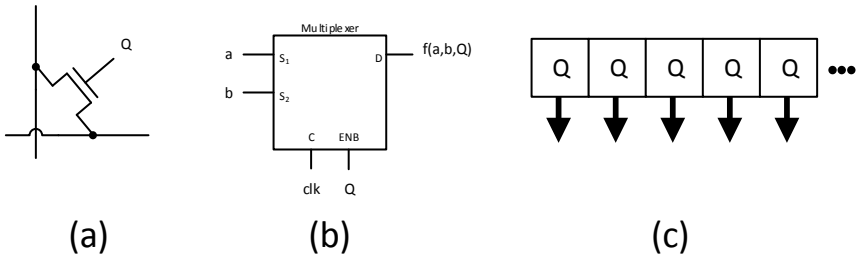

Figure 5 - A representation of a (a) Pass transistor, (b) programmable MUX and (c) function generation.

2.1.2 Function Generators

Two of the possible functions generators are multiplexers and look-up tables (LUT). A *multiplexor* is a switching device that, according to its control signal, will feed the appropriate input line to the output. An LUT is a group of memory cells, which stores functions' truth tables, and a decoder, which is used to access correct memory location and retrieve results (Figure 6) [BOB07]. An LUT is also called "universal logic gate", since it enables to implement any logic function of up to *n* variables, where 2^n represents the number of bits to store in the truth table.

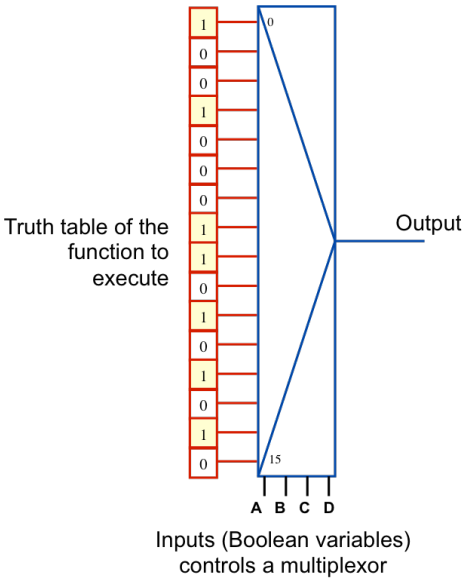

Figure 6 – An example of a 4-input LUT.

2.1.3 Interconnections

Through the combination of multiple LUTs and multiplexers it is possible to implement complex circuits. Combining multiple logic blocks requires programmable interconnection structures that appropriately interconnect LUTs.

Initially, the form of communication was called *nearest neighbor* [DEH08]. As the name suggests, each block may directly communicate with each immediate neighbor. The major disadvantage of this approach is scalability. The delay increases between distant nodes because it needs to traverse one or more logic blocks. A possible solution is to separate the communication from computation, just as *segmented* paradigm proposes. Two new blocks are required to achieve such purpose: the connection block (CB) and the switch box. Logic blocks, which contains one or more LUTs, access nearby resources through CB, and this links the first's I/O to routing resources through programmable switches, or multiplexors (Figure 7(a)).

Figure 7(b) details connections between such blocks in Xilinx devices. Configurable Logic

Blocks (CLB) are directly connected to switch matrixes (or switch boxes) through wires. Those may have different lengths depending on FPGA's size: single, hex (a signal can traverse up to 6 CLBs) and long (a signal can traverse one-fourth of the FPGA length).

There are several techniques available (such as hierarchal and row based) and this still is an interesting field of research, however, this document is restrained to this brief overview.

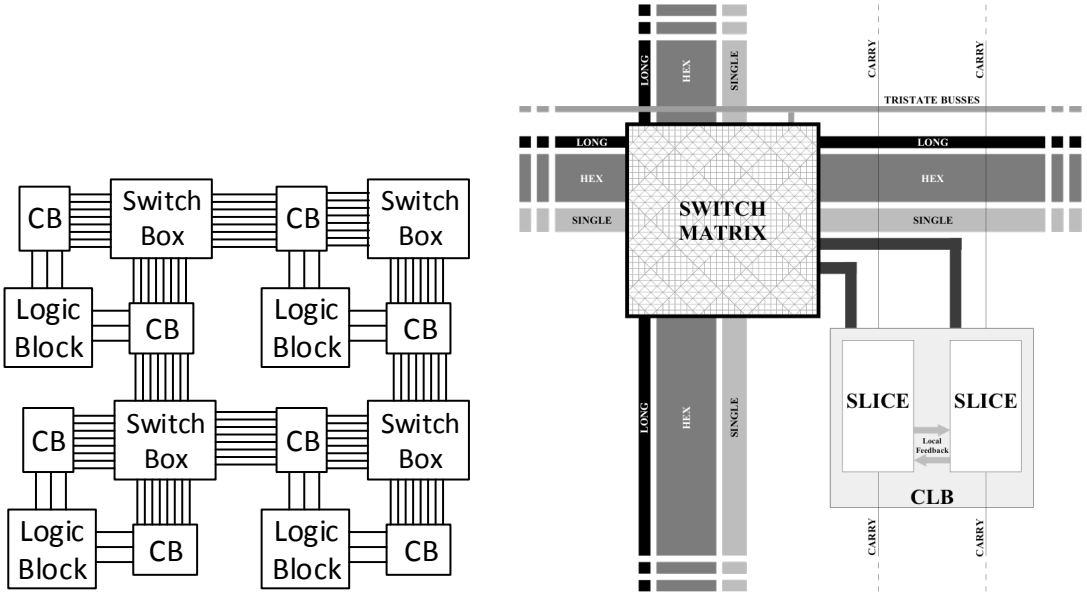

Figure 7 – An abstract representation of communication segmentation (a), and a detailed view of the routing segments.

2.1.4 7 Series Xilinx FPGA

Although any combinational and sequential logic may be implemented through the combination of several LUTs, the number of available resources restrains design size. Hence, to optimize device utilization, extending basic logic block with specific elements, e.g. fast carry chain, multipliers, and RAM; is current industry approach. To illustrate this, let's look over available industry technology.

Xilinx 7 Series FPGA basic block, the Configurable Logic Block (CLB), contains a pair of slices [XIL14]. Slices are not directly connected but organized as independent columns to create a carry chain (COUT and CIN) with other CLBs (Figure 8). They may easily be configured in several different ways to behave as shift registers, storage elements or arithmetical logic and thus enhance device's resource utilization.

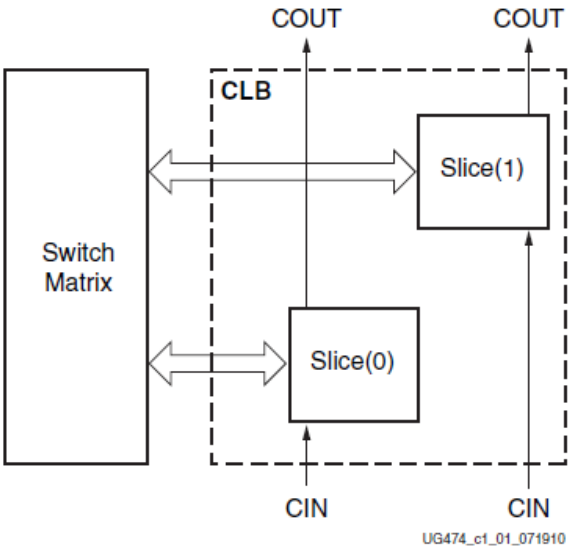

Figure 8 – Xilinx CLB layout [XIL14].

There are two different types of slices, i.e. SLICEM and SLICEL, each containing four LUTs, eight storage elements (flip-flops), wide-function multiplexers and carry logic. Although they virtually have the same elements, whereas both provide logic, arithmetic and ROM functions, only SLICEM supports distributed RAM and data shifter behavior. Each CLB is either composed by two SLICEL or one of each type.

The CLBs are arranged in a regular array, and may access general routing resources through the Switch Matrix that runs vertically and horizontally between CLB rows and columns. Xilinx also created the Advanced Silicon Modular Block (ASMBL) architecture to optimize FPGA for different application domains, and extended integration by using Stacked Silicon Interconnect (SSI) technology [XIL12]. Discussions about those topics are beyond the scope of this work.

Altera, for example, has a different solution from Xilinx, but both companies aim to achieve the same goal: enhance resource usability and thus increase performance. Xilinx's solution was presented in this work because target board uses their technology.

2.2 Hermes Multiprocessor System (HeMPS)

The result that this final project aims to achieve is an MPSoC managed by hard-core processors (Section 1.2). Hence, this section introduces the Hermes Multiprocessor System (HeMPS) MPSoC that is used throughout this work.

The HeMPS [CAR09] is a homogeneous NoC-based MPSoC. The main hardware components that HeMPS comprises are *(i)* Plasma-IP processing elements (PEs) and *(ii)* Hermes NoC [MOR04] communication infrastructure (Figure 9). In addition, there is a memory, called *task repository*, which holds all tasks' object codes that will be executed in the PEs.

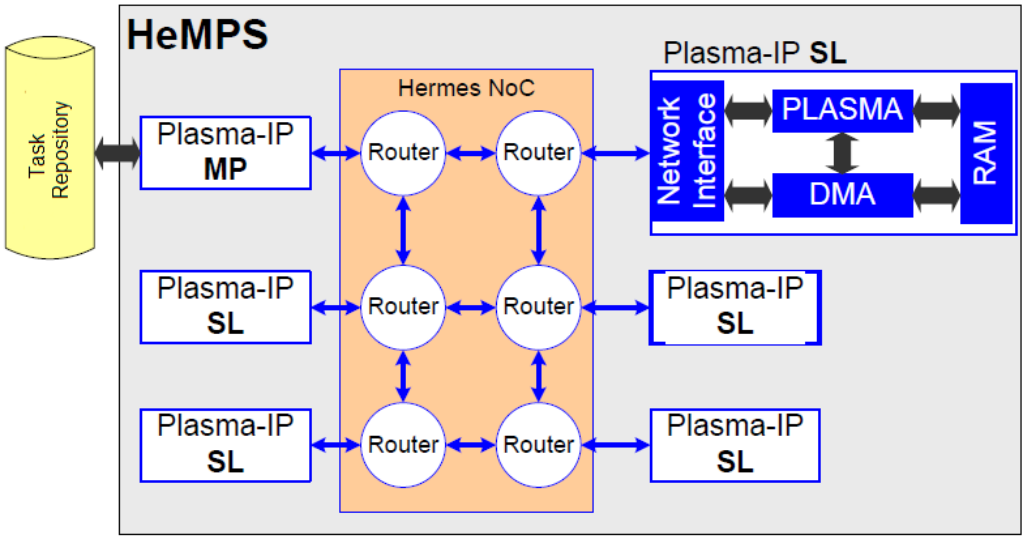

Figure 9 – HeMPS instance using a 2x3 mesh NoC [CAR09].

The platform is organized in a *master-slave* structure, which is composed by several slaves PE (Plasma-IP SL) and one master (Plasma-IP MP). The master processor is responsible for managing system resources since it is the only with access to the task repository. The slaves execute the tasks sent by the master. When HeMPS starts execution, the master allocates initial tasks, i.e. produce data to other tasks or do not communicate, to slave processors. Throughout execution, tasks will be continuously loaded from task repository to the slaves. Furthermore, the master is also responsible for responding requests from the slaves. Those may request information about other tasks or inform debug messages.

To simulate HeMPS, there are scripts that automatically generate the hardware and the repository filled with tasks (defined by scripts' input). The repository is a VHDL file that is accessed by the master processor through the testbench.

Further description about the two main HeMPS hardware components – the network and the processing element is covered in the following sections.

2.2.1 HERMES NoC

A NoC may be compared to traditional telecommunication network, where each host is connected to a single router, which is subsequently connected to others routers. Therefore, one host may send and receive messages from others hosts through the network. The same general idea is applicable to chips: microprocessors are connected to a simple routing hardware, i.e. the router, and those to another router, thus creating a network inside the chip.

HERMES NoC is a scalable 2D-mesh network. The NoC contains several routers, placed and identified by the Cartesian plane, where the leftmost lower corner represents the origin (Figure 10). Each node contains centralized switching control logic, four bidirectional ports to interconnect to other nodes (east, west, north and south), and one bidirectional port to communicate with local PE. Arbitration among those ports is done through *Round-Robin* algorithm, which provides a fairer service than static priority system (avoiding starvation).

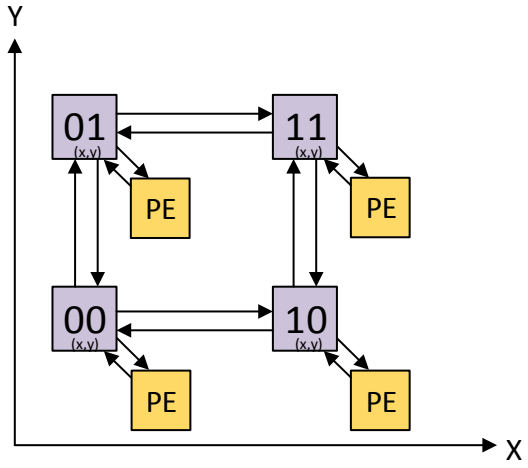

Figure 10 – 2x2 NoC mesh topology abstraction.

HERMES uses *wormhole* switching mode, in which packets are broken down into several *flits*. The first flit, known as header, contains packet destination information while the remainder, called *payload,* contains data itself. Based on header's information, nodes will know which way a packet must be sent by comparing current and requested Cartesian position. Since this implementation uses XY routing algorithm [CAR09], flits will first traverse horizontally until the right column, and then vertically. Besides that information, a node must know when communication is over. Hence, the first payload flit contains total data length that must be received. Consequently, the router is able to identify the last flit and finish communication.

2.2.2 Plasma-IP

Plasma CPU is a small synthesizable 32-bit RISC microprocessor encompassing a subset of the MIPS instruction set. Its code description is open source and is provided by *OpenCores* [RHO15]. The processor has a three-stage pipeline (fetch, decode, and execute), and uses Von Neumann memory architecture. The processor used by HeMPS contains modifications from original version, e.g. addition of interruption mechanisms, UART removal, and new memory-mapped registers.

To increase processing elements performance, Plasma-IP architecture separates communication from computation. While Plasma CPU is responsible for processing, the Network Interface (NI) and Direct Memory Access (DMA) modules manage communication. Moreover, local RAM is simultaneously accessible by the DMA and processor through a dual-port memory, which avoids extra hardware for elements like *mutex* or cycle stealing techniques. Nonetheless, area cost increases to provide this feature.

The NI is responsible for receiving data from the processor, segmenting into flits and sending it to the network. Naturally, it also handles the inverse process. When data is available, NI interrupts Plasma processor that consequently configures the DMA to transfer data from the buffer to the memory. Once data transfer finishes the DMA informs the processor through an interruption request.

