Entregable E1.1: Documento del diseño inicial del procesador y de las interfaces entre los diferentes componentes (jerarquía de memoria, core y aceleradores)

Información del Documento

Código del proyecto	001-P-001723
Página web del proyecto	https://drac.bsc.es/es
Plazo contractual	01/06/2019 - 31/05/2022
Nivel de difusión	Interno
Naturaleza	Entregable
Autor(es)	César Hernández, Abraham Ruiz, Carlos Rojas, Alberto González, Neiel Leyva, Joan Marimon, Xavier Carril, Vatistas Kostalabros, Oscar Palomar, Santiago Marco, Osman Unsal, Miquel Moretó, Adrián Cristal
Contribuyente(s)	BSC, UAB, UB, UPC y URV
Revisor(es)	Nehir Sonmez (BSC) y Antonio Espinosa (UAB)
Palabras clave	Lagarto Ka, procesador fuera de orden, jerarquía de memoria, interfaces con los aceleradores

El proyecto DRAC con número de expediente 001-P-001723 ha sido cofinanciado en un 50% con 2.000.000€ por el Fondo Europeo de Desarrollo Regional de la Unión Europea en el marco del Programa Operativo FEDER de Cataluña 2014-2020, con el soporte de la Generalitat de Cataluña.

Generalitat de Catalunya
Departament d'Empresa i Coneixement
Secretaria d'Universitats i Recerca

Unió Europea
Fons Europeu
de Desenvolupament Regional

Control de cambios

Versión	Fecha	Autor	Comentario
01	15/09/2020	César Hernández	Descripción del diseño de los elementos del procesador Lagarto Ka y de sus interfaces de comunicación.
02	30/09/2020	César Hernández	Incorporación de los cambios sugeridos por los revisores internos. Entregable enviado.
03	15/11/2020	César Hernández	Revisión final y cierre del documento

Content

Table Index

Figure Index

1. Lagarto Ka: Out-of-order General Purpose Processor

The Lagarto Ka out-of-order general purpose core designed in the DRAC project is planned to be a 2-way 64-bit out-of-order (OoO) superscalar core that implements the RISC-V instruction set architecture. This core is coupled to different accelerators specialized in emerging applications: a vector processing unit, a bioinformatics accelerator, a post-quantum cryptographic accelerator and an autonomous navigation accelerator. Section 1 describes the architecture of the out-or-order design, while Section 2 describes the interface with the memory hierarchy. Section 3 describes the three envisioned ways to incorporate accelerators to the out-of-order core. Finally, Sections 4-6 describe the different interfaces with the envisioned accelerators in the DRAC project. Figure 1 shows an example of the configuration with the accelerators with the Lagarto Ka out-of-order core.

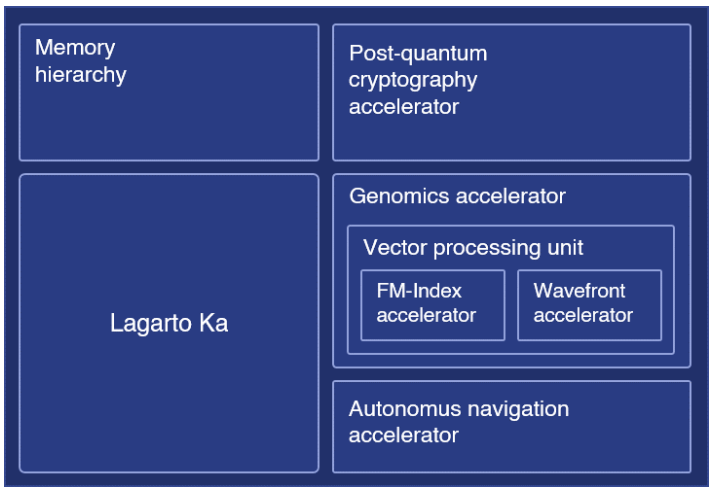

Figure 1. Lagarto Ka out-of-order core insights.

The Lagarto Ka microarchitecture is shaped by two main blocks: a sequential front-end, and an out-of-order back-end. On the front-end, the core fetches and issues two instructions each clock cycle, and is able to execute speculative datapaths, resolving instruction predictions by including a branch predictor coupled with a simplified recovery mechanism to handle mispredictions.

The instructions dispatched to the back-end of the Lagarto Ka are stored into different instruction queues, which are able to host up to 32 instructions. The design of the queues is focused on reducing energy consumption. In the first version of the design, the core is configured with a 5-instruction issue width, matching the instruction queues included; in consequence, this parameter can change to provide support for other accelerators.

The instruction execution is performed by different functional units compounded with a bypassing logic technique to effectively broadcast the source operands for dependent instructions. To preserve program order among the instructions in-flight and guarantee core recovery to a previous state, a 64-entry fully distributed reorder buffer is included. Finally, a group commitment mechanism is required into the back-end, looking for restoring the program order.

Table 1. Lagarto Ka out-of-order core parameters.

Feature	Parameter
L1 instruction cache	16 KB
L1 data cache	32 KB
L2 cache size	256 KB
Fetch-width	2 instructions
Branch predictor	128 entries
Issue-width	5 instructions
Register File	128 registers
Integer ALU latency	1 clock cycle
ROB entries	128
Recovery pages	1

Table 1 shows a more detailed specification of the parameters that are initially considered for the design of the core. Additionally, Figure 2 shows a general block diagram of the microarchitecture to implement.

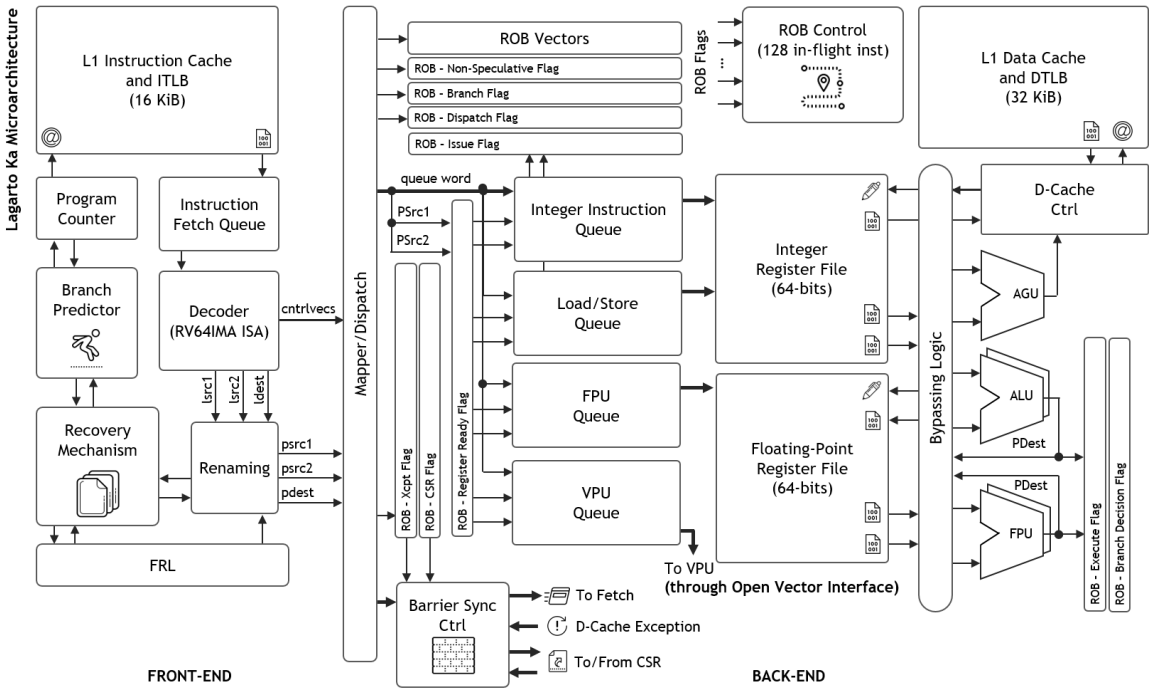

Figure 2. General overview of the Lagarto Ka out-of-order microarchitecture.

In order to achieve a proper program execution, the microarchitecture for both the core and the accelerators follows the specifications of the different RISC-V ISA extensions used. Table 2 describes the general features of the core and each accelerator, as well as the RISC-V extensions considered for its design.

Table 2. Lagarto Ka and accelerators features.

