T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Resource Efficient Processor Architectures with RISC-V Vector Processing Unit for FPGA
著者(和文)	ISLAMMD ASHRAFUL
Author(English)	Islam MD Ashraful
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12557号, 授与年月日:2023年9月22日, 学位の種別:課程博士, 審査員:吉瀬 謙二,宮崎 純,渡部 卓雄,横田 理央,金子 晴彦
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12557号, Conferred date:2023/9/22, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)

東京工業大学
Tokyo Institute of Technology

Resource Efficient Processor Architectures with RISC-V Vector Processing Unit for FPGA

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Engineering

Md Ashraful Islam

July 2023

Department of Computer Science
School of Computing
Tokyo Institute of Technology

Table of Contents

List of Tables

List of Figures

Abstract

The current expansion of the Internet of Things (IoT) is generating a large volume of data from the billions of sensors and devices connected to the network's edge. Processing such a large number of data continuously in the cloud is inefficient due to the network bandwidth and latency. These shortcomings of cloud processing push the computation from cloud to edge devices which insistence on more processing capabilities to meet the task requirements at the edge. Multicore with scalable architectures is a promising solution for this computation demand. Furthermore, heterogeneous multicore solutions consisting of different core types can offer a performance advantage.

FPGAs are reconfigurable devices and have grown in prominence for speeding computationally demanding tasks in particular applications. However, creating such an accelerator on an FPGA takes time and needs knowledge of hardware design and hardware description language. On the other hand, heterogeneous multicore integrates different processing cores, and sometimes custom hardware accelerators are most suited for edge computing requirements due to their diverse task requirements. Heterogeneous multicore systems may be constructed on an FPGA, giving the opportunity to use software to develop an application. The goal of this research is to create a resource-efficient heterogeneous multicore processor.

Several techniques have been applied to the microarchitecture to increase the instruction level parallelism (ILP), such as out-of-order super-scalar architecture to increase the processor performance. But implementation of these techniques in FPGA is resource inefficient and limits the operating frequency. Moreover, improving only a single core does not provide adequate performance for multiple-thread/task execution. The lack of single-core performance can be compensated by the multicore processor, where each core can be relatively less complex and resource-efficient in-order cores. Thus running these threads in parallel can multiply the overall performance, which is termed thread-level parallelism (TLP).

Applications with different task requirements can be broadly categorized as control-intensive and data-intensive. Heterogeneous multicore is supposed to take care of both kinds of task requirements by allocating the control-intensive tasks to the general-purpose CPU and data-intensive tasks to the accelerator or processor with SIMD/vector capability. An application-specific accelerator performs only a specific job, and users need to understand the requirement and develop the accelerator every time the requirement changes.

On the other hand, for massively data parallel workloads, vector processing is an efficient approach. A single vector instruction can process several data elements, thus amortizing the instruction fetch and decoding overhead. Vector processors can effectively exploit data-level parallelism (DLP). Moreover, the scalable vector architecture provides the flexibility to adapt the performance requirements by changing the parameters such as vector length and lanes.

The purpose of this research is to propose a resource-efficient heterogeneous multicore architecture that accommodates the vector processing unit that will make use of the data level parallelism (DLP) inside a given task/thread. At the same time, the scalar processor cores provide task/thread-level parallelism (TLP). I have proposed processor architectures in this thesis, including vector processing for FPGA.

In Chapter 3, 4-stage and 5-stage in-order scalar processor microarchitectures are presented that enable resource sharing. I design and implement a 4-core processor with multiple sharing options. I demonstrate that sharing resources in the multicore significantly reduces the logic utilization in FPGA. I have simulated the benchmark programs and shown that many different workloads have little performance impact due to hardware resource sharing. Then I present an efficient shared memory architecture with less RAM utilization in FPGA.

A scalable vector processing unit microarchitecture is introduced in Chapter 4. The architecture provides the flexibility to configure the vector lanes and vector length. I implement the vector architecture for multiple vector lengths and lanes. I have shown the implementation result and compared it with the other proposed vector processing unit. My proposed architecture has significantly less FPGA resource utilization and achieves much higher operating frequency. From the simulation results, I have shown that the proposed architecture performance is scalable with varying parameters.

I demonstrate a heterogeneous multicore architecture in Chapter 5. I design an efficient integration of the vector processing unit to the 5-stage pipeline RISC-V processor. I present the resource-shared four-core heterogeneous multicore processor with a vector processing unit. I implement the heterogeneous multicore architecture with and without resource sharing with multiple vector processing configurations. I have shown that the vector processing unit substantially boosts the multicore performance. Compared to the performance gain, the resource utilization for different vector processing unit configurations is much lower. And sharing the resources in multicore further reduces the logic utilization with a minimal performance drop.

1 Introduction

1.1 Motivation

With the proliferation of mobile devices and ubiquitous Internet connectivity, the world is now immersed in various wireless connectivity. This Internet of Things (IoT) connects these massive numbers of devices. The worldwide number of IoT devices is expected to increase from 9.7 billion in 2020 to more than 29 billion IoT devices in 2030[21]. IoT ecosystem with the newly developed protocols and System-on-Chips (SoCs) plays a critical role in data transmission from the billions of sensor nodes to the cloud.

However, cloud-based centralized data processing has significant limitations concerning security, power management, and end-to-end communication latency. In order to reduce the enormous overhead associated with the uplink and downlink transmissions, it has become necessary to push data processing to the edge, thus distributing the computational (and storage) resources. Besides exchanging data with other connected devices to centralized servers in the cloud, the IoT devices are expected to perform some computational tasks locally rather than cloud-based back-end processing for the data. Processing data at such IoT devices, which stays at the edge of the communication network, is referred to as edge computing.

There are different kinds of IoT devices with varied interfaces and processing requirements. They have common servicing requirements, such as capturing time-sensitive data from various input/output channels, processing these data streams in geographically distributed devices, etc. However, their workloads often vary with considerable data size and sometimes require extensive computation, such as in the applications of artificial intelligence, voice or image processing, etc. Moreover, these computations are sometimes latency-sensitive; in other words, they require a predictable performance with shorter execution times.

Usually, IoT devices are embedded devices, which contain embedded RISC processors with on-chip SRAM memory and some DSP blocks featuring multiply-accumulate (MAC) to achieve data processing capabilities. But due to the diverse requirement for edge computation, a single processor and only MAC operations may not be enough to meet the requirements. On the other hand, general-purpose CPUs and GPUs have higher performance because of multiple parallel processors and are optimized for batch processing. But they lack a wide range of I/O channels and can hardly provide consistent streaming

data processing. Furthermore, CPUs and GPUs are not energy efficient for edge devices.

Field Programmable Gate Array (FPGA) is a re-configurable hardware device that contains logic, memory, and a wide range of I/O. These make FPGA suitable for edge computation due to reconfigurability and adaptability to the various workload. FPGA design requires hardware description languages, such as Verilog, and knowledge of digital circuits, making it difficult for application developers to use.

On the other hand, heterogeneous multi-core architectures are recently being adopted for IoT to adapt to different kinds of workloads. There are different types of processor cores utilized in a heterogeneous multicore processor. Such heterogeneous multicore can be realized in FPGA by using the soft processor cores. Thus application developers can focus on software development without much getting involved in FPGA hardware design. FPGAs have limited resources, which is the main challenge of developing such heterogeneous multicore in FPGA. Secondly, achieving higher operating frequency in FPGA requires careful design and well-defined architecture. This thesis paper presents FPGA soft processor-based heterogeneous multicore architectures which achieve higher resource efficiency and better operating frequency.

1.2 Contribution

The contributions of this thesis are the following:

- I propose a resource-shared multicore processor for FPGA. First, I presented a soft processor microarchitecture that supports RISC-V ISA RV32IM, allowing sharing of functional units - shifter, multiplier, and divider with the other cores in the multicore processor. I have then implemented a 4-core processor with different sharing configurations. I study the logic reduction for different hardware sharing options from the FPGA implementation results. And finally, I compare the effect on performance due to different sharing configurations. I have also shown the block RAM (BRAM) utilization reduction by sharing the dual port of the BRAM in FPGA.
- I propose resource-efficient scalable soft vector processor architecture. I introduce a 5-stage pipeline-based vector processing unit with configurable vector length and lanes. To support different vector instructions, the design can accommodate various functional units. I present distributed vector register file-based architecture that provides flexibility to configure the number of lanes and the vector length. My vector processing unit performance is scalable with a number of lanes and

vector lanes. To reduce the latency of integer dot/convolution operation, I present an efficient reduction sum operation in conjunction with a vector multiplier. I have implemented the vector processing unit in FPGA for different parameters and demonstrated that FPGA resource utilization is much lower than the other vector processing unit designs. The operating frequency of my VPU is also higher than others.

- I demonstrate a heterogeneous multicore architecture with the vector processing unit. I propose an efficient integration of the vector processing unit to a 5-stage pipeline RISC-V core, which decouples the scalar and vector execution, thus increasing the processor's efficiency. I have presented the heterogeneous multicore architecture with resource sharing among the cores. I have implemented the 4-core processor for FPGA, studied the resource utilization, and compared the resource utilization with and without resource sharing. At the same time, vector processing units had different lanes and vector lengths. I have evaluated the performance of a heterogeneous multicore processor and compared it with a homogeneous multicore processor.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

In Chapter 2, I have described the background of processor microarchitecture, vector processing, and multicore processor. This chapter briefly discusses the RISC-V instruction set and the most commonly referred general 5-stage pipeline processor architecture. Then illustrate some related work on processor microarchitecture. The vector processing evolution and relevant architecture have also been discussed. And finally, the heterogeneous multicore architecture is described shortly with some state-of-the-art related works.

Chapter 3 proposes the novel microarchitecture of an in-order pipeline processor that enables the sharing of functional units for execution among the multiple cores as well as sharing the BRAM ports. This chapter describes some related works on sharing hardware resources and shows the distinction of my proposed architecture. I have described the FPGA implementation of four-core processors without and with hardware sharing. I presented the simulation results of the benchmark program and a detailed study of the sharing effect on the benchmark program execution.

In Chapter 4, I describe the novel microarchitecture of a soft vector processor for FPGA.

I briefly explain the state-of-the-art vector processing unit and show the merit of my proposed architecture. I have implemented the proposed architecture in FPGA and show its scalability when configuring different vector lengths and the number of lanes while utilizing fewer resources on the FPGA than related works. Results demonstrate that the proposed vector processing unit can achieve higher design frequencies than existing work due to its pipeline design and distributed vector register file while keeping scalability on FPGA resource utilization when increasing vector length and lanes. I have presented the simulation result, which shows the achieved giga operation per second for the convolution kernel, thanks to the optimized reduction tree, which reduces the computational latency.

Chapter 5 proposes the heterogeneous multicore architecture integrating resource-shared multicore with a vector processing unit. I have proposed the efficient integration of a vector processing unit to a 5-stage pipeline scalar processor. I have presented heterogeneous multicore architecture, which can share resources among the cores and utilize a vector processing unit as an accelerator in the multicore. I show the FPGA implementation result for heterogeneous multicore without and with resource sharing and vector processing units with multiple lane configurations. I demonstrated the performance of the vector processing unit compared to the scalar core and boosted the performance of the multicore system.

Finally, in Chapter 6, I have summarized the thesis and discussed future work.

2 Background and Related Work

2.1 RISC-V Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA) describes how a processor performs different executions based on the instructions. The ISA defines the set of registers owned by the processor and describes each machine instruction's binary and how they are encoded. Each instructions have a unique binary encoding. An ISA determines what each instruction does and what the corresponding processor states. It defines how to determine which registers will be accessed, immediate values, program counter value, memory access, the length of the instruction, and so on. Examples include the x86 ISA for Intel/AMD CPUs and the ARM ISA for mobile CPUs.

Hardware and software understand each other through the ISA. CPU Hardware is designed to implement the binary encoding of a given ISA. The software and application code are eventually translated into binary according to the ISA using a compiler/assembler. By following the specifications of the ISA, hardware and software can communicate with each other. The same ISA is executed in different CPU microarchitectures with varying pipeline stages, and the datapath will produce the same result while performance will be varied.

Existing commercial ISA have widely supported software ecosystems, development tools, and applications. However, the commercial instruction sets have the following disadvantages[22]:

- Commercial ISAs are proprietary; owners protect their ISA as intellectual property. Implementation of microarchitecture based on that ISA without licensing/permissions is prohibited.
- Commercial ISAs are usually targeted for particular application domains; for example, ARM ISA is popular for embedded domains, and x86 ISA is widely used for general-purpose computing.
- Predominant ISAs like x86 and ARM instructions are complex and comparatively hard to implement.
- The commercial ISAs instructions have increased over time, and to accommodate the new instructions, the binary encoding has become more complex and lacks the extensibility in ease.

- The benefits of using standard ISA quickly diminish if a small extension is added because it requires rebuilding the compilers and applications from the source code with the new extension.

To overcome those limitations, the RISC-V ISA was proposed by Andrew Waterman, Yunsup Lee, Krste Asanović, and David Patterson in 2010[23]. RISC-V evolved from the previous ISA called MIPS to overcome the shortcomings of MIPS-based designs. In 2015, the RISC-V Foundation (www.riscv.org) was established to build an open, collaborative community backed by software and hardware innovators.

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. A naming convention is used to identify the particular extension and ISA variations. The 32/64/128-bit ISA starts with RV32/RV64/RV128. The base integer ISA is defined as RV32I/RV64I/RV128I. Some other optional extensions are like

- M for multiply and divide instructions
- A for atomic instructions
- C for compressed instructions
- F for floating point instructions
- V for vector instructions

For example, RV32IM means the processor supports 32-bit RISC-V base integer, multiply, and atomic instructions. The ISA has defined 32 general-purpose registers as register files for integer extensions and separate register files for floating-point and vector extensions.

2.2 Processor Microarchitecture

A processor's microarchitecture details the implementation of the supported instruction set architecture (ISA). The design of the processor microarchitecture considers the datapath, control unit, caches or tightly coupled memory, and functional units. Usually, the microarchitecture uses some optimization techniques to improve performance and efficiency, such as pipelining.

Nowadays, pipelining is the key implementation technique used to make fast CPUs[1]. Pipelining is an implementation technique where instructions are executed over a few stages. The stages are connected one to the next to form a pipe, where instructions enter

at one end, progress through the stages, and exit at the other end. It exploits parallelism among the instructions in a sequential instruction stream. A classic five-stage pipeline is mostly referred to in computer architecture education which is most popular. The 5-stages are described below[1]:

- Instruction fetch stage (IF): Use the program counter (PC) to address the instruction memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.
- Instruction decodes stage (ID): Decode the instruction and read the registers corresponding to source operands from the register file. Decoding is done in parallel with reading registers.
- Execution stage (EX): The ALU operates on the operands prepared in the prior cycle, performing the data process or preparation for the next stage as
  - Memory reference—The ALU adds the base register and the offset to form the effective address.
  - Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
  - Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate value.
- Memory access stage (MEM): If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, the memory writes the data from the second register and reads from the register file using the effective address.
- Writeback stage (WB): Write the result into the register file, whether it comes from the memory system (for a load) or the ALU (for an ALU instruction).

Figure 1 shows the diagram of typical 5-stage pipeline execution, where each stage takes a single clock cycle to perform. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figure 1: Simple RISC pipeline[1]

2.2.1 In-order Scalar Processor

An in-order processor executes the instructions in the order of the way it comes across the program, that is, following the program's sequential order. The instructions are executed in pipeline stages. Depending on the number of pipeline stages, multiple instructions are being processed over the pipeline stages sequentially. Whenever a pipeline stage stalls due to a hazard between two consecutive instructions, successive instructions are blocked and cannot proceed until the hazard is resolved. Even though the later instructions do not have hazards with the stalled instructions, the instructions will be yet stalled.

Rocket[2] is a scalar processor core designed to support the RISC-V instruction set architectures RV32G (RV32IMAFD) and RV64G (RV64IMAFD). It was originally developed by the University of California Berkeley and implemented using Chisel[24], a high-level hardware construction language. The processor has a 5-stage in-order pipeline and supports virtual memory through its MMU, which uses page-based addressing. The pipeline diagram of the rocket core is shown in figure 2. It has a non-blocking data cache and a branch predictor that can be configured. The branch predictor consists of a branch target buffer (BTB), a branch history table (BHT), and a return address stack (RAS). Additionally, the processor makes use of Berkeley's Chisel-based implementation of floating-point units[25] for handling floating-point operations. The detailed microarchitecture[3] of the rocket core is presented in figure 3.

The Rocket Chip SoC generator[26] is a Chisel-based RTL generator for the SoC. The Rocket core is a key component of the Rocket Chip SoC generator. When a Rocket core is combined with L1 data and instruction caches, it creates a Rocket tile, which is a replicable unit within the SoC generator. The Rocket tile plays an important role in the overall architecture of the Rocket Chip SoC generator.

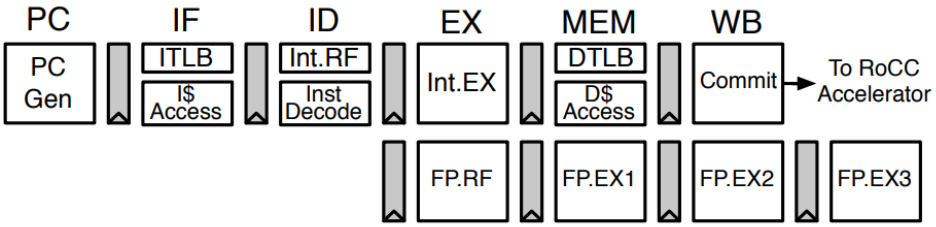


Figure 2: The Rocket core pipeline[2]

Through the Rocket Custom Coprocessor Interface (RoCC), a Rocket processor can be attached to other coprocessors like SHA36, vector processing units, etc, to communicate with these coprocessors independently. The RoCC interface facilitates the handling of coprocessor commands generated by the committed instructions from Rocket core. Furthermore, the RoCC interface provides mechanisms for the attached coprocessor to share the Rocket core's data cache and page table walker, as well as allow the coprocessor to interrupt the core.