2.3 Vivado Design Suite

Vivado Design Suite [FEI12] is a new IP and system-centric design environment that accelerates design productivity for All-Programmable devices, i.e. a new Xilinx technology. The

main tools provided in this suite are Vivado IDE (Integrated Development Kit) for hardware design and Vivado SDK (Software Development Kit) for software development. Also, there is a tool capable of converting C code to hardware descriptions (using VHDL or Verilog) with few changes in the source code, called Vivado HLS (High-Lever Synthesis). The advantages provided by Xilinx tools to this final project are the large number of available IP cores, tutorial material and previous students work. This Section covers basic information about this toolset, however Soares' contributions [SOA14] provides a complete overview.

The complete project flow, i.e. hardware and software, (Figure 11) begins with Vivado IDE to create the hardware. First, it is created a block diagram, which represents the interconnections between all modules. It is possible to customize available IPs or create a new in this step. The tool verifies if all interconnections are correct and warns the user of possible mistakes. The next steps are logical and physical synthesis. Vivado generates reports, e.g. timing and utilization, after each of these steps. Therefore, the user can identify synthesis timing problems, for example, and fix them before proceeding. Floorplanning is might be performed in either logical or physical synthesis. Furthermore, after the design is complete, i.e. software and hardware, Vivado IDE offers real time signal debugging capabilities. In order to use this feature, it is necessary to generate extra hardware that captures the signals that must be verified and transmits to Vivado.

After hardware design is complete, a bitstream (BIT) is generated and exported to Vivado SDK. The BIT file is used to configure the programmable logic with the created design. Now starts the software development step. Using the BIT file, SDK automatically generates a Board Supply Package (BSP) that contains register addresses, drivers from Xilinx IPs and other hardware settings. Moreover, it is possible to include automatically a few predefined libraries, e.g. Fat File System and lightweight Internet Protocol, to the project. Once software implementation is complete user will load the BIT file and the Executable and Linkable Format (ELF) object file [ARM98] to the board. The ELF contains the code (*.text*), initialized read-write data (*.data*) and statically allocated variables (*.bss*) for the executable. For further information about ELF characteristics and properties refer to ARM's documentation.

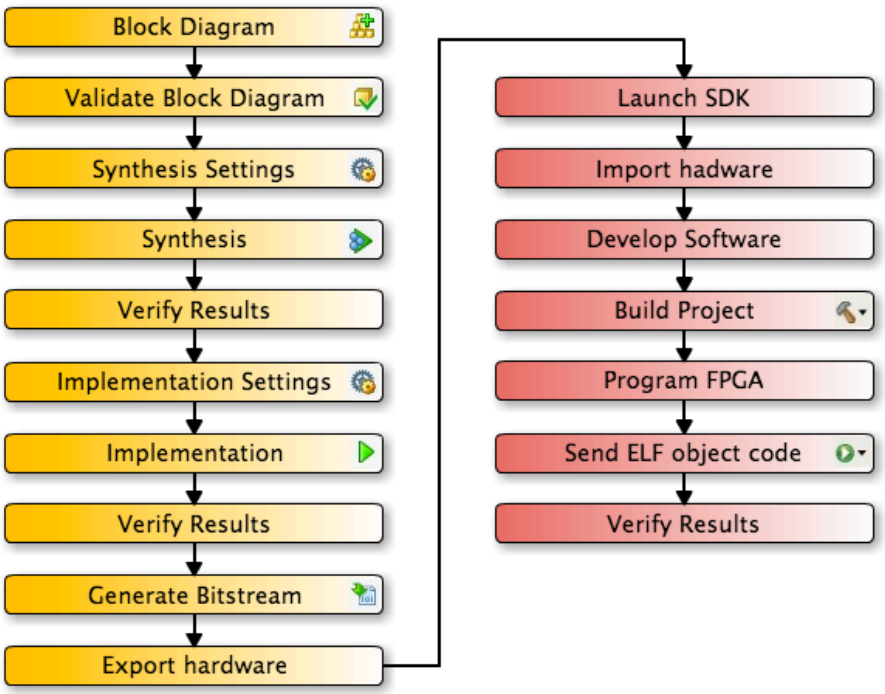

Figure 11 – Vivado design flow.

3 HARDWARE AND SOFTWARE INFRASTRUCTURE

This Chapter presents the target development board, Xilinx Zedboard APSoC [CRO14], and the peripherals used in this work. To prototype HeMPS MPSoC it is necessary to address a few issues such as task storage and HeMPS/ARM intercommunication. Therefore, it was necessary to use, modify and create software libraries that manipulate those peripherals and thus build an environment for HeMPS.

3.1 Zedboard

Zedboard is a low-cost development board for the Xilinx Zynq™-7000 All Programmable System-on-Chip (APSoC). It has a tightly coupled ARM® processing system and 7 Series programmable logic to create designs that combines software programming and hardware prototyping.

3.1.1 Processing System

The Zynq-7000 device contains a dual-core ARM Cortex-A9 processor with other processing resources forming an Application Processing Unit (APU). It further comprises peripheral interfaces, cache memory, memory interfaces, interconnects, and clock generation circuitry (Figure 12). This complete subsystem is named as Processing System (PS).

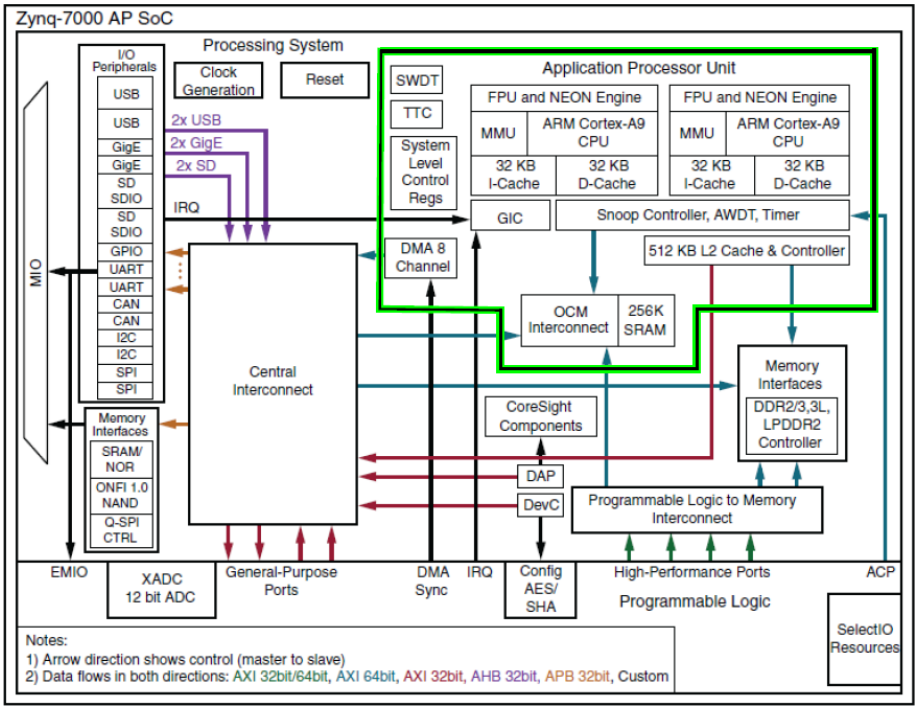

Figure 12 – The Zynq Processing System [CRO14].

The APU is primarily comprised of two ARM processing cores, each associated with a NEON™ Media Processing Engine (MPE) and Floating Point Unit (FPU) computational units; a Memory Management Unit (MMU); and Level 1 cache memory (Harvard architecture). It further contains a shared Level 2 cache memory, On-Chip Memory (OCM) and a Snoop Control Unit (SCU), responsible for memory coherency between L1 data cache and L2 cache. ARM processor maximum clock frequency is 866 MHz accordingly to Zynq-7000 specifications.

3.1.2 Programmable Logic

The Programmable Logic (PL) region is based on Artix®-7 FPGA fabric which primarily comprises several CLBs. In addition, it contains two special purpose components: Block RAMs for dense memory requirements and DSP48E1 slices for high-speed arithmetic. Both are integrated into the logic array and normally in proximity to each other because intensive computation and storage of data in memory are usually closely associated operations. Further information about Artix-7 FPGA capabilities may be obtained through Xilinx documentation [XIL14a].

3.1.3 Peripherals

This Section focus on defining which peripherals are necessary to create the system defined in Section 1.2. Moreover, HeMPS simulation configuration has a few characteristics that cannot be ported to the board. There is three questions that helps to understand which configurations must be changed and which peripherals were used:

- HeMPS' scripts create a VHDL file called *task repository* that contains information about all applications and their respective object code. A testbench uses this file to feed HeMPS MPSoC with data. However, this configuration is only valid for simulation. Therefore, where the applications will be stored and how a host computer will transfer those applications to the board?
- In HeMPS simulation the processing elements write their results into a *log* file and thus it is possible to verify that the system is working correctly. Nonetheless, when HeMPS is prototyped in the programmable logic, how the system behavior will be verified? Furthermore, how the developer will identify software problems in the ARM processor?
- This work main goal is to replace Plasma master IP, and its corresponding kernel, by the ARM Cortex-A9 core. How to connect this hard-core processor, located in the PS portion, with other Plasma IP cores, which will reside in the PL portion?

Those issues comprise essential design decisions for this project. The following sections answer each of those questions.

3.1.3.1 Smart Digital Card

As the first question poses, the testbench is an interface to access the repository. Scripts can generate different scenarios[1], but the system is restarted for each new simulation. Embedding HeMPS in the FPGA requires a flexible and runtime repository where it's possible to add new applications without power cycling, i.e. turn the board off, and on. Therefore, the platform should provide ways to easily insert, or remove, applications to a specific storage space.

Smart Digital Card (SDC) is a nonvolatile memory extensively used in portable devices. It offers 4 MBps transfer speed, low power consumption (ranges from 0.18W to 0.36W) and clock frequency of up to 50MHz [AVN12][SAN07]. Using external memory avoids consuming FPGA Block RAM (BRAM) resources. Moreover, placing the repository inside the FPGA logic might not even be interesting because the master is the only processing element that has access to it. Thus, that would create a loop in which data travels from PL to PS, i.e. the master reads the repository,

[1] A scenario contains the set of applications to be executed in the platform, as well as the moment they should start

and then back to the first, i.e. the master sends information to slave processing elements. The SDC provides enough transfer speed and more than sufficient storage space, given that the biggest application available in HeMPs library consumes 60 KB.

The used card has 4GB of memory and is configured with existing Fat File System [MIC05]. Xilinx provides a driver to use the SDC. Available routines were encapsulated into an Application Programming Interface (API) to automate read and write operations. Moreover, this API makes code cleaner and easier to manage SDC operations. The API routines were developed during this work.

3.1.3.2 Ethernet

Regarding the first and second questions, communication and observability respectively, both may benefit from Ethernet usage. A remote host can use Internet protocols to perform read and write operations on the SDC (3.1.4.1), issue command requests to the device (3.1.4.2), or even connect the board to other devices. Section 3.1.4 describes in more details the use of this peripheral.

3.1.3.3 UART

Universal Asynchronous Receiver/Transmitter (UART) is a hardware component used for serial communication. A software developer may use UART to create an environment similar to non-embedded software development where it is possible to visualize *printf* messages on the computer screen. Xilinx provides a function, i.e. *xil_printf*, which *transmit messages* through UART to a serial terminal, e.g. *gtkterm*, in the host computer. This feature significantly enhances software observability and *debugability*.

3.1.3.4 DMA

Direct Memory Access (DMA) allows certain hardware to access the memory independently of the CPU. Without DMA, the processor is usually occupied for the entire data read and write operation, and thus unavailable to perform other work. On the other hand, with DMA, the processor initiates the transfer and may perform any other operation while it is in progress. When it terminates, the DMA Controller will generate an interrupt request (IRQ) to inform the CPU. Similarly, when there is new information available for the CPU, an IRQ is generated.

Therefore, the DMA is used to answer the third question. Two buffers of 64KB each, inside OCM (On-Chip Memory), are used for write and read operations, respectively called *tx* (transmit) and *rx* (receive). The buffer size was inherited from available Xilinx tutorials, however it is possible to change it. The write operation should be understood as the ARM processor transferring data to a certain hardware, while the read, as ARM receiving data from that hardware. In write operations, data is written into the *tx* buffer and DMA is configured to perform the transmission. Whenever there is data available for reading, the DMA Controller interrupts ARM processor to process incoming data. Interrupts are configured to occur either after 125 cycles, after the first packet, [XIL10] or when 100 packets are received.

3.1.4 Software

Further addressing questions posed in 3.1.3, software implements necessary logic to use those peripherals. Aspects that were not completely discussed are which Internet protocols that control repository write and read operations, and command requests.

3.1.4.1 Trivial File Transfer Protocol

TFTP is a simple, lock-step, file transfer protocol that allows a host to download files to a remote host, in this case the board, or upload to it. Xilinx has its distribution that uses *Lightweight Internet Protocol* Library [DUN08]. It is a low memory footprint implementation suitable for embedded systems. However, it was necessary to modify the default storage system of Xilinx's implementation to use SDC peripheral. The SDC API, developed in this work, encompasses the necessary functions for those changes.

The repository combines both TFTP software and SDC peripheral. A computer, or other device, can send and receive files/applications through the network. For example, Linux OS provides a TFTP implementation that might be used. However, the user should be aware that the embedded implementation supports only *binary* operation mode. Therefore, OS software should be configured to match this requirement. This limitation doesn't have any performance impact since it concerns transfer characteristics only.

3.1.4.2 Command Interface

Telnet provides bidirectional interactive text-oriented communication using a virtual terminal connection over TCP/IP. The Command Interface uses this protocol to receive and send messages to a remote computer. To process incoming requests, there are three modules: Command Control, Parser and Command (Figure 13). The Command Control (*CmdControl*) unit listens on port 23, establishes telnet TCP connection and handles incoming requests (1). Next, the parser processes user's command (2) and identifies which command has been issued. If it is an unknown/invalid request or any parameter is missing no further processing takes place and the user is notified. On the other hand, it forwards requested code and necessary parameters to the Command unit (3). This executes user's command and sends the results back (4) to Command Control that will forward it to the user (5). This chosen port number was one the available and may be modified if necessary.

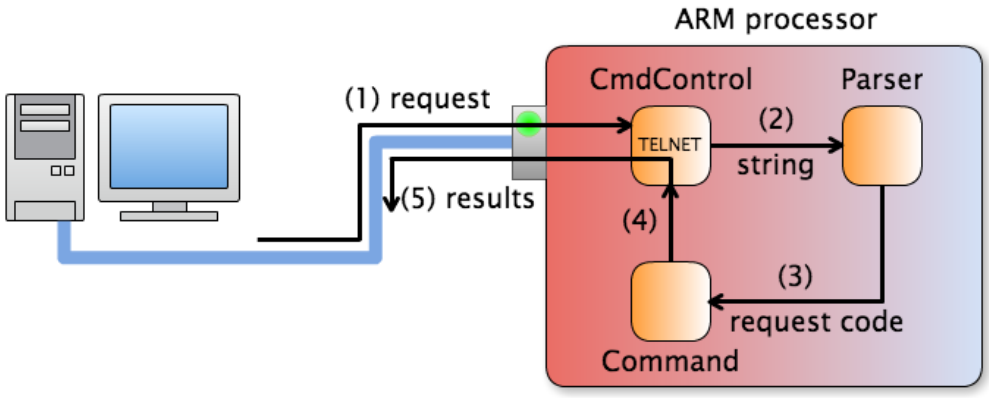

Figure 13 – Command Interface software flow.