Architecture specification	**Version**	**Features**
Lagarto Ka out-of-order	RISC-V I, M, A, H extensions	2-way 64-bit out-of-order architecture
Post-quantum cryptography accelerator	RISC-V Custom extension	Loosely and tightly coupled accelerators. ISA custom instructions, virtualization with device driver operated by OS kernel.
Autonomous navigation accelerator	RISC-V Custom extension	Loosely coupled accelerator. Systolic array with approximate compute ALUs.
Genomics accelerator: Vector Processing Unit	RISC-V Vector (V) extension version 0.7.1	4-lane 512-bit integer and floating-point vector architecture; FM-Index and WFA vector support
Genomics accelerator: FM-Index accelerator	RISC-V Custom extension	Context virtualization support
Genomics accelerator: Wavefront accelerator	RISC-V Custom extension	Loosely Coupled Accelerator Multilane parallel execution along with on-chip memory structures

1.1. Hypervisor support (RISC-V H extension)

The Hypervisor Extension aims to improve virtualization performance, mainly by reducing the frequency of traps that need to be handled by the Host OS. The approach the RISC-V specification defines is to virtualize the supervisor mode (S mode), changing the existing supervisor mode into an Hypervisor-extended supervisor mode (HS mode) and adding both virtual user mode (US mode) and virtual supervisor mode (VS mode). Additionally, the address translation mechanism is augmented with a second stage. This new setup virtualizes the memory and the memory mapped I/O devices for the guest OS. This extension is being developed as part of WP2.

1.1.1. Trap Handling

RISC-V features a trap delegation mechanism that allows different privilege levels to handle interrupts and exceptions instead of the M-mode. This is done by selecting which traps are delegated to a lower privilege level, going from M to HS to VS to VU if user interrupts are enabled.

1.1.2. Two-Stage Address Translation

When virtualization is enabled, all memory accesses go through two stages. In the first one, the virtual address is translated into a guest physical address. This stage is known as VS-stage. Afterwards, this address is translated again to a supervisor physical address. This is known as the G-stage. In this stage, all accesses are considered U-mode accesses, even those performed on VS-mode data structures; a guest page-fault must be handled by either M or HS and cannot be relegated further.

2. Memory Hierarchy Description of the Out-of-Order Processor

The memory system is in charge of supplying a constant stream of instructions to the pipeline, as well as managing data memory accesses (load/store), both performed based on the virtual address previously computed. For this, a memory hierarchy has to be included, designed to be able to support several requests; these requests could come from the Lagarto Ka out-of-order core, along with the accelerators.

2.1. On-chip memory hierarchy

The memory hierarchy for the Lagarto Ka is composed of a 4-way 16KB instruction cache and a 4-way 16KB data cache in level 1, both coupled to an 8-way and 64KB shared cache in a superior level (LLC).

The processor chip accesses main memory with several alternative interfaces for memory on the same PC board as the processor chip, or remote DDR memory on an FPGA board through a custom link. The different interfaces are explained below.

Table 3. Memory hierarchy features.

Instruction Cache	**Data Cache**
• 64 sets, 4-way • 16 instructions per set (64B cache lines) • Total size 16KB • 8-entry TLB	• 4-way 16KB • Non-blocking, with configurable # of MSHR • 64B cache lines • 8-entry TLB
LLC	
• 128 sets, 8 ways • 16 words (32 bits) per set (64KB) • Total size 64 KB • Accepts up to 2 concurrent requests	
Other features	
First level caches are Virtually Indexed Physically Tagged (VIPT). Caches are inclusive. Random replacement policy using lower bits. Page size: 1 GB Coherence protocol: MESI (a directory is used to maintain coherence, this is located in the LLC).	

Table 4. Memory hierarchy latencies.

Status	**Latency**
Hit IL1	3 cycles
Hit DL1	3 cycles
Miss IL2	20 cycles
Miss DL2	22 cycles

Finally, the communication between the core and the level 1 instruction and data caches is defined as shown in the following figures.

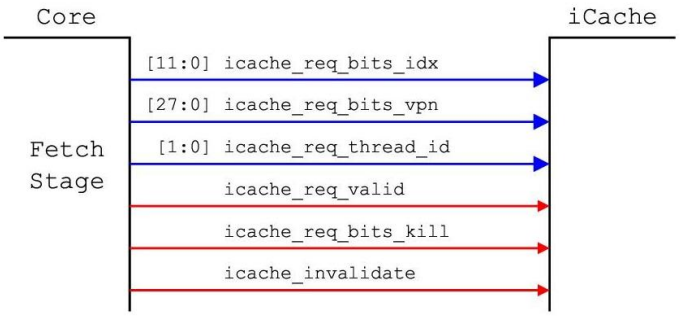

Figure 3. General overview of the control signals and data buses from the core to the Instruction Cache.

Figure 3 shows the data flow from the Lagarto Ka (core) to the Instruction Cache (iCache), while the data flow in the opposite direction, from the Instruction Cache to the Lagarto Ka, is shown in Figure 4. The communication is performed through data buses (in blue) and control signals (in red), keeping the information clearly separated. Table 5 describes in detail these interfaces.

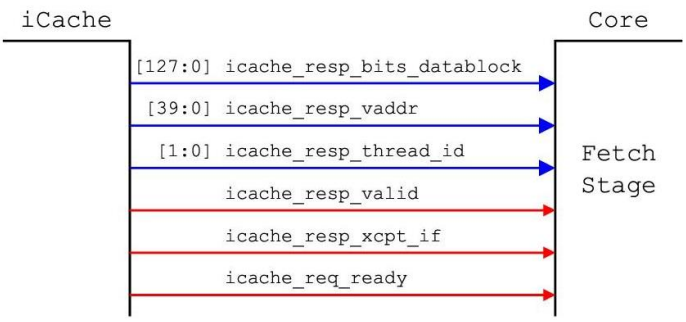

Figure 4. General overview of the control signals and data buses from the Instruction Cache to the core.

Table 5. Instruction cache memory signal interface definition.

Size	Signal	Direction	Description
12	icache_req_bits_idx	CORE->IMEM	Index of the requested address
28	icache_req_bits_vpn	CORE->IMEM	VPN of the requested address
2	icache_req_bits_thread_id	CORE->IMEM	Thread identifier
1	icache_req_valid	CORE->IMEM	A valid request
1	icache_req_bits_kill	CORE->IMEM	Kill request in flight
1	icache_invalidate	CORE->IMEM	Invalidate active cache line
128	icache_resp_bits_datablock	IMEM->CORE	Cache line delivery
40	icache_resp_vaddr	IMEM->CORE	Requested address
2	icache_resp_thread_id	IMEM->CORE	Applicant thread
1	icache_resp_valid	IMEM->CORE	A valid delivery or exception
1	icache_resp_xcpt_if	IMEM->CORE	An exception
1	icache_req_ready	IMEM->CORE	Cache ready to accept requests

Figure 5 shows the data buses that the Lagarto Ka uses to send data to the Data Cache (dCache). These buses not only contain the data computed by the core to be stored into the cache, but also include the memory operation type to be performed (load, store or atomic), the memory address, and the instruction tag identifier.

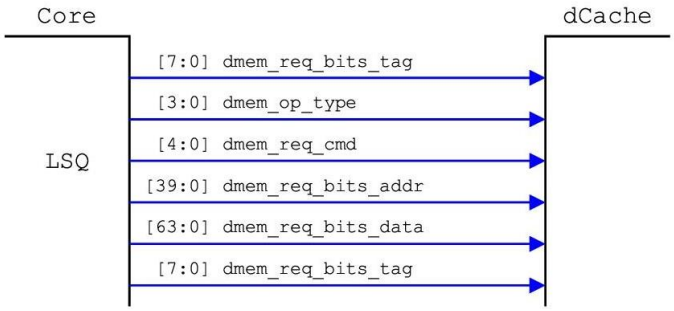

Figure 5. General overview of the data buses from the core to the Data Cache.

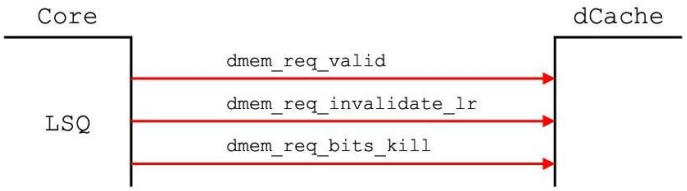

Figure 6. General overview of the control signals from the core to the Data Cache.

Figure 6 shows the control signals tied to the data buses that send information to the Data Cache. These signals indicate to the Data Cache when a request is valid, or whether it has to be discarded.

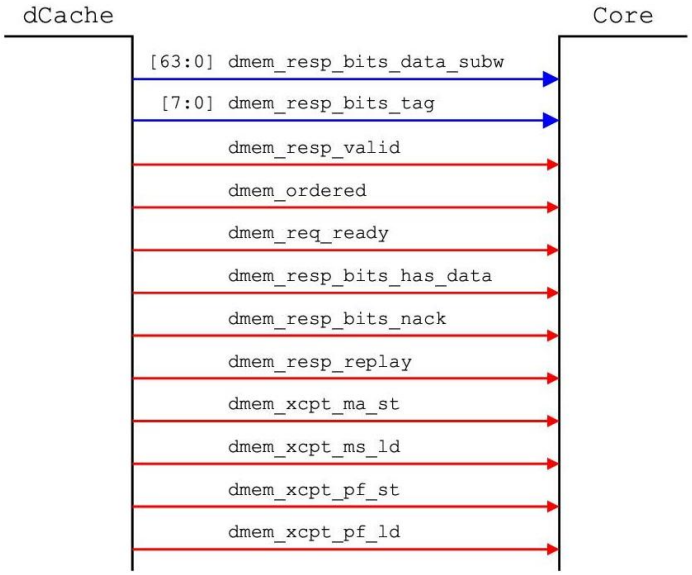

Figure 7. General overview of the control signals and data buses from the data cache to the core.