CV32E40P[27] is a 4-stage in-order 32-bit RISC-V processor core. The pipeline stages are shown in the figure 4, which are[4]:

- Instruction Fetch (IF): Using the prefetch buffer, it retrieves a single instruction from instruction memory in every cycle. This prefetch buffer can hold two 32-bit instructions. The compressed instructions are also pre-decoded into RV32I base instructions in the IF stage.
- Instruction Decode (ID): Fetched instructions are decoded, and register file operands are read in this stage. The ID stage also serves the decoded jump instructions.
- Execute (EX): Carries out the instruction execution. An ALU, multiplier, and divider are components of the EX stage. Branches are extracted from the EX stage. Multi-cycle instructions will halt EX, ID, and IF stages until they are completed. The results from the EX stage's ALU, Multiplier, and Divider operations are returned to the register file. The EX stage also determines the memory address for the load-store unit (LSU).
- Writeback (WB): The data that comes from the memory in response to load instructions are written back to the register file. Thus the register file has two write ports.

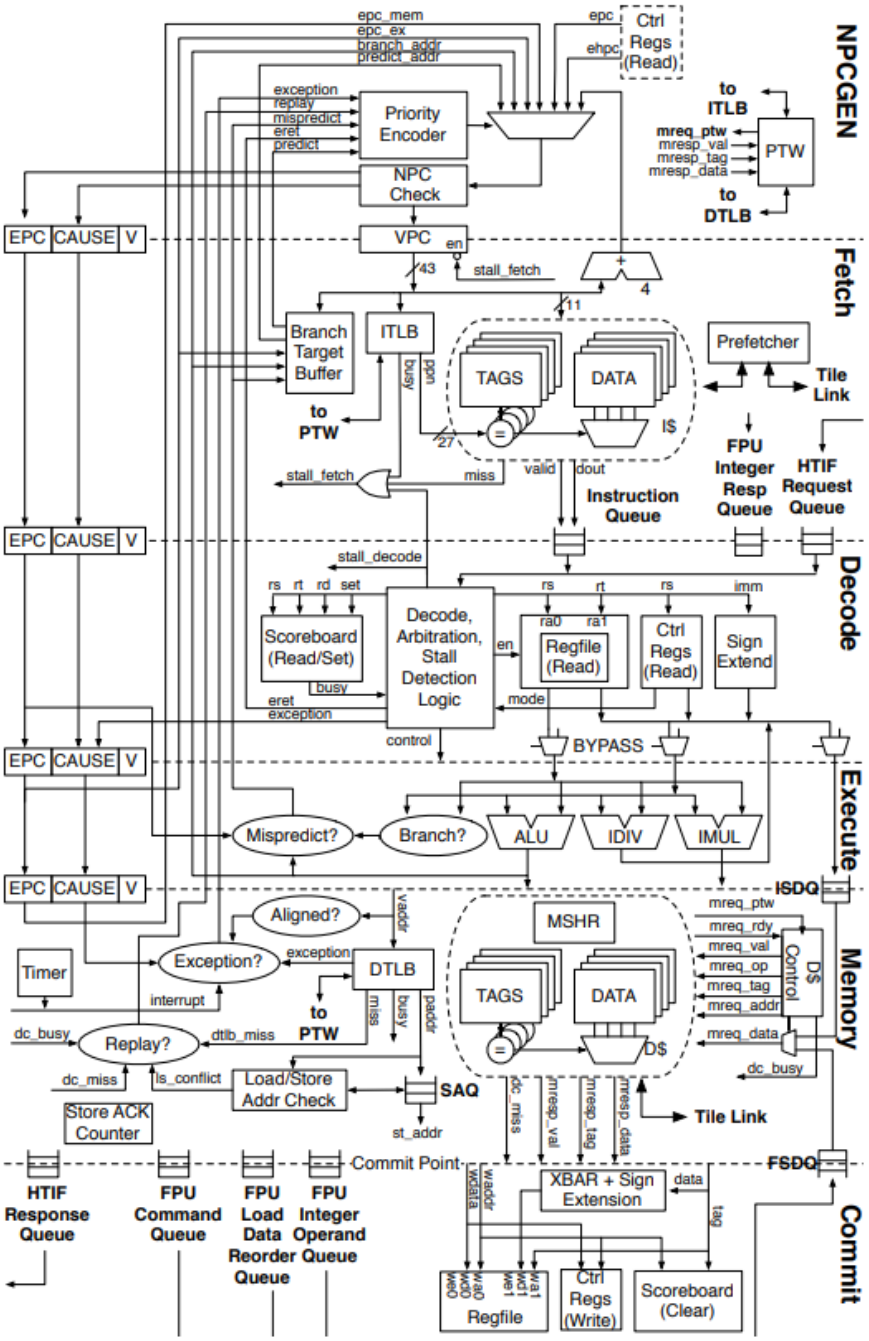


Figure 3: The Rocket microarchitecture[3]

RVCoreP-32I[5] a five-stage pipelined processor containing instruction fetch stage (If stage), instruction decode stage (Id stage), instruction execution stage (Ex stage), memory access stage (Ma stage), and write back stage (Wb stage). Figure 5 shows the block diagram of RVCoreP-32I. The RVCoreP-32I applied the following optimizations to the

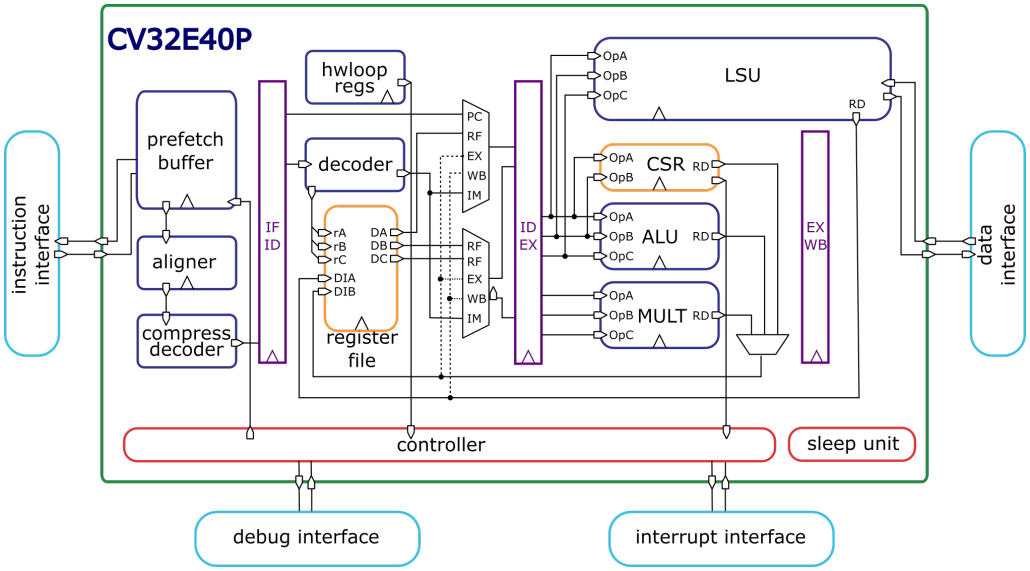


Figure 4: The microarchitecture of CV32E40P[4]

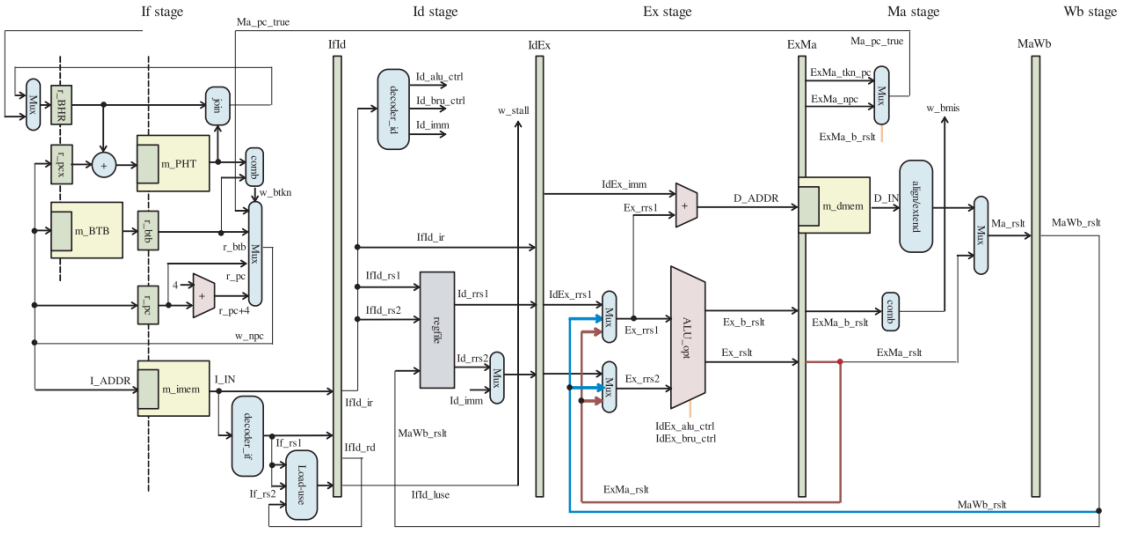


Figure 5: The block diagram of RVCoreP-32I[5]

traditional 5-stage pipeline processor:

- To improve the ALU operating frequency in FPGA, the ALU operations selection bits are encoded as one-hot encoding. For each individual operation of the ALU, the results are generated individually. If the selection bits of that individual operation are high, then it will be evaluated; otherwise, it will be 0. These individual results are then performed exclusive-OR to compute the final result for the ALU.
- For three load instructions of RV32I load byte unsigned (LBU), load halfword (LH),

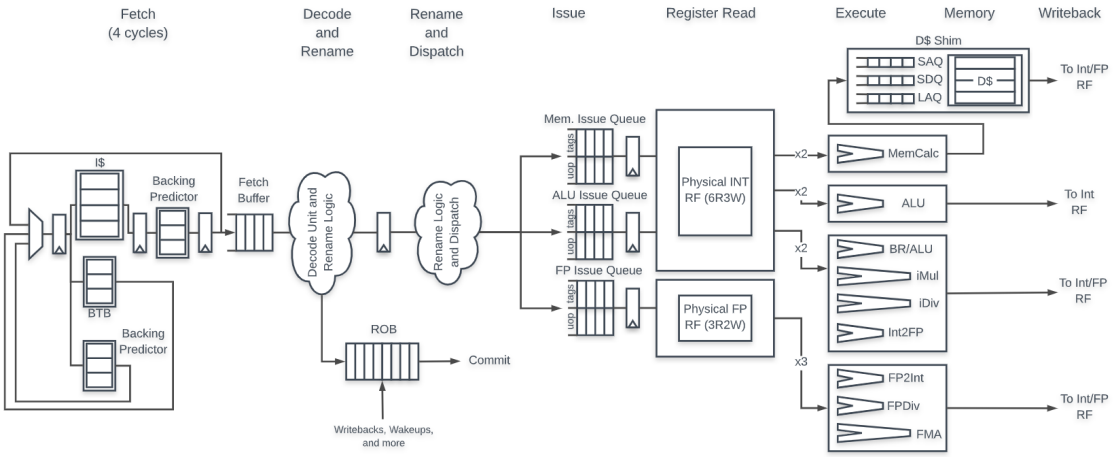


Figure 6: Simplified BOOM Pipeline with Stages[6]

load halfword unsigned (LHU) align/extend unit is required to align the loaded data by right shifting 8, 16, or 24 bits depending on the memory address and operation code of the load instructions. Similar to ALU optimization, these shifting operation is one-hot encoded and exclusive OR together.

- It applied pipeline to the gshare[28] brach predictor. It takes two cycles to determine the value of the next PC in the instruction fetch stage. In the first cycle (preIf stage), accessing the branch target buffer (BTB) and exclusive OR processing to determine the pattern history table (PHT) index are performed. In the second cycle in the If stage, the value of the next PC is determined by using the results from the preIf stage, and the instruction is fetched from the instruction memory.

2.2.2 Out-of-Order Processor

An out-of-order processor may process instructions in a different order from the one specified in the program sequence in as much as there is no data or control dependency. Instruction level parallelism (ILP) is increased when instructions are executed out of order. The instructions can be carried out independently of one another in each cycle by hardware functional units. More complicated hardware is needed for out-of-order processors than for ones that are in order.

BOOM[29] is an out-of-order microprocessor and implements the RV64G RISC-V ISA. The microarchitecture is designed with Chisel[24] language to generate the RTL for the core. Like Rocket[2] core, BOOM is also available in the Rocket Chip SoC generator as a component of a reusable library of different micro-architecture structures.

BOOM executes instructions in the pipeline of seven stages namely: Fetch, Decode/Rename, Rename/Dispatch, Issue/Register-Read, Execute, Memory, and Writeback (Commit occurs asynchronously, thus it is not considered as a pipeline stage). Figure 6 shows the simplified pipeline stages of BOOM. Instructions are pulled from instruction memory and placed into the fetch buffer, a FIFO queue. In this stage, branch prediction also takes place. Decoder then constructs the necessary Micro-Op by pulling instructions from the Fetch Buffer. Following that, the "logical" registers are mapped to "physical" registers. The detailed microarchitecture of BOOM is shown in figure7.

The uOP is thereafter sent to the issue queue(s). The uOPs kept in an issue queue are only issued to execute stage once all of their operands are available. Issued uOPs reach the corresponding functional units in the execute stage after reading their register operands from the unified Physical Register File (or through the Bypass Network). Memory operations that will be issued to the load/store unit of the memory stage calculate their addresses in the execute stage.

Load Address Queue (LAQ), Store Address Queue (SAQ), and Store Data Queue (SDQ) are three queues of load/store units. When an address is found in the LAQ, loads are launched to memory. At commit time, stores are fired to memory (and obviously, stores can't be committed until their address and data are in the SAQ and SDQ).

RSD[7] is a 32-bit RISC-V out-of-order superscalar processor core. The microarchitecture is shown in figure8. The microarchitecture consists of three blocks: a frontend block, a scheduling block, and an execution block. The front-end block fetches and decodes instructions from instruction memory in program order.

The dispatch unit, issue queue (IQ), rename unit and reorder buffer make up the scheduling block. The rename unit renames the logical registers of the source and destination operands to physical registers, which are acquired from the physical register file (PRF) free list. Register renaming is done in order to eliminate false dependencies (write after write and write after read) between instructions. The mapped physical registers are subsequently registered in a register map table (RMT). Depending on the type of instruction, the dispatch unit allocates an entry for a renamed instruction in a number of components, including the re-order buffer (ROB), instruction queue (IQ), load queue (LDQ), and store queue (STQ). When all of an instruction's source operands are accessible, the IQ is in charge of sending instructions to the execution block. The ROB maintains the state of dispatched instructions and commits an instruction once it has been correctly executed, becoming the

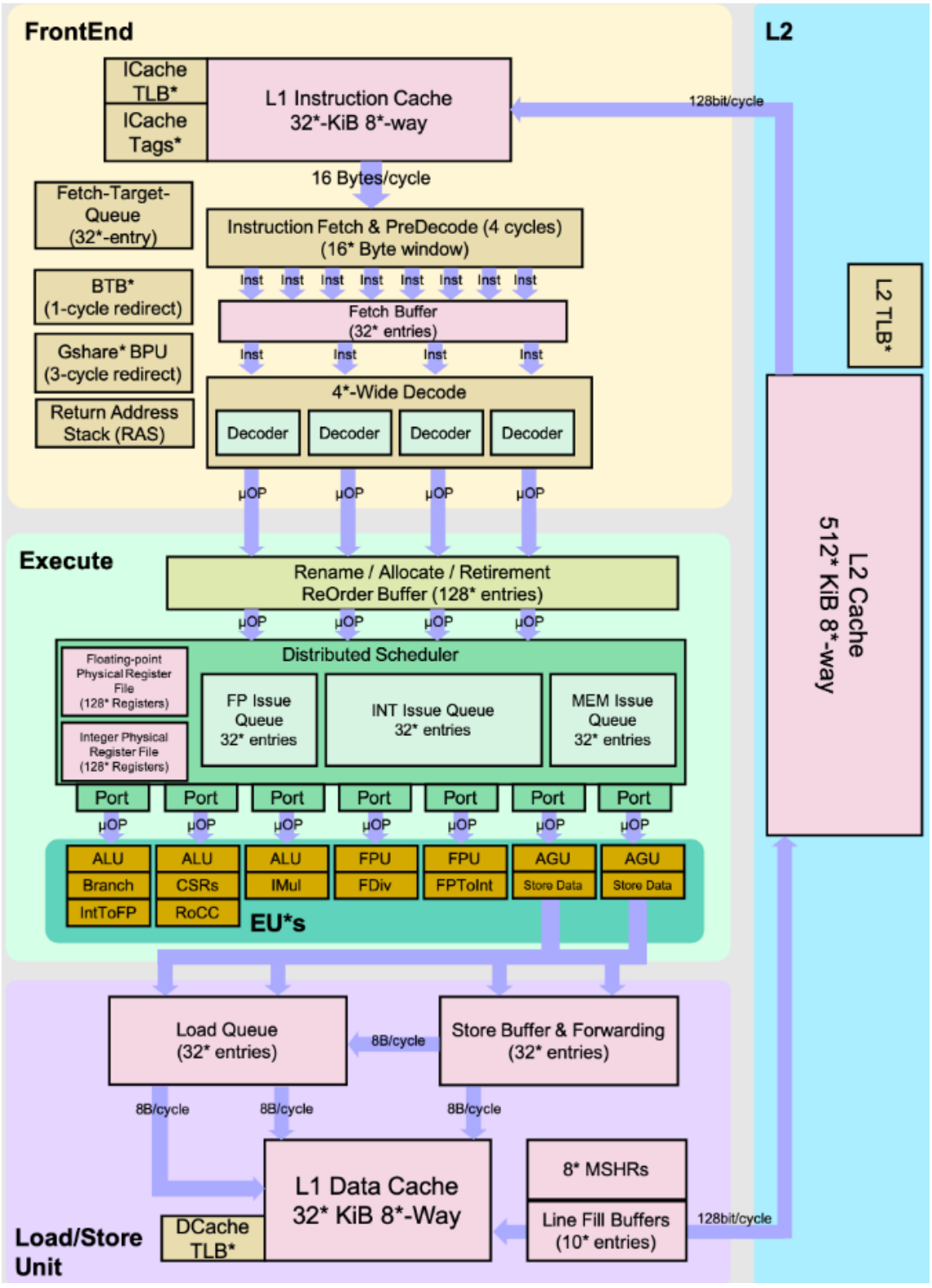


Figure 7: Detailed BOOM microarchitecture[6]

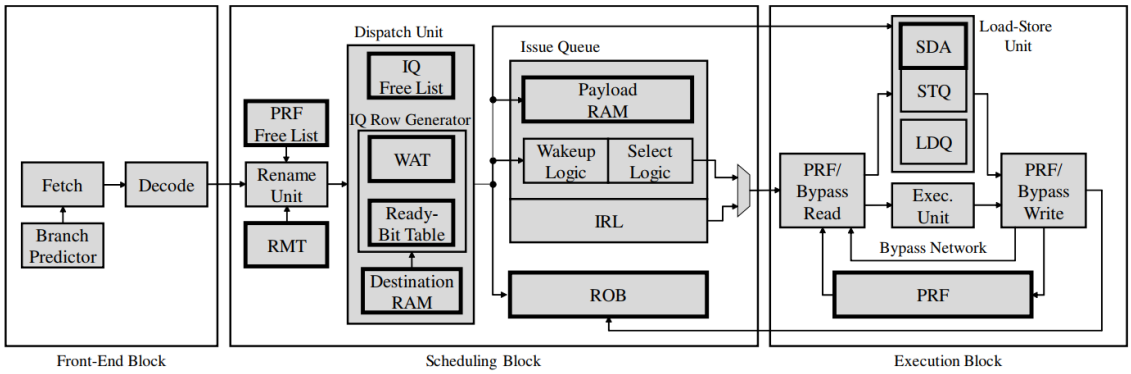


Figure 8: Block diagram of the RSD[7] © 2019, IEEE

oldest instruction in the processor.