3.1.4.3 Kernel

The Plasma kernel provides a system call, named *Echo*, which tasks might use to write messages to a log file during simulation. Whenever *Echo* is executed, the kernel executes a *puts* routine. This is an assembly code that writes data into Plasma IP UART (do not confuse with Zedboard UART). This message is then transmitted to the testbench file and will be written to a log file. A task example code and resulting log file is depicted are **Error! Reference source not found.**.

taskA.c

```
1  #include <task.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      int i;
7      unsigned int a;
8
9      Echo("taskA started!\0");
10     a = 0;
11     for(i = 0; i < 100000; i++)
12         a = a + 1;
13
14     Echo("taskA started!\0");
15     exit();
16 }
```

output0x1.txt

```
1
2  ECHO: task A started!
3
4  ECHO: task A finished!
5
6
7
```

Figure 14 – Example of a task (on the left) and resulting log file (on the right).

With this feature it is possible to analyze each task's execution flow and verify when it terminates. Zynq's PL portion does not have direct access to the UART. Nonetheless, it is possible to use HeMPS debug service to send messages to the ARM processor, which has access to it. Therefore, instead of using the internal UART, the kernel builds and transmits a debug packet.

The kernel has a reserved space in memory, called *pipe*, where messages are stored for transmission. DMA is responsible for reading the *pipe* and transmitting data to the NI that, consequently, forwards to the network. Debug packets are composed of:

- Destination address (*remote_addr*)
- Packet size (*pkt_size*)
- Service type (*service*)
- Processor identifier (*local_addr*)
- Task identifier (*target_id*)
- Message length (*length*)
- Message (*message*)

HeMPS kernel defines that a message may have a maximum of a hundred and twenty-eight flits. Since HeMPS is configured to use thirty-two bits flit size, each Message flit may hold four characters, given that one character requires eight bits. The implementation for Plasma slave kernel consists in procedures to received tasks from the master, execute them and transmit messages back. Figure 15 shows the *Echo* system call implementation for it. First, the task identifier and *Echo*'s message are extracted from the registers *arg1* and *arg0*, respectively. After header fields are properly set (lines from 233 to 237), *msg* is configured to point to the first pipe Message position. Then, while *Echo*'s message character (pointed by *ptr*) is different from *end-of-message* tag, i.e. *'\0'*, its value is stored in the pipe (*msg*). The variable *i* is used to count the number of characters stored in the message (*slot_ptr->message*). Based on its value, it is possible to calculate the packet length. Since each flit (thirty-two bits) holds four characters, *i* should be divided by four. Finally, *DMA_Send()* is invoked to perform data transfer.

This kernel modification will increase prototyped design observability and create a way to

determine whether tasks have been properly transferred to slave PEs. Messages from each task are forwarded to the ARM processor that will transmit to the user through UART. Therefore, it is possible to analyze whether tasks completed or not.

```
232 int Syscall(unsigned int service, unsigned int arg0, unsigned int arg1, unsigned int arg2)
233 {
234     char *msg;
235     // Syscall prepocessing method
236     switch(service)
237     {
238         // other Syscall services were ommited
239         case ECHO:
240             master_task = 0;                    // arm master address
241             target_id = (unsigned int) arg1;    // arg1 contains task id
242             ptr = (char *)((current->offset) | (unsigned int) arg0); // arg0 contains Echo message
243 
244             slot_ptr = GetFreeSlot();           // gets a temporary slot
245             slot_ptr->remote_addr = master_task;// packet destination
246             slot_ptr->service = DEBUG_MESSAGE;  // packet service
247             slot_ptr->local_addr = net_address; // local processor address
248             slot_ptr->target = target_id;       // task id
249             // each message character (ptr) uses 8 bits from
250             // slot_ptr->message, which is 32 bits
251             i = 0;
252             msg = (char *) &(slot_ptr->message[0]);
253             while(ptr[i] != '\0')
254             {
255                 *msg++ = ptr[i++];
256             }
257             *msg++ = '\n'; i++;
258             slot_ptr->length = (i) ? (i/4 + 1) : 0;
259             slot_ptr->pkt_size = slot_ptr->length + 5;
260 
261             DMA_Send(slot_ptr);
262         break;
263     }
264 
265     return value;
266 }
```

Figure 15 – *Echo* system call implementation.

4 HARDWARE INTERFACE

Both PL and PS communication are designed to benefit from the DMA. However, the DMA interface uses AXI-Stream protocol while Hermes uses credit-based protocol. This Chapter introduces the developed hardware that enables interconnecting those two standards. This new module, called Network Interface ARM-Hermes (NIAH), substitutes the Plasma IP Master to interconnect HeMPS router architecture with the new Master processor, i.e. ARM Cortex-A9. At one side, NIAH is connected to Hermes NoC (represented as a cloud), and on the other, to the DMA (Figure 16). Data from Hermes processing elements are converted to AXI-Stream signals and then written to the processor's OCM. The DMA reads the same OCM to transfers data to Hermes PEs through the NIAH, which now converts data from AXI-Stream to Hermes standards.

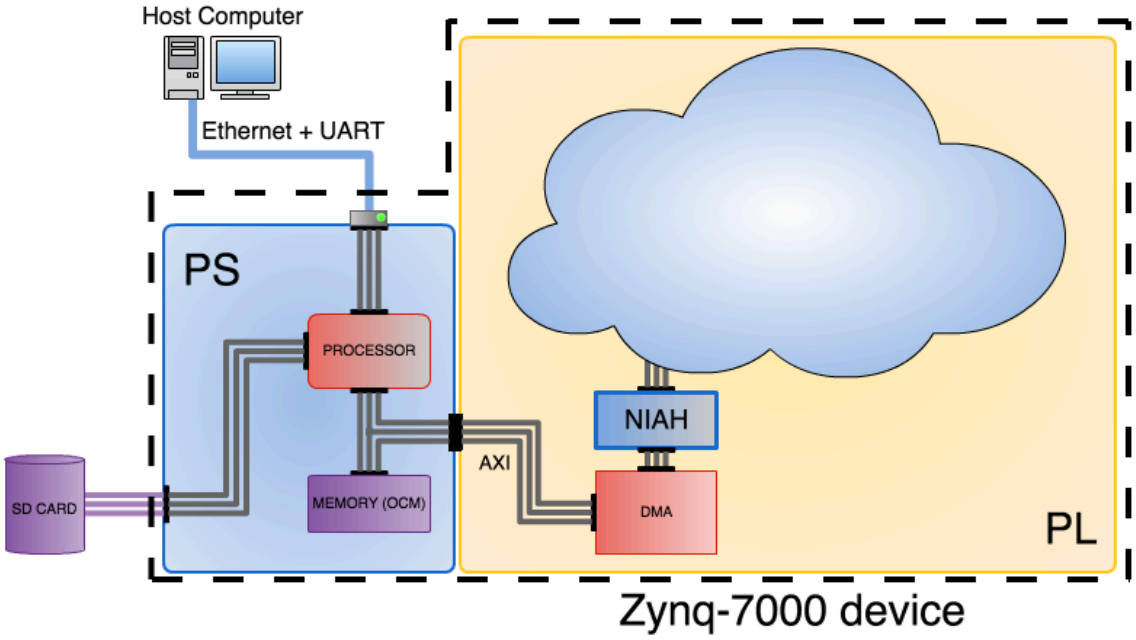

Figure 16 – Network Interface ARM-Hermes module interconnections.

The following sections introduce AXI-Stream protocol and Hermes credit based control flow. Then it presents NIAH module and validate its implementation.

4.1 Advanced eXternsible Interface Stream

AMBA ® AXI4-Stream protocol [ARM10] is a standard interface to connect components that wish to exchange data. The interconnect infrastructure supports multiple master to multiple slave configuration, arbitrary data width conversion, synchronous and an asynchronous clock rate conversion, multiple clock domains, among other capabilities.

The protocol consists of a two-way flow mechanism that enables both master and slave to control transfer rate across the interface. **TVALID** and **TREADY** handshake determines when information is transmitted. The required condition is that both signals are asserted, regardless of which was asserted first. After each transfer both signals are deasserted and, if there's more data, the process is repeated. **TLAST** indicates the last data and is set along with **TREADY**.

Table 1 summarizes the most important available interface signals and their purpose. Signals such as **TSTRB** and **TKEEP** offer advanced control over transmission, although NIAH implementation does not take advantage of them. Considering **TDATA** of thirty-two bits (four bytes), both **TSTRB** and **TKEEP** contains four bits, each associated with a byte of **TDATA**. Their values are fixed to logical one in this project because, otherwise, they indicate that associated byte is used for control purposes.

Table 1 – AXI4-Stream interface signals [ARM10].

Signal	Source	Description
ACLK	Clock Source	The global clock signal. All signals are sampled on the rising edge of **ACLK**
ARESETn	Reset Source	The global reset signal. **ARESETn** is active-LOW.
TVALID	Master	**TVALID** indicates that the master is driving a valid transfer. A transfer takes place when both **TVALID** and **TREADY** are asserted
TREADY	Slave	**TREADY** indicates that the slave can accept a transfer in the current cycle
TDATA[32]	Master	**TDATA** is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes
TSTRB[4]	Master	**TSTRB** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as a data byte or a position byte.
TKEEP[4]	Master	**TKEEP** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as part of the data stream. Associated bytes that have **TKEEP** byte qualifier deasserted are null bytes and can be removed from the data stream.
TLAST	Master	**TLAST** indicates the boundary of a packet.

4.2 Hermes Credit-Based Control Flow

When a given router A wishes to transmit data to its neighbor B, it must verify if the selected port has credit, i.e. there is space in B's target buffer. If that condition is true, router A properly sets **DATA_OUT** and signals router B through **TX** that there's data available. Router B decreases each transfer available credit, according to the buffer's space. While B offers credit, A may transmit new data on a cycle basis. Whenever credit is deasserted, router A must wait B to consume available data and increase available credit.

Hermes Credit-Based control flow interface comprises the signals described in Table 2. Analyzing two neighbor routers A and B, input signals from one are connected to the other's output. Thus, router A **CLOCK_TX** is connected to B **CLOCK_RX**; **TX** to **RX**, **CREDIT_I** to **CREDIT_O**, and **DATA_OUT** to **DATA_IN**.

Table 2 – Hermes Interface Signals

Signal	Source	Description
CLOCK	Clock Source	The global clock signal. Might be used as **CLOCK_RX** and **CLOCK_TX** source.
RESET	Reset Source	The global reset signal. **RESET** is active-HIGH.
CLOCK_RX	Target Router	The local clock between two routers. Used to synchronize Target Router transmission with Source Router reception.
RX	Target Router	**RX** indicates that Target Router is providing valid data. A transfer takes place when both **RX** and **CREDIT_O** are asserted.
DATA_IN[32]	Target Router	**DATA_IN** is the primary payload used by Source Router to receive data from Target Router.
CREDIT_O	Source Router	**CREDIT_O** indicates that Source Router is ready to accept data transfer.
CLOCK_TX	Source Router	The local clock between two routers. Used to synchronize Source Router transmission with Target Router reception.
TX	Source Router	**TX** indicates that Source Router is providing valid data. A transfer takes place when both **RX** and **CREDIT_I** are asserted.
DATA_OUT[32]	Source Router	**DATA_OUT** is the primary payload used by Source Router to transmit data to Target Router.
CREDIT_I	Target Router	**CREDIT_I** indicates that Target Router is ready to accept data transfer.

4.3 Network Interface ARM-Hermes

On one hand, AXI-Stream is a handshake protocol. On the other hand, Hermes is a credit-based control flow. Therefore, NIAH must operate accordingly on both interfaces. Two state machines control data conversion from DMA to Hermes, and from Hermes to DMA separately (Figure 17). AXI-Stream slave interface is connected to Hermes signals that drive data into the MPSoC, while AXI-Stream master is connected to those that drive data out, i.e. to the ARM processor.

Both state machines behave similarly, i.e. the same states and transitions (Figure 18), differentiating on data direction. One converts Hermes credit-based signals to AXI handshake, and the other performs the inverse operation. States correspond to expected packet format defined in Section 2.2.1. Hence, a packet should begin with the header and length flits, followed by a variable number of payload flits. For example, the FSM that receives data from the DMA, first processes the header (1 and 2) and the size (3 and 4) flits. Sequentially, it will loop over step (6) and (5) until all payload is received (7). Finally, the state machine terminates packet processing (8) and resets to wait a new packet (9). The other FSM performs in a similar manner because packets have the

same format independently of its source, i.e. DMA or Hermes. Table 3 summarizes the function of each state.

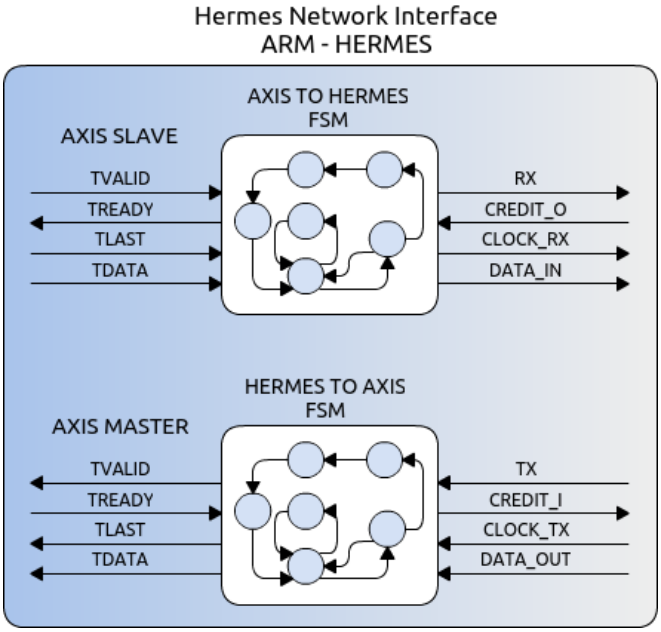

Figure 17 –NIAH module overview.

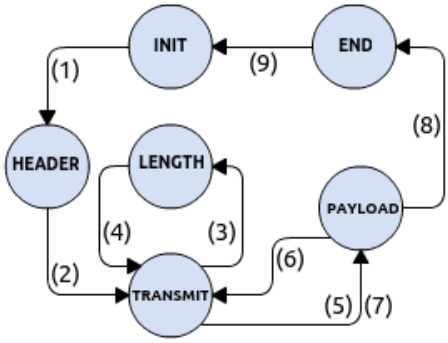

Figure 18 –NIAH Finite State Machine transition flow.

Table 3 – Hermes NIAH state and action correlation.