Finally, Figure 7 shows the data buses and the control signals sent to the core from the Data Cache. This information is received by the Load/Store Queue to determine whether it is possible to send a new request, or it is necessary to wait for the memory to be ready; similarly, the Load/Store Queue determines when a request must be resent or discarded, or if it has produced an exception. Table 6 describes in detail the signals of this interface.

Table 6. Data Cache memory signal interface definition.

Size	Signal	Direction	Description
1	dmem_ordered_i	DMEM -> ROB	Indicates that the memory is busy
1	dmem_req_ready_i	DMEM -> ROB	Set to '1' when the dcache is able to accept a new request.
64	dmem_resp_bits_data_subw_i	DMEM -> ROB	Data input bus
1	dmem_resp_bits_has_data_i	DMEM -> ROB	When is set to "1" together with a dmem_resp_valid_i indicates valid data in the dmem_resp_bits_data_subw_i signal.
1	dmem_resp_bits_nack_i	CORE -> DMEM	If true (1), indicates that the memory request must be issued again from the LSQ to the D-Cache
8	dmem_resp_bits_tag_i	CORE -> DMEM	The tag is used as an ID of the instruction.
1	dmem_resp_valid_i	CORE -> DMEM	Response valid from the dcache. This can be signaled 2 or more cycles after a request was done but not before.
1	dmem_xcpt_ma_st_i	CORE -> DMEM	Misaligned store exception
1	dmem_xcpt_ma_ld_i	CORE -> DMEM	Misaligned load exception
1	dmem_xcpt_pf_st_i	CORE -> DMEM	Store page fault exception
1	dmem_xcpt_pf_ld_i	CORE -> DMEM	Load page fault exception
1	dmem_req_valid_o	CORE -> DMEM	Request valid to the dcache
4	dmem_op_type_o	CORE -> DMEM	Data bit length
5	dmem_req_cmd_o	CORE -> DMEM	Contains the memory command, such as 00100 for AMOSWAP, 01011 for AMOAND, and so on.
64	dmem_req_bits_data_o	CORE -> DMEM	Store/amo data output
40	dmem_req_bits_addr_o	CORE -> DMEM	Memory request address
8	dmem_req_bits_tag_o	CORE -> DMEM	7-bit rob entry used as ID to identify the owner of the request

1	dmem_req_invalidate_lr_o	CORE -> DMEM	Set when exception
1	dmem_req_bits_kill_o	CORE -> DMEM	Kill in-flight instructions

2.2. Main memory access on the board

The main memory is considered as a 1 GB memory, in one or several modules external to the chip. There are several types of memory, from high bandwidth, like DDR, to moderate bandwidth, such as SDRAM or HyperRAM.

Memories such DDR3 that operate at high frequencies require an appropriate physical interface PHY to ensure data is transmitted correctly. Therefore, memory modules directly connected to the processor chip are considered to be in the SDRAM and HyperRAM technologies.

2.2.1. SDRAM

The purpose of the SDRAM controller is to provide an alternative way to access main memory without requiring an auxiliary FPGA board. The controller is made for SDR (Single Data Rate) DRAM, which operates at lower frequencies (typically up to 166 MHz) and does not require any specific physical interface.

Bandwidth depends on the operating frequency of the SDRAM and the width of the data bus. With an operating frequency of 166 MHz and a data bus of 16 bits, the bandwidth will be 332 MBps. The bandwidth scales linearly with the data bus (increasing the data bus to 32 bits, we obtain a bandwidth of 664 MBps). Capacity depends on the number of SDRAM chips used, each chip has up to 64 MB. Then, for example, 256 MB of memory can be obtained by using 4 SDRAM chips.

The SDRAM controller acts as an interface between the SoC and the external memory. For one side, the controller communicates with the main processor using the AXI protocol and, on the other side, it contains the logic required to manage the accesses to the SDRAM, generate the appropriate commands and control the refresh sequence. Additionally, the SDRAM controller contains an asynchronous FIFO used to synchronize the data between different clock domains, such as from the main clock to the SDRAM clock.

2.2.2. HyperRAM

The Cypress HyperRAM Self-Refresh Dynamic Random Access Memory (DRAM) is one of the few memories on the market that has a reduced number of bus signals. While DDR3 memories have 240 pins, the HyperRAM only has 12 pins. Also, the DDR3 memory controller IP fills a significant space on the floor plan if we want to introduce it on the ASIC, and it is protected from modifications with a costly license. Similar to the SDRAM controller, one of the main purposes of the HyperRAM controller is to provide an alternative way to access the main memory without requiring an auxiliary FPGA board.

With the memory of 1.8V, we can accomplish a maximum frequency of 166MHz, achieving then, a maximum bandwidth of 333MBps within 8-bit data bus. About the size of each module, there are two types, the 64Mb (8MB) one and the 128Mb (16MB) one. It should be noted that these memory modules can be merged together into a larger block. For example, we can achieve 512Mb (64MB) grouping four modules of 128Mb each.

This memory has the Double-Data Rate (DDR) characteristic. Additionally, we can do sequential burst transactions, meaning that for a given address, a consecutive number of bytes (more than four bytes) are read or written in a single transaction.

Like the SDRAM controller, AXI4 is the communication protocol to interface the core and the memory modules. Below in Figure 8, we can observe a brief outline of the logic used.

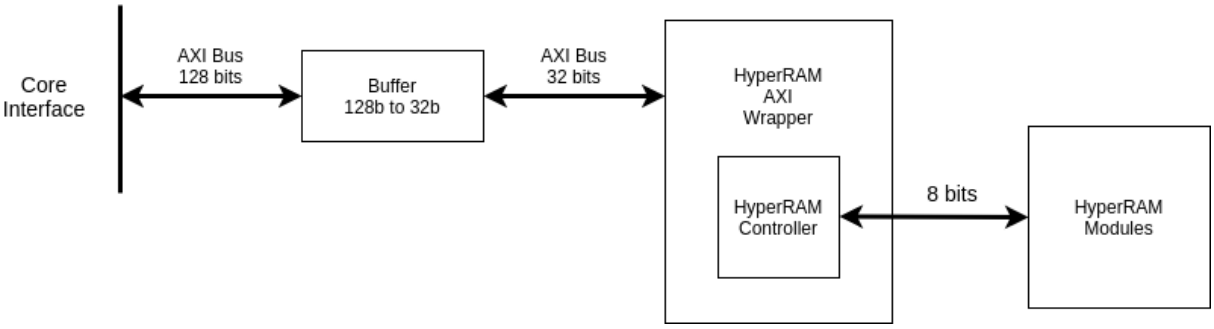

Figure 8. Block diagram of core to/from HyperRAM modules data path.

As the size of the bus of the core interface is about 128 bits, and the size of the bus of the HyperRAM AXI wrapper is 32 bits, a buffer was inserted between them in order to solve the change of bus size. Also, the HyperRAM AXI Wrapper contains FIFOs and a Finite State Machine (FSM) to arbitrate the sequence of commands to send to the HyperRAM Controller.

2.3. Main memory access on accessory FPGA board

High bandwidth memory standards such as DDR3 and DDR4 need a physical interface (PHY) on the chip, which is very expensive to acquire and very costly to develop internally. However, commercial FPGA evaluation boards (e.g. [2] or [3]) contain DDR memory on the board, connected to the FPGA device that integrates the necessary PHY and controller. In order to access DDR memory on the FPGA board, two interfaces between the out-of-order processor chip and the FPGA are defined (see Figure 9 below):

- Packetizer: a low bandwidth interface [5].
- SerDes: high bandwidth serial interface (HBWIF).