The scheduling block sends instructions to the execution block, which executes them. The operand physical register data is stored in the PRF. When an instruction enters this stage, it gets the data for its source operands from the PRF or a previous instruction via a bypass network. The relevant execution unit executes the instruction and transmits the output data to the bypass network and PRF. The load-store unit (LSU) manages the load, stores instructions, and communicates with the memory system.

2.3 Vector Processors

In many programs, the same function or process is applied to an array or a vector of data elements. CPU is required to process iteratively through each element of the array, in other words, executing the same code on a large number of data. Data level parallelism (DLP) is processing these data in parallel. Vector architectures exploit the DLP by gathering sets of data elements scattered in memory, bringing them into large sequential register files, operating on data in those register files, and then dispersing the results back into memory. A single instruction works on vectors of data elements. The first vector supercomputers were TI ASC and CDC STAR-100, introduced in 1972 and 1974, respectively. Both were memory-memory vector machines. They had relatively slow scalar units. Both devices have relatively long start-up overhead due to memory latency. They were able to process vectors of several hundred to several thousand elements.

In 1976, Seymour Cray from CDC designed and launched the Cray-1, the first commercially successful supercomputer with vector instructions. The CRAY-1 used a vector-register architecture to significantly lower start-up overhead. With the Cray-1, vector technology became mainstream in the supercomputer world. Compilers were developed

to vectorize loops automatically. The design used the pipeline technique to implement vector instructions, and the design had completely separate pipelines for different instructions; for example, addition/subtraction was implemented in different hardware than multiplication. This allowed a batch of vector instructions to be pipelined into each functional unit, a technique called vector chaining. Many subsequent vector machines are based on the architecture of this first commercially successful vector machine.

The primary components of the vector architecture are the following[1]:

- **Vector registers** — Each vector register holds a single vector, where vector elements are packed together. RISC-V defines 32 vector registers. The maximum vector element size that any operation can read or write is *ELEN*, which must be a power of 2 and greater than or equal to 8 ($ELEN \geq 8$). For instance, ELEN can be 8/16/32, and so on. The number of bits in a single vector register, *VLEN*, must be a power of 2 and $VLEN \geq ELEN$.

- **Vector functional units** — To perform different operations on the vector, there are some available units to perform these operations, such as ALU for arithmetic and logical operation, MUL for multiplication and accumulation operation, etc. These functional units are usually implemented in the pipeline and can start a new operation on every clock cycle. A control unit is needed to detect structural hazards for functional units and data hazards on register accesses.

- **Vector load/store unit** — The vector memory unit loads or stores a vector to or from memory. Depending on the type of memory accesses, the LSU may have different latencies to perform load or store.

- **Scalar and control status registers** - Some vector instructions return a scalar value, and some vector instruction receives scalar values from the scalar processor. There are some control and status registers (CSR) to indicate how to process the vector elements, such as elements width, number of elements, etc.

- **Vector Lane** — Vector lanes are multiple parallel vector units structured as parallel datapaths. Lane reduces the number of clock cycles to execute the vector instructions as multiple elements are processed in parallel.

One of the common approaches to handling a large number of elements is "strip-mining," where in each iteration of a loop, some number of elements are processed, and the iterations continue until all elements have been processed[30]. The RISC-V vector specification sup-

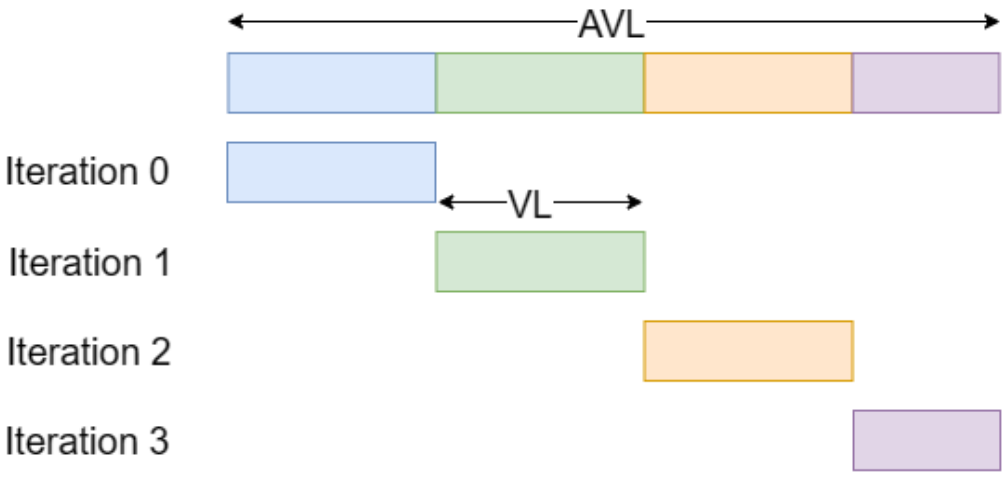


Figure 9: An example of vector strip-mining

ports this approach, making porting the application program to different vector processors easier. The ISA defines *vl* CSR, which holds an unsigned integer specifying the number of elements to be updated with results from a vector instruction. This vl CSR can only be updated by the vset{i}vl{i} instructions. The application specifies the total number of elements to be processed (the application vector length or AVL) as a candidate value for vl CSR. The hardware responds via a general-purpose register with the number of elements that the hardware will handle per iteration (stored in vl) based on the microarchitectural implementation and the vtype setting. Figure 9 shows how an AVL larger than the vector length is processed iteratively using the strip-mining technique.

VEGAS[8] is a scalable soft vector processor designed for FPGA, implementing custom vector instructions, and it was integrated with Nios II[31] processor. Figure 10 shows the VEGAS architecture. There are two separate 16-entry queues to dispatch instructions from the Nios II: one for vector instructions and another for DMA block-transfer operations. VEGAS uses a scratchpad memory instead of a vector register file to access the working vector data set. Vector lanes read directly from the scratchpad memory, and the results are returned to the same memory. When the vector data operands addresses are aligned to the lane structure, the read and write latency is lower than unaligned accesses. A data alignment crossbar network is used for unaligned addresses to align the data. It uses a fracturable ALU, which can operate on four-byte, two halfwords, or word of 32-bit data based on the selected vector element width.

Hwacha[9] decouples the vector pipeline from the scalar core developed by the University of California Berkeley. It does not implement the RISC-V standard vector extension

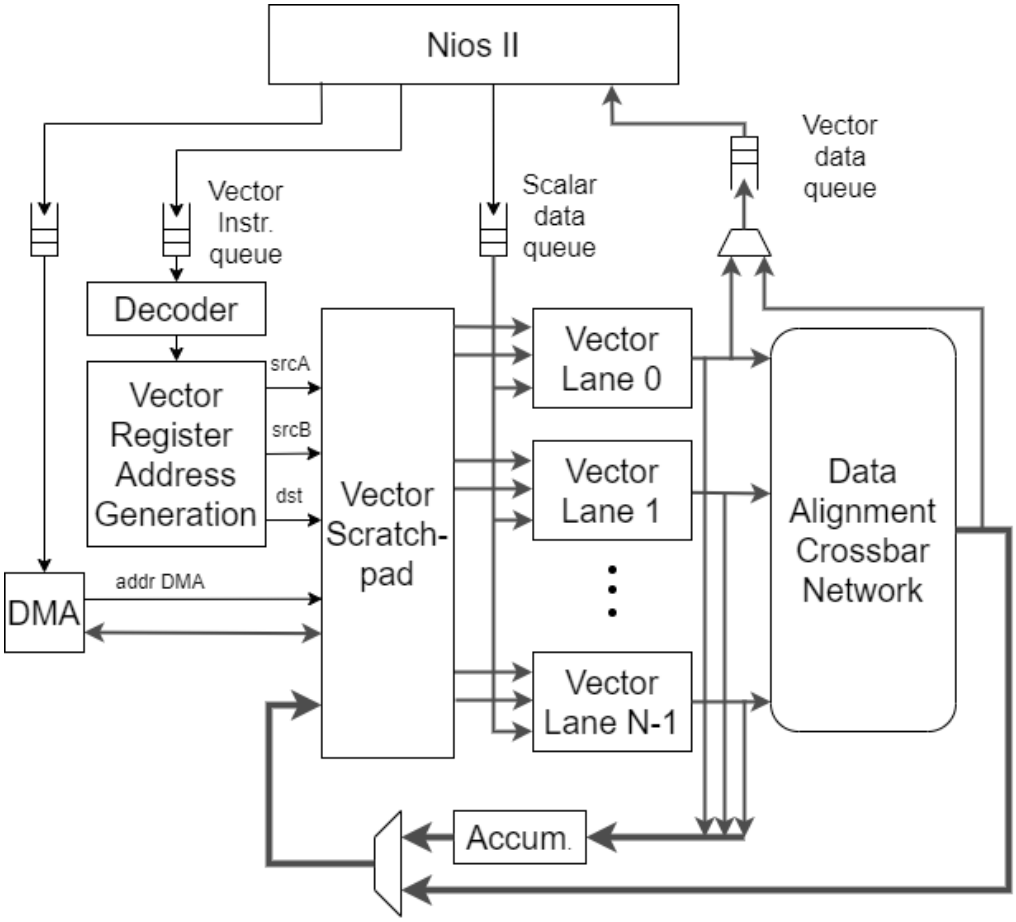


Figure 10: VEGAS Architecture [8]

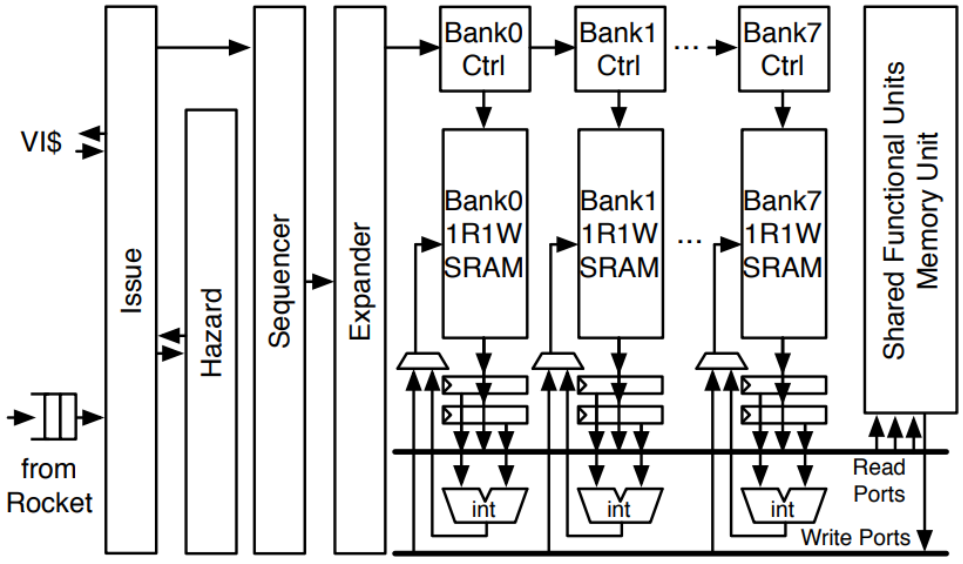


Figure 11: Hwacha Vector Accelerator Microarchitecture[9] © 2014, IEEE

proposal. Figure 11 shows the microarchitecture of the Hwacha vector accelerator. Scalar RISC-V core Rocket[2] delivers Hwacha a vector instruction at the scalar commit stage once all exceptions are cleared. The Hwacha issue unit fetches and decodes a vector instruction and waits until all the previous instructions-related hazards (e.g., data, structural, and bank conflicts) are resolved. Once all hazards are cleared, the vector instruction is issued to the sequencer. The sequencer sequentially executes a decoded vector instruction until all elements in a vector are executed. Vector instructions are divided into bank micro-operations (micro-ops) by the expander. Each bank performs the micro-operation on its own bank before transferring control synchronously to the next bank. There are eight banks of 1R1W SRAM in the vector register file (VRF).

Integer ALUs are directly coupled to each bank's read and write ports to reduce the structural hazards on integer execution units and boost integer performance. Larger functional units, such as the 64-bit integer multiplier and the single- and double-precision floating-point units, are shared by all banks. To further minimize size, the floating-point functional units are shared with the Rocket core. The memory unit allows vector gathers, scatters, and unit-strided and strided vector memory accesses.

Pluggable VPU[10] is based on RISC-V vector ISA extension v1.0. The block diagram is shown in figure 12. Vector instruction from the scalar core is fed to the vector processing unit through an instruction queue. The vector instructions are executed in a 3-stage pipeline - sequencer, execute, and write back. Sequencer decodes the instructions, creates micro-operations, and are sequenced and dispatched to the vector functional units for the execute stage.

The execution stage consists of VRF and functional units. The VRF, a multi-banked SRAM memory, receives the requests from each functional unit and arbitrates them by an allocator. The allocator implements a custom locking protocol to track data dependencies between micro-operations. Each functional unit performs SIMD Read Operands, SIMD Execute (specific to function), and SIMD Write-Back. The design features three parallel vector functional units - Vector Integer Unit (VINT performs the integer and fixed-point computations and integer reductions), Vector Permutation Unit (VMOV performs permutation and mask instructions), and Vector Load/Store Unit (VLSU for memory access instructions). These functional units can execute and write back out-of-order.

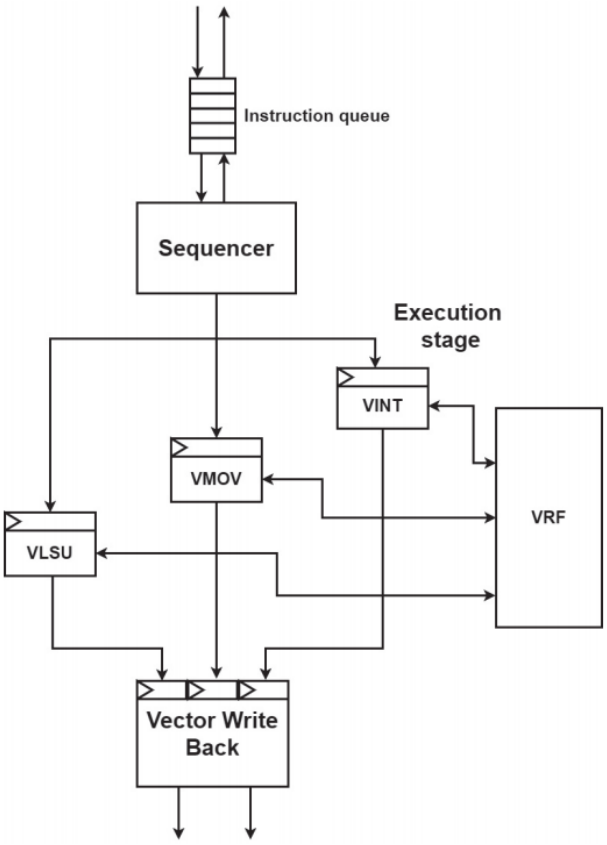


Figure 12: Block Diagram of the vector unit[10] © 2022, IEEE

2.4 Multicore Processor

The multicore processor contains multiple processing cores in a single chip that can execute its own programs individually. The world's first multicore processor was introduced by IBM in 2001, a VLSI (very-large-scale integration) chip containing two 64-bit microprocessors comprising more than 170 million transistors. Intel launched its first dual-core processor Intel Pentium Processor Extreme Edition 840, in 2005, which can process four threads.