State	Action
INIT	Initialize state machine control variables.
HEADER	Receives packet header from DMA and forwards to Hermes, or vice-versa.
LENGTH	Stores payload size and forwards to the other interface. This information is used by **PAYLOAD** state to process the right amount of data. Furthermore, data flow from Hermes to AXIS, generates **TLAST** when all packet have been received.
PAYLOAD	Loops in this state until all payloads have been transferred. Control when NIAH is ready to receive data.
TRANSMIT	Control when the destination, i.e. either DMA or Hermes, is ready to receive and data transfer.
END	Deassert signals and returns to the initial state.

Using the same example, Figure 19 represents the equivalent signals that are generated from the DMA (*Slave AXI-Stream*) and NIAH's respective converted signals (*Hermes input*). To not overweight the picture, some state names were abbreviated. Header was changed to "HDR", payload to "PLD" and transmit to "TX". Note that it takes two cycles to transfer data from one interface to another, but it could take more depending on Hermes credit (*credit_o*). NIAH does not have storage logic. Therefore, when Hermes is ready to receive data from NIAH, it is consequently able to accept data from the DMA. Hence, data throughput depends on both modules availability, i.e. *ready* and *credit* signals.

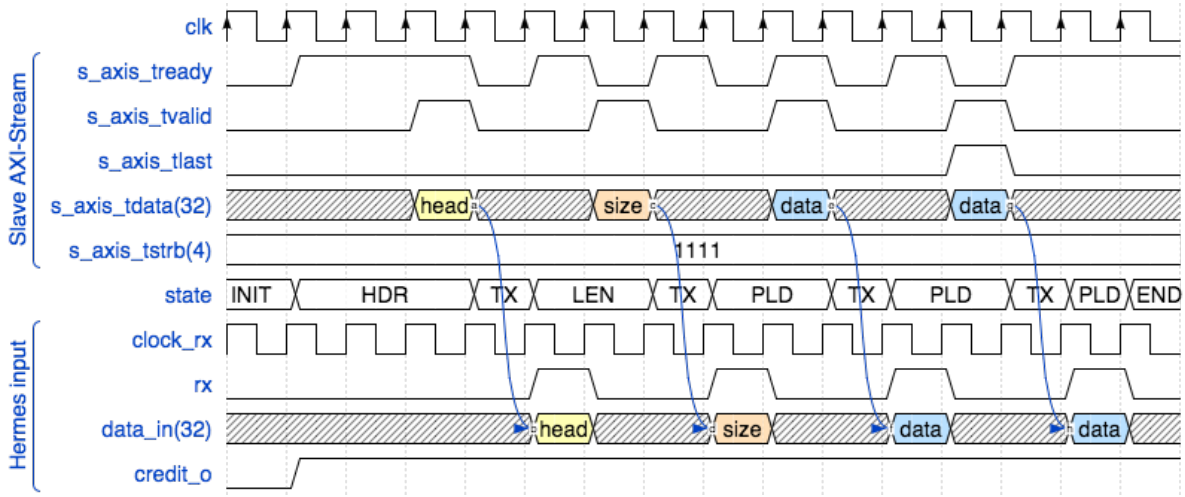

Figure 19 – NIAH behavior: data transfer from slave AXI-Stream to Hermes input.

AXI-Stream data width is equal to Hermes for this project, however they may differ. This design assumes both are configured with thirty-two bits size. It is possible, for example, to configure Hermes with bigger or smaller flits but NIAH must be modified. Thus, data should be split up or concatenated accordingly to receiver's data width.

4.4 NIAH Validation

NIAH validation was carried-out by prototyping a 2x2 Hermes NoC, which is connected to simplified processing elements. There are three PEs available, where each performs a different kind of operation. The first executes sequential add operation over the packet's payload; the second, subtract operation; and the third, combines add and shift operation. Those operations were implemented in hardware to temporarily substitutes Plasma IPs and simplify NIAH's validation. Complete Vivado design is depicted in Figure 20. The blocks in the Figure corresponds to:

1. *acc_dma_0*: includes Hermes NoC and NIAH; *M00_AXIS* and *S00_AXIS* are NIAH's master and slave AXI-Stream interface respectively
2. *axi_dma_0*: Xilinx DMA module;
3. *processing_system7_0_axi_periph*: AXI interconnection for control signals between the DMA and the PS;
4. *axi_mem_intercon*: provides the DMA direct access to read and write to PS' memory;
5. *xlconcat_0*: transmits the interrupt signals from the DMA to the PS;
6. rst_processing_system7_0_102*M*: the reset system;
7. *processing_system7_0*: processing system instance

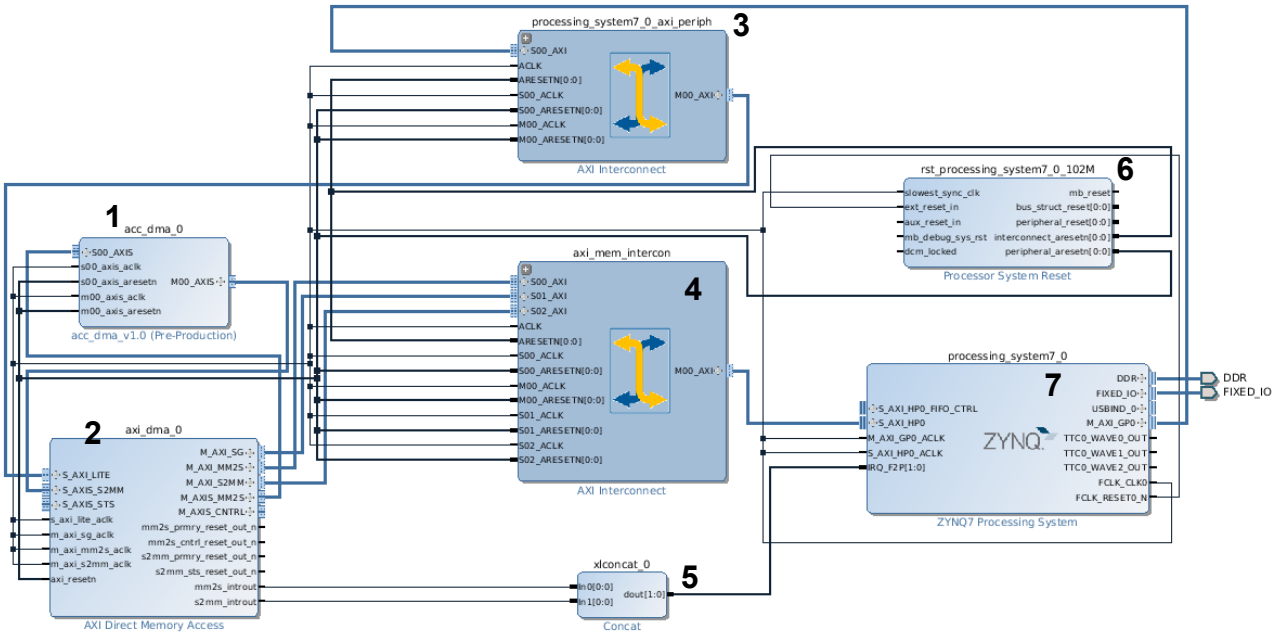

Figure 20 - Hermes prototyping blocks using NIAH to communicate with the PS.

Figure 21 is an abstract model that helps to understand the validation process. Suppose that the SDC has already stored a set of files. The host computer, then, issues a command (1) to perform one of the available operations over a certain file. Consequently, the ARM processor reads from the SDC and writes the requested file in OCM. Next, the DMA is configured to read the memory (3) and transfer the file to the network. Consequently, the packet will pass through NIAH and reach its destination (4). Whenever computation is finished, the packet will travel the inverse path (5) and hence being written again into OCM by the DMA (6). The processor is interrupted, it reads the memory and sends the results back to the host computer (7). Since each PE performs a different operation, the user is capable to check if they match expected answer.

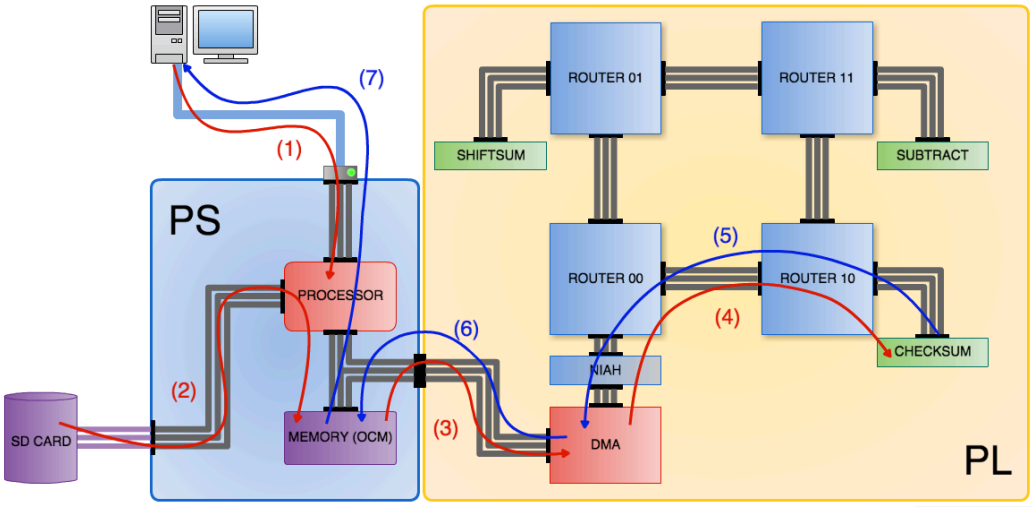

Figure 21 – NIAH validation flow.

This validation flow generates the screen captures depicted in Figure 22. First, the host computer uses TFTP to transfer *LOG.TXT* file to the SDC. This file contains data to be processed by the PEs (shiftsum, checksum, subtract). Next, it issues one command for each PE, i.e. (1), (2) and (3); with telnet and verifies if results match with those from *verify_results* software. This performs the same operations as the hardware. On UART communication is possible to identify

the moments when the command is received, data is written into memory, how many flits are transferred and when it terminates.

This setup validates the new hardware module developed in this work, i.e. NIAH, the Hermes NoC, the TFTP protocol, developed API, and all other structures described thus far. Additionally, comparing it with HeMPS setup, the major difference is the simplified processing elements that should be replaced by Plasma Slave IPs.

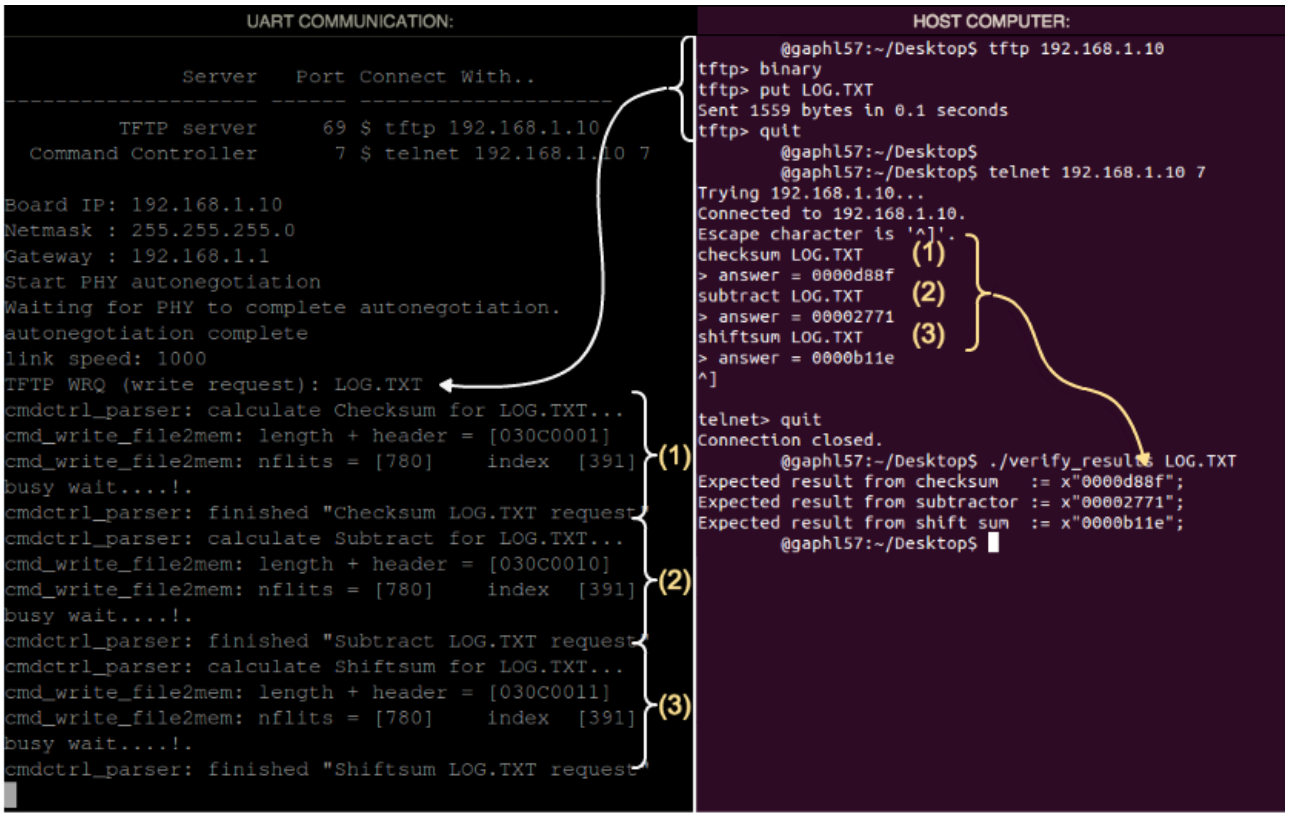

Figure 22 – UART communication and host computer terminals.

5 HEMPS DESIGN

Once validated the hardware-software interface (DMA and NIAH) and the communication interface (Ethernet and RS232), the next step is to prototype HeMPS MPSoC. This Chapter describes the hardware design in Section 5.1, and software used to control the MPSoC in Section 5.2.

5.1 Hardware

This Section discusses from HeMPS MPSoC generation, modifications, simulation and finally prototyping.

5.1.1 Architecture Characteristics

The first step is to generate HeMPS original hardware using available scripts. The input for those is a file that contains information about the hardware characteristics and a set of applications to simulate. It is possible to configure the following hardware parameters: (i) NoC size, (ii) Plasma memory size and (iii) page sizes, (iv) flit size, (v) NoC buffer size and (vi) global master location. The corresponding settings used for this final project are presented in Table 1. Plasma IP is configured with four pages and thus may execute three tasks at the same time. Two pages are reserved for the kernel object code. Therefore, it is necessary to have, at least, 32KB for the kernel. With 64 KB is possible to execute two tasks simultaneously.

Scripts compile both master and slave kernels, which are implemented in C language, to generate a pre-initialized BRAM memory. Those comprehend Plasma IP first two memory pages. Similarly to the kernel, applications are compiled to generate a VHDL file, however the resulting object code is transformed into a single RAM and not BRAM. Applications consist of one or more tasks, written in C language, that are going to be simulated.

Table 4 – HeMPS defined parameters.