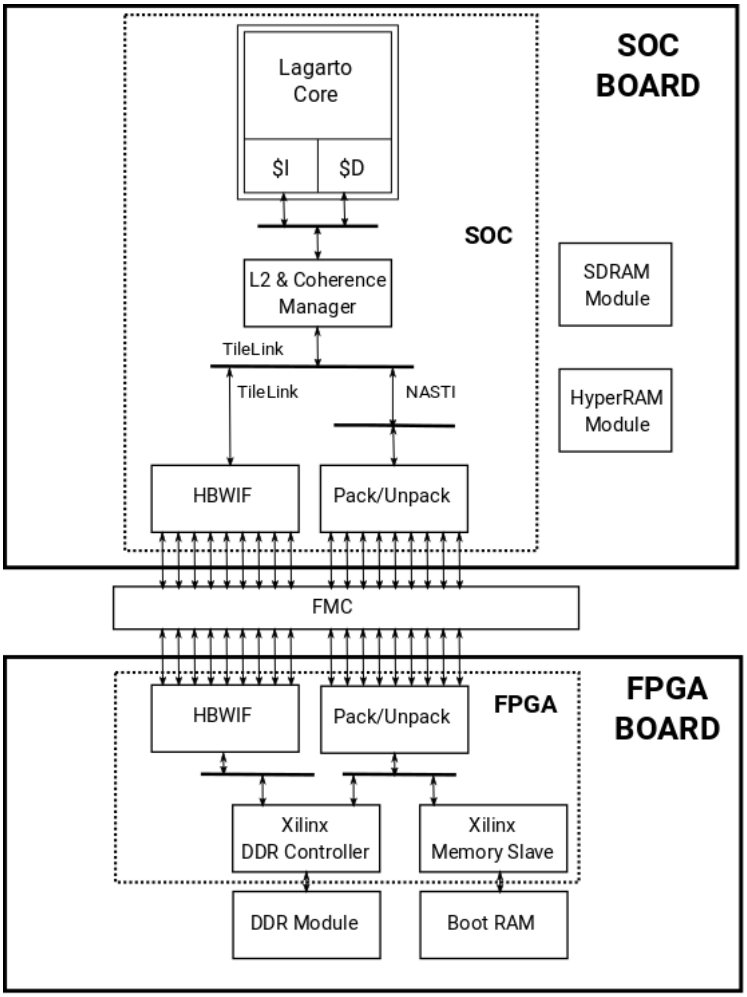

Figure 9. Block diagram of the connection to main memory on FPGA using the two interfaces: serders on the left of the FMC, packetizer on the right handside. Additional on-board memory SDRAM and HyperRAM are considered (interfaces not included in the schema) [5].

2.3.1. Packetizer interface [5]

The lack of a physical interface for a DDR3 memory controller can be overcome by the design of a custom interface to communicate with memory using the physical DDR3 memory from an external FPGA board. Memory access is then split into two parts: one on the FPGA that contains the memory controllers to access the main memory, and a second one on the chip containing the core and rest of the uncore system, including L1 and L2 cache memories.

Both parts are connected with an FPGA Mezzanine Card (FMC) connector. Based on our empirical evaluation, we achieved a steady transfer rate operating at 50 MHz using an FMC cable with a 128-bit bus split into 4 transactions of 32 bits each.

2.3.2. SerDes interface

Another method to connect the FPGA containing DDR controllers and PHY is using high speed serial interfaces, which can achieve data rates close to 10 Gbps per channel. This interface needs two components integrated in the processor chip:

- PMA (Physical Media Access): PHY interface capable of transmitting in the Gbps range. This is an analog high frequency component that uses differential signaling: two wires for TX and two wires for RX in each channel.
- PCS (Physical Coding Sublayer): Digital layer interfacing between the PHY and the L2 memory.

On the FPGA side, a compatible PHY I/O (GTX or GTY depending on the FPGA device on the board) communicate with the on-chip PHY through an FMC connector, and are linked to the digital control that manages the DDR controller and PHY (see Figure 9). Several lanes can be considered to increase data throughput.

PMA architecture and specification

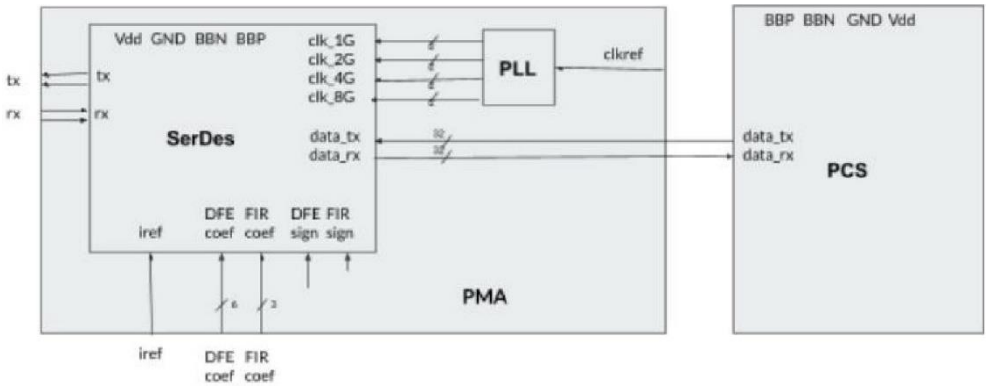

Figure 10. Connection between PMA and PCS.

The PMA is a serializer-deserializer designed to work at 8 Gbps with a parallel bit width of 32. It contains an 8 GHz PLL.

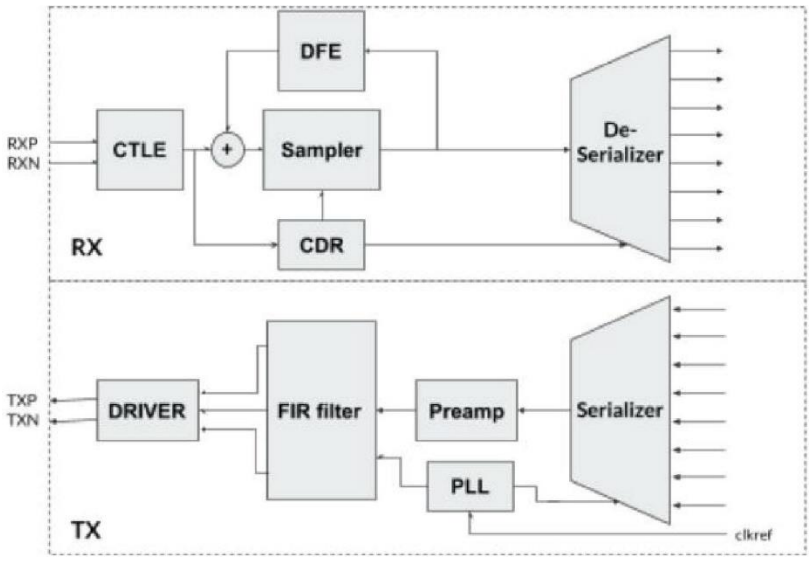

Figure 11. Architecture of the PMA.

PCS architecture

The architecture of a single PCS lane is depicted in the image below.

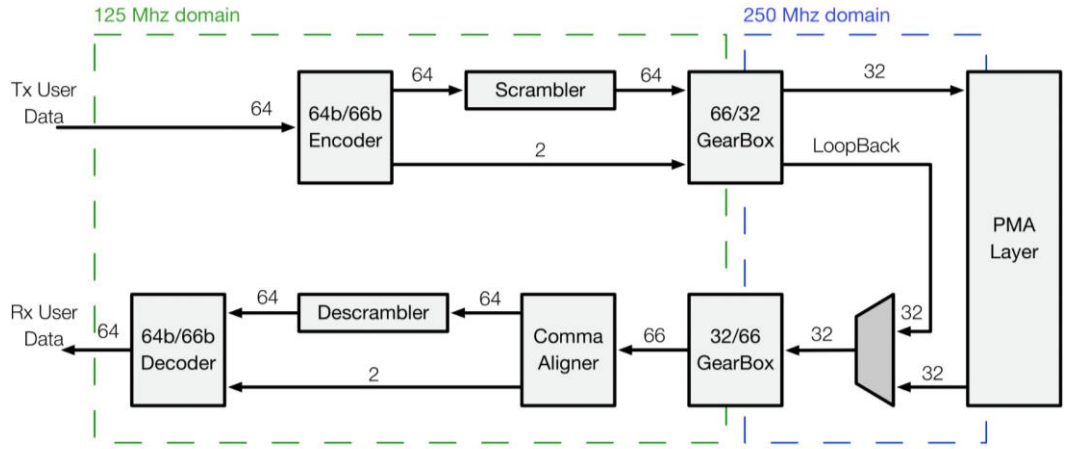

Figure 12. Architecture of the PCS.

Tx User Data (TxUD) and Rx User Data (RxUD) are Axi-Stream busses, composed of the following signals:

- **tdata [63:0]:** payload;
- **tvalid:** triggers valid data;
- **tready:** only on the TX side, determines if the Slave is able to receive data in a given clock cycle;
- **tlast:** last element of a packet;
- **tkeep, tuser, tid:** not mandatory;

Incoming data in the TxUD bus are firstly sent to the 64b/66b encoder, that adds 2-bit of encoding information to the 64-bit payload (e.g., setting them to 10 in case of idle packets, and to 01 in case of data packets). According to the 64b/66b specs, the 64-bit related to TxUD data are then scrambled, allowing to ensure an adequate number of 0-1 (or 1-0) transitions of the serial data, thus preventing the CDR losses in the receiver side.
Scrambled data, together with the unscrambled 2-bit sync signal created by the encoder, are sent to the 66/32 gearbox, whose purpose is to convert the 66-bit bus in the format expected by the PMA layer (i.e., 32-bit).

On the receiver side, the 32-bit bus from the PMA layer is converted in a 66-bit format by the 32/66 gearbox, and aligned by the comma aligner. The comma aligner checks that, for a given time of iterations (e.g., for 64 66-bit packets), the two MSBs of the 66-bit signal contain encoded data (e.g., bit 65 and 66 must be equal to 10 for 64 incoming packets). Otherwise, the aligner sends a shift_req to the 32/66 gearbox, which shifts the incoming data by 1 bit. The gearbox should have an internal shift register of 256 bits, to allow selecting the whole word of 66 bits without overflows in the shift mechanism. Aligned data are descrambled and decoded to create the RxUD 64-bit bus.

