Performance Estimation of Novel 32-bit and 64-bit RISC based Network Processor Cores

Danijela Jakimovska, *Student Member, IEEE,* Aristotel Tentov, *Member, IEEE,* Sashka Gjorgjievska, Goce Dokoski, Marija Kalendar, *Member, IEEE*

***Abstract*—The rapid expansion of computer networks in number of users, servers, connections and demands for new applications, services and protocols, along with the tremendous growth in data traffic has claimed the development and deployment of high-speed telecommunication systems. At the same time the fiber optics has shown significant bandwidth increase, providing multi-Gb/s line rates. However, present network devices still have limited processing power, which makes them unable to satisfy these demands. Therefore, telecommunication industry is looking forward for more innovative ways of designing flexible, scalable and high performance routers architecture. A solution to this problem is to use specialized processors, called network processors (NPs). These application specific instruction processors (ASIPs) are specially tailored to perform packet processing operations and their architecture is usually a question of different trade-offs between performance, flexibility and price. NPs from various vendors have different architectures, and the appropriate NP design choice can significantly affect the router architecture and its performances. In this paper we give an overview of the current trends in NP design, emphasizing the fact that a vide variety of NP architectures are composed of multiple equal general purpose RISC processor cores. Having this in mind, we consider the possibility of augmenting and modifying 32-bit and 64-bit RISC processor cores, for packet processing application. This would require some minor architectural changes in the initial RISC cores architecture and their instruction set. The proposed NP cores would be implemented in language for instruction set architectures (LISA), so their functionality could be tested and verified. Furthermore, we would evaluate their performance capabilities, by comparing them with the initial simple general purpose RISC cores, and with the micro engine processor cores of one of the most famous multi-gigabit network processor, Intel IXP1200. Designed NP cores could be further used for homogeneous multi-processor NP organization, where each core would be processing packets independently of the others. We believe that the proposed NP cores would be capable to deal with multi-gigabit (10/100 Gb/s) links of Next Generation Networks.**

***Index Terms*— IP packet processing, LISA, network processor, next generation networks, RISC**

Manuscript received May 10, 2011. This work was supported by the Faculty of Electrical Engineering and Information Technologies – Skopje, as part of the project "Routing and Traffic control in Next Generation Computer Networks", 2011-2012.

Danijela Jakimovska, MSc, is teaching and research assistant at the Faculty of Electrical Engineering and Information Technologies, Ss. Cyril and Methodius University, Skopje, R. Macedonia (phone: +3892-3099-153, fax: +3892-3064-262, e-mail: danijela@feit.ukim.edu.mk).

Aristotel Tentov, PhD, is full professor at the Faculty of Electrical Engineering and Information Technologies, Ss. Cyril and Methodius University, Skopje, R. Macedonia (e-mail: toto@feit.ukim.edu.mk).

Sashka Gjorgjievska, final year student in BSc, is laboratory assistant at the Faculty of Electrical Engineering and Information Technologies, Ss. Cyril and Methodius University, Skopje, R. Macedonia (e-mail: Saska.GJorgjievska@feit.ukim.edu.mk).

Goce Dokoski, MSc, is teaching and research assistant at the Faculty of Electrical Engineering and Information Technologies, Ss. Cyril and Methodius University, Skopje, R. Macedonia (e-mail: gocedoko@feit.ukim.edu.mk).

Marija Kalendar, MSc, is teaching and research assistant and PhD student at the Faculty of Electrical Engineering and Information Technologies, Ss. Cyril and Methodius University, Skopje, R. Macedonia (e-mail: marijaka@feit.ukim.edu.mk).

I. Introduction

THE trend of data, voice and video traffic convergence in the Internet, has caused bandwidth requirements growth in the data and telecommunication networks that form the backbone of the Internet. Therefore, telecommunication links and devices should be able to transmit and process huge amount of converged data traffic at very high speeds, up to multi Gb/s. These requirements are not limitation for the telecommunication links, due to the use of fiber optic transmission technology. On the other side, network devices must provide a solution that would satisfy the requirements for high throughput, but as well would provide flexibility in supporting new protocols, services and applications (QoS, firewalls, VPN, scheduling, flow controls etc), [1] - [3]. Additionally, as a result of the changing requirements for cost, performance and flexibility, the proposed solution should be capable to reach the market rapidly.

Almost all network devices, including ATM switches, Ethernet switches, IP routers, web servers, hardware firewalls etc. provide some kind of packet processing. In the past, packet processing was implemented in software run on general purpose processors (GPP), because that time the performance requirements were very low and the networking protocols were very simple. However, GPP couldn't provide high performance computing at wire rates. This became a huge problem, so the network engineers decided to develop hardware-based solutions using application specific integrated circuits (ASICs), [1], [4]. Although ASIC circuits could reach high speed and processing power, they were very specialized, and almost impossible to change, once they have been designed. Consequently, re-programmability couldn't be provided by use of ASIC circuit. This implied development of some new technologies such as System on Chip (SoC) design, Field programmable gate architecture (FPGA) as well as complex programmable logic device (CPLD). All these technologies have enabled many new possibilities in processor design area. A performance comparison between different technologies for network processing implementation is given in Fig. 1.

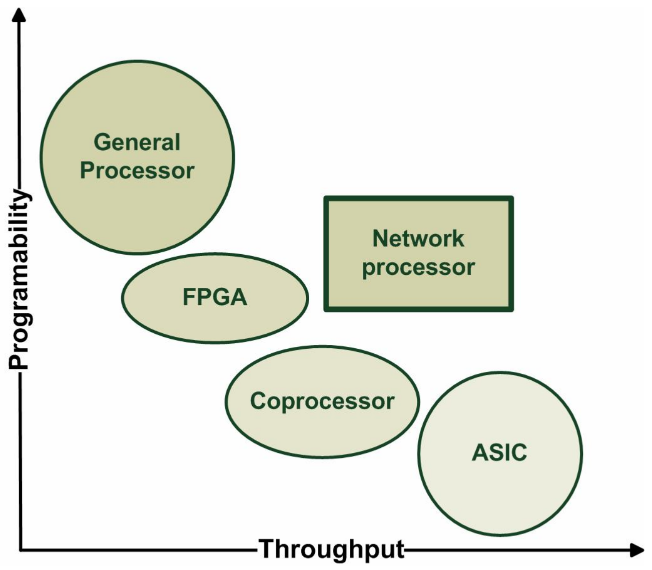


Fig. 1. Comparison between different technologies for network processing implementation. As shown, ASICs achieve highest performance, on the cost of flexibility, whereas GPPs provide most flexibility, but less performance. On the other side, NPs characteristics are somewhere between FPGAs and Coprocessors, so NPs have proved themselves as the best solution for performance/flexibility achievement.

This evolution resulted in the concept of network processor as hardware unit optimized for packet data processing at wire rates (multi Gb/s). In general, NPs are defined as chip-programmable devices, particularly tailored to process network packets at very high speeds, [1] - [5]. They are part of various networking devices, and are usually implemented as application specific instruction processors (ASIP), with customized instruction set that may be based on RISC, CISC, VLIW etc., [2]. Furthermore, many NP architectures employ some improvements, such as parallel computing and pipeline techniques. This paper examines current architecture trends, prior to evaluating the novel NP cores.

NPs have proven themselves as the best solution since they provide the flexibility of GPP, while keeping high performance of ASIC. They also provide many other important capabilities, such as scalability, product differentiation, reduced cost of ownership, and a faster time-to-market. In this paper we would provide an outline of the achievements in NP design, in order to propose some ideas for further improvements.

NPs architecture design is an ongoing field of research, expecting that the NPU market will show strong growth in the near future. Over the last few years many companies developed their own NPs, so many various NP architectures have been applied. What is more, many new ideas, such as the NetFPGA architecture, [6], [7], or software routers, [8] are constantly emerging.

The aim of this paper is to propose novel 32-bit and 64-bit RISC based NP cores design, as well as to make performance estimation of their network processing capabilities. In order to achieve the goal, we would try to understand the NP processing operation requirements, so that we could research the current NP architecture trends. Besides that, we would examine various processor design techniques and tools in order to choose the most appropriate one, for implementing the proposed NP cores. Afterwards we would research the opportunity to extend 32-bit RISC based processor, and specify it for networking application. To achieve that, we would use the well known DLX processor architecture. Later we would consider the possibility to augment 64-bit RISC processor architecture, so it could be adjusted for network processing application. The proposed NP cores would be implemented in LISA, chosen as the most appropriate language for NP architecture description. The implemented NP cores might be tested, and verified within the used processor designer tool, so we could indicate that each of them is working properly. Additionally we would evaluate the achieved network processing performances of the proposed NP cores, by executing and analyzing IP packet processing programs. The performance estimation is given by comparing the number of IP packet processor cycles for the initial 32-bit and 64-bit RISC cores, the proposed 32-bit and 64-bit RISC based NP cores, and one micro engine of some of the most famous multi-gigabit NPs, such as Intel IXP1200.