Parameter	Configuration
Plasma memory size	64 KB, for each Plasma processor (requires 32 16Kb BRAMs per Plasma processor)
Plasma page size	16 KB
Flit size	32 bits
NoC buffer size	16 flits, for each available port (E, W, N, S, L)
NoC **x** dimension	2
NoC **y** dimension	2
NoC master location	Lower left corner (LB – lower bottom)

All applications that will be simulate constitute the repository. This has information regarding each task individually, e.g. object code, and the application, e.g. number of tasks. The repository is a 32 bits word line memory, which the first address is *0x04000000*. The first position

contains the number of tasks that the application has. The next ten memory addresses store initial task identifier, notifying the master PE which tasks should be allocated first. Next positions comprise important task information, called task headers. Each has reserved twenty-four memory positions to store the following data:

- Identifier;
- Code size;
- Code memory address;
- Static address;
- Ten spaces for dependent tasks:
  - Identifier;
  - Communication load

The repository characteristics were already defined. This work just presented how it is structured.

5.1.2 Modifications Targeting FPGA Prototyping

To prototype HeMPS MPSoC, it is necessary to adapt the generated hardware. As previously discussed, inside each Plasma IP there is an UART that is used to generate the *log* files during simulation. Since in this work the *debug* method was modified, the UART is no longer necessary. Moreover, as Plasma IP master is replaced by ARM processor, it is also removed from the design. Nonetheless, master's router is kept to be used by the ARM.

To create this connection, NIAH is incorporated into HeMPS MPSoC replacing the master PE. This new design is referred as *A-HeMPS* from now on (Figure 23). *S00_AXIS* and *M00_AXIS* signals represent slave and master AXI-Stream interfaces discussed in Section 4.1. The slave interface receives data from the DMA module and forwards to HeMPS, which is represented inside the module in the Figure. The master interface is used to transmit data in the opposite direction.

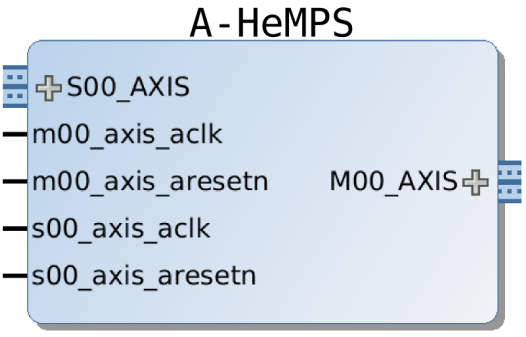

Figure 23 – A-HeMPS Vivado top module view.

An abstract view of the resulting design is illustrated in Figure 24, showing only A-HeMPS and the PS. The task repository is implemented in the SDC and the computer represents the host connected to the board. The Figure also highlights the OCM, with the TX and RX buffers used for DMA read and write operations.

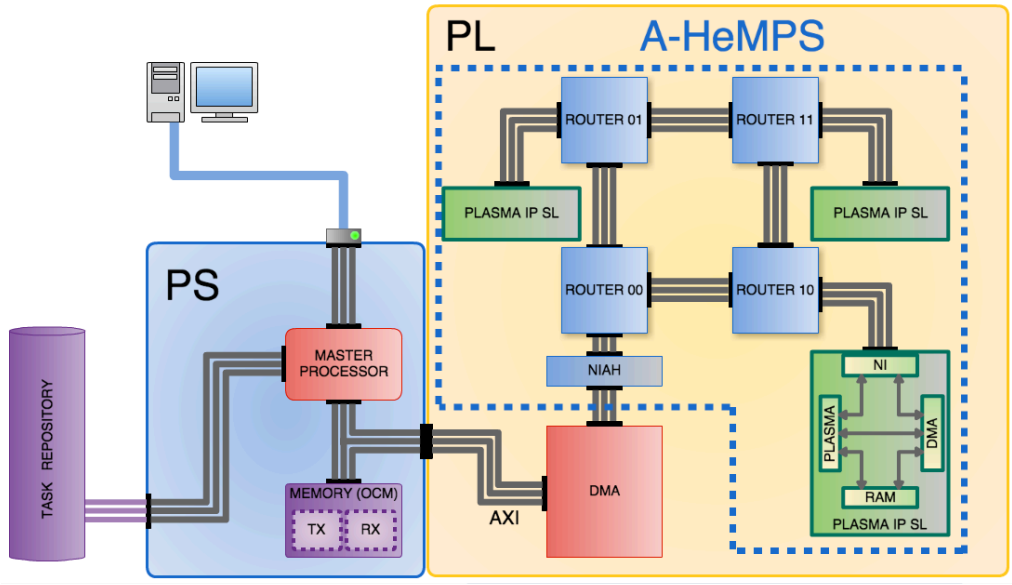

Figure 24 –A-HeMPS abstract design overview.

5.1.3 Simulation

Before proceeding to Vivado project creation and prototyping, A-HEMPS simulation should validate that all modifications are consistent, i.e. it works. The testbench simulates DMA AXI-Stream signals to send tasks for all PEs thus simulating the ARM processor. Besides usual task termination messages, in the modified slave kernel (3.1.4.3) *Echo* messages will further provide ways to verify the correct functionality of the design. Moreover, in this step the UART should not be removed to view the PE messages in the log file.

The tasks that were used during simulation are depicted in Figure 25. Each task sends an *Echo* message to the master processor on task startup and conclusion (lines 12 and 16). The *exit* function is a system call that sends a packet to the master indicating that there's an available page in the memory to receive another task.

taskA.c

```
1  #include <task.h>
2  #include <stdlib.h>
3
4  #define ITERATIONS 10000
5
6  Message msg;
7
8  int main()
9  {
10   int i, j;
11
12   Echo("Beginning taskA.\0");
13   // executes tasks...
14   // ...
15   //Send(&msg,taskD);
16   Echo("Finished taskA.\0");
17   exit();
18 }
```

taskB.c

```
1  #include <task.h>
2  #include <stdlib.h>
3
4  #define ITERATIONS 10000
5
6  Message msg;
7
8  int main()
9  {
10     int i, j;
11
12     Echo("Beginning taskB.\0");
13     // executes tasks...
14     // ...
15     //Send(&msg,taskD);
16     Echo("Finished taskB.\0");
17     exit();
18 }
```

taskC.c

```
1  #include <task.h>
2  #include <stdlib.h>
3
4  #define ITERATIONS 10000
5
6  Message msg;
7
8  int main()
9  {
10   int i, j;
11
12   Echo("Beginning taskC.\0");
13   // executes tasks...
14   // ...
15   //Send(&msg,taskD);
16   Echo("Finished taskC.\0");
17   exit();
18 }
```

Figure 25 – A-HeMPS' tasks for simulation.

In the simulation results, depicted in Figure 26, it is possible to verify that NIAH Slave interface receives data from the testbench, which simulated the DMA, and forwards to Hermes NoC. Task A, B, and C are transferred to slave Plasma IP *0x0001*, *0x0101* and *0x0100* respectively. During task execution, all *Echo* messages are stored in the pipe. When it terminates,

all messages are sent to NIAH master interface and a packet from *exit* system call. In the end, the testbench receives back nine packets, where six derives from *Echo* and three from *exit*.

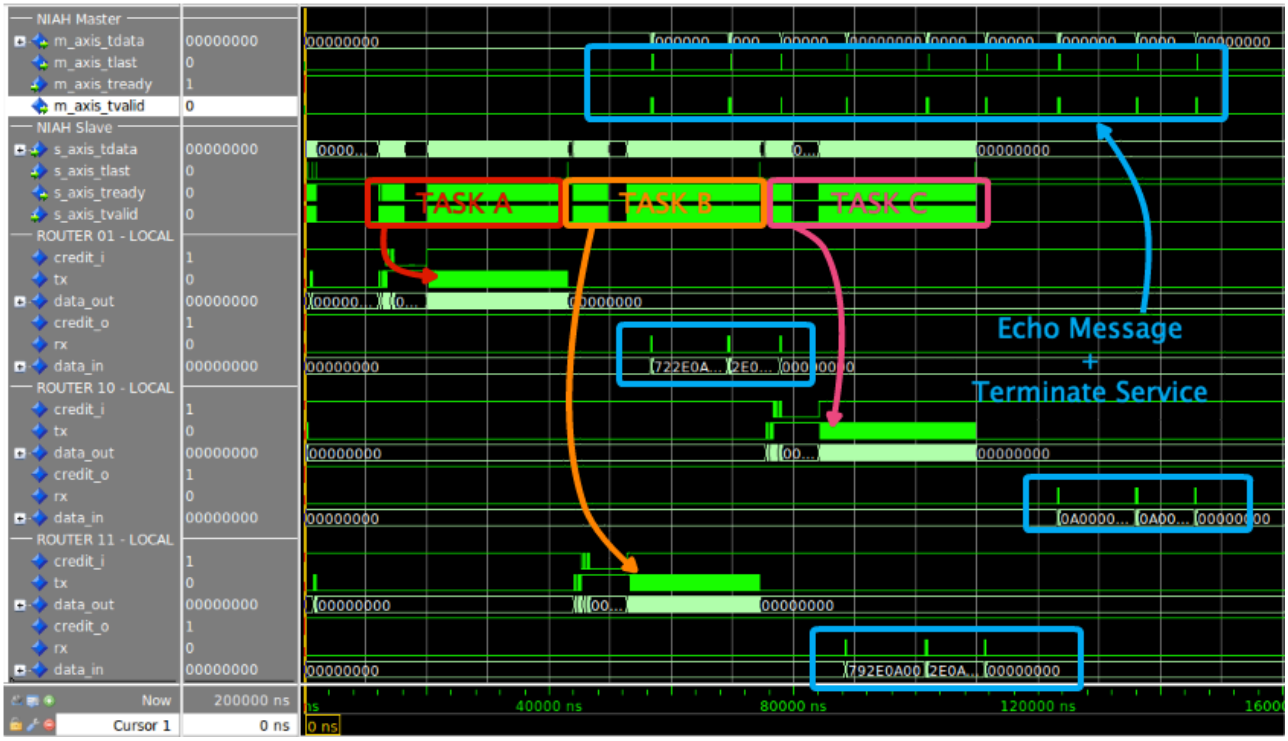

Figure 26 – A-HeMPS simulation waveform.

5.1.4 Design

Once simulation is set forth, and A-HeMPS is validated, the final project advances to design. Vivado IDE has a feature that creates a new AXI4 peripheral interface. This automatically generates logic that is not necessary and may be removed (Figure 27). The input/output ports represented in the Figure are the same of NIAH. This is the hardware module that creates the connection between the processor and the MPSoC.

A-HeMPS_Vivado.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.HeMPS_defaults.all;

entity A_HeMPS is
    generic (
        -- Parameters of Axi Master Bus Interface M00_AXIS
        C_M00_AXIS_TDATA_WIDTH  : integer    := 32;
        C_M00_AXIS_START_COUNT  : integer    := 32;

        -- Parameters of Axi Slave Bus Interface S00_AXIS
        C_S00_AXIS_TDATA_WIDTH  : integer    := 32
    );
    port (
        -- Ports of Axi Master Bus Interface M00_AXIS
        m00_axis_aclk     : in std_logic;
        m00_axis_areset   : in std_logic;
        m00_axis_tvalid   : out std_logic;
        m00_axis_tdata    : out std_logic_vector(C_M00_AXIS_TDATA_WIDTH-1 downto 0);
        m00_axis_tkeep    : out std_logic_vector((C_M00_AXIS_TDATA_WIDTH/8)-1 downto 0);
        m00_axis_tlast    : out std_logic;
        m00_axis_tready   : in std_logic;

        -- Ports of Axi Slave Bus Interface S00_AXIS
        s00_axis_aclk     : in std_logic;
        s00_axis_aresetn  : in std_logic;
        s00_axis_tready   : out std_logic;
        s00_axis_tdata    : in std_logic_vector(C_S00_AXIS_TDATA_WIDTH-1 downto 0);
        s00_axis_tkeep    : in std_logic_vector((C_S00_AXIS_TDATA_WIDTH/8)-1 downto 0);
        s00_axis_tlast    : in std_logic;
        s00_axis_tvalid   : in std_logic
    );
end A_HeMPS;
architecture arch_imp of HeMPS_v1_0 is