The number of cores in a multicore processor varies depending on the architecture and target applications. These cores can be homogeneous (symmetric) cores; all of the cores are of the same type; or Heterogeneous (asymmetric) cores with different kinds of cores. Communication and coordination among the cores are performed through shared memory. Some multicore processor allocates dedicated local memory for each core. The cores can cache the shared memory; if so, the cache coherency techniques must be applied to keep data consistent. Heterogeneous multicore is becoming popular due to better performance

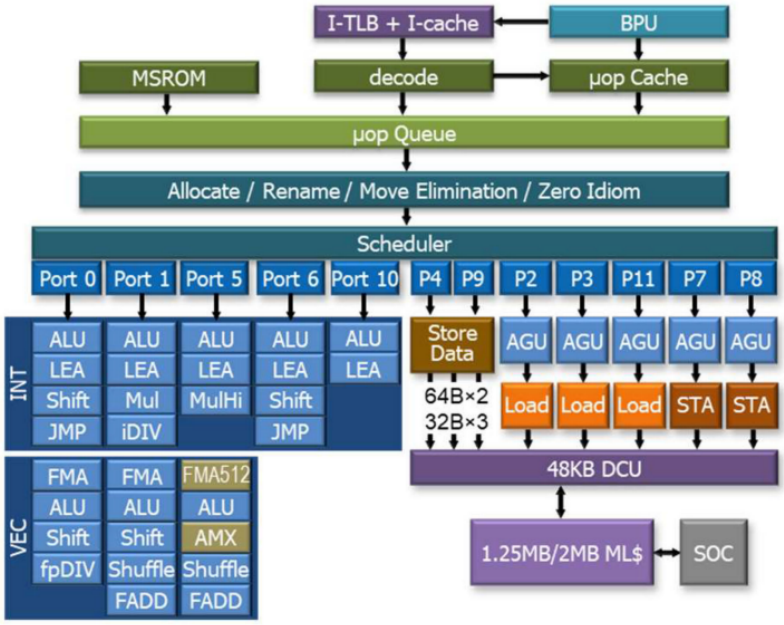


Figure 13: P-core block diagram[11] © 2022, IEEE

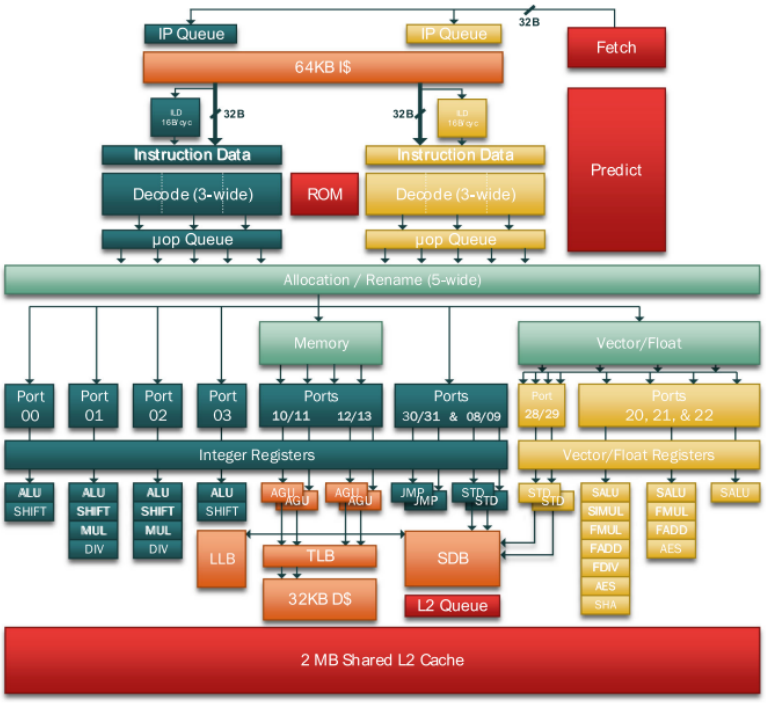


Figure 14: E-core block diagram[11] © 2022, IEEE

and energy consumption for different domain-specific applications.

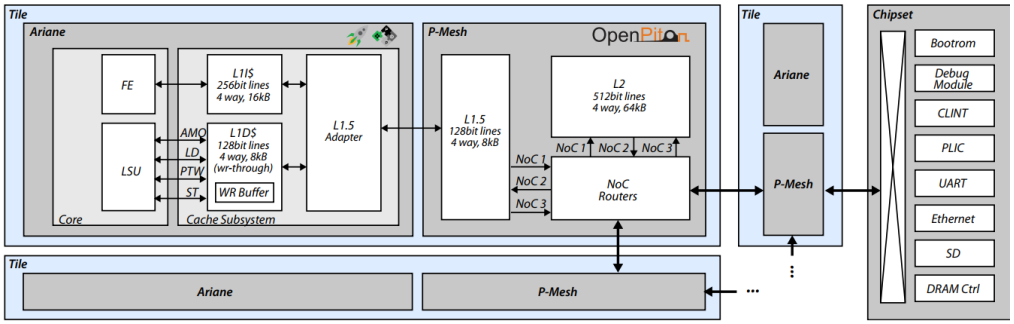


Figure 15: Overview of the OpenPiton+Ariane architecture[12]

Intel alder lake[11] is a recently launched CPU architecture to support hybrid CPU in multicore processors. They introduced a performance core (P-core, as shown in figure 13) to deliver higher IPC and essential core (E-core as shown in figure 14) with typical x86 architecture. Thus heterogeneity comes from accommodating multiple P-core and E-core in a single-chip processor.

The P-core microarchitecture supports deeper out-of-order schedulers and buffers, with more physical registers, a wider allocation window, and more execution ports. The P-core is enhanced with improved branch prediction. P-core architecture adopted new advanced matrix extensions, which define new registers and instructions that natively process matrix multiplication operations. Using these new instructions, p-core can achieve eight times better performance AVX-512 vector instructions. On the other hand, E-core is designed to achieve traditional x86 performance, with computing scalability for multithreading. Compared to the P-core, the E-core's geometric-mean performance for integer workloads is 80%, and the floating point computes performance is 62% of a P-core running at the same frequency.

OpenPiton+Ariane[12] is a tiled architecture that supports network-on-chip (NoC) with 2D mesh topology to interconnect a configurable number of processor tiles as shown in figure 15. It uses Ariane RISC-V core within the tiles, a 64-bit single-issue, in-order RISC-V core supporting RV64GC ISA. Each tile contains a local private cache L1.5, the NoC routers, and a slice of distributed, shared last-level cache L2. When a cache miss happens in L1.5, the L1.5 translates and forwards the request to the L2 cache. The cache coherence is maintained with a directory-based MESI coherence protocol. The coherence protocol operates between the private L1.5 cache, distributed shared L2 cache, and the memory controller. The directory information is embedded with the L2 cache line, and the L2 cache is inclusive of L1 and L1.5.

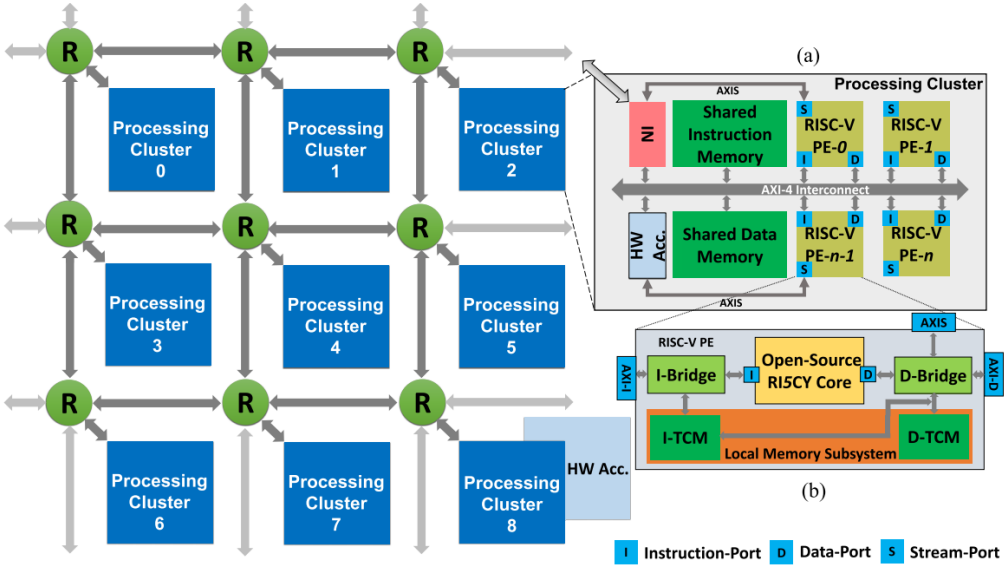


Figure 16: Overview of the RISC-V based many-core architecture (a) Processing cluster block diagram, (b) RISC-V PE block diagram[13]

RISC-V based many-core architecture [13] proposed a modular and hierarchical interconnect to design an accelerator for domain-specific and general-purpose applications on FPGA. It is a cluster-based architecture, as shown in Figure 16, which consists of a scalable number of processing clusters connected by a network-on-chip interconnect. The processing element (PE) consists of a soft-core processor RI5CY[32] and a local tightly coupled memory (TCM) subsystem for data and instructions as shown in Figure 16(b). The RI5CY core is a 4-stage in-order 32-bit pipeline processor. The core implements an RV32IMC ISA extension.

The processing cluster consists of multiple PEs with tightly coupled shared instruction and data memories connected by a shared bus interconnect. Therefore, the PEs can communicate between them by accessing shared memories and shared memory-mapped peripherals by utilizing common address space. The shared data memory is used for communication and synchronization between the PEs inside a single cluster, while the shared instruction memory is read-only memory. NoC architecture is used for inter-cluster communication through the router (R).

Each task is mapped to one or more PEs or processing clusters based on the computation requirements. The communication happens in a message initiated by the transmitting cluster and ending by the receiving cluster. Figure 17 shows a sequence diagram of synchronous message-based communication between two processing clusters. The transmission is ini-

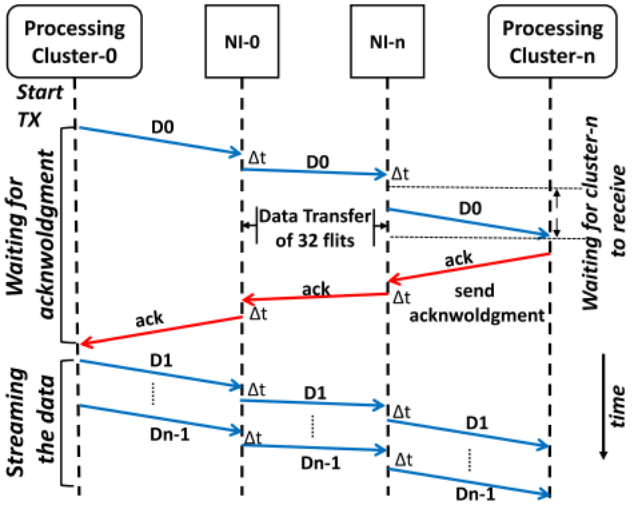


Figure 17: Sequence diagram of the synchronous message-based communication model[13]

tiated by any PE in the sender cluster with a packet containing the X-Y coordinate of the destination followed by 32 packet flits containing the first 32 payload packets. The sending PE is blocked until it receives an acknowledgment (ack) packet from the receiving cluster. Once the sending cluster receives the ack, it starts to stream the following data packet-flits, and the receiving PE in the receiving cluster is blocked until the successful receiving of the complete size of data. From this table, we can see that there is a need for processor and vector architecture, which will require fewer FPGA resources and higher operating frequency. Moreover, the architecture should be adaptable to the multicore architecture at lower FPGA resource overhead without degrading the operating frequency.

2.5 Summary

This chapter briefly introduces the RISC-V instruction set, different processor microarchitectures, and a few relevant works.

To increase the performance of the processor core, instruction level parallelism (ILP) is widely used in commercial microprocessors, utilizing the out-of-order super-scalar microarchitecture. But the hardware resource and complexity of such microarchitecture proliferates with the number of pipeline stages, number of parallel instruction issue/execute units, register renaming, etc. On the other hand, the in-order scalar processor is less complex and has relatively fewer hardware requirements.

For data level parallelism (DLP), vector processing is prominent due to less instruction

Table 1: Summary of reviewed architectures and FPGA adaptibility

Core	Architecture	Strength	Weakness
Rocket[2]	In-order scalar	Supports multiple extensions like RV32IMAFC	Higher logic usage for FPGA
CV32E40P[27]	In-order scalar	Less complex and smaller core	Lower operating frequency in FPGA
RVCoreP-32I[5]	In-order scalar	Lower logic usage and higher frequency in FPGA	Implements only RV32I extension
BOOM[29]	Out-of-order superscalar	Exploit ILP	Higher logic usage for FPGA
RSD[7]	Out-of-order superscalar	Exploit ILP and moderate FPGA logic utilization	Lower operating frequency in FPGA
VEGAS[8]	Vector Processing	Exploit DLP and designed for FPGA	Lack of standard ISA and tools support
Hwacha[9]	Vector Processing	Less complex	Not designed for FPGA
Pluggable VPU[10]	Vector Processing	Exploit DLP and designed for FPGA	Higher logic usage for FPGA
Intel alder lake[11]	Heterogeneous Multicore	Exploit ILP and DLP	Can not be used for FPGA
OpenPiton+Ariane[12]	Homogeneous multicore	Exploit TLP and cache coherent	Higher logic usage for FPGA
Many-core [13]	Homogeneous multicore	Exploit TLP and moderate FPGA resource	Lower operating frequency in FPGA

overhead and parallel data processing. It improves the program's performance when large regular data structures are being processed in the same manner. Vector processor architectures' performance is scalable. Vector processor performance can be boosted by increasing the number of vector lanes, which can perform more element operations in parallel. To achieve scalability, the vector processor's microarchitecture should be adaptable to adjust the number of lanes and vector length without much complexity and resources.

Heterogeneous computing is a widely used approach to distribute tasks among different kinds of devices like CPU, GPU, DSP, etc. There are two mainstream approaches to heterogeneous multicore design, one is heterogeneity obtained by incorporating different ISA-based processor cores or with different microarchitectures, and another one is coupling the custom accelerator to the processors. Software development for the first approach,

heterogeneous processors, is relatively more straightforward than the custom hardware accelerator.

In table 1, I have summarised the architectures discussed in this chapter. I have also listed the strength of these architectures and their weakness regarding FPGA implementation. The table shows that the in-order processor core requires lower FPGA resources than the out-of-order cores, which is a trade-off with ILP. There is a lack of Vector processors with standard ISA and potential architecture with fewer FPGA resources. As for heterogeneous multicore architecture, there is a need for flexible architecture with TLP and DLP while achieving higher frequency and lower resources. In this thesis, I have addressed the issues and proposed resource-efficient architectures, including scalar and vector processing and accomplishing the heterogeneous multicore.

3 The Resource Shared RISC-V Multicore Architecture

3.1 Motivation

The Internet of Things (IoT) devices at the edge are getting insistence to perform a diverse and processing-intensive range of tasks, including executing machine learning (ML) inferences. This has put stress on the microprocessor's operating frequency to maintain performance at the required level. However, the operating frequency is limited by the device technology. To address this issue, a more flexible approach to the design of multicore processors. It offers a potential solution with multiple processing cores built into a single FPGA.

Hard processors are available in modern FPGA, which are much faster than soft processors. However, they have a few drawbacks [33]. Firstly, the number of built-in hard processors in the FPGA is limited and may be sub-optimal for the application. Secondly, the configuration of each processor is fixed and is not configurable according to the application. Thirdly, each hard processor is placed in a fixed position, which may lead to difficulties in routing between the processors and the custom logic or accelerators.

The soft processor cannot easily match the performance, area, and power of a hard processor. However, soft processors have several advantages. Designers can implement the required number of soft processors for their applications. EDA tools can be used with different options to find the optimal place and route to maximize the throughput. Moreover, the developers can exploit heterogeneous computing, such as tightly integrating the custom acceleration functions to compete with the complexity of the application.

3.1.1 Purpose

In this work, I present a 4-stage and 5-stage in-order RISC-V soft processor for multicore architecture in FPGA. I have chosen the in-order scalar processor because out-of-order processors consume significant FPGA resources. The architecture is based on the idea of sharing hardware resources among the cores in a multicore system. I have named my proposed soft processor microarchitecture *RVCoreP-32IM*.

3.1.2 Contribution

The key contributions of this paper are as follows:

- I present a novel microarchitecture for RISC-V soft processor-based multicore sup-

porting RV32IM extension with a hardware resource-sharing option. My proposed architecture allows latency-insensitive execution of the shifter, multiplier, and divider as a result this architecture can share the shifter, multiplier, and divider in the multicore.

- I investigate the logic reduction for different hardware sharing options from the FPGA implementation results.
- I show a detailed study of the performance impact of this hardware sharing using the standard benchmark programs.
- I have shared the Instruction memory between 2 cores by utilizing dual port BRAM which reduces 50% of BRAM utilization for Instruction memory. I have also used the dual-port BRAM to implement the Data memory that is utilized for both local memories as well as shared memory for the multicore. This further reduces the BRAM utilization for Data memory.

3.2 Sharing Resource in Multicore

Sharing of large but less utilized resources between cores in a multicore processor has been promoted in the past by several studies [14, 15, 34, 35] to reduce silicon area at a marginal loss of performance. For instance, AMD implemented sharing of the entire floating-point unit including reservation stations and execution units in its BullDozer architecture [15]. In addition to FP units, several research publications [14, 34] suggest sharing of caches, crossbars, branch predictors, and large latency units.

Figure 18 shows the key concept of sharing hardware resources among the processor cores [14]. Resources such as the L1 instruction cache (IL1), the L1 data cache (DL1), and the branch predictor (BP) can be shared to reduce hardware costs. They also proposed to share long-latency functional units such as integer multipliers and dividers.

Figure 19 show the block diagram of the AMD Bulldozer processor presented in [15]. The microarchitecture has shared the issue queue and the floating-point datapath with two cores. They have also shared the L2 cache.

The graphics processing unit (GPU) also shares resources among the multiprocessors. As an example, figure 20, shows the Nvidia GPU of Fermi architecture which has a shared L1 cache and special functional unit (SFU) in the streaming multiprocessor (SM). SFU can handle special operations such as sin, cos, exp, and rcp (reciprocal).