The rest of this paper is organized as follows: Section II gives an overview of NPs, their key characteristics and methods of operation, while outlining current architecture trends in NP design. Section III gives an overview of the techniques for processor design implementation. Section IV clarifies the design and implementation of the 32-bit NP core. Section V explains the development of 64-bit NP core and its implementation. Afterwards, in section VI, a performance evaluation of the novel NP cores is given. The paper concludes in section VII. The conclusion outlines the benefits of the proposed NP cores.

II. State of the Art

NPs have emerged in the late 1990s as potential technology that could handle the complex network processing requirements, [2]. Until now, there is no standard architecture for NPs, but most of the NP designs share some common characteristics that make them some how similar. Accordingly, all NPs are usually composed of: many processing engines (PE), dedicated hardware accelerators, memory resources, network interfaces, and software support, [1], [3]. Furthermore, NPs architecture is often improved by use of parallel processing, specialized coprocessors, and different techniques for achieving parallelism. According to the Flynn's classification scheme, NPs can be categorized in the multiple instruction stream, multiple data stream class (MIMD), [2].

The majority of the commercial NPs, such as EZChip's NP-1-4, Intel's IXP1200, 2400, 2800, 2850 NPs, IBM's Power NP, Motorola's C-5 NP and many others, are designed as multi-core parallel processors. Generally, they could be separated in the following two categories: ones that use a number of high-end, special-purpose processing cores, like EZChip's NPs and those who use a large number of simple RISC processing cores, such as Intel's IXP NPs. However, all NPs are system-on-chip (SoC) designs that employ processor cores, memory and I/O on a single chip. In many NPs architectures (like Intel's, Motorola's, Sitera's) the processing engines are RISC based cores, augmented with specialized instructions, multithreading, or pipeline implementation.

Network devices are usually composed of four functional blocks: physical interface, data plane, control plane and switching interface, [1]. The basic function of each network device is to process the ingress data flow accepted by the physical interface, and then forward the packets to an outbound port, after the processing is completed. Basically network processing can be divided in two categories: control plane and data plane processing. Usually, NPs are responsible for the fast packet forwarding, executed in the data plane. The processing operations performed in the data plane are performance critical and must be completed very fast. On the other side, the slow packet processing implemented in the control plane, doesn't require high performance computing. Accordingly, the control plane performs operations for control, configuration and management of the network device and is also responsible for routing protocols execution and management of routing tables. These control functions support and adjust the performing of data plane processing operations. Therefore, the control plane is generally implemented as a GPP. The switching fabric is another part of the network devices, responsible for network traffic forwarding from ingress to egress ports.

Basically, the NP operation begins with receiving an input stream of data packets from PHY interface or switch fabric. During the packet processing usually only the IP header of the received packets is being processed, by parsing, analyzing, and modifying its content. Additionally, NPs do various packet processing operations such as classification of packets, IP route lookup and pattern matching, queue management and traffic control. After completing all the required processing operations, the packet is sent out through the switching fabric to appropriate outbound port, [1] - [3].

According to [9], the proper selection of networking protocols functionalities, considering the most time-consuming packet processing operations could significantly simplify and accelerate the forwarding process. As a result, many different approaches have been used, such as speeding-up the look-up operations by label concepts use, or avoiding some very complex operations like checksum calculation. However, it has been shown that the routing table look-up operation is the most time consuming operation that has to be taken into consideration. Having this in mind, network designers should focus their research in two directions: speeding-up or avoiding the table search. Therefore, the authors of papers [10] and [11] describe and propose faster table look-up algorithms, but in paper [9], the authors suggest avoiding the time-consuming table searches, by employment of source routing. Anyway, speeding up the routing process depends not only on the routing algorithm, but on the NPs architecture, as well. Consequently, in this paper we would try to improve the network processing performances, by proposing novel 32-bit and 64-bit RISC-based NP cores, specified for network processing application.

At the present, there are many different NP architectures, each using different organization and concepts. However, a key point to all of these architectures is that they employ multiple processing cores (micro engine), that can process the input data streams in parallel. Besides data level parallelism, NPs can implement other parallel techniques for achieving thread or instruction level parallelism (ILP), [3].

Very often, NPs packet processing can be accelerated by employing, hardware assistance (co-processors, functional units), adjusted memory architectures and interconnection mechanisms, [2]. Hardwired or reconfigurable coprocessors are usually in conjunction with processing cores, performing some functions that are computationally intensive to be implemented in softer. These special purpose hardware blocks are generally used for implementing some time-consuming operations like traffic management, packet classification, table lookup, and crypto graphing functions, [1], [2]. NP's memory is another very important part of their architecture, especially because the processor frequently interacts with it. Not to mention that memory performs many important operations like storing the program and registers' content, buffering packets, keeping intermediate results, holding, and maintaining potentially huge tables and trees for look-ups, maintaining statistical tables, and so forth. Consequently, the communication between memory and processor should be very fast, so that memory could fulfill the speed requirements. However, memory size and speed would always be a huge trade off. Major improvements can be achieved, by use of memory coprocessors for lookup, fast memories like CAM and SRAM, or various caching mechanisms, [3].

All these functionalities result in a NP architecture that can achieve high performances. However, there are additional requirements that should be satisfied as well, like flexibility, ease of programmability and fast time-to-market, [3]. Currently, NP design companies are paying much more attention to the programmability, enabling network software to be written in a high-level language such as C, and the core routines in microcode, [3], [12]. Since there are much different architectures, with complex design and performance constraints, current trend is to achieve software uniformity and design portability.

Nowadays, the most famous NPs implement multi-core architecture that can operate in parallel, pipeline or hybrid mode, [2]. Additionally, according to NPs level of operation they are divided in three categories: – entry-, mid- or core-level NPs. Mid-level NPs do packets processing at higher layers, so they implement parallel architectures. On the other side, core-level NPs require highest processing speeds at the lower network layers, thus they implement pipeline architectures, [2].

As given in [2], NPs can be classified according to the organization of their processing cores and hardware accelerators: in pipelines or parallel pools of processors. Parallel architectures usually include multiple RISC-based homogeneous processing cores, which implement some particular instructions for packet processing. Their NP cores usually have small data and instruction cache, since they don't interact with each other. Pipeline architectures usually employ heterogeneous processing cores optimized for specific processing function. The packet processing in the pipeline model is divided into multiple stages, whereas each stage is responsible to handle some specific networking operation. Hence, each processing core has optimized instruction set for executing specific pipeline stage function.

The Intel IXP2800 processor is composed of 16 identical multi-threaded RISC processors, organized as a pool of parallel homogeneous processing cores, [13], and an additional 32-bit XScale processor responsible for control plane management. Intel's architecture organization is advantageous in the simplicity of programming the processing elements, as they all use the same instruction set, and it also allows great flexibility towards ever-changing services and protocols. Therefore, Intel IXP2800 can achieve up to 10 Gb/s processing speed. On the other hand, Agere's NPs employ pipeline model, where each heterogeneous processing core is responsible for one pipeline stage, [2]. Furthermore, the EZChip's NP-1-4 processors are an example of a pipeline of heterogeneous multi-core processors. The used processing cores are optimized for specific tasks and are called Traffic Optimized Processors (TOP cores). Each stage of the pipeline is consisted of more duplicated cores, thus if one packet takes more time for processing, the whole pipeline would not be stalled. The heterogeneousness complicates the programming, but allows near-ASIC processing speeds to be reached. The newest EZChip NP-4 processor can achieve a total throughput of 100 Gb/s, that can be arranged among one or more line cards, [14], [15].

Our goal would be to design two different network processing cores that could further be used in homogeneous multi-core parallel organization. Therefore, each NP core would process the packets separately of the others, so packets that need more processing time won't stall the packet flow.