3. Integrating Multiple Accelerators in an Out-of-Order Processor

The process of integrating an accelerator with the processor is as crucial as the actual design of the accelerator. Certain design choices have to be made early on, in the design phase in order to interface the hardware accelerator to the core in an efficient, yet resource savvy manner. Following are the two main ways that an accelerator can be coupled with the main processing element.

3.1. Tightly Coupled Accelerators (TCA) to the Processor's Pipeline

The first case of the interfaces is tightly-coupled [5] to the core's pipeline accelerator. This scenario adds one or more computational units (CUs) to the execution stage of the pipeline. The CUs share key-resources with the core (i.e. register file and MMU) and finish their computation in a few clock cycles. They rarely have internal memory, since they use the L1 cache of the core, but do use configuration registers to customize their computation. The resulting wider pipeline is able to execute the new computations performed by the CUs, provided that they are exposed to the Instruction Set Architecture (ISA) as new instructions. This case is depicted in Figure 13.

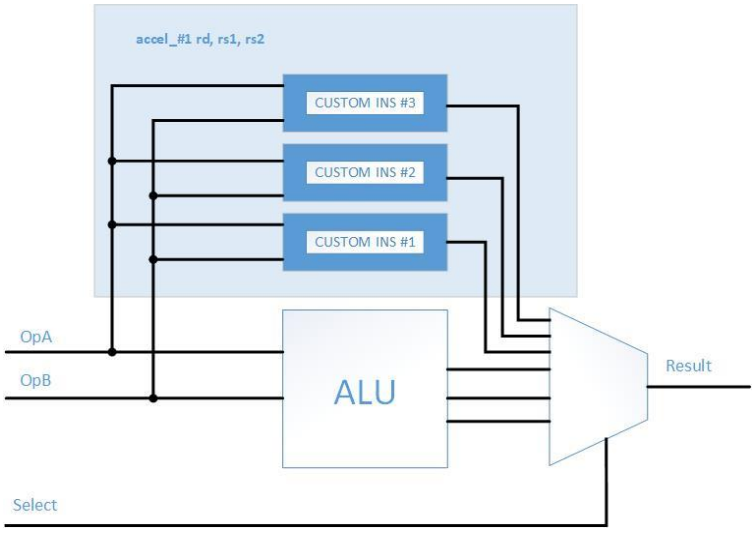

Figure 13. Tightly coupled accelerators (TCA) to the processor's pipeline.

The interface with the core is the ISA extension itself, since the hardware addition needs no more than a bigger multiplexer for the output of the execution stage and a modification at the decoding stage reflecting the addition of the new instructions. Such alterations of the ISA are well documented for the RISC-V ISA case. Additionally, separate ISA extensions provided by the RISC-V community (i.e vector and bit manipulation extensions) can be used in this acceleration scenario as long as there is an underlying hardware module performing the computations. Special care has to be taken at the software stack, since the ISA extension requires modifications at the compiler. Moreover, timing closure of the accelerator has to be taken care of, so as not to interfere with the critical path of the core, thus affecting the operating frequency. The latter issue also puts constraints on the area budget of the accelerator.

3.2. Loosely Coupled Accelerators (LCA) from the Processor's Pipeline

The case of a LCA best describes a hardware module that performs coarse-grained computations, thus consuming a bigger area budget than the TCA. Their computation takes at least tens of clock cycles to finish. The LCA is usually equipped with its own internal private memory (i.e scratchpad) in order to hold data values and store intermediate results. Typically, the accelerator is also coupled with either the Last Level Cache (LLC) of the core or the DRAM itself in order to load and store the results it computes, also making them available to the core. That interface can either be coherent or not. This communication is often handled by a Direct Memory Access (DMA) mechanism, thus preventing the stall of the computational core as long as the hardware accelerator performs the memory access. The latter is taken care of by the DMA controller (DMAC) that resides in the hardware accelerator and is part of its interface with the core. A generic view of a LCA can be seen in Figure 14.

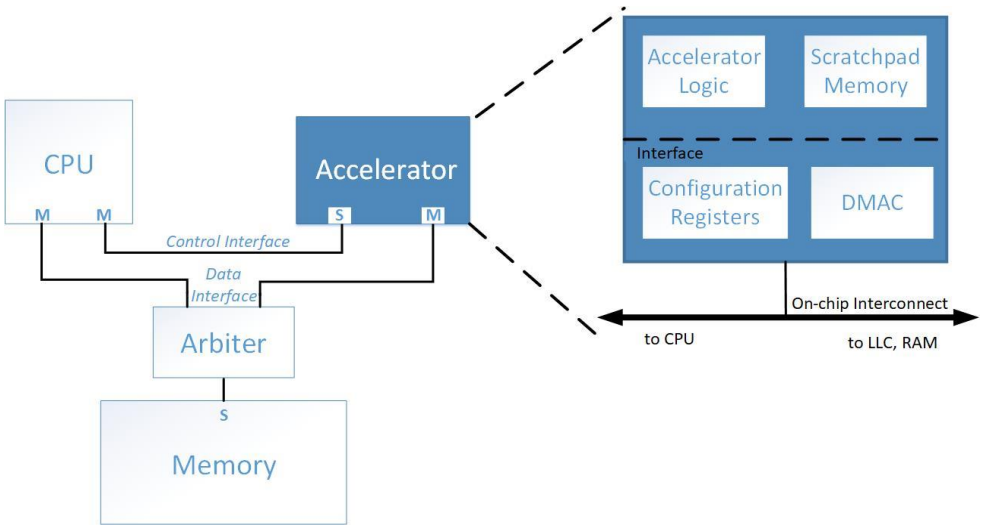

Figure 14. Loosely Coupled Accelerators (LCA) from the processor's pipeline.

Unlike the TCA, a LCA has to be controlled from the kernel space by a device driver. The user application issues a system call that invokes this driver and takes care of moving data to the accelerator and configuring its computation. When the computation finishes, the accelerator issues an interrupt, signaling the end of the respective computation. That communication can be based on a polled or interrupt-based interface from the core's perspective. An interrupt service routine (ISR) might induce a timing overhead, but it lets the core perform separate computations or even shut down, while the accelerator performs its own computations. Polling offers the quickest response and less timing overhead, but keeps the core busy waiting for the results.

Regarding the programming model of the accelerator in the LCA scenario, there are several ways it can be constructed. In a simple embedded processor scenario there might be no OS service and the accelerator is used in bare metal with ad-hoc custom protocols. In the case we have an OS enabled system, the accelerator can either be a memory mapped device controlled by the user space or the kernel can be issuing commands to the accelerator via virtualization of the latter and the use of a device driver.

4. Genomics Accelerator

To search sequence databases that may contain billions of sequences, different genomic algorithms are implemented, these algorithms become computationally expensive. Consequently, in this design, we focused on accelerating the most fundamental genomics algorithms by offloading the computationally repeated portion of the algorithms to custom hardware instructions. These simple modifications accelerate the algorithm runtime compared to the pure software implementation. Therefore, further design of hardware offers a promising direction to seeking runtime improvement of genomics database searching.

4.1. Interface with the Genomics Accelerator

We will follow the Open Vector Interface (OVI) defined in the European Processor Initiative (EPI) to connect the out-of-order core with the VPU. This interface is open [7] and has the following input and output signals:

Table 7. Vector Processing Unit Interface.

Size	Signal	Direction	Description
1	issue_valid_i	CORE -> VPU	Indicates valid data on issue group
32	issue_instr_i	CORE -> VPU	Instruction fetched by the core
64	issue_data_i	CORE -> VPU	Scalar value from the core
4	issue_sb_id_i	CORE -> VPU	Scoreboard ID used to identify the instruction while on the fly
40	issue_csr_i	CORE -> VPU	Vector CSR to be used by the VPU
1	dispatch_nxt_sen_i	CORE -> VPU	An instruction becomes next senior
1	dispatch_kill_i	CORE -> VPU	An instruction must be killed
4	dispatch_sb_id_i	CORE -> VPU	ID of the instruction in reference by the dispatch group
1	memop_sync_end_i	CORE -> VPU	All data has been transmitted on current memory operation
4	memop_sb_id_i	CORE -> VPU	ID of the instruction in reference by the memop group
15	memop_vstart_vlfof_i	CORE -> VPU	vstart/vl value after memory operation. It can contain only on f-o-f loads

1	load_valid_i	CORE -> VPU	Whether there is a valid data on the bus
1	load_mask_valid_i	CORE -> VPU	Whether the mask is valid
512	load_data_i	CORE -> VPU	Data fetched from memory
33	load_seq_id_i	CORE -> VPU	Sequence ID for a given chunk of data
64	load_mask_i	CORE -> VPU	Mask assigned to the load operation
1	store_credit_i	CORE -> VPU	Core returns a store credit
1	mask_idx_credit_i	CORE -> VPU	Core returns a mask credit
1	core_stall_o	VPU -> CORE	Request core stall from the VPU
1	issue_credit_o	VPU -> CORE	VPU returns a credit to the core
1	completed_vxsat_o	VPU -> CORE	Fixed-point accrued exception flags
1	completed_valid_o	VPU -> CORE	Valid data on completed group
1	completed_illegal_o	VPU -> CORE	Illegal instruction
4	completed_sb_id_o	VPU -> CORE	ID of the instruction in reference by the completed group
5	completed_fflags_o	VPU -> CORE	Floating-point accrued exception flags
64	completed_dst_reg_o	VPU -> CORE	Result scalar value
14	completed_vstart_o	VPU -> CORE	vstart value in case of traps or retry
1	memop_sync_start_o	VPU -> CORE	VPU is ready to execute a memory op
1	store_valid_o	VPU -> CORE	Whether the store data is valid
512	store_data_o	VPU -> CORE	Data to store on memory
1	mask_idx_valid_o	VPU -> CORE	Whether item contains valid data

1	mask_idx_last_idx_o	VPU -> CORE	Whether the mask/index being transmitted is the last one
65	mask_idx_item_o	VPU -> CORE	Mask/Index shared bus for indexed memory operations

In addition, the idea of including support to the VPU to perform direct memory accesses without requiring the intervention of the out-of-order processor is explored; this implies that there is no need to add an extra interface between the VPU and the memory hierarchy.