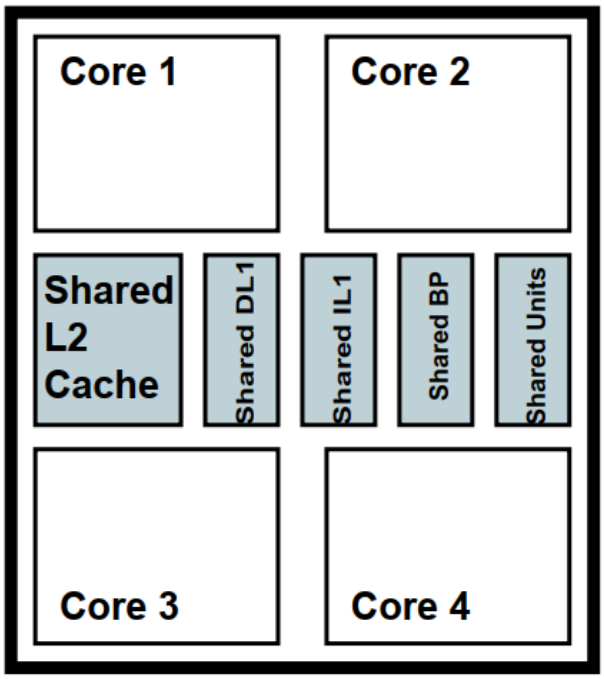


Figure 18: Diagram of the sharing proposed in [14]

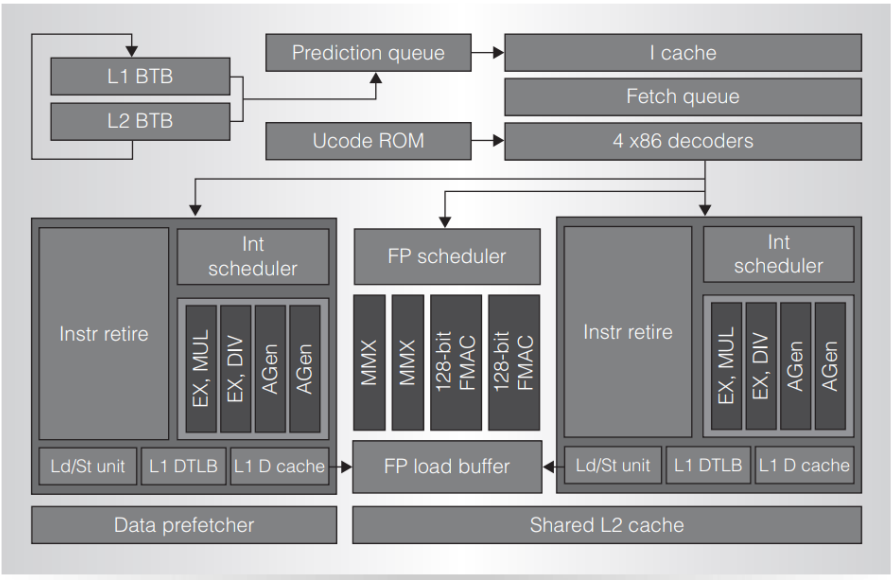


Figure 19: High-level block diagram of the Bulldozer module [15] © 2011, IEEE

However, the FPGA is a resource-constrained device. The multicore implementation for FPGA needs special consideration of logic utilization and achievable maximum operating frequency. The soft processors used for FPGA are usually scalar RISC processors due to the reduced logic consumption for implementation. Many embedded applications rely

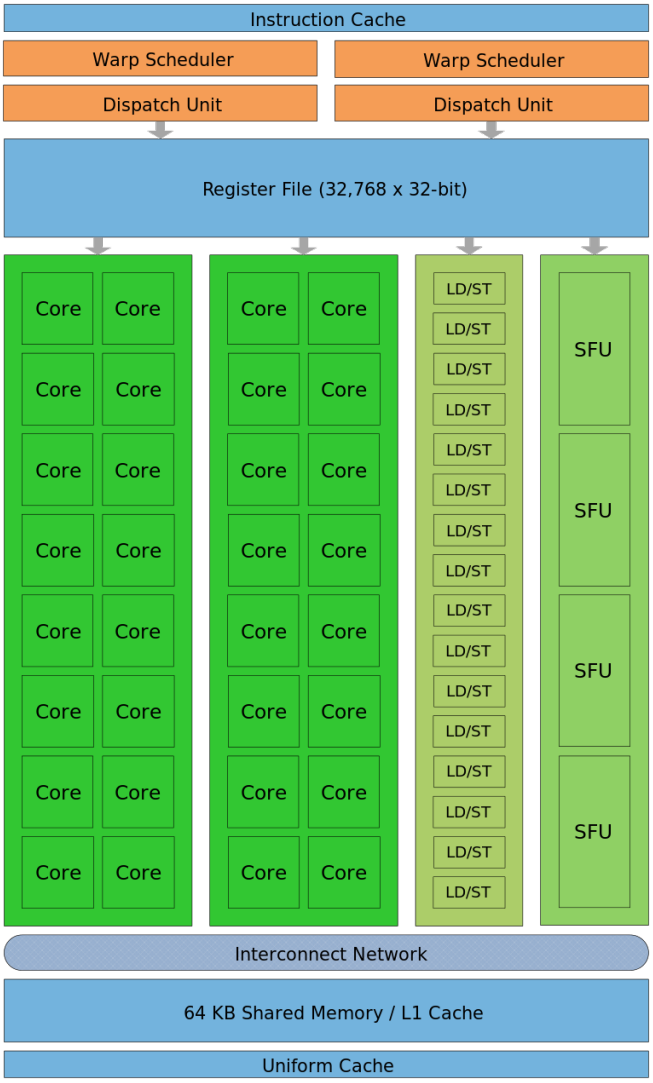


Figure 20: Nvidia Fermi architecture of streaming multiprocessor[16]

on integer computation only where floating-point units might not require. For resource sharing among multicore, the functional units which have larger logic are and less utilized should be chosen. To share the functional units the microarchitecture/pipeline of the processor must have to be adapted to the sharing requirements.

Implementation of cache-based multicore design might not be practical due to the logic overhead in the cache controller. On the other hand, the BRAM available in the FPGA can be used as tightly coupled memory (TCM). Rather than sharing a cache, sharing TCM would be more beneficial for the FPGA-based multicore architecture.

3.3 Microarchitecture of RVCoreP-32IM

Figure 21 shows the 4-stage and 5-stage pipeline diagram whose units are IF for instruction fetch, ID for instruction decode, EX for execute, ME for memory, MUL for a multiplier, DIV for a divider, SHIFT for the shifter, and WB for the write back.[1] [2] The processor core supports instructions of RV32IM extension.

Figure 21(a) shows the conventional pipeline stages where each stage takes a single cycle. For RV32I instructions implementation, it is easy to consider that every stage will take only one clock cycle. On the other hand, it is unreasonable to assume the single clock cycle execution unit to implement M-extension (that is, performing multiplication and division operations in a single clock cycle). Because it will affect the maximum operating frequency for the FPGA.

In the 5-stage pipeline of VexRiscv (shown in figure 21(b)), apart from the integer execution unit, the shifter, multiplier, and divider results are forked and committed into the write-back stage. This pipeline allows the shifter, multiplier, and divider to perform operations in multiple clock cycles. In my proposed RVCoreP-32IM, I consider the execution stage as multi-cycle execution units (i.e. some instructions take more than 1 clock cycle to execute) for both 4-stage and 5-stage pipeline configurations.

Figure 21(c) presents the 5-stage pipeline configuration where, integer ALU, shifter, multiplication, and division units are forked in the execution stage and then joined into the memory stage. And the 4-stage pipeline is shown in 21(d), where integer ALU, shifter, multiplication, division, and memory units are forked in the execution stage and then joined into the write back stage

In the microarchitecture description, I use the following abbreviated terms

- AGU is an address generation unit for memory instructions.
- BRU is a branch unit for branch instructions.
- ALU is the arithmetic and logical unit for RV32I integer arithmetic and logical instructions.

[1]Copyright © 2021, IEEE, Reprinted, with permission, from Md Ashraful Islam, Kenji KISE, Efficient Resource Shared RISC-V Multicore Processor, 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), December 2021.

[2]Copyright © 2022, IEICE, Reprinted, with permission, from Md Ashraful ISLAM, Kenji KISE, An Efficient Resource Shared RISC-V Multicore Architecture, IEICE Transactions on Information and Systems, 2022, Volume E105.D, Issue 9, Pages 1506-1515

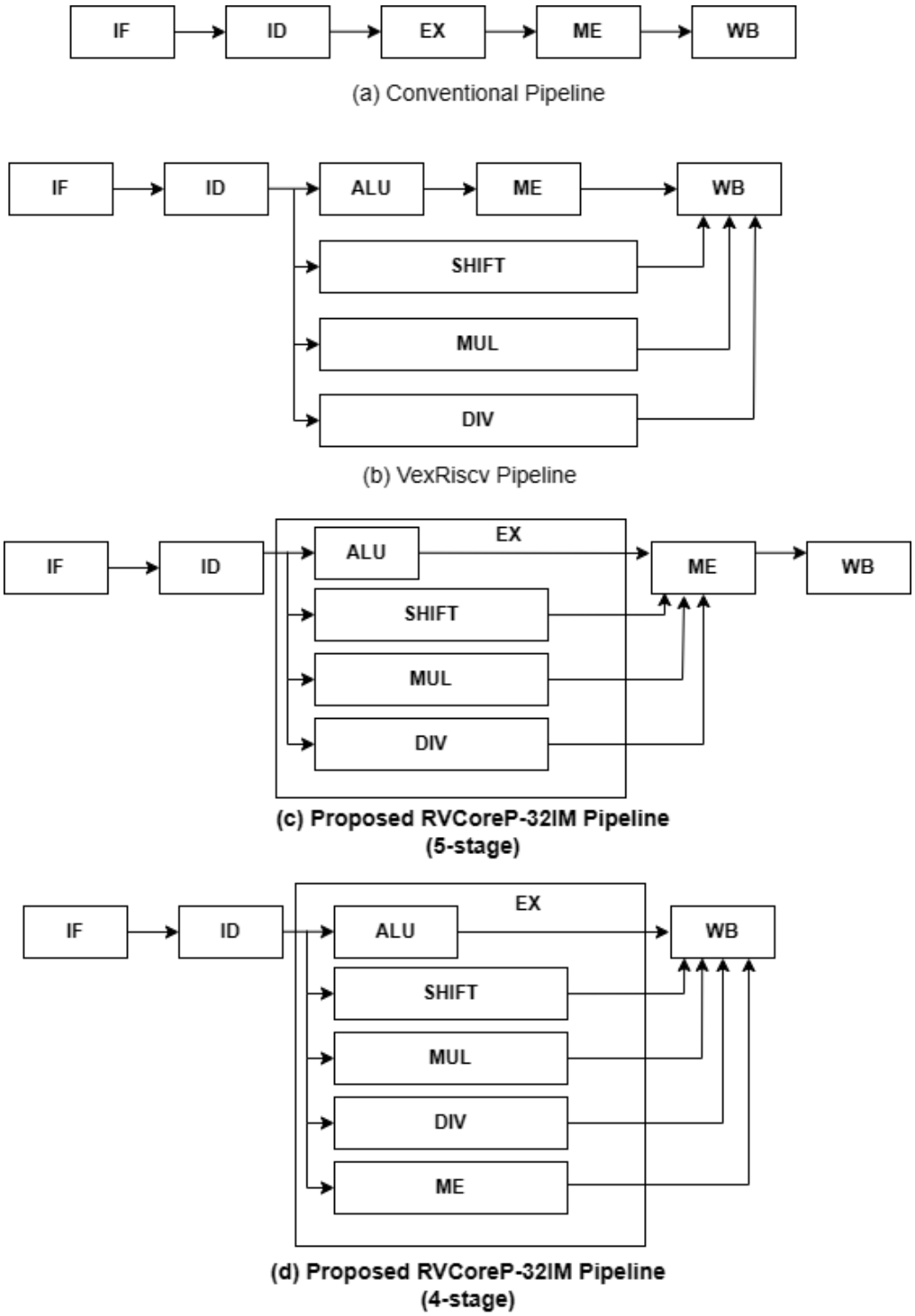


Figure 21: The pipeline diagrams of microprocessors

- SHIFT is a shifter for left and right shifts.
- MUL is a multiplier for multiplication instructions.

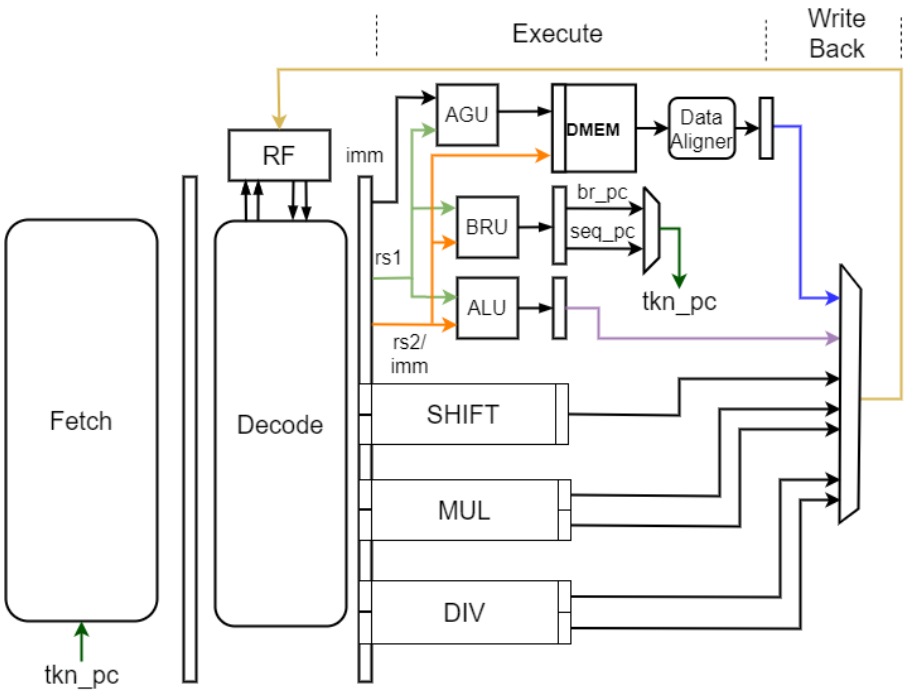


Figure 22: The microarchitecture of the 4-stage RVCoreP-32IM

- DIV is a divider for division and remainder instructions.

Instruction fetch units for 4-stage and 5-stage pipeline configurations are the same which uses the pipelined gshare branch prediction as proposed in RVCoreP[5]. According to RVCoreP, the fetch and decode stage is stalled only when there is a data dependency hazard due to load instructions. However, RVcoreP-32IM stalls when there is a data dependency that exists with hazard, as well as execute stage performs a multi-cycle operation.

Figure 22 shows the microarchitecture of the 4-stage pipeline RVCoreP-32IM consists of fetch, decode, execute, and write back. Execute stage performs branch/jump, arithmetic, logical, shift, multiplication, division, load, and store operations.

Execute stage of the 4-stage pipeline does not have data forwarding logic, therefore, the fetch and decode stage stalls when there is a data dependency in two consecutive instructions. Memory operations (i.e. load and store) take 1 clock cycle for address calculation and at least 1 clock cycle to perform read or write operations to memory. Division operations take 3 (if any operand is zero) or 35 cycles to calculate the quotient and reminder. As a result, when execute stage performs memory or division operations, the fetch and decode stage also stalls until their operations are done. All other operations take a single clock cycle to execute.

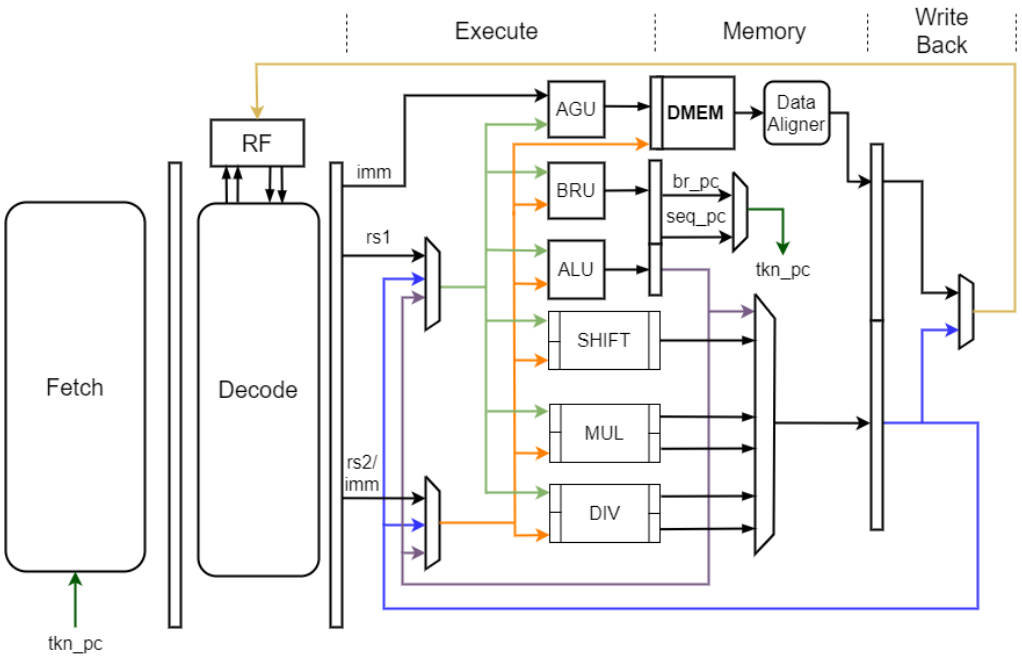


Figure 23: The microarchitecture of the 5-stage RVCoreP-32IM

In the 4-stage pipeline of cv32e40p[27], execute stage results are directly passed to the register file for writing. While in my proposed architecture I first capture the execute stage result into the pipeline register and then write it back to the register file. I used this pipeline register to improve the clock frequency.

Figure 23 shows the microarchitecture of 5-stage RVCoreP-32IM consists of fetch, decode, execute, memory, and write back. Results from ALU, shifter, multiplier, and divider of execute stage are stored in pipeline registers and then multiplexed into the memory stage. Write back stage multiplexes data from load data and data result from other instruction execution. The 5-stage pipeline has a data forwarding path from the memory stage containing ALU results and from the write back stage containing execution results except for load data.

To avoid the critical timing path in the execution stage, the shifter, multiplier, and divider are also pipelined. As a result shift and multiply operation takes 2 clock cycles and the divide operation takes 4 and 36 clock cycles in execute stage. Since the results from the shifter, multiplier, and divider from the memory stage are not forwarded into execute stage, the fetch and decode stage will be stalled 1 additional clock cycle when the next instruction has a data dependency on these results.