III. Processor Design Techniques and Tools

Processor design is a long, tedious, and error-prone task consisting of typically four design phases: architecture exploration, software design (assembler, linker, loader, and profiler), architecture implementation (RTL generation for FPGA or cell-based ASIC) and verification. Therefore, the selection of appropriate processor design techniques and tools is one of the most important things that should be done, prior to the processor design implementation. Furthermore, the chosen processor designer tool is supposed to satisfy some requirements like optimal processor performance, small die size, low power consumption etc. However, there is still no solution which enables all these requirements to be satisfied at the same time.

When it comes to processor design, the classic HDL methodology, that uses languages such as VHDL, SystemC or Verilog, can be a long and tedious process that can be usually carried out by only very experienced designers. This is how the Architecture Description Languages (ADL) have emerged. They enable processor design at higher level of abstraction with high degree of automation and unified development environment, thus more flexible development process. For example, the Language for instruction-set architectures (LISA) allows modeling a processor not only from instruction-set but also from architecture description including pipelining behavior. This allows design and development tool consistency over all levels of the design, [16].

From their introduction until today, ADL's were subject to great number of changes. Today they are becoming an irreplaceable modeling tool for special purpose processors (like NPs) in academic research and commercial usage. In this paper we are going to provide an overview of the available processor modeling tools and techniques, in order to select the most appropriate one for the NP cores implementation. Furthermore we would consider the possibility to use FPGA board, as an experimental platform for real hardware simulation.

A. Design Techniques

In order to design a processor, several stages are required including, but not limited to: architecture design, architecture implementation, software development, and instruction and system verification. Architecture design and implementation is typically done by use of processor design tool, which supports special description languages purposed for representing the hardware operation. Currently, two types of description languages are available: Hardware Description Languages and Architecture Description languages.

The complexity of modern processors requires their logic design to be extensively tested before they are manufactured. Hardware Description Language (HDL) models are created to allow this simulation. Compared with going directly to circuit design, HDL modeling dramatically reduces design time and logic bugs. HDLs are also designed to be independent of the manufacturing process, so that logic designs are moved easily from one manufacturing generation to the other, [17]. The most commonly used forms of HDL are Verilog and VHDL.

Architecture Description Languages (ADLs) are becoming popular recently because of their quick and optimal design convergence achievement capability during the design of different kinds of processors. Out of the many available ADLs, the preferred one in most cases is Language for Instruction Set Architecture (LISA). It provides a lot of flexibility and powerful manipulation and verification tools that can be used through the design process while reducing the gap between the traditional design of a processor using VHDL or Verilog and instruction set languages for architecture exploration, [18].

The main characteristic of LISA is the operation-level description of the pipeline which is able to model even complex interlocking and bypassing techniques. Instructions consist of multiple operations which are defined as register transfers during a single control step. Depending on the requested accuracy, a control step can be an instruction-, clock-, or phase-cycle. Operation scheduling in LISA is based on modified Gantt charts (L-charts) specifying time and resource allocation of operations and an operation sequencer with an ASAP (As Soon As Possible) operation sequencing strategy, [19].

B. Automatic Design Tools

As processor designs have steadily increased in complexity, processor design teams have also grown. To allow continuous increases in complexity while preventing design teams from growing even further, engineers must rely upon design automation, [17]. There are several automated design solutions available today, including Tensilica's Xtensa Processor Developer's Toolkit, MetaCore Application-Specific Programmable DSP Development System, architectural level processor design environment PEAS III, tool for automated multiprocessor system design ESPAM, Synopsys (former CoWare) Processor Designer and a few others.

Synopsys Processor Designer is an automated, application-specific embedded processor design and optimization environment that slashes months from processor hardware design time and engineer-month from the creation of application processor-specific software development tools. Processor Designer's high degree of automation enables design teams to focus on architecture exploration and application-specific processor development, rather than on consistency checking and verification of individual tools.

The key to Processor Designer's automation is its Language for Instruction Set Architectures, LISA 2.0. In contrast to SystemC, which has been developed for efficient specification of systems, LISA 2.0 is a processor description language that incorporates all necessary processor-specific components such as register files, pipelines, pins, memory and caches, and instructions. It enables the efficient creation of a single "golden" processor specification as the source for the automatic generation of the instruction set simulator (ISS) and the complete suite of software development tools, like Assembler, Linker, Archiver and C-Compiler, and synthesizable RTL code. The development tools, together with the extensive profiling capabilities of the debugger, enable rapid analysis and exploration of the application-specific processor's instruction set architecture to determine the optimal instruction set for the target application domain. Processor Designer enables the designer to optimize instruction set design, processor micro-architecture and memory sub-systems, including caches, [20]. Considering the suitability of this processor design tool, we decided to use it for LISA implementation of the novel NP cores architecture, we are proposing.

C. Field Programmable Gate Array (FPGA)

FPGA is a logic device which can be custom configured and programmed after its manufacturing ("in the field"). Basically, it serves as platform for simulating and testing, any kind of custom designed hardware. Its ability for reconfiguration, offers various advantages for many applications, [21].

The architecture of FPGAs has become increasingly complex and no longer consists of a simple array of lookup tables and flip flops connected by programmable routing. FPGAs now include on-chip RAM blocks, and multipliers. FPGA devices and their architectures vary across device families and across vendors. In this work we focus on Xilinx's Virtex 5 FPGA family and hence we discuss its architecture in more detail.

XUPV505-LX110T is general purpose evaluation and development platform for hardware simulation. It is consisted of many components, such as: Xilinx Virtex-5 XC5VLX110T FPGA, Flash PROMs, 64-bit wide 256Mbyte DDR2, 10/100/1000 tri-speed Ethernet PHY interfaces, RS-232 port, 16x2 character LCD, and many other I/O devices and ports. This platform is very suitable for education and research in many different areas like digital system design, embedded systems, computer architecture, networking etc., [22].

IV. Implementation of 32-bit RISC Based Network Processor Core

To facilitate with the increasing demands for high performance network processing, we suggested novel NP core organization, based on 32-bit RISC architecture. Assuming that the Intel IXP multi-gigabit NPs use 32-bit RISC micro engines, we decided to use a very simple DLX architecture, for the novel RISC based NP core design. DLX is an academic hypothetical architecture, developed by John L.

Hennessy and David A. Patterson, presented in [23]. In fact we are going to augment and extend the initial DLX architecture, so it could be capable for performing multi-gigabit network processing.

A. DLX Basic Architecture

DLX is a 32-bit load-store multi-cycle RISC architecture. It provides 32 general purpose registers, all of which are 32 bits wide. Three types of 32-bit instructions are defined, as shown in Table 1.

TABLE I
DLX INSTRUCTION FORMATS

Format	Bits					
	31-26	25-21	20-16	15-11	10-6	5 - 0
R-type	0x0	rs1	Rs2	rd	not used	op. code
I-type	op. code	rs1	Rd	immediate		
J-type	op. code	value				

As can be seen from Table 1, an R-type instruction code consists of six zero-bits, followed by the two registers containing the source operands. Next instruction field is the destination address register, followed by five unused bits. The operation to be executed is encoded in the least significant six bits. Unlike R-type, I-type and J-type instructions format encode the operation type using the most significant six bits of the instruction code. Following instruction fields are source register, destination register and immediate value which are later sign extended at I-type instructions and the actual memory address value on which the program counter would point at J-type instructions.

B. DLX Pipeline

Earlier we mentioned that DLX is a pipelined architecture. Pipelining is one of the key concepts in Computer Architecture, defined as an implementation technique in which multiple instructions can be executed in overlapping fashion. This is possible if the operation tasks performed in each cycle of a multi-cycle architecture are clearly defined and independent from each other, [24].

The DLX pipeline is a five stage pipeline. Each instruction's execution follows some or all of the pipeline stages corresponding to these cycles: Instruction Fetch (IF), Instruction Decode/Register Read (ID), Execution/Effective Address (EX), Memory Access/Branch Completion (MEM), and Write-back (WB) cycle. Communication among the various stages is accomplished using pipeline registers, as shown in Fig.2.