Independent memory accesses from accelerators without going through the out-of-order core relieve the core's task load and reduce the memory penalties that hinder the processor's performance.

With this approach, the accelerators are able to share a specified cache level, where each handles the data through scratchpad memory, a buffer, or a local cache, depending on the necessities for each accelerator design.

A priority manager handles the order of the requesters dynamically to have a fair distribution of the total bandwidth for every accelerator. If an accelerator modifies its priority level, this does not imply any restriction to keep performing requests at any time. If a requester with a high priority is not making a petition, others can use the unused bandwidth.

When a memory petition from any of the requesters causes an exception, this exception is bypassed to the main processor to perform the corresponding service routine.

Currently, this strategy is under evaluation since it is necessary to perform a deeper analysis to determine the cache level directly connected to the accelerators, depending on the data rate consumption. However, for the first implementation, the out-of-order processor will perform the memory accesses to the L1 caches, including those requested by the accelerators.

4.2. Wavefront accelerator and FM-Index custom vector instructions

Bio-computation kernels such as FM-Index search and pairwise alignment with Wavefront present disjoint needs from the architectural hardware point of view. The nature of the FM-Index algorithm presents random memory access, which will constantly cause memory access delays, causing the algorithm to be memory bound. On the other hand, the Wavefront algorithm presents a compute-bound behavior requiring several parallel computations, making it affordable for vector processing. Nevertheless, both kernels are essential in genomics mapping applications and are intended to be run on this hardware.

The FM-Index accelerator module will look to apply data prefetch techniques in order to improve as much as possible the number of memory requests looking to consume the available memory bandwidth without interfering with the main core execution. Moreover, we will also use custom vector instructions to accelerate the index calculation.

For the Wavefront accelerator, an initial proposal is intended to extend the Vector Instruction ISA and the OVI to issue specific instructions that will fetch complete memory regions that contain pairs of text strings. Then, other computation instructions will use this data as operands. We will perform evaluations of the advantages of using the Vector register file or other specific memory structures as a scratch-path memory to this particular process. The evaluations could include system-level simulations and RTL descriptions or how the hardware behavior will execute the specific accelerator custom instructions. Finally, we will define the specifications of the ISA extension and the hardware microarchitecture.

5. Autonomous Navigation Accelerator

The goal is to design a particularly efficient hardware accelerator to perform object detection from camera images by trading off accuracy and power consumption. The accelerator will speed-up the operations related to the YOLO system [8], used by the Apollo AD framework [9] to handle camera-based object detection. YOLO (You Only Look Once) is an award-winning, widely-used object detection system. Its most computationally-intensive function is a Convolutional Neural Network inference algorithm. However, the accelerator is conceived not to be restricted to this particular object detection system and be usable for any application building on stochastic processes implemented with matrix multiplications, thus neither requiring a specific standard for floating point numbers (e.g. IEEE754) nor bit-level precision (e.g. some inaccuracies are acceptable).

A systolic-array-based architecture has been designed to this end, containing a 2D array of processing elements, plus several peripheral blocks, as seen in the figure below.

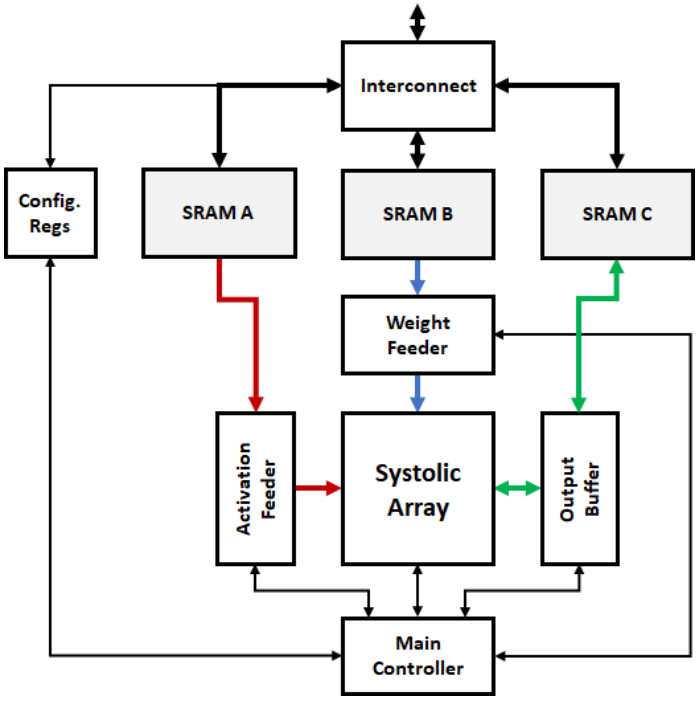

Figure 15. Block diagram of a systolic-array-based architecture.

The Systolic Array block contains the 2D array of processing elements, which perform MAC operations in parallel. The Activation and Weight Feeder modules provide the input streams to the Output Stationary (OS) systolic array. The Output Buffer block handles the extraction of results from the array, as well as the insertion of preload values to accumulate over, if needed. The Main Controller module orchestrates the operation. Three independent and double-buffered SRAM blocks store the operands to

load and the computation results. A set of configuration registers is used to control the accelerator, accessible from a unified interface.

Thanks to the novel architecture of the feeder modules, the accelerator is able to natively handle both general matrix-matrix multiplications (GEMM) and full convolutions, eliminating the need to convert the convolutional tensors to a matrix equivalent via convolutional lowering (im2col algorithm), and the software and memory overheads that come with it.

5.1. Interface with the Autonomous Navigation Accelerator

The full address space of the automotive accelerator will be accessible from the rest of the chip. This includes the internal SRAM memories that hold the tensors/matrices to operate with and the configuration registers of the accelerator. The processing core(s) will be able to configure the accelerator and monitor its status using regular Instruction Set Architecture (ISA) instructions, such as load and store operations. For optimal performance, the input and output data will be moved to and from the internal (non-coherent) SRAM memories of the accelerator via DMA, even though they are accessible by the core(s) as well.

A unified 32-bit interface with a 16-bit address will be used to access all the memory resources of the accelerator. The 2 MSBs select the resource to access: the configuration registers (0), or each of the three SRAM blocks (A, B or C; 1, 2 or 3 respectively). If the config registers are selected, the next 4 bits discriminate between different register regions.

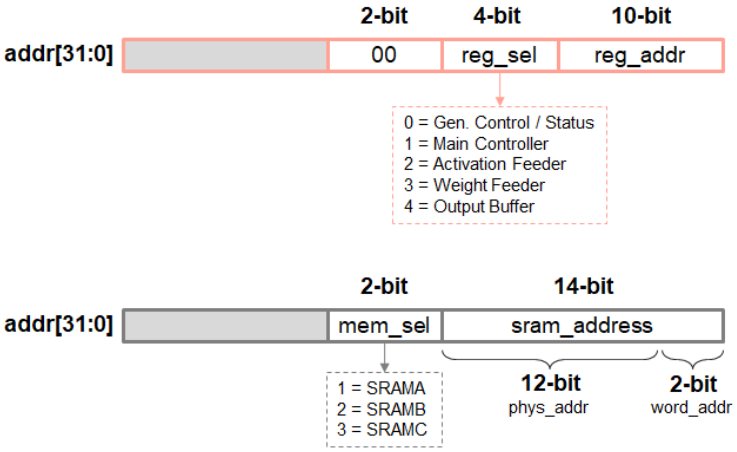

Figure 16. Memory Addressing Control Scheme.

After configuration is complete and the input operands are ready, the computation will start when the core sets the START signal (address=0x0). Upon finalization, the accelerator will set the DONE signal (address=0x1) and optionally raise an interrupt signal. Both START and DONE will be accessible as 32-bit unsigned integers, thus the operation of starting and polling the accelerator will be atomic.