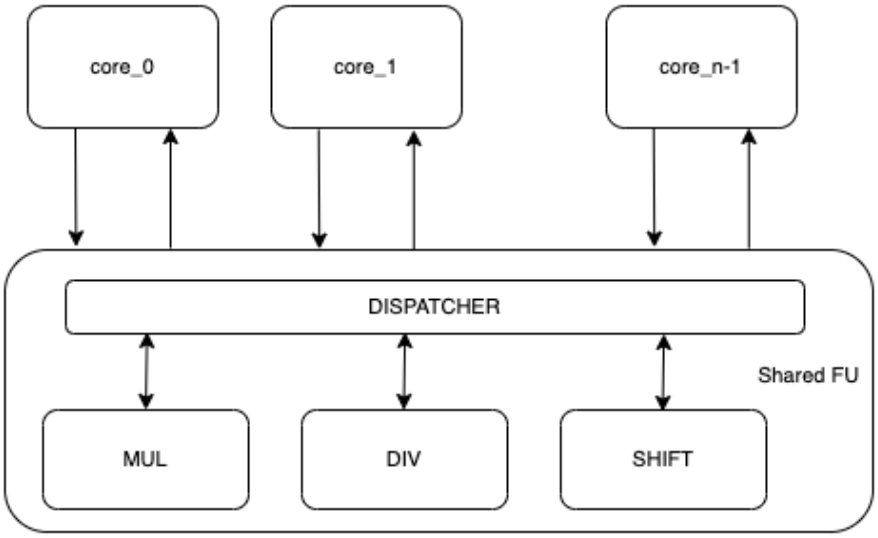


Figure 24: The sharing functional units among multiple cores

3.3.1 Latency Insensitive Execution Unit

In my proposed architecture execution stage requires multiple clock cycles for some operations regardless of pipeline stages. To support such different latency for different operations I implemented an execution unit as latency insensitive[36]. I use the elastic protocol SELF as proposed in [37], with a simple fork and join control based on “valid" and “stall" signals. The “valid" signal propagates forward to indicate that new data is available for the next stage. And the “stall" signal propagates backward to signal the previous stage to halt and insert a bubble.

In RVCoreP-32IM, the decode stage passes on the decoded signals to the execution stage when a valid instruction is decoded. If the decoded instruction is a single-cycle instruction, then in the next clock cycle, the execution results from ALU, BRU, and AGU will be available to the memory stage. When the decoder decodes multiplication, division, and shift operations it asserts “mul_op", “div_op", and “shift_op" signals to the execution stage. These signals act as valid signals for the multiplier, divider, and shifter units. When a mul, div, or shift operation is requested in the execution stage, the fork-join controller in execute stage asserts a stall signal to the decode stage and keeps asserting until the mul, div, or shift operation is done and outputs valid data for the memory stage. The stall will be one clock cycle for mul and shift operation and 34 clock cycles for div operation in my design.

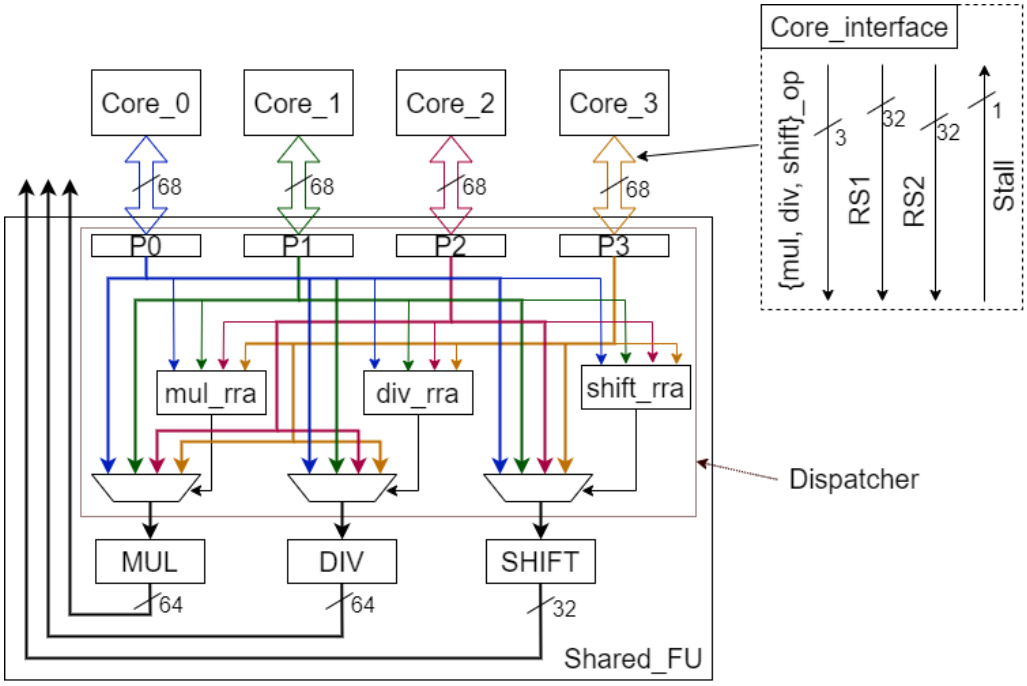


Figure 25: The shared functional unit (FU) for four cores

3.3.2 Sharing Functional Units

The multiplication, division, and shift operations require a considerable amount of resources compared to the ALU, and they are used less frequently than ALU. Therefore, I propose a shared FU (Functional Unit) that will contain these units like a multiplier, divider, and shifter that would be shared among multiple cores as shown in Figure 24. Since my microarchitecture is the in-order scalar processor, one core will request any function at a given time and will not request other functions until the previous operation is done. This characteristic motivates us to share the multiplier, divider, and shifter among the cores.

Each core can independently request any functional unit by asserting their corresponding mul/div/shift_op signal to the dispatcher. As I have mentioned that my core execution unit is latency insensitive, so the corresponding core's fork-join controller will stall until the valid out is returned from the shared FU. The dispatcher unit in the shared FU arbitrates the request from multiple cores. If there is a contention between multiple cores requesting the same functional unit, then the dispatcher arbitrates the request based on a round-robin scheme.

Figure 25 is the shared FU architecture for four cores. Each core request to shared FU through their associated interface channel (Core_Interface). The request for a specific

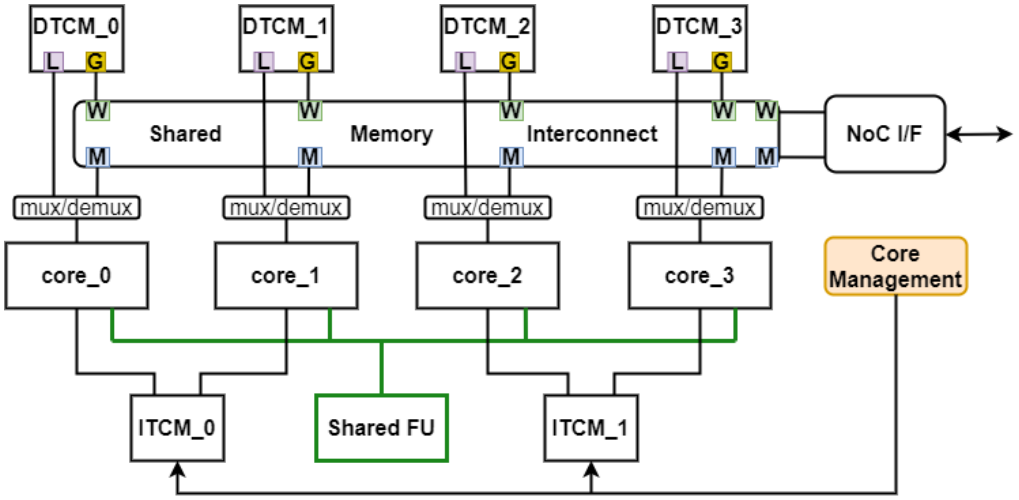


Figure 26: The Shared multicore architecture

functional unit like multiplication, division, or shift is signaled by mul, div, shift_op. The RS1 and RS2 of core interface signals are 32-bit operands for the requested operation. The stall signal indicates that the requested operation result is not available yet, which is used by the fork_join_control in the associated core's execution unit.

Core interface signals are captured by port register (P0 to P3), which are then requested to associated functions like a multiplier, divider, or shifter's round-robin arbiter (mul_rra, div_rra, shift_rra). According to the arbitration result, operands from port registers are multiplexed to the input of the functional unit.

In my design, I can configure the sharing options for the individual functional units. For example, I can configure to have MUL and DIV only in the shared FU or DIV and SHIFT only in the shared FU, and so on.

3.4 Multicore Architecture

Figure 26 shows my multicore architecture for 4 cores (core_0 to core_3). Despite the instruction and data cache, I use Instruction Tightly Coupled Memory (ITCM) and Data Tightly Coupled Memory (DTCM) directly connected to cores owing to the fact that embedded applications memory utilizes small on-chip RAM. All the recent FPGAs have embedded Block RAM (BRAM) that can be configured as either single-port or as dual-port RAM. Therefore DTCM and ITCM are composed of BRAMs, which I configured as dual-port RAM. It does not have memory address translation so the memory addresses are directly mapped.

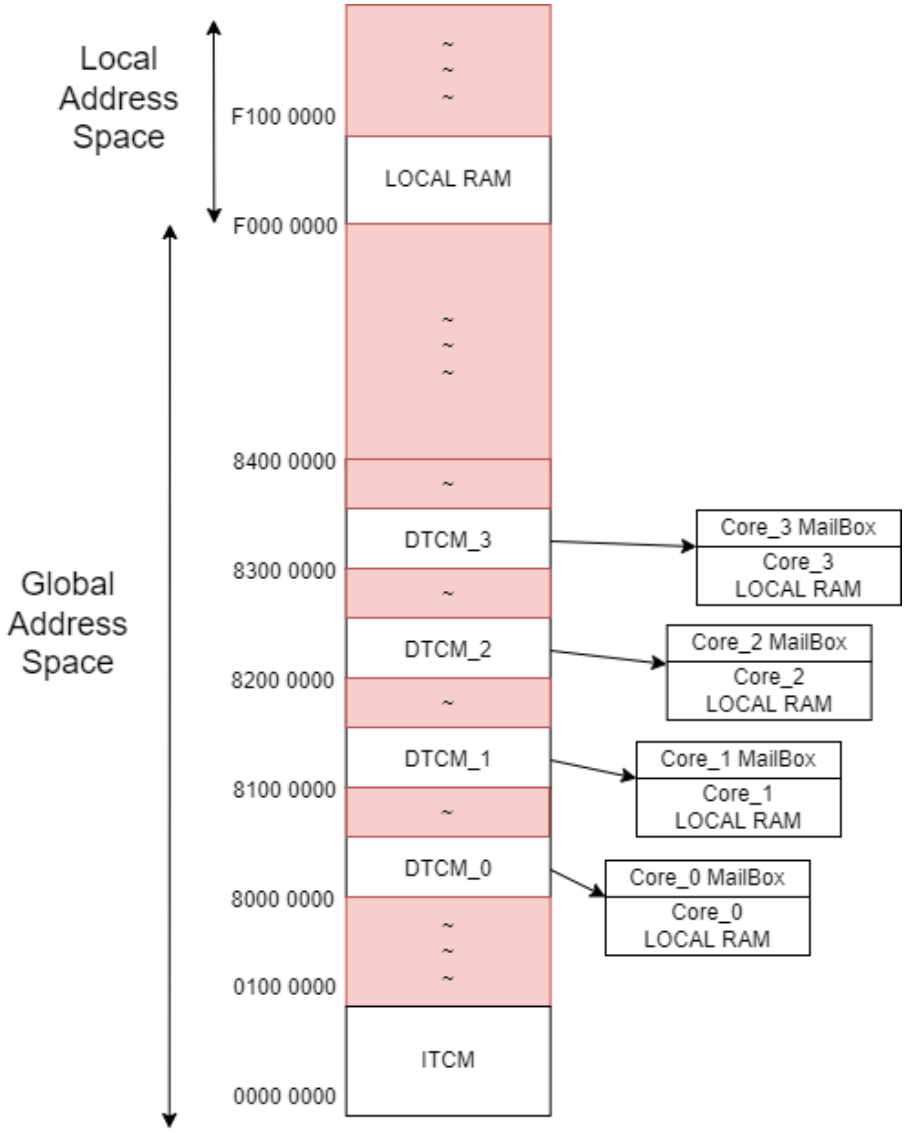


Figure 27: The simplified view of memory regions

As ITCM has dual-port BRAM, I have shared ITCM among two cores, ITCM_0 is connected to core_0, core_1, and ITCM_1 is connected to core_2, core_3 instruction fetch unit. This reduces the BRAM utilization for ITCM to 50%.

It does not have any memory address translation therefore all the memories have fixed address allocation. I utilize the dual-port BRAM of DTCM as a local port and global port, which is shown in figure 26 as L and G. The same DTCM memory can be accessed through the local address (L port) and global address (G port). Each core is connected to the associated DTCM local port which provides a low-latency access to that DTCM when accessed through the local address. From Figure 26, core_0 is connected to DTCM_0

local port, core_1 is connected to DTCM_1 local port, and so on. For a given core, the local address is an alias of its associated DTCM global address.

In order to perform the memory operations on these DTCMs using global address space, I employ shared memory interconnect. In Figure 26, “M" represents the master port, and “W" represents the worker port for the shared memory interconnect. DTCM’s global port is connected to this shared memory interconnect. I use the crossbar as shared memory interconnect. Moreover, increasing the number of cores in the system can be done by adding several such multicores connected to Network-on-Chip (NoC) through the network interface.

To give an illustration of memory address partition I presented an example of memory mapping in Figure 27. Each core’s local addresses are the same but physically located in different regions in the global memory. The benefit of this address scheme is when utilizing the local address the core directly accesses the BRAM, minimizing access latency. During global memory accesses depending on the destination and arbitration at interconnect, it may have several cycles of latency. Another aspect of local address space is simplifying the program development which utilizes the local memory.

Shared memory interconnect enables the cores to send and receive messages from other cores, which increases the flexibility of core-to-core communication. As shown in Figure 27, DTCM memories can be divided into the local ram and mailbox region. How much memory will be allocated for these regions is user-defined (like declaring into the linker script and defining into a C header file). In this architecture, I have not implemented the logic to support atomic instructions. However, the message passing among the core facilitates the developer to perform the data transfer between processes executing in different cores.

The core management performs the reset tasks and loads the program binary into the ITCM. Loading the same program on ITCM_0 and ITCM_1 by core management lets the 4 cores execute the same program but on a different portion of data and their branch of instruction execution sequence can be different. Which offers the Single Program Multiple Data (SPMD) programming models. However, Core management can load 2 different programs on ITCM_0 (for core_0, core_1) and ITCM_1 (for core_2, core_3) to execute 2 different programs. In other words, the architecture can be utilized for Multiple Program Multiple Data (MPMD) programming as well.

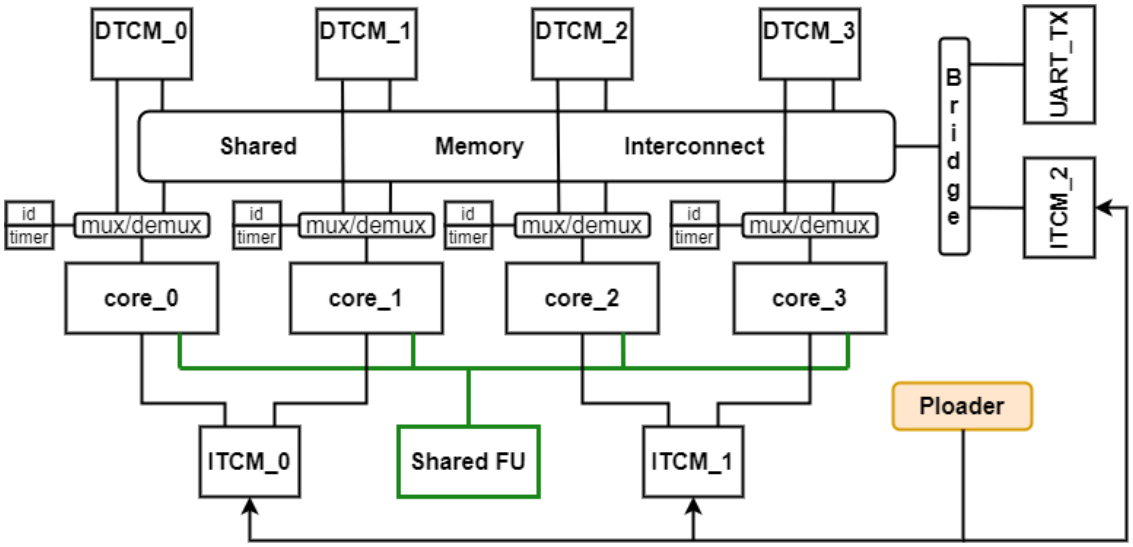


Figure 28: The simple SoC for evaluation in FPGA

3.5 Evaluation

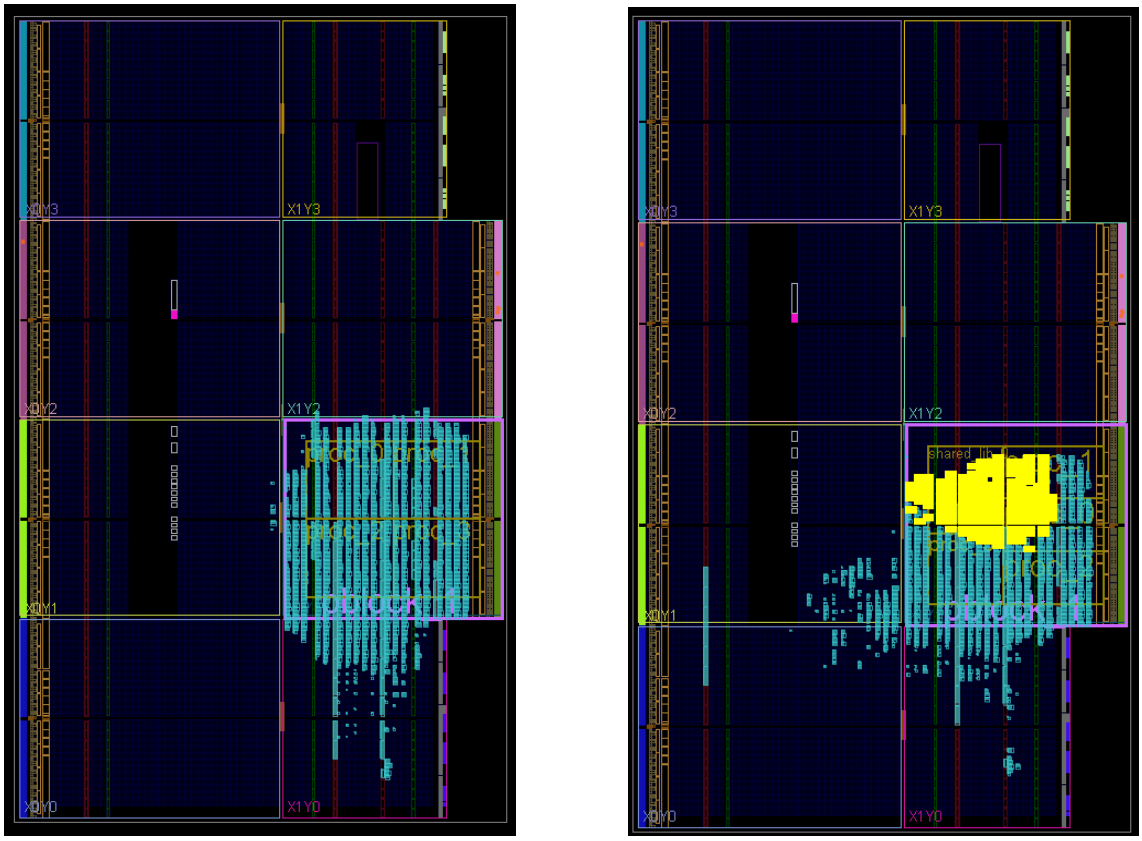
3.5.1 FPGA Implementation of Shared Multicore

RVCoreP-32IM uses a two-stage pipelined gshare branch prediction with a pattern history table (PHT) of 8,192 entries and a BTB of 512 entries. We prepare a simple SoC to evaluate the FPGA implementation of these processors. The SoC block diagram is shown in Figure 28, where a UART transmitter (UART_TX) and an instance of ITCM are connected to multicore. Each core has its own timer and id register. The id register returns the unique id of individual cores like 0 for core_0, 1 for core_1, and so on.