Instruction Fetch is a primary pipeline cycle in which a new instruction is fetched from the instruction memory and then sent to the Instruction Register (IR), while Program Counter (PC) is incremented by 4. Next, during the Instruction Decode/Register Read (ID) cycle, the fetched instruction is decoded and register file is accessed in order to initialize internal registers. The following cycle is

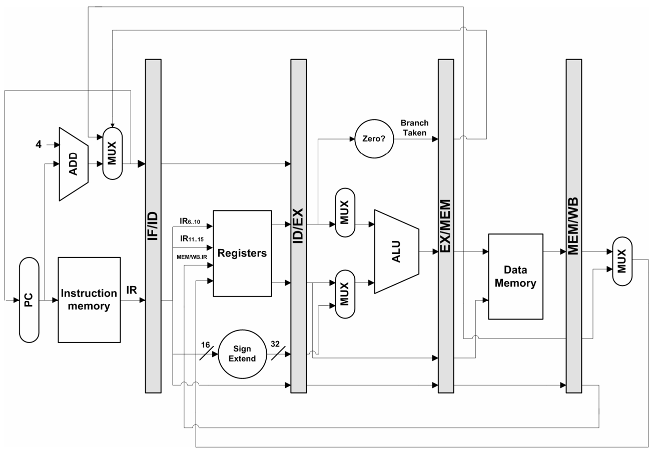


Fig. 2. Pipelined DLX architecture. As shown, DLX load/store architecture features a 5-stage instruction pipeline including: instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM) and write back (WB). The communication between pipeline stages is realized by registers, placed between two sequential stages.

Execution/Effective Address, during which the ALU operates on the operands prepared in the previous cycle. This operation could be computing for the target branch address, a register-register ALU operation, a register-immediate ALU operation or memory reference address. Next is Memory Access/Branch Completion Cycle, during which only those instructions requiring memory access are active. Branch completion (i.e., branch decision) and updating of the PC are also done during this cycle. The last pipeline cycle is Write-back, during which the result either coming from the ALU or the memory is written back to the register file.

C. LISA Implementation

Considering the above description of the DLX processor, we implemented it through definition and ADL implementation (in LISA 2.0) of several groups of instructions. We primarily focused on integer instructions, since those are the ones needed for processor extension. DLX architecture also supports floating-point operations.

In order to ease the implementation, we divided integer instructions into four groups, based on their function: arithmetic instructions, memory-access instructions, control-flow instructions and test-and-set instructions. All of them have some distinctive characteristics, as we will see shortly.

Instructions are not the only resource needed for DLX processor to function properly. Some registers and memory are also obligatory. Therefore, 32 general-purpose registers (GPRs) were implemented as well as 4 pipeline registers. Each of them is 32 bits wide. All of the GPRs can be used as sources as well as destination registers, except GPR [0] (which is zero-register) and GPR [31] (reserved for copying the Instruction Register during jumps).

Regarding memory, two memory blocks were initially defined: program memory and data memory. Each of the two is four KB in size. Later, during the extension process, three additional memory blocks would be defined.

D. *Arithmetic Instructions*

There are two arithmetic instructions variants: I-type and R-type. The former has one of the operands hard coded in the instruction code as an immediate value and in latter both source operands are registers. The result of the operation is always placed back into a register.

TABLE II
IMPLEMENTED ARITHMETIC INSTRUCTIONS

Instruction	Syntax	Operation	Type
add	add rd,rs1,rs2	rd = rs1 + rs2	R
sub	sub rd,rs1,rs2	rd = rs1 - rs2	R
and	and rd,rs1,rs2	rd = rs1 && rs2	R
or	or rd,rs1,rs2	rd = rs1 || rs2	R
xor	xor rd,rs1,rs2	rd = rs1 ^ rs2	R
addi	addi rd,rs1,imm	rd = rs1 + extend(imm)	I
subi	subi rd,rs1,imm	rd = rs1 – extend(imm)	I
andi	andi rd,rs1,imm	rd = rs1 && imm	I
ori	ori rd,rs1,imm	rd = rs1 || imm	I
xori	xori rd,rs1,imm	rd = rs1 ^ imm	I
sll	sll rd,rs1,rs2	rd = rs1 << (rs2 % 8)	R
srl	srl rd,rs1,rs2	rd = rs1 >> (rs2 % 8)	R
sra	sra rd,rs1,rs2	same as srl*	R

Table 2 depicts implemented arithmetic instructions. Some of their characteristic properties are as follows. Extend (imm) extends the immediate operand using its most significant bit so that it is 32 bits wide. SRA and SRAI are arithmetic right shifts. This means that, instead of shifting in zeroes from the left, the sign bit of the operand is duplicated. SRL and SRA perform identically if Rs1 is positive. If Rs1 is negative (bit 31 == 1), 1's are shifted in from the left for both SRA and SRAI.

E. *Memory-Access Instructions*

Basic memory-access instruction set was implemented, which enables word loading and word storing. Details are given in Table 3. Extend (imm) has the same function as defined for the arithmetic instructions.

TABLE III
IMPLEMENTED MEMORY-ACCESS INSTRUCTIONS

Instruction	Syntax	Operation	Type
lw	lw rd,rs1,imm	rd = MEM[rs1 + extend(imm)]	I
sw	sw rd,rs1,imm	MEM[rs1 + extend(imm)] = rd	I

F. *Control-Flow and Test-and-Set Instructions*

Control instructions can be classified into two groups: conditional jumps and unconditional jumps (branches). Unconditional jumping (branching) is actually jumping from the current position forward (if c > 0) or backward (if c < 0) by c instructions (words). Since instructions can only begin at a word boundary, it makes sense giving the offset in words rather than in bytes, as this increases the range of the jump by a factor of four. Jump means going to an absolute address. Here, the address is given as a byte address, and not as a word address. Implemented control-flow instructions are given in Table 4.

TABLE IV
IMPLEMENTED CONTROL-FLOW INSTRUCTIONS

Instruction	Syntax	Operation	Type
bnez	bnez rd,rs1,imm	PC += (rs1 != 0 ? extend(imm) : 0)	I
beqz	beqz rd,rs1,imm	PC += (rs1 == 0 ? extend(imm) : 0)	I
blt	blt rd,rs1,imm	PC += (rd < rs1 ? extend(imm) : 0)	I
jump	jump val	PC+= extend(val)	J
jal	jal val	r31=PC+4; PC+= extend(val)	J

Test-and-set instructions, on the other hand, do not directly change the program flow. Instead, they are used before a control-flow instruction is executed, to see whether some condition(s) are met, and set the flags accordingly. Several test-and-set instructions were implemented, but not shown in this paper due to lack of space.

G. *Other Resources*

As with any processor, the DLX CPU needs a way to communicate to the outside world. This can be done via a few simple signals. In addition to a data and address bus (with the necessary control signals), DLX needs a RESET signal and a clock. These signals and buses are described in this section.

After powering up the CPU, it needs some way of getting into a known state. This can also be done at any point of execution when the CPU needs to restore to a reasonable starting point. For that purpose, DLX uses the RESET signal. When the RESET signal is asserted (high), the CPU loads the program counter with 0. After RESET is deasserted, execution begins at location 0, which should probably be the address of the first instruction of a program to be executed.

An external clock signal is also required so that DLX processor can function the way it is supposed to. This can be provided in one of two ways. For debugging, a "manual" clock signal is likely to be the best. This signal should allow us to manually set the clock signal to high and low alternately. Since this CPU is being designed in a simulator and cycle time is unimportant, this method will allow us to take the time examining the CPU after each clock cycle. Once the CPU is working, however, we should use a "real" clock. For that purpose, we can create one in the simulator. However, we should make sure the cycle time is sufficiently long. If it isn't, the CPU may not work

H. *Processor Verification*

The verification of implemented DLX processor functionality is done by comparing the result obtained in simulation and the one attained using existing and already verified processor architecture. The same result achieved in both simulation and existing hardware, proves the functionality of the novel implemented NP core. According to that, we simulated the proposed NP core in Synopsys processor debugger tool, which is an integrated part of the processor designer tool, used for the NP core design. This simulation environment allowed us to analyze and test the execution of assembly programs, written using the NP core instruction set. We executed the assembly programs, and during the simulation, we could examine when and how the processor resources (registers, pipeline registers or memory) were updated, which stage of the pipeline was performed, which operation was executed at that moment, which instruction could stall the pipeline and etc.

The DLX operation was verified with two different assembly language programs, one of which contains simple mathematical operations, while the second one implements the bubble-sort algorithm. Using the former, every instruction of the processor's instruction set was proven to be working properly. The latter program verified the processor's ability to cope with more time-consuming tasks, while generating the expected results. Therefore, we can outline that the DLX processor hardware implementation is correctly completed. The DLX processor provides all the required functionalities.

I. DLX Extension for Network Processing Application

After verifying processor's functionality, our next task was its extension, so that it could work as a NP core. This extended processor would be able to process network packets in hardware, speeding up the routing process.