end arch_imp;
```

Figure 27 – A-HeMPS Vivado top VHDL file (simplified code).

Similarly to the validation scenario, represented in Figure 20, A-HeMPS interconnects with the same elements. Figure 28 represents Vivado block diagram, which was abstractly demonstrated by Figure 24. The processing system is further configured to use UART, Ethernet, and USB (for programming) board peripherals necessary for this project. DMA is set to operate on Scatter Gather mode, which enables segmented memory data transfer.

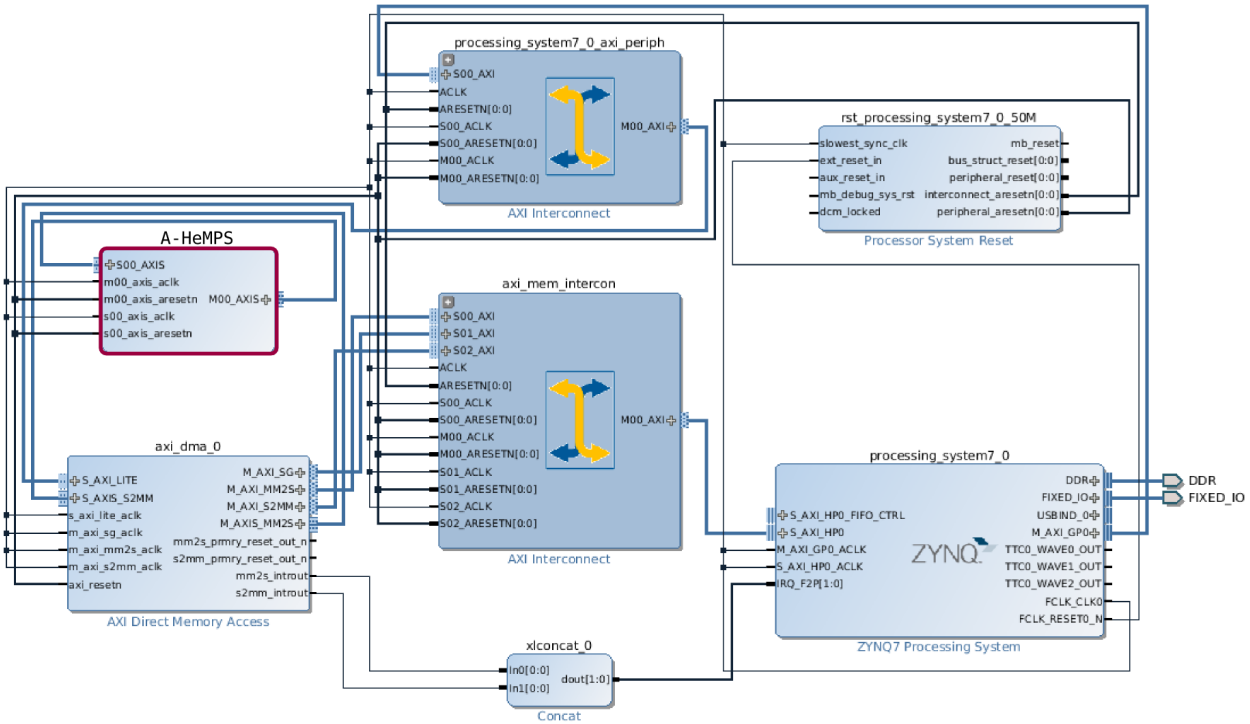

Figure 28 – A-HeMPS Vivado block design diagram.

After the hardware definition, the designer proceeds with logic and physical synthesis. It was not possible to meet timing constraints only using default settings on both steps. Even significantly reducing operating clock frequency, from 100 MHz to 20 MHz, time slack was always negative. There are two main critical paths in this design: (*i*) Plasma register bank; (*ii*) delay due to Zynq-7000 device BRAM disposition. Those are column oriented, and thus forces prototyped logic to create long wires to access certain Plasma memory segments (Figure 29). It might not be interesting to use LUTs as BRAMs because it would consume too much resources. Each CLB contains 8 LUTs that can hold 256 bits [XIL14]. Since each Plasma IP requires 64 KB of BRAM logic, it would be necessary 4,096 LUTs for each (dual port). Considering available resources, it would consume 24% of available LUTs.

One technique used in Electronic Design Automation to enhance timing constraints in the design is floorplan. It consists in delimiting where some major functional blocks should be placed. Hence, blocks that intercommunicate might be closely placed to avoid long wires.

Looking back to hardware (Figure 28) and abstract (Figure 24) design overview, it is possible to observe which blocks communicate more often and thus should be kept close. Starting from PS physical location, which is the orange portion in Figure 29, the next blocks are AXI interconnections. Those connect the PS to the DMA; therefore they should be placed near. Inside HeMPS routers should be close to each other and their respective Plasma IP. Between the master router and the DMA, NIAH should be placed. The resulting floorplanning, using those considerations, are depicted in Figure 30. A further division that is not represented in this Figure is

Plasma IP Network Interface (NI). It is possible to observe, mainly in the right PEs, that there is a separated portion of programmable logic placed close to the router. Those delimitations grants, at a certain level, that logical blocks are placed together and hence avoid long wires.

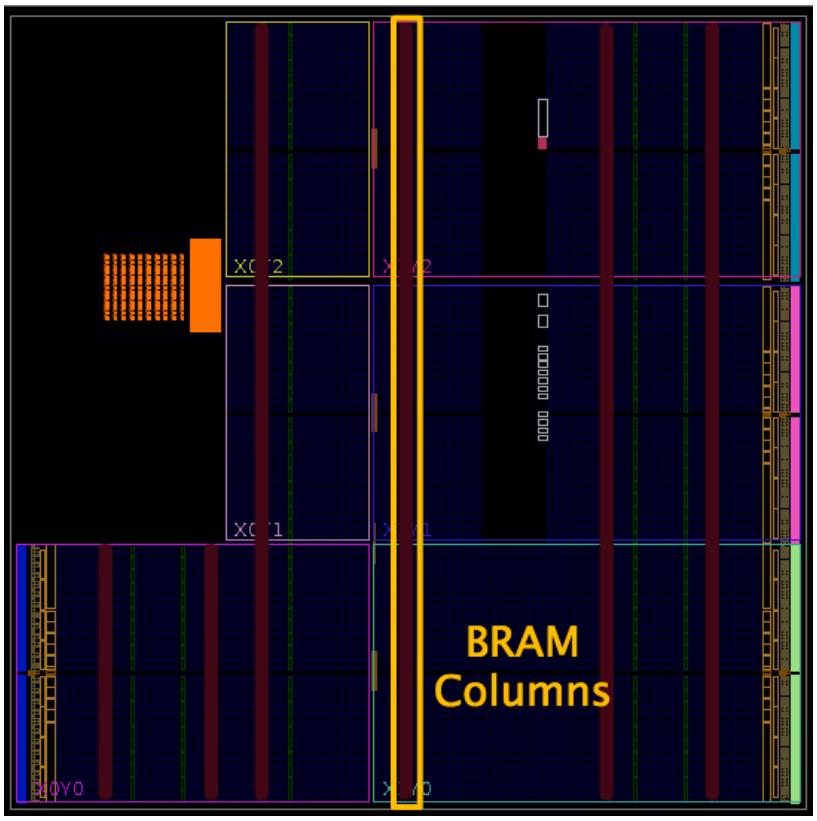

Figure 29 – Zynq-7000 device programmable logic, with the BRAM logic block placement.

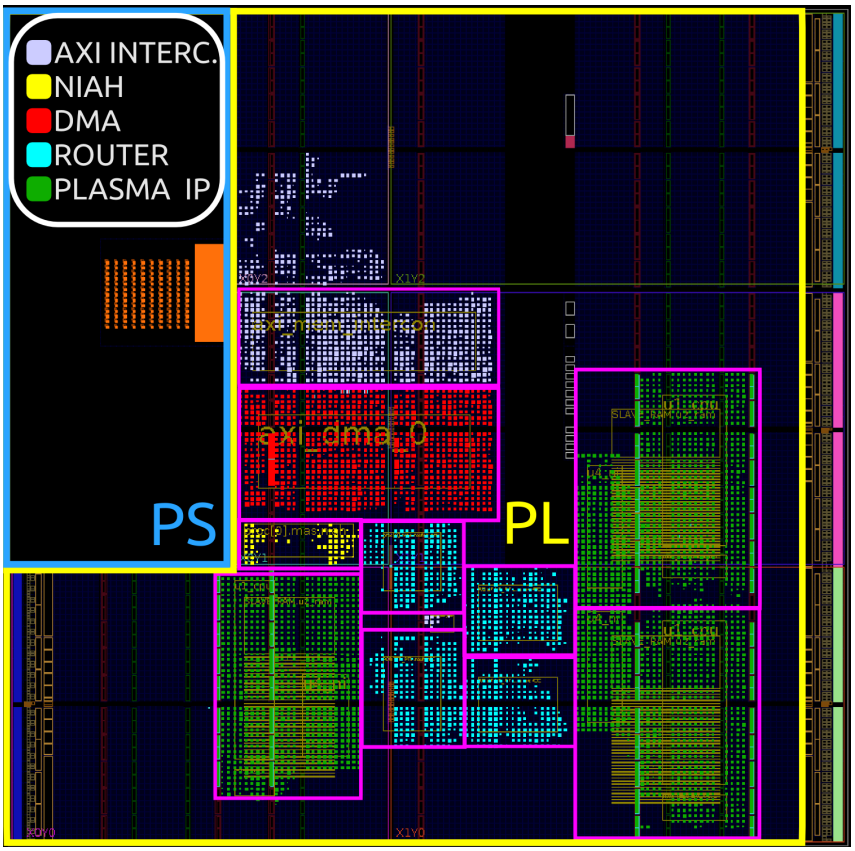

Figure 30 – HeMPS floorplanning.

All modules from this design operate in the same frequency. DMA AXI4 and AXI4-Stream interconnections, respectively connected to PS and HeMPS, are capable to support different clock frequencies. However current architecture may not benefit from this feature. AXI4-Stream speed represents how much data is possible to inject into HeMPS network, and AXI4 speed, read and write operations. HeMPS is both the center and slowest module, thus constraining maximum design clock. As pointed out, only increasing AXI4, i.e. PS, interconnection speed wouldn't add much to current architecture, i.e. memory operations speed, and hence was not considered at this time.

Clock frequency, area, power and other results are reserved for the next Chapter.

5.2 Software

The project objective is to prototype the HeMPS MPSoC and to develop a software manager using ARM Cortex-A9 cores. The processor is capable to communicate with the slaves and allocate tasks on them. Moreover, it offers an interface to users to control which applications the system will run. This Section describes the software implementation.

5.2.1 Initialization

When the ELF file is loaded into Zynq-7000 device, the first actions executed are related to initializing hardware components. Those consist in configure and start: (*i*) timers, e.g. for Snoop Control Unit (SCU); (*ii*) interrupt system, e.g. for timers and DMA; (*iii*) DMA; (*iv*) Ethernet Interface, e.g. IP address.

Afterwards, interrupt requests are enabled and the management modules execute their startup routines. Those are TFTP, Command Interface (telnet) and MPSoC Master. A wrapper unit called *dispatch* is responsible for sequentially initializing each routine. The Internet protocols, TFTP and telnet, routines consist on basic UDP and TCP procedures to bind a specific port to device's IP address. Port numbers are 69 and 23 respectively. Furthermore, it is defined which function handles incoming requests on those ports. Finally, the MPSoC Master allocates memory for internal control units.

After all peripherals and software modules are properly initialized, execution waits a user request to continue. Default IP address is 192.168.0.1/24; therefore the user should properly configure the computer interface to communicate with the board.

5.2.2 File Transfer Requests

Complementing discussion in Section 3.1.4.1, the TFTP protocol is responsible to handle user's read and write requests to the SDC. It is not related to any MPSoC or repository concept but only to receive and send files.

5.2.3 Command Interface

Described in Section 3.1.4.2, the Command Interface is responsible to interpret the user commands and forward to HeMPS master. This work defined three commands, represented in Table 5. They aim to provide a flexible interface to users create execution scenarios on the fly.

Table 5 – HeMPS available commands.

Command	Parameters	Description
hemps start	None	Sends HeMPS initialization packets to each slave. Those inform where Master processor is located in the NoC.
hemps load	Repository File	Reads parameter, which should be a repository file, and loads it. This command is used to create execution scenarios with one or more applications.
hemps run	None	Executes the scenario loaded by **hemps load** command. Applications are allocated to slaves in the same order that the users issued **hemps load** requests.

5.2.4 HeMPS Master

HeMPS Master needs to control applications, loaded by the user, and executes Plasma IP Master original functions. Those consist of:

1. Task allocation;
2. Mapping heuristics;
3. Handling slave PE requests:
    a. Debug messages;
    b. Task location;
4. MPSoC control operations

Let's discuss those items in the same manner that the device is used. After the software is loaded in the PS, the user connects with TFTP to transfer repository files, or telnet to issue commands. The Master expects repository files in a similar format used for simulation with a few modifications. HeMPS scripts generate two types of repositories, one described in C language and the other in VHDL. Any of those might be used because what is removed from them is language syntax defines. This approach makes easier and more straightforward for the Master to process the files and takes advantage of existing scripts. Linux *sed* command is used to convert the generated file to the new format. Figure 31 demonstrates the difference between C/VHDL and A-HeMPS file. Each character is interpreted as a byte and thus creates a 32 bits per line "memory".

After files are loaded into the SD Card, the user issues Table 5 commands to execute applications. The form they are presented in the Table corresponds to the order in which actions take place.

5.2.4.1 Hemps start

The MPSoC should be started prior to any other action. That means all slave processing elements must be notified to perform startup routines and receive the master address, i.e. address. This ensures that slaves know who they should obey and inquire about any control status. The corresponding function to this command is depicted in Figure 32. First, it erases any old values that OCM might have, and then builds a packet called *Initialization Packet* and configures the DMA to perform the transfer. An *Initialization Packet* contains the master address and identified as *Initialize Slave* service. This operation is repeated for all available slaves, which is statically

defined accordingly to hardware specifications.

repository.h

```
#ifndef _repository
#define _repository

    #define NUMBER_OF_APPS 1
    float appstime[1] = {0,};
    #define REPO_SIZE   60000
    unsigned int repository[REPO_SIZE] = {

        0x00000001,
        0x00000000,
        0xffffffff,
        0xffffffff,
        0xffffffff,
        0xffffffff,
        0xffffffff,
        0xffffffff,
        0xffffffff,
        0xffffffff,
        0x00000000,  //taskA.c
        0x00000139,  //code size
        0x0000008c,  //initial address
        0xffffffff,  //static position
        0x027ba688,
        0x00000064,
          ....
        0x00000000,
        0x00000000
};

#endif
```

repository.vhd

```
library IEEE;
use IEEE.Std_Logic_1164.all;

package memory_pack is

constant NUMBER_OF_APPS         : integer := 1;
type timearray is array(0 to NUMBER_OF_APPS) of ti
constant appstime : timearray := (0 ms,0 ms);
constant REPOSITORY_SIZE    : integer := 60000;
type ram is array (0 to REPOSITORY_SIZE) of
            std_logic_vector(31 downto 0);

signal memory : ram := (

    x"00000001",
    x"00000000",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"ffffffff",
    x"00000000",
    x"00000139",
    x"0000008c",
    x"ffffffff",
    x"027ba688",
    x"00000064",
       ...
    x"00000000",
    x"00000000"
);
end memory_pack;
```

a-hemps_repository.txt

```
00000001
00000000
ffffffff
ffffffff
ffffffff
ffffffff
ffffffff
ffffffff
ffffffff
ffffffff
00000000
00000139
0000008c
ffffffff
027ba688
00000064
```

Figure 31 – Comparison among C, VHDL and A-HeMPS repository files.

```
void
master_init_slaves(void)
{
    size_t i;
    struct packet *pkt;
    uint32_t *rxmem, *txmem;

    txmem = (uint32_t *) Dma_GetTxAddress();
    rxmem = (uint32_t *) Dma_GetRxAddress();
    pkt   = (struct packet *) Dma_GetTxAddress();

    hemps_debug("%s: Cleaning Tx and Rx buffers.\r\n", __FUNCTION__);
    clean_memory(rxmem, Dma_GetFreeSpace());
    clean_memory(txmem, Dma_GetFreeSpace());

    hemps_debug("%s: Initializing slaves.\r\n", __FUNCTION__);
    for(i = 1; i <= HEMPS_SLAVES; i++)
    {
        pkt->daddr   = mapper_get_slaveaddr(i);
        pkt->length  = PACKET_PL_INIT_LEN;
        pkt->service = INITIALIZE_SLAVE;
        pkt->init_maddr = HEMPS_MADDR;

        master_send_pkt(pkt->length + PACKET_HEADER_LEN);
    }
}
```

Figure 32 – hemps start command implementation.

5.2.4.2 Hemps load

Afterwards, the user is capable to load repository files that will make part of the execution scenario. The load operation corresponds to reading the repository file and extract number of tasks, where is their object code, i.e. “memory address”, and with which other tasks they communicate. Those parameters are stored in a set of structures so that Master does not have to read the file again, unless to get tasks’ object code, to execute them. Moreover, there are other information that identify which tasks are initials and where tasks have been allocated. An abstract

overview of **hemps load** implementation is represented in Figure 33. Note that the master is not concerned with any particular structure, it just uses repository API to control them.