We have a single depth buffer between the interface of shared memory interconnect to CPU cores. Shared memory interconnect has a crossbar. The minimum latency for load and store instruction execution is 3 clock cycles (1 clock for buffering at shared memory interconnect, 1 clock for BRAM access, and 1 clock for memory stage execution).

A RISC-V program binary is loaded into ITCMs by the ploader which writes to the BRAMs of ITCM. Once the ITCMs are initialized by the ploader the cores resets are de-asserted and start execution.

I evaluate the operating frequency and hardware resource utilization for Nexys A7 board [38] containing xc7a-100tcsg324-1 FPGA, which is a family of Xilinx Artix-7 FPGA of speed grade -1. In this FPGA for 32-bit data width, BRAM size is 4KB each. Xilinx Vivado 2021.1 is used to evaluate operating frequency and hardware resource utilization.



(a) no_shared configuration

(b) s_m_d shared configuration

Figure 29: Artix-7 FPGA implementation layout for the 5-stage 4-core processor. A shared functional unit is highlighted in yellow color

Synthesis, placement, and routing strategy were set by Vivado default.

The design uses the physical constraints to place all four cores and shared functional units in the same clock region X1Y1 which contains 7600 LUTs and 40 DSP blocks. I performed the logic synthesis and placement, and routing by incrementally stepping the clock constraint in the 5MHz scale. The highest frequency that achieves the positive timing slack is used as the operating frequency in the table. For hardware resource evaluation, I used the result of placement and routing at the maximum operating frequency. I set both ITCM and DTCM sizes to 32KB for FPGA implementation. Figure 29 shows the FPGA implementation layout of a 5-stage multicore processor without sharing and with sharing (s_m_d) configuration.

FPGA implementation result is shown in Table 1, where I take LUT, register, and DSP as FPGA resources. In Table 1, the logic resources required by cores and shared functional units (FU) have shown separately. For the shared FU the resources required for the shifter,

multiplier, and divider have been shown in shift, mul, and div rows. And the resources required for arbitration and data multiplexing and demultiplexing are shown in the table as “dispatch”, which I can consider the sharing overhead. When s_m_d configuration is used the sharing overhead is higher, which is expected.

For the multiplier sharing, the DSP block usage is significantly reduced to 75% compared to no_shared. The reductions of LUT and register utilization for different sharing configurations compared to 5-stage no_shared are presented in Figure 30. The figure reveals that the “s_m_d" and “s_d" configuration substantially lowers the LUT usage for both 4-stage and 5-stage.

Table 1: The FPGA implementation result of multicore (4-core) for different sharing configuration

			no_shared		s_m_d		m_d		d		s_d	
			4-stage	5-stage	4-stage	5-stage	4-stage	5-stage	4-stage	5-stage	4-stage	5-stage
LUT as Logic	Cores		5165	**5967**	2737	**3070**	3904	4112	4148	4427	3213	3405
	Shared FU	shift	0	0	276	**282**	0	0	0	0	272	288
		mul	0	0	166	**204**	201	200	0	0	0	0
		div	0	0	508	**455**	457	338	466	345	335	459
		dispatch	0	0	407	**396**	309	305	158	146	256	249
	Total		5165	5967	4094	**4407**	4871	4955	4772	4918	4076	4401
Slice Registers	Cores		3692	3685	2701	2885	2705	2881	2849	3021	2845	3029
	Shared FU	shift	0	0	82	82	0	0	0	0	82	82
		mul	0	0	44	44	44	44	0	0	0	0
		div	0	0	211	211	211	211	211	211	211	211
		dispatch	0	0	316	316	296	296	280	280	300	300
	Total		3692	3685	3354	3538	3256	3432	3340	3512	3438	3622
DSP			16	16	4	4	4	4	16	16	16	16
Fmax (MHz)			115	100	115	115	115	115	115	115	115	115

I checked the maximum operating frequency for all configurations which is shown as Fmax in the table 1. Interestingly I found that only the 5-stage no_shared configuration achieved 100MHz and all other configurations achieved the same frequency which is 115MHz.

3.5.2 Benchmark Simulation

Dhrystone and Coremark is a synthetic benchmark program used for a long time in the industry and academy. Opposed to the synthetic benchmark, the Embench is relatively new and was released in 2019. It provides actual application programs to exercise branch, memory, and integer compute type instruction intensively. Due to this variability of an instruction stream, I use Embench for performance study by simulation.

To compile the program, I use GCC tool version 9.2.0 with machine ABI switch “-mabi=ilp32" and optimization switch “O2". We use the Verilator [39] tool for simulating RTL with generated program binary.

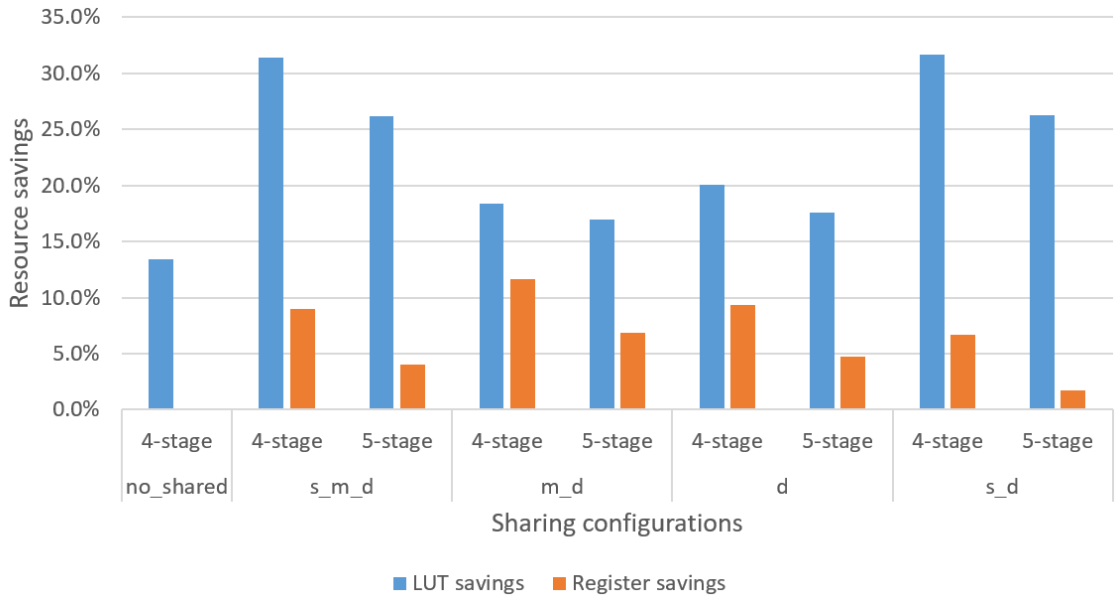


Figure 30: The reduction of FPGA resource utilization for resource sharing

In this study, I particularly explore the comparatively worst case of performance penalty for resource sharing. All four cores are executing the same benchmark programs, so they request the same functional units at the adjacent time. For each benchmark program simulation, I measure the number of clock cycles required to execute the program. Then the execution time for the benchmark is calculated from the number of cycles required for simulation and the clock period. The clock period for RVCoreP-32IM is obtained from Table 1. I then normalized the execution time of each configuration by comparing it with the 5-stage no_shared execution time.

In Table 2, I have shown the CPU clock cycles required to execute corresponding Embench programs by single-core VexRiscv, 5-stage, and 4-stage RVCoreP-32IM. I have shown the 5-stage and 4-stage RVCoreP-32IM performance gain with respect to VexRiscv by comparing the execution time. To measure the execution time, I took the maximum operating frequency of single-core VexRiscv, 5-stage, and 4-stage RVCoreP-32IM implementation, which are 143MHz, 164MHz, and 125MHz, respectively. From Table 2, I can see that 5-stage RVCoreP-32IM has achieved the highest performance compared to VexRiscv.

To quantify the multicore performance, I used single-core 5-stage RVCoreP-32IM as a baseline. Embench simulation cycles and performance gains are shown in Table 3. Single-core processor is directly connected to the BRAM and load/store instruction can execute in a single clock. On the other hand, the multicore processors are connected via shared memory interconnect and require at least 3 clock cycles to execute load and store

Table 2: Single-core Embench program simulation cycles (in Million) and performance gain compared to VexRiscv

	VexRiscv	5-stage		4-stage	
program	cycles	cycles	gain	cycles	gain
mont64	5.292	5.065	1.20	5.925	0.78
crc_32	5.086	4.212	1.39	7.368	0.60
libedn	4.399	4.929	1.02	7.578	0.51
libhuffbench	3.511	3.481	1.16	5.974	0.51
matmult-int	4.786	5.239	1.05	9.364	0.45
libminver	7.064	7.039	1.15	9.278	0.67
nbody	7.820	7.870	1.14	9.721	0.70
nettle-aes	4.846	4.872	1.14	8.783	0.48
nettle-sha256	4.234	4.179	1.16	6.499	0.57
libnsichneu	4.185	4.482	1.07	6.627	0.55
libpicojpeg	5.220	4.952	1.21	9.830	0.46
qrduino	4.407	4.441	1.14	7.463	0.52
combined	3.984	4.010	1.14	7.042	0.50
libslre	3.225	3.325	1.11	6.038	0.47
libst	5.471	5.492	1.14	7.051	0.68
libstatemate	1.946	1.921	1.16	3.982	0.43
libud	5.277	5.386	1.12	6.931	0.67
libwikisort	3.729	3.740	1.14	5.627	0.58

instructions (1 clock for buffering at shared memory interconnect, 1 clock for BRAM access, and 1 clock for memory stage execution). As a result, a multicore processor requires higher CPU cycles compared to a single core. For the multicore performance comparison, I have considered the same program execution for four times.

We calculated the geometric mean of the performance gain and found that single-core 5-stage RVCoreP-32IM is 1.15x better than VexRiscv. For multicore, 5-stage RVCoreP-32IM is 2.1x and 4-stage RVCoreP32IM is 2x better than single-core 5-stage RVCoreP-32IM.

Table 4 shows each benchmark program's ratio of executed shift, multiply, divide, and other instructions. This table helps us to correlate the sharing effect in the multicore program execution. In Table 5, I presented the Embench program simulation results for the multicore with different sharing configurations. From Table 5, I can find the effect of sharing on the execution cycles and pertain to Table 4. For instance, the libnischneu program has a negligible effect on sharing because shift, mul, and div instructions ratios are imperceptible. On the other hand, the mont64 program has a significant amount of shift instructions; as a result, the s_m_d and s_d configuration has much more effect on

Table 3: Multicore Embench program simulation cycles (in Million) and performance gain compared to single-core 5-stage RVCoreP-32IM

	Single-core	Multicore			
	5-stage	5-stage		4-stage	
program	cycles	cycles	gain	cycles	gain
mont64	5.065	5.248	2.70	5.964	2.73
crc_32	4.212	4.915	2.40	7.544	1.80
libedn	4.929	7.549	1.83	7.923	2.00
libhuffbench	3.481	5.064	1.92	5.974	1.88
matmult-int	5.239	8.071	1.82	9.742	1.73
libminver	7.039	8.011	2.46	9.378	2.42
nbody	7.870	8.484	2.60	9.929	2.55
nettle-aes	4.872	7.216	1.89	8.882	1.77
nettle-sha256	4.179	4.865	2.41	6.500	2.07
libnsichneu	4.482	6.165	2.04	6.627	2.18
libpicojpeg	4.952	7.673	1.81	9.952	1.60
qrduino	4.441	6.119	2.03	7.530	1.90
combined	4.010	6.017	1.87	7.042	1.83
libslre	3.325	5.161	1.80	6.038	1.77
libst	5.492	6.084	2.53	7.187	2.46
libstatemate	1.921	3.249	1.66	3.982	1.55
libud	5.386	6.307	2.39	7.036	2.46
libwikisort	3.740	4.927	2.13	5.684	2.12

execution cycles. Similarly, the libwikisort program is also influenced by the division instructions execution ratio.

Figure 31, 32, 33, and 34 represents the performance effect as normalized execution time of individual benchmark programs from my simulation results. In these figures, all the sharing configuration uses the 5-stage pipeline no_shared configuration's execution time of the corresponding benchmark program as reference performance or speedup as 1.0. If any program in a sharing configuration requires more time to execute than 5-stage no_shared, then the performance will be less than one and vice versa.

This figure shows that m_d and d sharing configurations performance are comparable to the 5-stage no_shared configuration. For s_m_d and s_d sharing configuration, most programs have notable performance drops.

For various sharing configurations, nearly all programs executed faster in a 5-stage pipeline than in a 4-stage pipeline. In some cases sharing configuration runs faster than no_shared configuration due to the increased frequency of FPGA implementation.

Table 4: Shift, multiplication, and division instructions ratio in Embench program execution

Program	Shift	Mul	Div	Others
mont64	**34.0%**	0.8%	0.0%	65.2%
crc_32	21.7%	4.3%	0.0%	74.0%
libedn	3.6%	**15.9%**	0.0%	80.5%
libhuffbench	9.3%	0.0%	0.0%	90.7%
matmult-int	0.0%	11.1%	0.5%	88.4%
libminver	20.7%	1.4%	0.7%	77.2%
nbody	24.8%	3.6%	0.3%	71.3%
nettle-aes	24.8%	0.0%	0.2%	75.0%
nettle-sha256	30.6%	0.0%	0.0%	69.4%
libnsichneu	0.0%	0.0%	0.0%	100.0%
libpicojpeg	19.7%	2.8%	0.0%	77.5%
qrencode	9.3%	2.8%	0.0%	87.9%
combined	6.2%	0.0%	0.4%	93.4%
libslre	0.4%	0.0%	0.0%	99.6%
libst	23.5%	3.5%	0.6%	72.4%
libstatemate	0.0%	0.0%	0.0%	100.0%
libud	9.0%	3.9%	0.9%	86.2%
libwikisort	14.0%	2.3%	**1.0%**	82.7%

Table 5: Embench program execution cycles (in Million) for multicore with different sharing configuration

	no_shared	s_m_d		m_d		d		s_d	
Program	5-stage	4-stage	5-stage	4-stage	5-stage	4-stage	5-stage	4-stage	5-stage
mont64	5.248	9.438	10.400	6.006	6.909	5.964	5.248	9.404	10.298
crc_32	4.915	8.949	8.244	7.719	6.316	7.544	4.916	8.770	7.719
libedn	7.549	8.777	9.152	8.625	8.886	7.924	7.550	8.102	7.955
libhuffbench	5.064	6.461	6.062	5.993	5.412	5.993	5.104	6.461	6.063
matmult-int	8.071	11.415	10.581	11.415	10.581	11.039	9.453	11.039	9.453
libminver	8.011	11.900	12.254	10.130	9.713	10.041	9.379	11.715	11.870
nbody	8.484	13.378	14.217	10.303	10.860	10.088	8.590	13.139	13.659
nettle-aes	7.216	10.948	10.778	8.954	8.408	8.954	7.296	10.946	10.788
nettle-sha256	4.865	8.788	8.894	6.501	6.091	6.500	4.865	8.787	8.865
libnsichneu	6.165	6.627	6.166	6.627	6.166	6.627	6.165	6.627	6.166
libpicojpeg	7.673	10.317	10.917	10.102	8.982	9.979	7.705	10.210	10.550
qrduino	6.119	8.075	7.299	7.647	6.647	7.549	6.139	7.993	7.065
combined	6.017	7.576	6.818	7.400	6.574	7.400	6.432	7.576	6.875
libslre	5.161	6.053	5.194	6.038	5.172	6.038	5.161	6.053	5.194
libst	6.084	10.016	10.379	8.281	8.510	8.135	7.313	9.855	9.889
libstatemate	3.249	3.983	3.249	3.983	3.249	3.983	3.249	3.983	3.249
libud	6.307	8.377	7.990	7.563	7.338	7.442	7.127	7.964	7.737
libwikisort	4.927	7.353	7.084	6.672	6.472	6.760	6.024	7.291	6.940