The bare DLX is general purpose architecture. As mentioned before, NPs do make use of general purpose processing functionality, but on the other hand, they must provide application specific functionalities, as required in network devices. These application specific functionalities include header processing, classification and routing, policing, queuing and finally packet forwarding.

In order to meet these requirements, we implemented two new, network processing specific instructions. Outline of their characteristics is given in Table 5.

TABLE V
IMPLEMENTED NETWORK PROCESSING INSTRUCTIONS

Instruction	Syntax	Operation	Type
ldhdr	ldhdr r1 cnt dst	[dst..dst+cnt]= MEM[r1..r1+cnt]	I
sthdr	sthdr r1 cnt src	MEM[r1..r1+cnt]= [src..src+cnt]	I

Once a new packet arrives, it should be buffered in order to be processed and later sent to the output. When it comes to buffering input packets, there are two main approaches, as discussed in [25]. One alternative is to write packets directly to the input buffer and then process the headers, reading them in from there. A second alternative is to stream packets through the header-processing unit before they are stored in the buffer. Here, we use a combination of the two: as packets arrive, their header is written in a small, but fast header input buffer, while the packet content is written in larger, but slower buffer.

Header input buffer is implemented as memory range of 6000B, allowing maximum 100 packet headers to be stored simultaneously. In order to enable header loading from this buffer, a specific instruction was implemented (LDHDR). It places every header word into a separate, predefined register, so that after its execution the whole header is available in registers. While loading the header, this instruction also computes its checksum. Invalid headers are not processed any further. Once the header is completely processed, the corresponding packet should be stored into an output buffer, waiting to be forwarded. This output buffer is implemented as memory range too, this time 512KB in size, which is enough for 8 complete packets. Again, we have implemented a specific instruction for this purpose (STHDR). It reads all predefined registers, storing their content in consequent memory locations.

All those modifications and newly implemented instructions should enable the novel NP core to process network packets, provided they are already somehow placed in memory. Additionally, we improved the NP core architecture by defining alias registers, which would provide direct access to the appropriate IP header fields, placed in the header input buffer. All instructions could use this alias registers as operands. The extended 32-bit RISC-based architecture is shown in Fig. 3.

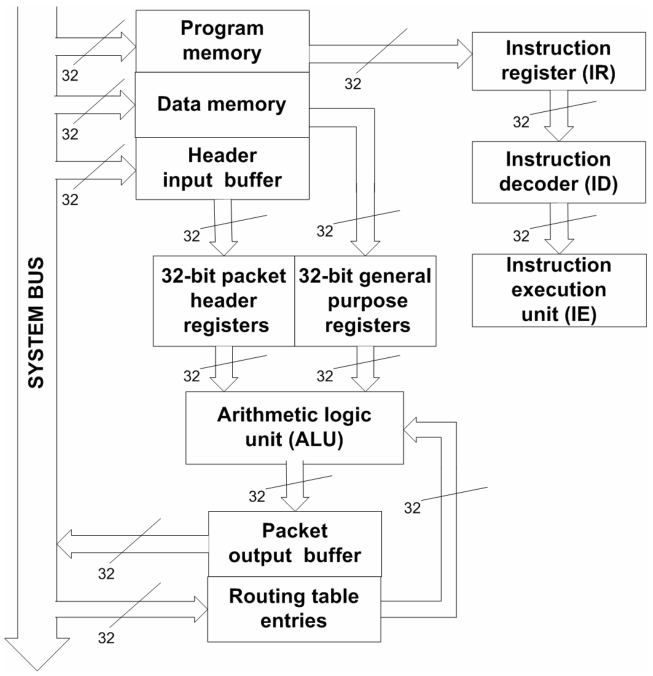

SYSTEM BUS
Program memory
Data memory
Header input buffer
Instruction register (IR)
Instruction decoder (ID)
Instruction execution unit (IE)
32-bit packet header registers
32-bit general purpose registers
Arithmetic logic unit (ALU)
Packet output buffer
Routing table entries
32

Fig. 3. Proposed architecture of 32-bit RISC based NP core. The DLX RISC core was extended with input and output buffer, intended for storing the IP header. Additionally, the NP core instruction set was augmented with some novel instructions that simplify the packet processing. Each NP instruction could directly manipulate with the IP header fields, addressed by alias registers.

In order to verify processor's functionality, a custom made IP processing program was written. In it, we statically place packets in memory. These packets are to be processed and forwarded as needed. At the beginning of the program, 32 bits (the first word) of each packet header is read into processor registers. Packet version (4 bits), Header length (4 bits), Type of service (8 bits) and Packet length (16 bits) are then available in general purpose register (GPR[0]). Until now, only IP packets can be processed, so once the correct version is ensured, the rest of the header is loaded. This is done using LDHDR instruction. As a result, the rest of the header is now available in registers 1 to 4 (GPR[1]-GPR[4]). While loading the header, a checksum is also computed. Two conditions must be met in order to continue processing: checksum must be correct and packet's destination address should not match any IP address of the processing machine.

After executing this program using a simulator for the new processor, we were able to compare results obtained from the simulation with those we had computed. Results matched, packets were correctly routed (written to output buffer), and routing table searches yielded correct next hops. This verifies correctness and applicability of the novel NP core.

V. Implementation of 64-bit RISC Based Network Processor Core

In order to achieve better network processing performances, we decided to implement 64-bit RISC based NP core, so we could measure its processing speed and compare it with the previous designed NP core. We believe that data-width incensement would influence on the packet processing speed. Once again, we use a very simple RISC architecture with Harvard organization. The use of RISC architecture has shown many advantageous in various multi-gigabit NPs, such as Intel IXP1200. During the NP core design, we would try to enrich the initial architecture of a 64-bit RISC core, and augment its fundamental instruction set. We hope that with appropriate internal hardware and software interventions, the proposed NP core could be involved in a multi-Gb/s routing applications.

A. Basic NP Core Architecture

The proposed NP core is based on standard 64-bit RISC processor architecture, augmented with several hardware accelerators and adjusted for IP packet processing. The NP core uses Harvard organization, which is very advantageous, since read/write operations to the data/program memory can be performed at the same time. Furthermore, the NP core employs RISC architecture, which allows execution of short and simple one cycle instructions, and implements ILP by 5-stage (fetch, decode, execute, memory access and write back) pipeline. Additionally, the RISC based 64-bit NP core is capable of transferring 64-bits at a given moment. Actually, the main idea for designing this NP core is the possibility of modifying 64-bit general purpose RISC processor and adapting it for network processing application. We believe that the proposed NP core would be able to speed up the routing process and would achieve high network processing performance.

The proposed NP core architecture is composed of: internal program and data memory (instruction and data cache), 64–bit ALU, two operand and one result register, 128 general purpose registers and 64 packet header registers (packet header buffer). All the data paths between processor register, memory and other structures are 64-bits wide. The processor core, as usual, includes program counter and instruction register, responsible for instruction execution control. Additionally, the NP core employs buffer status register, which can be used for storing some important information (IP version, header length etc.) during the packet processing. Furthermore, the NP core implements hardware accelerator for checksum calculation. This way, the processor speeds-up the execution of a very complex and time-consuming operation. The proposed internal architecture of the 64-bit NP core is presented in Fig. 4.

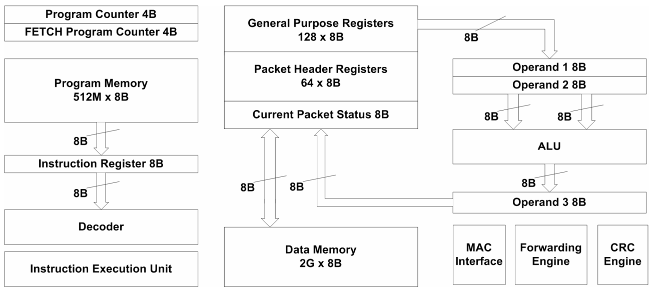


Fig. 4. Proposed architecture of 64-bit RISC based NP core. The initial 64-bit RISC core was extended with packet header buffer, responsible for storing the IP header. The NP core could perform very fast manipulation with the IP header, since the header was already placed in the NP core registers and thanks to the alias registers defined for each IP header field. The instruction set was extended with some network instructions, such as for checksum calculation.

B. Network Processing

We consider general packet processing for both IP versions. IPv4 packets are processed by executing the following operations: verification of packet header fields (version, packet length, source and destination address), CRC validation, route table look up, simple changes of some header fields (decrement of time-to-live), calculation of new CRC value and forwarding to outbound port, [25]. On the other side, IPv6 packets processing excludes some of these operations, such as CRC code validation and calculation. Additionally, IPv6 header, employs hop count limit field (its value is incremented during the packet processing), instead of time-to-live (TTL), [26].