```
226  int
227  master_load_application(char *appfile)
228  {
229      int addr = 0;
230      // create new application
231      app = repo_new_app(master->repo, appfile));
232      // open file inside the sd card
233      ret = sdca_open(&fd, appfile, (FA_OPEN_EXISTING | FA_READ));
234      // read number of tasks
235      task_count = sd_read32(&fd, addr, &br);
236
237      // read all initial task id
238      for(addr = 1; addr <= REPO_MAX_INITIAL_TASKS; addr++)
239          initial[(addr-1)] = sd_read32(&fd, addr, &br);
240
241      // load application's tasks
242      while(task_count > 0)
243      {
244          task = app_new_task(app);
245
246          task_set_id(task, app->id, sd_read32(&fd, addr++, &br));
247          task_set_codesize(task, sd_read32(&fd, addr++, &br));
248          task_set_address(task, sd_read32(&fd, addr++, &br));
249          task_set_stposition(task, sd_read32(&fd, addr++, &br));
250          app_verify_initial(initial, task, REPO_MAX_INITIAL_TASKS);
251          // read dependencies
252          for(i = 0; i < TASK_MAX_DEPENDENCIES; i++)
253          {
254              tid   = sd_read32(&fd, addr++, &br);
255              tload = sd_read32(&fd, addr++, &br);
256              if(task_is_valid_id(tid))
257              {
258                  task_add_dependence(task, tid, tload);
259              }
260          }
261          task_count = task_count -1;
262      }
263
264      sdca_close(&fd);
265      return HEMPS_SUCCESS;
266  }
```

Figure 33 – hemps load abstract command implementation.

5.2.4.3 Loaded Applications

Applications' information is stored in a three level hierarchy: repository, application and task, from higher to lower levels. The repository is composed of all applications that the user has issued a **hemps load** command, which are subsequently formed by several tasks. Besides the task header, defined in Section 5.1, there are two other important parameters stored for each task. One identifies whether it is initial or not, and the other contains information about in which slave PE it is allocated. Table 6 summarizes all tasks parameters using implementation names.

To support multiple application execution, each has an associated identifier. They are assigned accordingly to **hemps load** call order. Therefore, first loaded application is identified by one, the second by two, and so on.

5.2.4.4 Hemps run

At this moment, the repository has applications loaded by the user and the Master will allocate them to the slave processing elements. Initially, it gets from the repository the first application and allocates all initial tasks. During this procedure, *initial_allocated* variable is deasserted. This is a measure to ensure that, if DMA interrupts initial task allocation, no other

request is processed. After this first step, all pending requests are then processed. This procedure is repeated for all repository applications. Thereafter, the master waits all allocated tasks to indicate that they have finished, and consequently all applications. The functions that perform those actions are abstractly represented in Figure 34.

Table 6 – Task's defined parameters.

Parameter	Type	Description
id	32 bits integer	Identifier
code_size	32 bits integer	Object code size
address	32 bits integer	Object code address (line) in the memory (file)
static_position	32 bits integer	Defines task position. This HeMPS version does not use this parameter.
initial	Boolean[2]	Identifies which tasks should execute first
dependencies	List	It is a list of all other tasks that this one will communicate. Consists in target task identifier and communication load information.

```
109 void
110 master_run_app(struct application *app)
111 {
112     // allocate initial tasks
113     master->initial_allocated = 0;
114     LIST_FOREACH(task, &app->tasks, next)
115     {
116         if(task_is_initial(task))
117             master_allocate_task(master, task);
118     }
119     master->initial_allocated = 1;
120 }
121
122 void
123 master_run_all(void)
124 {
125     if(LIST_EMPTY(&master->repo->apps))
126     {
127         hemps_debug("%s: No applications loaded.\r\n", __FUNCTION__);
128         return;
129     }
130     // executes on application per time
131     LIST_FOREACH(app, &master->repo->apps, next)
132     {
133         master->apps_running = 1;
134         master->pending_requests = 0;
135         master_run_app(app);
136
137         // waits all tasks to terminate
138         while(master->apps_running)
139         {
140             // executes pending requests
141             if(master->pending_requests > 0)
142                 master_pending_requests(master);
143         }
144     }
145 }
```

Figure 34 – hemps run abstract command implementation.

[2] Boolean is represent as integer. Since ARM Cortex-A9 is a 32 bits architecture, this type is actually a 32 bits variable.

More details about where tasks are mapped and how DMA interrupt requests are handled is described next.

5.2.4.5 Mapper

The Mapper responsibility is to control slave PE status, allocate tasks and store this information, i.e. where they were allocated. There is three important information for each processing element: identifier, address and available memory pages. To determine where a task is going to be allocated, the Mapper verifies which PE has more available pages (Figure 35). The implementation of a complex mapping heuristic is out of scope of this work. Nonetheless, the basic necessary structure, accordingly to Plasma master kernel, was developed.

```
76  uint32_t
77  mapper_task_map(struct mapper *map, struct task *task)
78  {
79      uint32_t mapped_pages;
80      struct slave_processor *sp;
81      struct slave_processor *mapped;
82
83      mapped_pages = 0;
84      LIST_FOREACH(sp, &map->slaves, next)
85      {
86          if(sp->free_pages > mapped_pages)
87          {
88              mapped = sp;
89              mapped_pages = sp->free_pages;
90          }
91      }
92
93      mapped->free_pages--;
94      task->slave_addr = mapped->address;
95
96      return (hemps_saddr[mapped->id]);
97  }
```

Figure 35 – Mapper slave PE allocation decision.

After every task is mapped (*mapper_task_map*), the number of target PE available pages is decremented. As already commented, when a slave kernel executes an *exit* system call a *Task Terminated* packet is sent to the master. This will inform the Mapper to increment the number of available pages and hence, if there is tasks available, allocate them.

Even though the repository supports several applications, the Mapper implementation does not handle all possible scenarios. For example, the Mapper does not handle cases where a task must be allocated and there is no available PE. This project achieved more than expected, therefore there was no available time to implement all new necessary functionalities. Further discussion is reserved to the final Chapter.

5.2.4.6 MPSoC message handler

During tasks' execution, a slave processor may send messages that should be one of the following three different types of service:

- Task Request: Whenever a task wants to communicate with another, it asks the Master for its location. In turn, the Master verifies if the target task has already been allocated. If it has, Master requests the Mapper for its location, else allocate target task and then, in both cases, informs the requesting PE its location.

- Task Terminated: Indicates that a specific task has finished execution (*exit* system call). Whenever this type of service is received, the Master forwards packet information to the Mapper.
- Debug Message: Slave processor use this service to send *Echo* messages to the master that, consequently, transmits to the user through the UART. This is related to the kernel modifications explained in Section 3.1.4.3.

When there is a request, the DMA interrupts the Master to handle them. Both *Task Terminated* and *Debug Message* may be processed any time without problems. However, it might not be possible to handle a *Task Request* service at that moment, i.e. when interruption occurred. The difference between those services is that *Task Request* has to answer a requesting slave, while the other two don't.

To answer the slave PE, it is necessary to write a packet to memory, i.e. *tx* buffer. The problem arises if the Master was already using the same buffer to send another packet. Therefore, there are two flags that implement a *mutex* logic:

- *initial_allocated* : Indicates whether all initial packet have already been allocated or if is doing so. Even if a PE requests an unallocated task, beginning tasks should be allocated prior the others.
- *handling_pr* : Handling Pending Requests flag, indicates that another request that could not be processed at interruption, is under processing. Therefore, the new request must be stored.

The appropriate action, when one of those flags is asserted, is to store incoming request for future processing. The list where this information is stored is called *pending_requests*. After initial tasks have being allocated, Master checks if there is any pending request and take appropriate actions.

5.2.5 HeMPS Debug

Additionally to the software project, a debug mechanism was created to help this and future development. As previously discussed, the UART peripheral is used to *print* messages in the remote computer and thus understand how software is executing. Therefore, another component was included in the API to ease its usage. This new routine is called *hemps_debug*. If this functionality is used instead of directly accessing the peripheral, it is possible to enable/disable all messages just changing one define:

```
#define HEMPS_DEBUG_MODE
```

With this define all messages are active, on the contrary, all deactivate.

After all applications have finished executing, the Master returns a message through Command Control (3.1.4.2) to the user. Now, with both hardware and software design explained, the document advances to its final part: Validation, Result Analysis and Conclusions.

6 DESIGN ANALYSIS AND VALIDATION

After the design set forth, this Chapter considers final project results, imposed limitations and how to possibly overcome those. In addition, discussion follows to general design validation, i.e. overall functionality.

6.1 Hardware Characteristics

This Section analyzes the developed hardware design characteristics, i.e. area, power and timing, for the final project.

6.1.1 Timing

Using optimized logical and physical synthesis settings and floorplan technique the hardware design achieved 51 MHz clock frequency. The Processing System AXI interfaces are capable to operate at a maximum of 250 MHz, hence it is possible to further enhance resulting frequency. In spite of working with synthesis settings and floorplannig represented an important part of this work, it is out of scope to prototype HeMPS hardware at the highest possible speed.

6.1.2 Area

Zynq-7000 programmable logic utilization report, for the complete design, is shown in Table 7. Special blocks, e.g. DSP and XADC, where not used and thus do not integrate this table.

Table 7 – Complete design area utilization report.

Resource	Utilization	Available	Utilization (%)
Slice LUTs	18,045	53,200	33.92
Slice Registers	10,467	106,400	9.84
Memory	50	140	35.71
Clocking	1	32	3.12

Using Vivado™ utilization reports it is possible to further analyze how much HeMPS module contributes to the complete design area. Considering Table 8 results for each resource, HeMPS complete block represents 62.43% of all used logic.

Comparing both tables, it is clear that Plasma IP memory significantly contributes to BRAM usage (96%).

Table 8 – HeMPS area utilization contribution.

Resource	Utilization	Utilization (%)
Slice LUTs	13,030	72.20
Slice Registers	4,760	45.44
Memory	48	96.00
Clocking	0	0.00

6.1.3 Power

Vivado™ power analysis feature is used to estimate system power consumption after routing. Vectorless Estimation [XIL13] engine is capable of generating circuit activity through algorithms and heuristics that can even approximate glitching rate for any nodes in the netlist. Results are not as accurate as post-route simulation with reasonably long duration and realistic stimulus, but it provides a good estimation for this work.

Results depicted in Figure 36 shows that the Processing System represents 90% of Dynamic Memory Consumption. HeMPS module consumes 7% of total dynamic power, which is the sum of BRAM and Logic consumptions.

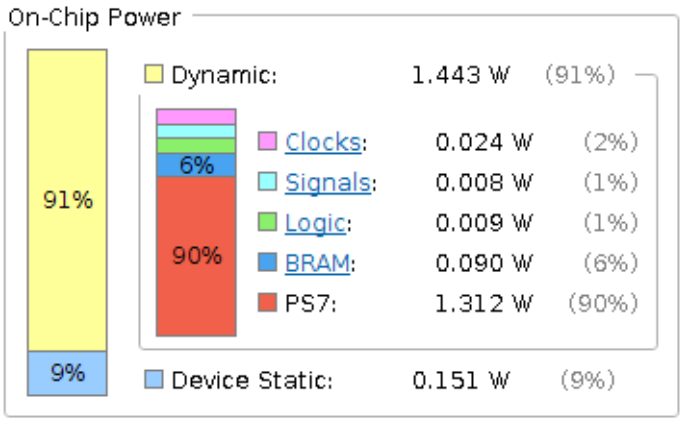

Figure 36 – Design power analysis.

6.2 Software Characteristics

This Section covers software analysis in a structural form of view, i.e. memory usage and leakage. Behavioral verification is discussed in Section 6.3. Vivado™ does not provide software analysis tools in this level. Therefore, it is necessary to verify implemented software separately from the SDK. There are several available tools in Linux OS; hence all tests were performed in it.

One of embedded system's concerns regards memory usage. Problems might remain hidden during test phase and only manifest after a long time that the product in the field. For this reason all software that manipulates memory should be verified. Xilinx libraries did not integrate testing scenarios because coarse errors are more likely due to new developed code. Even though this might be true for the scope of this work, it does not mean that those libraries are faulty-free. The modules that were tested are:

- Command Interface:
  - Parser;
  - Command Control;
  - Command;
- Mapper;
- Scenario structures:
  - Repository;
  - Application;
  - Task;
- Master.

6.2.1 Valgrind

Valgrind [VAL14] is an instrumentation framework for building dynamic analysis tools. This final project uses Memory Check and Massif Heap Profiler built in tools to evaluate memory leakage and usage respectively.

6.2.1.1 Memory Check

The Memory Check tool verifies if allocated memory during program execution is completely freed in the end, and all memory accesses are aligned. Considering a real scenario for this final project, applications are included and removed from it. Consequently, it should be verified that removal routines work properly, i.e. free all memory. Figure 37 depicts the testes used to evaluate memory. It reads 100,000 times two different applications, simulates allocation and task termination.

```
185 int
186 main(void)
187 {
188     size_t rep;
189     struct master master;
190
191     // Create Structure
192     start_master_application(&master);
193
194     for(rep = 0; rep < 100000; rep++)
195     {
196         //  Read Repository
197         master_load_application(&master, APP1_FILE);
198         master_load_application(&master, APP2_FILE);
199
200         //  Map and Terminated Cycles
201         validate_debug_cycle(&master, 1, 2);
202     }
203
204     // Remove structures
205     repo_delete(master.repo);
206     free(master.repo);
207     master.repo = NULL;
208
209     mapper_delete(master.map);
210     free(master.map);
211     master.map = NULL;
212
213     return (1);
214 }
```

Figure 37 – Test routine used to verify memory requirements.

After all tests were executed in the final version no memory leaks or unaligned memory accesses were found. This results are demonstrated in Figure 38.

```
==32518== Memcheck, a memory error detector
==32518== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==32518== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==32518== Command: ./a.out
==32518==
==32518==
==32518== HEAP SUMMARY:
==32518==     in use at exit: 0 bytes in 0 blocks
==32518==   total heap usage: 960,005 allocs, 960,005 frees, 87,140,120 bytes allocated
==32518==
==32518== All heap blocks were freed -- no leaks are possible
==32518==
==32518== For counts of detected and suppressed errors, rerun with: -v
==32518== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 38 – Valgrind *MemCheck* report.

6.2.1.2 Massif Heap Profiler

Massif gives a graph detailing memory allocation throughout test time. Even though the Cortex-A9 is 32 bits architecture and machine used for testing is 64 bits, results are useful to estimate memory usage. Repository concentrates memory operations and after creating ten applications, each with three tasks and two dependences, around 5KB was used. Disregarding

minimal Mapper's contribution and other control structures, each application consumes around 500B for this scenario.

However, those results do not apply for other cases because application contain different number o tasks. This tool was mainly used to analyze how much memory task structures consume. Conclusion was that each task uses approximately 144 bytes, plus 12 bytes per dependence, considering memory alignment and administrative bytes. This indicates that structures do not require much memory and thus benefiting embedded systems.

6.3 Validation

The main goal of this Section is to validate the final project design functionality. NIAH hardware and master software are the major modifications from original architecture. Therefore, the tests focus on those, but are not restricted to them.