Double buffering will be used at the data and configuration levels, so the core can modify the configuration values and the SRAM contents while the computation takes place without any side-effects. Therefore, the core and the DMA can fetch the data needed for the next computation while the accelerator is busy, enabling continuous operation. The new values written after a START signal is set will only be effective after the next START rising edge.

To provide a fine-grained control over the accelerator and optimize hardware resources, the configuration will be performed at a very low level: the register signals directly control hardware parameters such as the step size and limit of the tensor readout counters. A software task will be in charge of compiling the user parameters (convolution options and tensor sizes) into the appropriate values for the configuration registers. Since these values only depend on the CNN to execute, this operation can be performed at compile time and hence has zero performance cost during inference.

A detailed specification of the configuration signals, the address map and the compiling process will be provided. The user settings that completely define these signals are as follows:

- **Cw - Output Tensor Width**: size of the x (width) dimension of the resulting tensor after the convolution.
- **Ch - Output Tensor Height**: size of the y (height) dimension of the resulting tensor after the convolution.
- **Cc - Output Channels**: number of output channels.
- **Bw - Kernel Width**: x-dimension of the convolutional kernel.
- **Bh - Kernel Height**: x-dimension of the convolutional kernel.
- **Bc - Input Channels**: number of input channels.
- **Dilation:** Dilation coefficient of the dilated convolution (atrous convolution). A value of 1 implements a regular convolution.
- **Strides:** Stride coefficient of the convolution.
- **Xused - Used columns**: number of columns of the systolic array used to map the computation
- **Yused - Used rows**: number of rows of the systolic array used to map the computation
- **Preload Enable:** a value of True enables the preload of values into the array accumulators to start summing over.

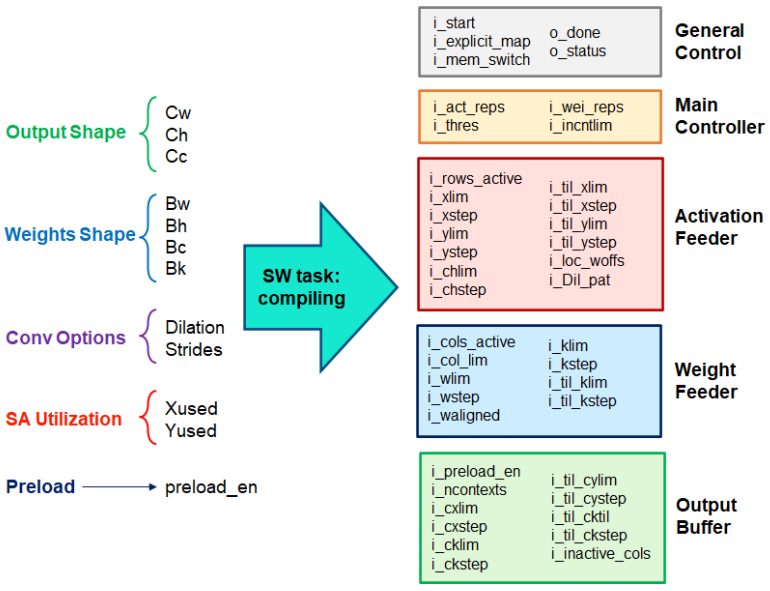

Figure 17. Accelerator configuration signals.

Direct matrix-matrix multiplications, needed to implement the Fully-Connected (FC) layers of many networks, can be implemented by generalizing the convolution operation.

6. Post-Quantum Cryptography Accelerator

The Post-Quantum (PQ) Cryptography accelerator is a co-processor designed by the WP2 team. This hardware module is intended to foster the efficient use of PQ cryptographic schemes by speeding up computational bottlenecks of the latter, as well as optimizing their memory access patterns to fit our needs.

6.1. Interface with Post-Quantum Cryptography Accelerator

There may be several Key Encapsulation Mechanism (KEM) algorithms, or functionalities (i.e ciphertex encapsulation/decapsulation, key generation, signature check, etc.) accelerated at the final SoC. Therefore, depending on the individual acceleration needs, there may be several interfaces defined for each acceleration scenario. The section below will try to depict as accurately as possible; a general interface that reflects a possible acceleration scenario, based on the up-to-date research activities of the WP2 team.

In Table 8 we summarize the main features of the accelerator interface with the core.

Table 8. Basic interface features for TCA and LCA modules.

Interface Features	**Comments**
Accelerator type	TCA & LCA
Control Interface	SCRs, Memory mapped registers (via AXI Full/Lite)
Data Interface	Register file, DMA coupled with Scratchpad
Data Coherency	Common with core, ACP if needed
Memory coupling	Common with core, Last Level Cache (LLC)
Programming model	ISA custom instructions, Virtualization with device driver operated by OS kernel

Computational Granularity: Current software profiling results have shown that the PQ KEM algorithm acceleration will benefit both from a coarse-grained acceleration mode and a fine grained one. For this reason, we will employ TCA and LCA design techniques equally.

Control Interface: Regardless of whether TCA or LCA unit, the computational module will need some configuration registers to customize and initiate its computation. For the case of the TCA, we can add custom SCR registers, while for an LCA module we can have memory mapped registers controlling the configuration and the status of the accelerator.

Data Signals: As for the actual data the TCA accelerator will simply be supplied by the register file. On the other hand, an AXI-Full interface can be utilized for the data transfer to/from the LCA accelerator.

Regarding whether we will be able to exploit any inherent parallelism of the KEM, we can add Direct Memory Access (DMA) support. Such a feature would free up the processor to perform independent processes, while the accelerator is busy loading/saving its results to the memory hierarchy. Moreover,

the memory interface can be coherent or not. In our case, no coherency support will be needed, since the tasks performed by the core while the accelerator is busy will operate on independent data. In the case that we identify the need for coherency, though that would complicate the interface, we can add an Accelerator Coherency Port (ACP) to the accelerator's side. This interface will be operated via an AXI link and connected to the DMA controller.

The connection of the LCA accelerator to the memory hierarchy will be at the Last Level Cache (LLC). Any intermediate results, not needed by the core, can be temporarily saved to the internal private scratchpad memory. The final result will be transferred to the common LLC for further continuation of the KEM mechanism. The TCA will be using the core's caches and MMU.

Programming Model: The LCA will likely be exposed to the OS via a device driver operated by the kernel space. Virtualizing the device and keeping the control of the CU at the kernel space provides us with portability across systems and an elevated level of security. The TCA will simply be exposed to the ISA via custom ISA instructions. Compiler support should also be taken care of.

Code examples: In the following code we will try to give examples of both a TCA and LCA API as seen by the developer. The TCA contains simple galois field arithmetic operations while the LCA performs a more complex gaussian elimination on a given matrix.

Example #1 Galois Field Arithmetic acceleration (TCA)

The code below shows the galois field inversion function that uses the squaring (gf_sq()) and multiplication (gf_mul()) functions. Their implementation is made in hardware as TCA.

```
gf gf_inv(gf in)
{
   gf tmp_11;
   gf tmp_1111;

   gf out = in;

   out = gf_sq(out);
   tmp_11 = gf_mul(out, in); // 11

   out = gf_sq(tmp_11);
   out = gf_sq(out);
   tmp_1111 = gf_mul(out, tmp_11); // 1111

   out = gf_sq(tmp_1111);
   out = gf_sq(out);
   out = gf_sq(out);
   out = gf_sq(out);
   out = gf_mul(out, tmp_1111); // 11111111

   out = gf_sq(out);
   out = gf_sq(out);
   out = gf_mul(out, tmp_11); // 1111111111

   out = gf_sq(out);
   out = gf_mul(out, in); // 11111111111

   return gf_sq(out); // 111111111110
}
```

Figure 18. Example #1 Galois Field Arithmetic acceleration (TCA).