All the IP processing operations are supported by the 64–bit ALU, working with two sources, and one destination operand register. The ALU allows basic arithmetic/logic operations and additional simultaneous shifting of the second operand.

C. Packet Header Buffer

Packets received on the MAC interface, are stored in the data memory, and after their processing is finished, they are sent out to the forwarding engine. Usually, only the IP packet header is being processed by the NP core, so in order to accelerate its processing, the header is loaded and stored in additional 64 packet header registers. The packet header registers enable completely storing of each IPv4 header (including the option fields) or IPv6 header (including one extension header). This header registers provide fast data access and manipulation, since each IP header field can be individually and directly accessed, changed, and afterwards, restored in memory with the new (re)computed values. The NP core allows direct specifying of each IP header field as operand in NP instructions for IP packet processing.

D. Alias Registers

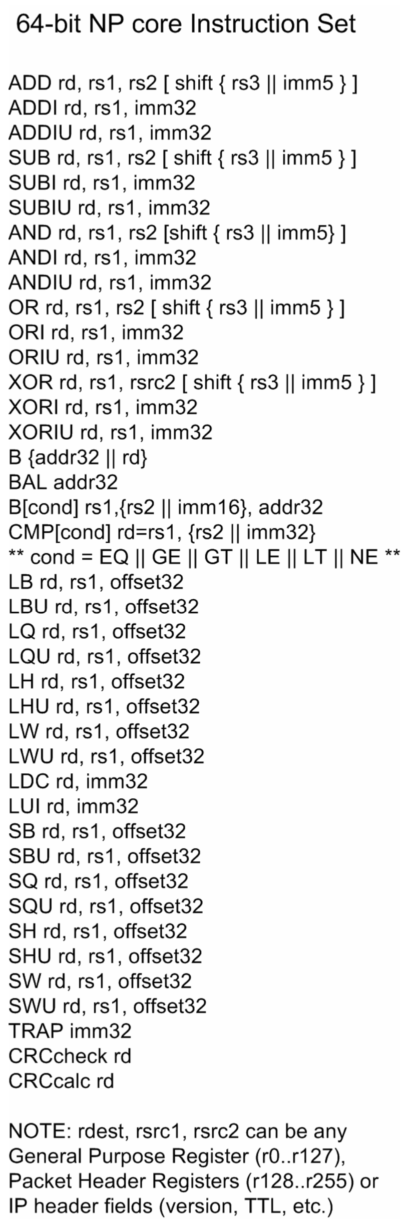
The NP core is composed of 128 general purpose registers, and 64 packet header registers. All of them can be addressed with completely 8 bits. The codes starting with b00, b01 or b10 are used to denote register indexes, and the remaining 64 codes, starting with b11 are used for addressing the packet header fields, that are in fact alias registers. Therefore, the last 64 codes are divided, one half for IPv4 fields, and the other for IPv6 fields. Then, for example, the first general purpose register is addressed as b000000000, and the first field of the packet header (IPv4), is addressed as b11000000. All instructions can work with these alias registers as operands. This allows for a more flexible (and faster) packet header processing and greater convenience to the programmer. When the compiler is built, this kind of access to the packet header fields will be allowed via system calls.

E. Instruction Set

The NP core implements enchanted instruction set which is specially tailored to network processing application. Accordingly, it employs several specific instructions for hardware accelerators control, and CRC code validation and calculation. The NP core instructions have RISC based format, and they are additionally adjusted for IP header fields (for both IP versions) manipulation during the network processing. This involves utilization of an additional addressing mode that allows direct access to the IPv4 and IPv6 packet header fields by specifying their names (ex. ip4_ver, ip4_header_length, etc.).

The instruction set is composed of some very simple general purpose and several special purpose instructions. The instructions are 64-bits wide, and can be given in one of the following three instruction formats: register, immediate or control (R, I and C format, accordingly) type.

The register instructions such as sub, add, xor etc. are three-address instructions, which operate with register value operands. Additionally, these instructions allow shifting of the second operand, before the execution of some arithmetical/logical instruction. On the other side, the immediate instructions (load, store, add, etc.) are responsible for register-to-memory or memory-to-register transfer, and conditional brunches. These instructions always include at least one immediate value operand. However, according to the operands used, some of the instructions (ex. comparison, addition) can be implemented as either R-type or I-type. The last instruction format, C – type, is used to express: unconditional branching, procedure calls, CRC code validation and calculation, and trap instructions. The instruction set of the proposed NP core is shown in Fig. 5.


64-bit NP core Instruction Set

```
ADD rd, rs1, rs2 [ shift { rs3 || imm5 } ]
ADDI rd, rs1, imm32
ADDIU rd, rs1, imm32
SUB rd, rs1, rs2 [ shift { rs3 || imm5 } ]
SUBI rd, rs1, imm32
SUBIU rd, rs1, imm32
AND rd, rs1, rs2 [shift { rs3 || imm5} ]
ANDI rd, rs1, imm32
ANDIU rd, rs1, imm32
OR rd, rs1, rs2 [ shift { rs3 || imm5 } ]
ORI rd, rs1, imm32
ORIU rd, rs1, imm32
XOR rd, rs1, rsrc2 [ shift { rs3 || imm5 } ]
XORI rd, rs1, imm32
XORIU rd, rs1, imm32
B {addr32 || rd}
BAL addr32
B[cond] rs1,{rs2 || imm16}, addr32
CMP[cond] rd=rs1, {rs2 || imm32}
** cond = EQ || GE || GT || LE || LT || NE **
LB rd, rs1, offset32
LBU rd, rs1, offset32
LQ rd, rs1, offset32
LQU rd, rs1, offset32
LH rd, rs1, offset32
LHU rd, rs1, offset32
LW rd, rs1, offset32
LWU rd, rs1, offset32
LDC rd, imm32
LUI rd, imm32
SB rd, rs1, offset32
SBU rd, rs1, offset32
SQ rd, rs1, offset32
SQU rd, rs1, offset32
SH rd, rs1, offset32
SHU rd, rs1, offset32
SW rd, rs1, offset32
SWU rd, rs1, offset32
TRAP imm32
CRCcheck rd
CRCcalc rd
```

NOTE: rdest, rsrc1, rsrc2 can be any General Purpose Register (r0..r127), Packet Header Registers (r128..r255) or IP header fields (version, TTL, etc.)

Fig. 5. Instruction set of the 64-bit RISC based NP core. The instruction set was extended with some specific instructions, such as CRC code check and calculation. All the instructions could operate with the IP header fields (alias registers), as instruction operands.

F. Addressing Modes

The NP core is RISC based, so it should support very simple addressing modes, [2]. As a result, it implements simple addressing modes like register, immediate and index addressing. Additionally, the most of the instruction operations are executed by memory or register accesses. Some of the instruction operands can be provided as alias registers, specified by the appropriate IP header field name.

G. LISA Implementation and Processor Verification

The proposed NP core architecture was modeled using the language for instructions set architectures - LISA. This modeling language is general enough to model any kind of instruction set driven processors, and yet powerful enough to model highly specific instruction set processors, [27].

Therefore we used it to model the proposed 64-bit NP core, and analyze its characteristics. We defined its memory and bus architecture, a standard 5-stage instruction pipeline and the instruction set specific to the network processing. From the LISA model we simulated the processor within the Synopsys processor designer tool, and afterwards we verified its functionality and performance within the Synopsys processor debugger tool. Accordingly, we wrote some assembler programs to prove that the NP instructions are working properly. The obtained simulation results allowed us to verify the NP core capability to perform network processing.

H. FPGA Implementation

The processor designer environment includes processor generator tool, which allows automated HDL code generation from the LISA model. The attained HDL code can be used for investigation at a lower level.

In very near future, we are going to simulate the proposed NP core on a Xilinx VIRTEX 5 FPGA board. This could help us to achieve performance estimation that would be closer to real hardware. Hence, there are some issues that need to be taken in consideration, such as circuit complexity, power consumption and overheating. These characteristics might significantly influence on the overall performance that could be achieved.

VI. PERFORMANCE ESTIMATION

The present transition from circuit switch to packet switch networks has caused network traffic doubling every 12 – 18 months, [2]. Consequently, it is expected that until 2015 the Internet throughput would increase to 1 Tb/s. At the same time, processors performance is limited by Moore's law and power constraints. As a result, NPs should provide high speed computing, while overcoming all these limitations, and as well they should scale with the increasing computing performances.