6.3.1 First Scenario

The first scenario, depicted in Figure 39, consists of three independent tasks. All are *initial* and thus the master sequentially allocates them. Mapper's logic assures that each task is mapped to a different processing element to force all to participate. Tasks perform three actions:

1. *Echo* - begin message;
2. *Performs computation;*
3. *Echo* – terminate message

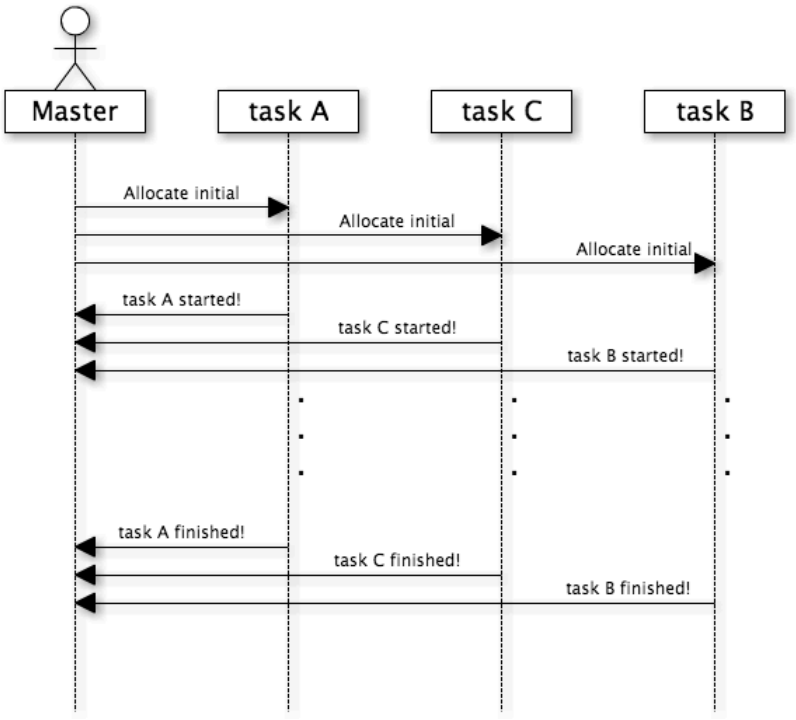

Figure 39 – Independent tasks application

User controls the system through the Command Interface and TFTP. Possible actions are (i) transfer the application to the board; (ii) connect to the command interface; (iii) start A-HeMPS MPSoC; (iv) load application file to current scenario; and (v) run.

```
       @gaphl57:~/Desktop$ tftp 192.168.1.10 (i)
tftp> binary
tftp> put simul1.txt
Sent 16010 bytes in 0.6 seconds
tftp> quit
       @gaphl57:~/Desktop$ telnet 192.168.1.10 23  (ii)
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.
hemps start              (iii)
> HeMPS architecture started with 'hemps start' command.

hemps load simul1.txt    (iv)
> Loading application to repository with 'hemps load' command

hemps run                (v)
> Running application with 'hemps run' command.

^]

telnet> quit
Connection closed.
```

Figure 40 – User using A-HeMPS command interface.

The master processor transmits through UART all steps required to execute user's commands (Figure 41). Since tasks are short, i.e. do not perform heavy computation; their *Echo* messages arrive right after they were allocated. It is so fast that there is not enough time to allocate another task before those messages arrive. Note that ARM processor does not individually waits any task. This happens here because there is not enough time to send a new task before debug arrives.

```
TFTP WRQ (write request): simul1.txt
master_init_slaves: Cleaning Tx and Rx memories.
master_init_slaves: Initializing slaves.
cmd_hemps_load: loading 'simul1.txt'...
master_load_application: application loaded
   > master_allocate_task: task [0] allocated to [00000100]
   > task [00000000], on PE [00000100], says:
        "Beginning taskB."
   > task [00000000], on PE [00000100], says:
        "Finished taskB."
   > master_allocate_task: task [1] allocated to [00000100]
   > task [00000001], on PE [00000100], says:
        "Beginning taskC."
   > task [00000001], on PE [00000100], says:
        "Finished taskC."
   > master_allocate_task: task [2] allocated to [00000100]
   > task [00000002], on PE [00000100], says:
        "Beginning taskA."
   > task [00000002], on PE [00000100], says:
        "Finished taskA."
   > application [0] terminated.
   > all applications finished!
```

Figure 41 – MPSoC reports for *simul1.txt*: independent tasks.

6.3.2 Second Scenario

The second scenario depicted in Figure 42, is a traditional consumer-producer application. Two tasks, i.e. A and B, generated data and forward to another, i.e. C, to be consumed. The consumer depends on producers' data; therefore only A and B are initially allocated. Eventually,

they will need to transfer generate data to the consumer. Consequently, both send a *Task Request* message to the ARM processor that will allocate task C. Afterwards, tasks A and B are informed C's location, which also receives information regarding A and B's position. Once tasks A, B and C knows each other's' location, the producers send all data to the consumer. In the end, task C informs the master if the expected data format from both A and B were received correctly.

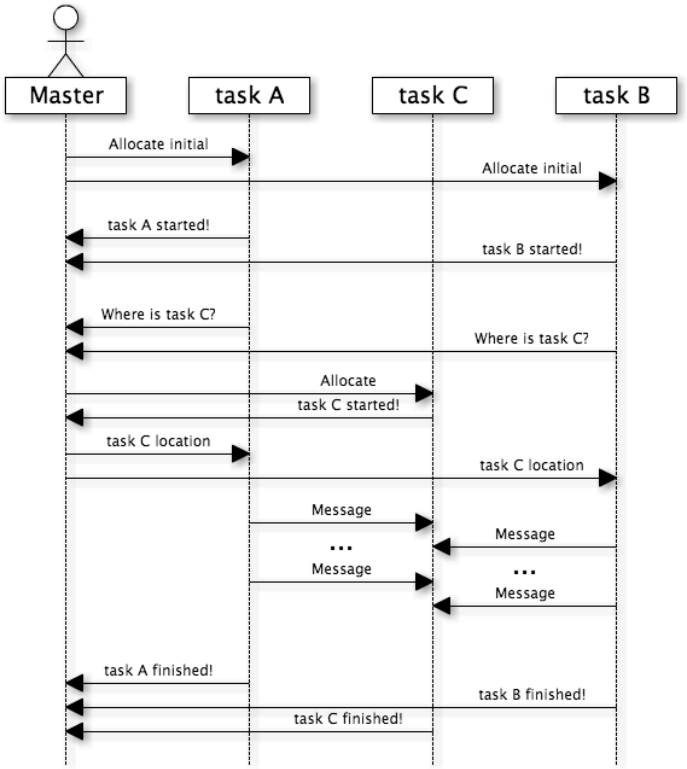

Figure 42 – Dependent tasks application

Results from the second application are depicted in Figure 43. After the user has issued the same command set described for the first test case, the following actions take place (the Figure has the corresponding messages from each step):

1. Allocate task B (identifier 0) to *0x0100* PE;
2. Allocate task A (identifier 2) to *0x0010* PE:
    a. During this stage, task B requests where task C is located;
    b. Saves this request, because not all initial tasks have been allocated;
3. Start pending packets handling, the first is from B:
    a. During this stage, task A also requests where task C is located;
    b. Saves this request, because another is being processed;
    c. Task C (identifier 1) was not allocated, thus allocates to *0x0101* PE;
    d. Sends to task B and C, each other's location;
4. Task A also has a pending request:
    a. Task C is already allocated;
    b. Sends to task A and C each other's location;

5. After consumer and producers finish communicating (MPI), each sends a *Terminate* service packet to the master

Note that there is an (**) in this Figure, right after task C allocation (3.c), indicating that one duplicated request was avoided. When C starts execution, it needs to receive data from task A and thus needs to know where task A is located. As task A already made a similar request, master does not need to answer it twice.

```
TFTP WRQ (write request): simul2.txt
master_init_slaves: Cleaning Tx and Rx memories.
master_init_slaves: Initializing slaves.
cmd_hemps_load: loading 'simul2.txt'...
master_load_application: application loaded
(1)> master_allocate_task: task [0] allocated to [00000100]
   > task [00000000], on PE [00000100], says:
        "task B started."
(2)> request context saved! (2.a-b)
   > master_allocate_task: task [2] allocated to [00000001]
   > task [00000002], on PE [00000001], says:
        "task A started."
(3)> request context saved! (3.a-b)
   > master_allocate_task: task [1] allocated to [00000101]
   > task [00000001], on PE [00000101], says: (3.c)
        "task C started."
   > avoided adding duplicated requests!(**)
   > Replied task request message to [00000100]. (3.d)
   > Replied task request message to [00000101].
(4)> Replied task request message to [00000001].
   > Replied task request message to [00000101].
(5)> task [00000002], on PE [00000001], says:
        "task A finished."
   > task [00000000], on PE [00000100], says:
        "task B finished."
   > task [00000001], on PE [00000101], says:
        "task C finished perfectly."
   > application [0] terminated.
   > all applications finished!
```

Figure 43 – MPSoC reports for *simul2.txt*: consumer and producer.

6.4 Limitations

Final project duration is short and sometimes it is not possible to cover all cases. This Section explains this design's limitations.

6.4.1 Application

Suppose the scenario previously described is executed. As presented, the scenario executes correctly, but what happens if a *new* scenario is loaded? The answer is that it will not execute correctly due to two issues. One is related to HeMPS scripts that generate applications

and the other one related to the kernel master.

The current implementation requires that all applications must be compiled together to generate an unique repository. All tasks compiled in the current implementation receive an unique ID, inserted in the object code. It is not possible to insert dynamically a new application into the system because it will generate a conflict in the tasks ID. So solve this issue it would be necessary to generate on-the-fly the tasks' IDs to avoid such conflicts. The HeMPS 7.0 (new HeMPS version) already supports this feature.

The second issue is related to the kernel master. When a new scenario is loaded, the previous mapping tables are not cleaned. Therefore, a given initial task will receive a wrong task position when it tries to communicate with a new task. To solve this issue, it is required to implement a simple function to clean the master structures when a new scenario is loaded.

6.4.2 Mapper

The Mapper is responsible for deciding where tasks are allocated. This decision is based on which processing element has more available memory pages. However, when all PEs are busy this task should be scheduled to execute later. Due to limited time to develop this project, this feature was not implemented.

7 CONCLUSIONS AND FUTURE WORK

The work described in this document produced the A-HeMPS platform, which comprises an MPSoC prototyped in the programmable logic and a manager using ARM®Cortex-A9 cores. Moreover, this system is connected to a remote computer capable to create application scenarios to evaluate the design.

The proposed objectives demanded the author to practice and improve knowledge learned throughout the Computer Engineering course. The complete process to design A-HeMPS fulfilled the project goals:

(i) understand an industrial design flow to create complex designs;

(ii) understand a modern FPGA technology with embedded ARM cores;

(iii) understand the communication techniques between software and reconfigurable components of modern FPGAs;

(iv) prototype an MPSoC in the target FPGA programmable logic;

(v) use the available ARM processors to control the MPSoC.

Additionally, this final project was able to surpass the initial goals, creating a larger and more complete case-study design, with tasks exchanging messages.

This project's main contribution was to create a prototyping platform for HeMPS with a command interface to execute applications. To achieve this, it was necessary to understand HeMPS hardware design to modify it; implement a system call in the kernel to debug the MPSoC; gather available software, i.e. for peripherals, and implement Plasma IP master functionality. This implementation was based on existing master kernel, however it was designed from scratch. It is different from original implementation and does not have all functionalities.

It is possible to use different devices based on ARM processor for this project. Nonetheless, it was not evaluated which modifications would be necessary because it depends on each different device. It is important that it offers the same peripherals, otherwise it would be necessary to make bigger changes in the design, i.e. regarding peripherals. Moreover, the new device should be compatible with Vivado IDE to use A-HeMPS and Xilinx DMA IPs.

A possible future work for this final project would be to develop State-of-art MPSoC solutions. Expand developed software to include mapping heuristics, fault tolerant techniques and others. Moreover, it would be interesting to implement ways to extract relevant information about the MPSoC, e.g. network traffic, to evaluate their impact in field scenarios.

REFERENCES

[ARM98] ARM. "*ARM ELF File Format".* 1998, 10p.

[ARM10] ARM. "*AMBA 4 AXI4-Stream Protocol Specification"*. Version 1.0, 2010, 42p.

[AVN12] AVNET Electronics marketing. "*Zedboard – Zynq™ Evaluation and Development Hardware User's Guide"*, Version 1.1, August 2012, 38p.

[COL99] Collins, C. "*Putting Things in Perspective"*. Xcell Journal, v.(32), 1999, p. 2.

[CRO14] Crockett, L.; Elliot, R.; Enderwits, M; Stewart, R.; "*The Zynq Book"*. Strathclyde Academic Media, 1st Edition, 2014.

[BOB07] Bobda, C. "*Introduction to Reconfigurable Computing: Architectures, algorithms and applications*". Dordrecht: Springer, 2007, 375p.

[DEH08] Hauck, S.; Dehon, A. "*Reconfigurable Computing: The Theory and Practice of FPGA BasedComputation*". Burlington: Elseiver, 2008, 945p.

[DUN08] Dunkels, A. "*A Lightweight TCP/IP stack"*. Available at: www.nongnu.org/lwip/, March 2008

[JER05] Jerraya, A.; Wolf, W. "*Multiprocessor Systems-on-Chips"*. Morgan Kaufmann Publishers Inc, 2005.

[MIC05] Microsoft Corporation. "*Microsoft FAT Specification*". August 2005, 34p

[MOR04] Moraes, F.; "*HERMES: an Infrastructure for Low Area Overhead Packet switching Networks on Chip*". Integration the VLSI Journal, v.38(1), Oct. 2004, pp. 69-93.

[RHO15] Rhoads, S. "*Plasma CPU*". Available at: http://plasmacpu.no-ip.org/, April 2015

[ROE99] Roelandts, W. "*15 Years of Innovation"*. Xcell Journal, v.(32), 1999, p. 4.

[SCA97] Schaller, R. "*Moore's law: past, present and future*". Spectrum IEEE, v.34(6), Jun 1997, pp 52-59.

[SAN07] SanDisk Corporation. "*SanDisk SD Card Product Family – OEM Product Manual*", Version 2.2, Jun 2007, 40p.

[SOA14] Soares, E. "Embedded Systems development using FGPA platforms with ARM devices", End-of-term work, PUCRS, 2014, 70p.

[SCH10] Schranzhofer, A.; Jian-Jia C.; Santinelli, L.; Thiele, L. "*Dynamic and adaptive allocation of applications on MPSoC platforms"*. In: ASP-DAC, 2010, pp. 885-890

[FEI12] Feist, Tom. "*Vivado Design Suite*". Xilinx, 2012, 14p.

[VAL14] Valgrind Organization. "*Valgrind.org*". Available at: http://valgrind.org/, Jun 2015.

[WOL04] Wolf, W. "*The Future of Multiprocessors System-on-Chips".* In: DAC. 2004, pp. 681-685

[XIL10] Xilinx Inc. "*LogiCORE IP AXI DMA (v2.00a)",* December 14, 2010, 63p.

[XIL12] Xilinx Inc. *"Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency"*, 2012, 10p.

[XIL13] Xilinx Inc. *"Vivado Design Suite User Guide – Power Analysis and Optimization "*, Jan 2014, pp16-19.

[XIL14] Xilinx Inc. *"7 Series FPGA Configurable Logic Block – User Guide"*, 2014, 74p.

[XIL14a] Xilinx Inc. "*7 Series FPGAs Overview",* 2014, 17p.