Example #1 Gaussian elimination acceleration (LCA)

The code below generated the public key of the Classic McEliece KEM. The part of the gaussian elimination of the matrix mat is being accelerated in a LCA manner. In the code we can see that the developer uses the functions provided by the device driver (i.e gaussian_elimination_init(), @security_lvl(), and gaussian_elimination()).

```
int pk_gen(unsigned char * pk, unsigned char * sk, uint32_t * perm)
{
   int i, j, k;
   int row, c;
   uint64_t buf[ 1 << GFBITS ];
   unsigned char mat[ GFBITS * SYS_T ][ SYS_N/8 ];
   unsigned char mask;
   unsigned char b;

   gf g[ SYS_T+1 ]; // Goppa polynomial
   gf L[ SYS_N ]; // support
   gf inv[ SYS_N ];

    g[ SYS_T ] = 1;

   for (i = 0; i < SYS_T; i++) { g[i] = load2(sk); g[i] &= GFMASK; sk += 2; }

   for (i = 0; i < (1 << GFBITS); i++)
   {
       buf[i] = perm[i];
       buf[i] <<= 31;
       buf[i] |= i;
   }

   sort_63b(1 << GFBITS, buf);

   for (i = 0; i < (1 << GFBITS); i++) perm[i] = buf[i] & GFMASK;
   for (i = 0; i < SYS_N;         i++) L[i] = bitrev(perm[i]);

   // filling the matrix

   root(inv, g, L);

   for (i = 0; i < SYS_N; i++)
       inv[i] = gf_inv(inv[i]);

   for (i = 0; i < PK_NROWS; i++)
   for (j = 0; j < SYS_N/8; j++)
       mat[i][j] = 0;

   for (i = 0; i < SYS_T; i++)
   {
       for (j = 0; j < SYS_N; j+=8)
       for (k = 0; k < GFBITS;  k++)
       {
           b  = (inv[j+7] >> k) & 1; b <<= 1;
           b |= (inv[j+6] >> k) & 1; b <<= 1;
           b |= (inv[j+5] >> k) & 1; b <<= 1;
           b |= (inv[j+4] >> k) & 1; b <<= 1;
           b |= (inv[j+3] >> k) & 1; b <<= 1;
           b |= (inv[j+2] >> k) & 1; b <<= 1;
           b |= (inv[j+1] >> k) & 1; b <<= 1;
           b |= (inv[j+0] >> k) & 1;

           mat[ i*GFBITS + k ][ j/8 ] = b;
       }
       for (j = 0; j < SYS_N; j++){
           inv[j] = gf_mul(inv[j], L[j]);
       }

   }
   gaussian_elimination_init();
   @security_lvl(3);
   if (guassian_elimination(mat)== -1) return -1;

   for (i = 0; i < PK_NROWS; i++)
       memcpy(pk + i*PK_ROW_BYTES, mat[i] + PK_NROWS/8, PK_ROW_BYTES);

   return 0;
```

Figure 19. Example 2: full matrix multiplication.

In Figure 20 we picture the interface described above. Since we expect significant changes to be made at the crypto accelerator design in the next phases of the WP2 the current visualization may not accurately reflect the final implementation.

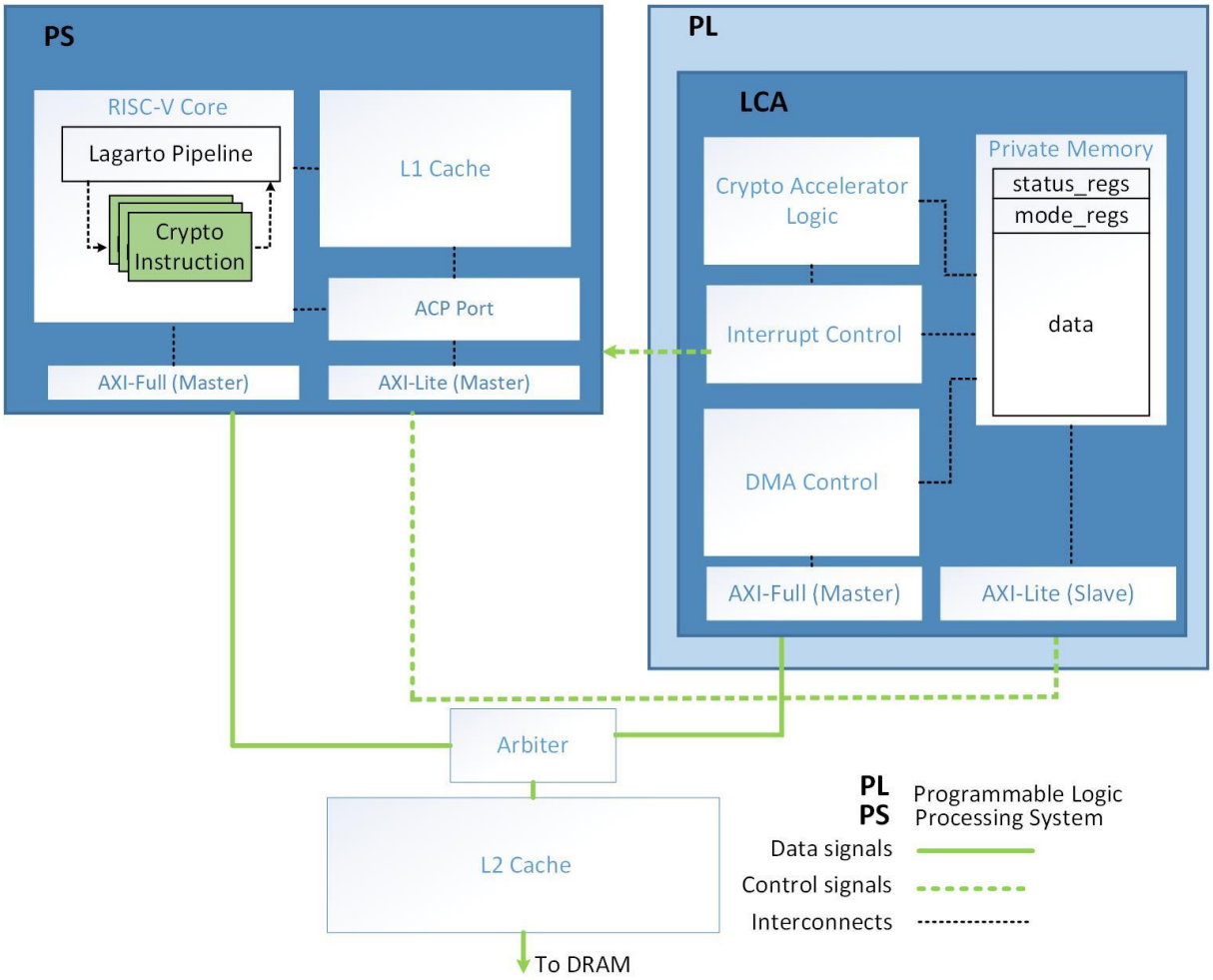

Figure 20. Generic view of the interface between the cryptographic accelerator(s) and the RISC-V core.

7. Conclusions

This document presents the different design proposals that will be the basis for achieving the DRAC objectives.

First, the design of the Lagarto Ka processor is presented, a 2-way 64-bit out-of-order processor capable of running the open-source ISA RISC-V. This processor includes in its design dynamic planning techniques with which it is sought to achieve high performance and low energy consumption, maintaining the trade-off between complexity and performance.

Coupled with the Lagarto Ka processor, a set of specialized accelerators are designed in the DRAC project to facilitate the execution of specific applications. In this document we have defined all the interfaces required to integrate such accelerators in the Lagarto Ka processor.

The first accelerator targets genomic analysis applications. Such accelerator is based on a novel high performance parallel architecture for processing and analyzing genomic data at a large scale. These applications are a relevant component in current and future personalized medicine applications: the sequencing phase is an essential part of most genomic data analysis pipeline. The reduction in its processing cost directly impacts the treatment of health data.

The final DRAC design also includes an accelerator for automotive applications with approximate computing in FDSOI technology. The advent of autonomous driving requires relatively straightforward (and inexpensive) hardware that gives a very high performance within a stringent consumption limit to meet the requirements of automotive systems. The current high-performance systems that could provide the necessary performance do so with very high consumption.

Thus, this project is focused on designing an accelerator that uses the approximate computation for complex functions and the FDSOI technology (oriented to low consumption). It will be capable of introducing alterations in the results when operating at low supply voltage; without worsening the global precision of the prediction process. In particular, this project will design and implement the indicated calculation unit and define its integration into future generations of the European processor, which attacks the supercomputing and automotive segments.

At last, a cryptographic accelerator is integrated. In the DRAC project, different candidate schemes are analyzed. Additionally, the corresponding RISC-V extensions will be designed to be incorporated into the processor. Classical security techniques such as randomization will also be developed and implemented to achieve a safe out-of-order processor and the implementation of hardware security levels (rings) together with the hardware support of the virtualization mechanisms present in the set of instructions for RISC-V.

Finally, a memory hierarchy capable of maintaining the correct coherence and consistency of the data will be implemented through its different cache levels (L1, L2, main memory) and the accelerators developed in this project. Likewise, the interfaces of the possible direct communication schemes with the accelerators are proposed, in order to achieve an improvement in the final performance of the system.

8. References

[1] Xilinx, "AMBA AXI4 Interface Protocol", available on-line: https://www.xilinx.com/products/intellectual-property/axi.html

[2] Xilinx, "KC705 Evaluation Board for the Kintex-7 FPGA".

[3] Xilinx, "VCU128 Evaluation Board".

[4] Carlos Rojas Morales, Master Thesis: "From FPGA to ASIC: A RISC-V processor experience", Instituto Politécnico Nacional de México - Universidad Politècnica de Catalunya, 2019.

[5] Toshihiro Hanawa et al, "Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators", 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum, USA, 2013.

[6] Luca Piccolboni et al, "PAGURUS: Low-Overhead Dynamic Information Flow Tracking on Loosely Coupled Accelerators", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 37 (11), Nov 2018.

[7] SemiDynamics, "AVISPADO-VPU Interface", available on-line: https://github.com/semidynamics/OpenVectorInterface

[8] YOLO: Real-Time Object Detection, available on-line: https://pjreddie.com/darknet/yolo/

[9] Apolo: Smart Transportation Solutions, available on-line: https://apollo.auto/