In order to estimate the performance trade-offs for the proposed NP RISC based cores, we provide some simple computations for calculating the theoretical maximum of instruction cycles allowed for each IP packet processing at the desired speeds of 10/100 Gb/s. The results of the equations given in (1) and (2) can be used as a theoretical limit which can be compared with the NP cores results. Consequently, this way we can estimate the network processing performance capabilities of the proposed NP cores.

$$\text{Average rate of packets} = \frac{\text{data rate}}{\text{average size of packets}}\ [\text{number of packets/s}] \quad (1)$$

$$\text{Average time for processing one packet} = \frac{1}{\text{average rate of packets}}\ [\eta\text{s}]$$
$$= \frac{\text{average time for processing one packet}}{\text{one processor cycle time in } \eta\text{s}}\ [\text{number of cycles}] \quad (2)$$

Therefore, if the NP cores are working at 2GHz frequency, and average data packet size is 512B, the theoretical number of processor cycles acceptable for multi-gigabit processing of single packet is: 820/82 processor cycles for attaining 10/100 Gb/s packet processing speeds, accordingly. In order to satisfy these high performance requirements, NP cores must implement some parallelization techniques or hardware accelerators. Accordingly, the both NP cores employed a standard 5 stages pipeline, which allowed us to increase the packet processing throughput. Additionally, we could minimize the dependences between sequential instructions, and therefore pipeline stalls, by reorganizing and reordering the IP processing assembler programs code, executed on the NP cores.

Assembly programs execution was monitored using Synopsys processor debugger tool. While proving processor's design correctness and testing its functionality, this tool allowed us to also estimate its performance. As mentioned earlier, it allows a designer to track all changes raised in memory and registers (including pipeline registers), as well as additional custom resources. We used custom resources section to monitor values in global variables, as well as their effect over program execution. We took advantage of another possibility the tool offers – tracing. This allowed us to estimate the number of cycles needed for each instruction execution.

Considering the proposed 32-bit and 64-bit RISC based NP cores, we evaluated their performance by analyzing assembler programs for general IPv4 and IPv6 packet processing. We simulated these programs and estimated the number of processor cycles needed for each type of IP processing. Additionally we considered that the NP cores can achieve different results, according to the memory type used (DRAM/SRAM), [11]. In our analyses, we compared the network processing performance of the proposed 32-bit and 64-bit RISC based NP core with the general 32-bit and 64-bit RISC core, and afterwards we measured the possible improvements. Attained results for each of the processing cores, using DRAM/SRAM memories are shown in Fig. 6 and 7, accordingly.

Fig 6 depicts that the modified 32-bit NP RISC core, utilizing DRAM memory, achieves IPv4/IPv6 processing for 435 and 665 processing cycles, accordingly. Furthermore, the modified 64-bit NP RISC core attains better results, finishing the IPv4/IPv6 processing for only 355, and 500 processor cycles. On the other side, Fig. 7 presents that the modified

32-bit NP RISC core, using SRAM memory, achieves IP processing for 135, and 185 processor cycles, for IPv4, and IPv6 packets, accordingly. Additionally, the modified 64-bit NP RISC core achieves better results, finishing the IPv4/IPv6 processing for only 115, and 140 processor cycles.

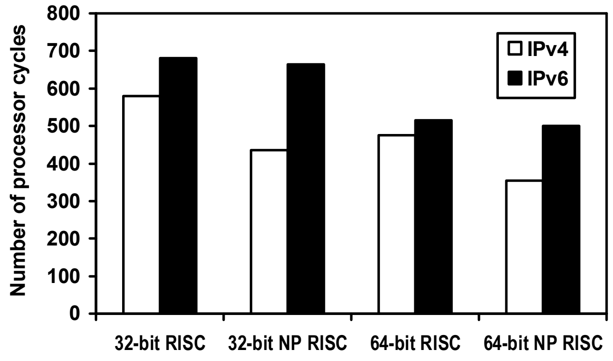


Fig. 6. Number of IP packet processing cycles (IPv4 and IPv6), for each of the defined processing cores: 32-bit GPP RISC, 32-bit NP RISC, 64-bit GPP RISC and 64-bit NP RISC. All these processing cores utilize slower DRAM memory. The initial results show that the proposed 32-bit and 64-bit NP RISC cores improve the network processing performance, compared to the general purpose RISC cores.

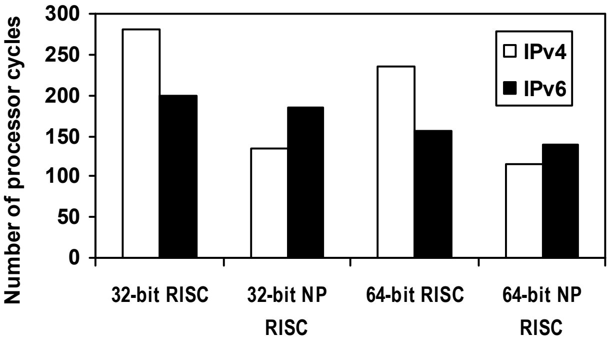


Fig. 7. Number of IP packet processing cycles (IPv4 and IPv6) for each of the defined processing cores: 32-bit GPP RISC, 32-bit NP RISC, 64-bit GPP RISC and 64-bit NP RISC. All these processing cores utilize faster SRAM memory. The initial results show that the proposed 32-bit and 64-bit NP RISC cores improve the network processing performance, compared to the general purpose RISC cores.

In the both cases, the modified RISC NP cores achieve better results, compared to the general purpose RISC processing cores. The processor cycles gain, by utilizing the novel NP cores is given in Fig. 8.

The results given in Fig. 8 prove that the proposed NP cores achieve some performance improvements. Actually, the proposed 32-bit NP RISC core, using DRAM memory, accelerates the packet processing by 25% and 2% for Ipv4, and IPv6 packets, accordingly. The same NP core, using SRAM memory achieves 52% and 7,5% processor cycles gain, for Ipv4, and IPv6 packets, accordingly. For the 32-bit NP RISC core using DRAM memory the achieved improvements are: 25% and 3% for Ipv4, and IPv6 packet processing, accordingly. Additionally, the same NP core using SRAM memory achieves 52% and 10% processor cycles gain, for Ipv4, and IPv6 packets, accordingly.

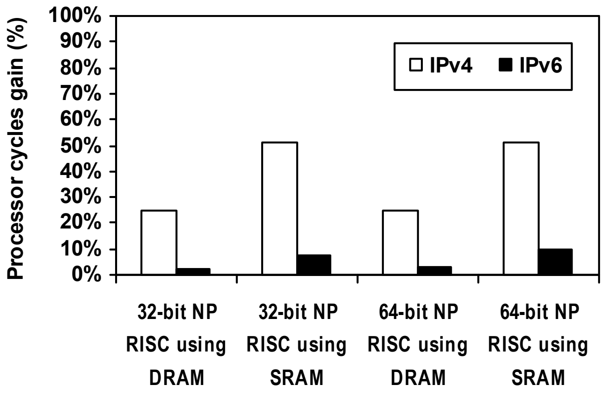


Fig. 8. Ipv4 and IPv6 processor cycles gain, attained with the novel NP cores. The results show that the proposed NP cores achieve faster IPv4 and IPv6 processing, and consequently better network processing performances. At least, they are within 12-19 Gb/s boundaries of the theoretical limit.

These results show that the proposed NP cores, can achieve multi-gigabit processing in the theoretical boundaries of at least 12-19 Gb/s. The initial results are satisfying, so in very near future we would consider the possibility to design multi-core NP architecture, with homogeneous NP cores. For that reason, we made an investigation of the performance achievements of the micro engines in the very well known Intel IXP1200 network processor. According to [28], a single Intel IXP1200 micro engine needs totally 710 processor cycles, for packet processing. These processor cycles include: 280 cycles of registers instructions, and 430 cycles of memory delay. According to that, the proposed NP cores achieve very reasonable performances, similar to the Intel IXP 1200 micro engines, as shown in Fig. 6 and 7. Therefore, the authors should continue the research and consequently design multi-core NP architecture.

VII. Conclusion

In this paper, we are proposing two novels RISC based NP cores that should be able to cope with multi-gigabit networks. We initially described the NP cores architecture, including their instruction set, registers and additional resources, and afterwards we implemented them in Language for Instruction Set Architecture, using the Synopsys processor designer tool. This environment allowed us to verify the NP cores functionalities and measure their network processing performances.

The proposed NP cores are specialized for network processing application. Their key architectural aspects are: enhanced instruction set, implementation of five stage pipeline, execution of complex instructions in one cycle, use of packet header buffer for holding the IP header, and use of alias registers for easier manipulation with the IP header fields. We have shown that the proposed architectural modifications have significantly improved the network processing capabilities of the initial general purpose RISC processors. Designed NP cores are able to fulfill the current network processing speeds and could be able to cope with multi-gigabit (12-19 Gb/s) links of Next Generation Networks.

Since the NP cores, can achieve similar network processing performance as a single micro engine of the very well known Intel IXP1200 NP, in a very near future we would consider the possibility of designing multi-core NP with homogeneous NP cores. Additionally, we could investigate the ability to use the designed NP cores as hardware support for novel routing protocols intended to speed-up the network routing. Therefore, there is ongoing work for performing additional hardware and software simulations in order to accomplish these aims.

References

[1] H. Jonathan Chao, Bin Liu, *High Performance Switches and Routers High speed switches and routers,* Wiley-IEEE Press, May 2007

[2] Ran Giladi, *Network Processors - Architecture, Programming and Implementation,* Morgan Kaufmann Publisher, Ben-Gurion University of the Negev and EZchip Technologies Ltd., 2008

[3] Mahmood Ahmadi, Stephan Wong, "Network Processors: Challenges and Trends", Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing,, ProRisc, Veldhoven, The Netherlands, November 2006, pp. 222-232

[4] Panos C. Lekkas, *Network Processors: Architectures, Protocols and Platforms,* McGraw-Hill Professional, 2003

[5] Mohammad Shorfuzzaman, Rasit Eskicioglu, Peter Graham, "Architectures for Network Processors: Key Features, Evaluation, and Trends", Proc. on Communications in Computing, 2004, pp.141-146

[6] NetFPGA Online Guide, [online]. Available: http://netfpga.org/ [Accessed 10 May 2011]

[7] Jad Naous, Sara Bolouki, Glen Gibb, Nick McKeown, "NetFPGA: Reusable Router Architecture for Experimental Research", Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow, Stanford University, California, USA, 2008

[8] Michele Petracca, Robert Birkea, Andrea Bianco, "HERO: High-speed enhanced routing operation in software routers NICs" , Proceedings of the 4th international telecommunication networking workshop on QoS in multiservice IP networks, Politec. di Torino, 2008

[9] Simon Hauger, Thomas Wild, Arthur Mutter, Andreas Kirstädter, Kimon Karras, Rainer Ohlendorf, Frank Feller, and Joachim Scharf, "Packet Processing at 100 Gbps and Beyond—Challenges and Perspectives", in Proceedings of the 10. ITG Symposium on Photonic Networks, May 2009

[10] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in Proc. IEEE INFOCOM'98, Session 10B-1, San Francisco, CA, 1998, pp. 1240–1247.

[11] W. Eatherton, G. Varghese, Z. Dittia, "Tree bitmap: Hardware/Software IP Lookups with Incremental Updates", SIGCOMM Comput. Commun. Rev., vol. 34, no. 2, 2004.

[12] M. Gries, C. Kulkarni, C. Sauer, K. Keutzer, "Exploring Trade-Offs in Performance and Programmability of Processing Element Topologies for Network Processors", 2nd Workshop on Network Processors (NP2) at the 9th International Symposium on High Performance Computer Architecture, 2003, pp. 75–87

[13] *Intel IXP2800 Network Processor® Product Brief, For OC-192/10 Gbps network edge and core applications*, Intel Corporation, 2004

[14] *NP-4, 100-Gigabit Network Processor for Carrier Ethernet Applications, Product Brief* , EZchip Technologies, 2010

[15] *NP-3, 30-Gigabit Network Processor with Integrated Traffic Management, Product Brief,* EZchip Technologies,2010

[16] Uve Meyer-Base, Alonzo Vera, Suhasini Rao, Karl Lenk, Marios Pittichis, " FPGA wavelet processor design using language for instruction-set architecture (LISA)", Proc. SPIE Int. Soc. Opt. Eng., April 2007, Vol. 6576, pp. 65760U-1-12

[17] Grant McFarland, *Microprocessor design: a practical guide from design planning to manufacturing*, The McGraw-Hill Companies, 2006

[18] Umakanta Nanda, Kamalakanta Mahapatra, "Design of an application specific instruction set processor using LISA", International conference on Advanced Computing and Communication, 2010.

[19] Vojin Zivojnovic, Stefan Pees, Heinrich Meyr, "LISA - Machine Description Language and Generic Machine Model for HW/ SW CO-Design", White paper, 1996.

[20] Automating the Design and Implementation of Custom Processors, [online]. Available: http://www.synopsys.com [Accessed 10 May 2011]

[21] Pong P. Chu, *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*, John Wiley & Sons, 2008

[22] Xilinx University Program XUPV5-LX110T Development System, [online]. Available: http://www.xilinx.com [Accessed 10 May 2011]

[23] Hennessy, John L., and Patterson, David A., *Computer Architecture: A Quantitative Approach 2nd Edition*, Morgan Kaufmann Publishers, 1996

[24] Roger Luis Uy, Jonathan Lee, Jonathan Ray Roque, " DARC: DLX Architecture Simulator", Proceedings of the 4th Philippine Computing Science Congress, 2004

[25] Andreas Moestedt, Peter Sjödin, Torsten Köhler, "Header Processing Requirements and Implementation Complexity for IPv4 Routers", White paper, HP Laboratories Bristol, September, 1998

[26] *Internet Protocol, Version 6 (IPv6) Specification*, IETF Standard RFC2460. Available: http://www.ietf.org/rfc/rfc2460.txt

[27] *LISA Language Reference Manual, Product Version V2009.1.1, CoWare,* CoWare Processor Designer Product Family, 2009

[28] Niti Madan, "Asynchronous micro engines for network processing", Master Thesis, School of Computing, University of Utah, 2006

Danijela Jakimovska obtained bachelor and master degree at the Faculty of Electrical Engineering and Information Technologies, Univ. "Ss. Cyril and Methodius", Skopje, R. Macedonia, in 2008, 2010, respectively. Her major fields of studies include computer engineering, and information technologies

She currently works as teaching and research assistant in the computer science department at the Faculty of Electrical Engineering and Information Technologies, Univ. "Ss. Cyril and Methodius" – Skopje, R. Macedonia. She started to work at this Faculty in 2008, as Laboratory assistant. During here work experience she has participated in some national and international projects. In 2010 she made two month scientific research in Processor architectures laboratory at École Polytechnique Fédérale de Lausanne (EPFL). She has also participated in the DAAD founded project "Embedded System Design" from 2009-2011. She has published several scientific papers as author or coauthor on national and international conferences. Currently her main areas of research include network processors, computer architectures, processor design, multi-gigabit networks etc.

Msc Jakimovska is member of IEEE Women in Engineering, IEEE Circuits and Systems Society, IEEE Computer Society and IEEE Communications Society, since 2010, member of Association of Computer Machinery since 2011, and alumnae of Board of European students of Technology Skopje, since 2009.

Aristotel Tentov obtained bachelor, master and Ph.D. degree at the Faculty of Electrical Engineering and Information Technologies, Univ. "Ss. Cyril and Methodius", Skopje, R. Macedonia, in 1983, 1989 and 1994, respectively. His major fields of studies include computer engineering, information and communication technologies, system-on-chip, computer-communication systems performance analysis and modeling, and embedded systems design.

He is currently a full professor in the computer science department at the Faculty of Electrical Engineering and Information Technologies, Univ. "Ss. Cyril and Methodius" – Skopje, R. Macedonia. He is an author/coauthor of more than 35 scientific papers on conferences, symposiums and journals, and author/coauthor on more than 40 national/international projects and technical reports. He is a member of the Program Committees on more than 20 International conferences. His main areas of research include: Computer Architectures; Processor Architectures; Wired, wireless, and mobile networking; Mission critical systems and networks; Avionics; Multiprocessor and Multi-core Systems; Embedded systems; High Performance Computing; System-on-chip; RFID devices and environments; and Process Control;

Dr. Tentov is a member of IEEE since 1988 and of ACM since 2001. He is a member of IEEE Technical Committees on: Computer Communications, Distributed Processing, Real-Time Systems and Simulation.