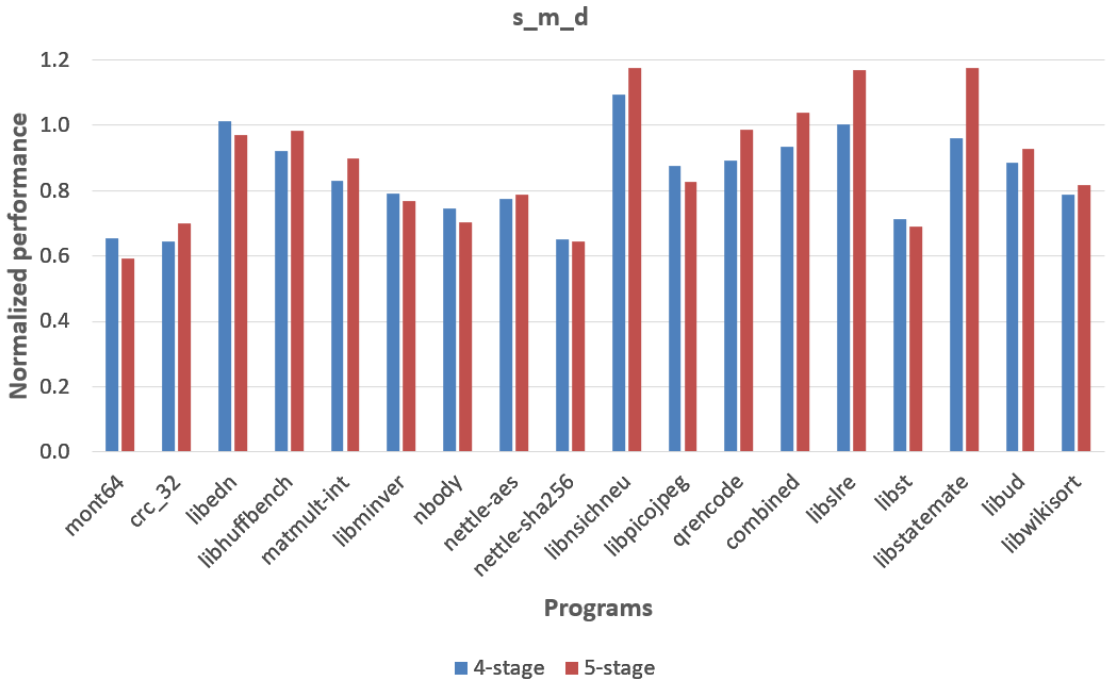


Figure 31: The normalized performance for s_m_d configuration w.r.t 5-stage no_sharing

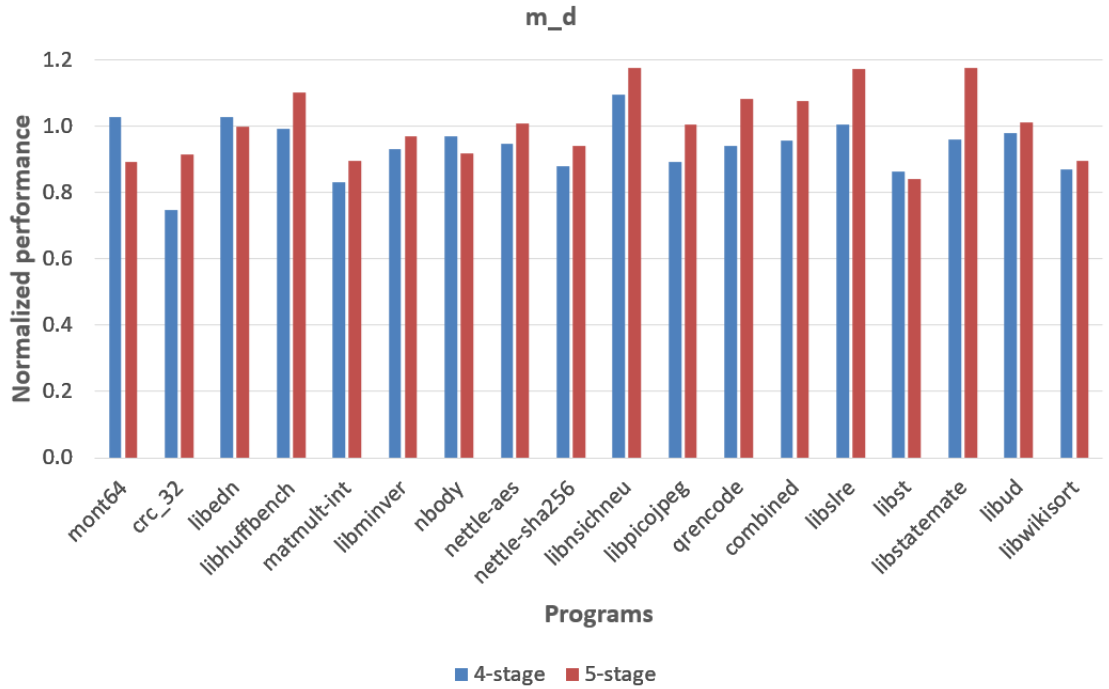


Figure 32: The normalized performance for m_d configuration w.r.t. 5-stage no_sharing

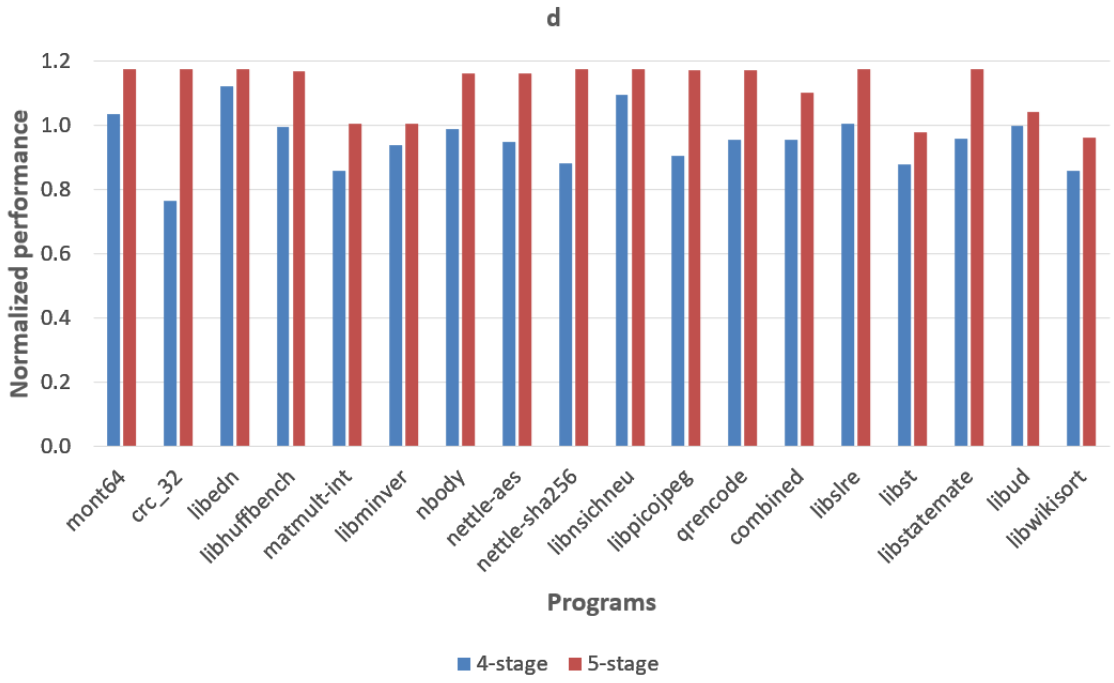


Figure 33: The normalized performance for d configuration w.r.t. 5-stage no_sharing

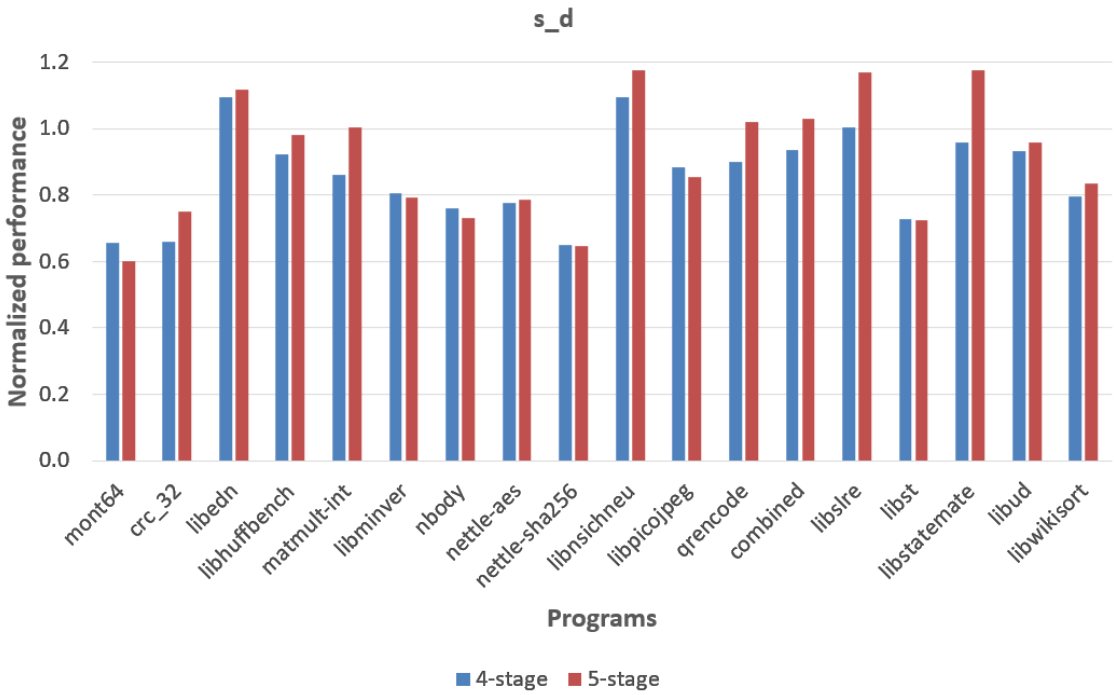


Figure 34: The normalized performance for s_d configuration w.r.t. 5-stage no_sharing

Table 6: The summary of 4-core evaluation results

Sharing configuration	s_m_d		m_d		d		s_d	
	4-stage	5-stage	4-stage	5-stage	4-stage	5-stage	4-stage	5-stage
Performance drop	**16.67%**	**13.73%**	6.32%	0.05%	5.14%	**-11.73%**	15.43%	11.00%
LUT savings	**31.4%**	**26.1%**	18.4%	17.0%	20.0%	17.6%	31.7%	26.2%

Table 6 summarizes the evaluation result of my experiment. I took the geometric mean of the benchmark programs' performance for sharing configurations. In table 6, I showed the performance drop with respect to the 5-stage no_shared configuration due to sharing resources. Since LUT is a fundamental resource for FPGA, I also put the LUT resource savings in the summary table. Interestingly, for the 5-stage d sharing configuration, the performance drop is negative; that is, it performs better than the no_shared configuration.

3.6 Related Works

Sharing resources among the cores can reduce the multicore FPGA logic utilization and fit more cores in the same FPGA. Dolbeau et al. proposed CASH[14] parallel processor, which shares the memory structures (instruction and data caches and branch predictors) and sparsely used functional units like a divider. At the same time, FPGA soft processors can be tightly coupled with the available SRAM on fabric that does not use the cache memories. Sharing the hardware like a multiplier, divider, and shifter among two or more soft processors, termed conjoining soft processors[35], can reduce the size. However, that study does not provide the actual implementation results as the sharing of resources has logic overhead.

There are many works on the RISC-V processor core, and are available as open-source and commercial cores. Some of those cores are listed in table7. Among them, some of the RISC-V cores that are intended for FPGA soft processors are PicoRV32 [48], Taiga [57] VexRiscv [46] and RVCoreP [5]. PicoRV32 is RV32I with an optional M and C extension focused on the smaller logic resources but not optimized for performance. Taiga is a RISC-V soft processor consisting of variable latency parallel execution units. It has higher performance but with additional logic overhead. VexRiscv, as a soft processor, has a good balance of performance and logic resource utilization, and it is written in Scala. RVCoreP, which is written in Verilog, is found to have better performance than VexRiscv for RV32I. However, RVCoreP does not support M extension. It is noteworthy that both VexRiscv and RVCoreP is five-stage pipeline processor. Furthermore, none of these soft processors have studied resource sharing in multicore processors.

Table 7: List of some available RISC-V cores

Supplier	Core	Language
Berkley [40, 41]	BOOM, Rocket	Chisel
ETH [42]	CV32E40P, CVA6, Ibex	SystemVerilog
MIT [43, 44]	Riscy, RiscyOO	Bluespec
Cornell [45]	Lizard	PyMTL
SpinalHDL [46]	VexRiscv	Scala
IIT Madras [47]	Shakti	Bluespec
Clifford Wolf [48]	PicoRV32	Verilog
Western Digital [49]	SweRV EH1/2, EL2	SystemVerilog
University of Tokyo [50]	RSD	SystemVerilog
Tokyo Tech [5, 51]	RVSoC, RVCoreP	Verilog
SiFive [52]	E2/3/7, S2/5/7, U5/7	Chisel
Codasip [53]	L10/30/50, H50, A70	Verilog
Andes [54]	N/D/A/NX/AX 25/27/45	Verilog
T-Head Alibaba [55]	XuanTie 902/906/910	Verilog
Syntacore [56]	SCR 1/3/4/5/7	SystemVerilog

Several kinds of research propose heterogeneous multicore clusters for FPGA with TCM as shared or local memory. HERO[20] proposed a heterogeneous many-core research platform on FPGA. It integrates the RISC-V core into a shared memory cluster-based programmable many-core accelerator (PMCA). The work presented in [13] proposed both local and shared TCM in the cluster of their many-core architecture. However, even for a single cluster of 4 to 8 cores, the FPGA implementation requires a considerable amount of LUT.

A. Rahimi et al. [58] have proposed the RISC-V based multicore cluster where only the floating-point unit was shared among the cores in the cluster. The GRVI[19] is an FPGA-based accelerator framework based on scalar RISC-V cores of a 2/3-stage pipeline supporting mainly the RV32I instruction set and that coupled with accelerators. To our knowledge, GRVI is the first RISC-V based multicore implementation that shared the integer functional units like a multiplier, divider, and instruction and data RAM. But it does not provide a detailed study on the reduction of logic resources and the performance impact of resource sharing. While recent work Snitch[59] proposed the hardware multiplication, division, and floating-point sharing among the cores, yet the research is focused on ASIC, and the core is a single-stage pipeline. Both GRVI and Snitch proposed sharing the TCM into the cluster.

3.7 Summary

The RV32IM extension has adequate instructions for embedded soft computing and can be implemented into FPGA with a reasonable resource. Moreover, higher performance requirements can be met by utilizing multicore architecture.

I have presented 4-stage and 5-stage pipeline architecture for FPGA with hardware resource sharing in this paper. Compared to nothing shared, I studied the resource utilization implication for sharing functional units like the shifter, multiplier, and divider. Sharing allows for reducing the DSP usage to 75% regardless of pipeline architecture. LUT usage is reduced to 31.7% for 4-stage and 26.1% for 5-stage pipeline architecture. To summarise, the 4-stage pipeline is slightly smaller than a 5-stage pipeline for any sharing configurations in multicore. But still, FPGA logic utilization is a crucial design factor because of fixed and constrained resources. Moreover, when the design is packed with more LUTs, the operating frequency decreases.

The simulation results presented in this paper illustrate that all kinds of workloads are not affected by resource sharing. Some programs are affected by resource sharing, while others provide a good balance of resource savings compared to performance. The results of this study suggest that sharing divider or multiplier and divider has little performance drop due to sharing.

In all sharing configurations, the 5-stage has better performance than the 4-stage. And 4-stage has area benefits over the 5-stage. My proposal on BRAM port sharing maximize the BRAM utilization for multicore architecture. It is worth to be noted that hardware logic utilized by four cores with sharing consumes similar resources for three soft cores without any sharing. Since the soft processor cores mapping to FPGA are relatively straightforward, users can choose the appropriate sharing option for RVCoreP-32IM according to the application requirement.

4 Resource Efficient Vector Processing Unit (VPU)

4.1 Motivation

Many of the edge-embedded applications are data-intensive, but the computation performance is limited by the scalar processor due to a lack of data-level parallelism. In FPGA-based design, such data-parallel applications can be addressed by custom-designed hardware accelerators. However, the development of embedded applications with such a custom accelerator requires hardware design experience. On the other hand, a vector processing unit allows the developer to exploit such data-level parallelism without redesigning the accelerator every time.

The advent of quantized integer-based AI inference enables the development of different applications such as image classification, voice recognition, and many others at the edge. These applications require integer computation such as multiplication, addition, dot operation (for convolution), min/max, etc., with varied numbers of parallel operations. Single instruction multiple data (SIMD) allows executing operations over the wide registers of a fixed length such as ARM NEON[60], AVX-512[61] etc. On the other hand, in a Vector Architecture, vector length can be changed by application up to a predefined maximum vector length (MVL) such as ARM SVE[62], RISC-V vector extension[30].

RISC-V vector extension (RVV) is an open-source instruction set architecture(ISA) that allows researchers to develop their own architecture to implement the vector processing unit. This ISA defines arithmetic instructions for integer, fixed point, and floating point data types. Since FPGA is a resource-constrained device implementing the vector processor with all data types will consume a lot of resources. Support for all of those data types may not be required for a specific application. In this research, I have focused on the integer data type implementation, though the same architecture can be extended for other data types as well. The ISA also defines the vector memory load and store instructions for memory operations, reduction instructions to process the contents of a single vector register, permutation instructions to shuffle the elements in the vector register, and vector mask instructions for conditional executions.

4.1.1 Purpose

The purpose of this research is to propose a scalable soft vector processing unit (VPU) supporting a subset of RISC-V Vector extension ISA. It would be pluggable into any scalar RISC-V processor core. Development of an embedded application for such VPU is easy

due to the support of the compiler. Exploiting data-level parallelism will accelerate the performance of such applications.

4.1.2 Contribution

The main contributions of this research are summarized below:

- I introduce a 5-stage pipeline-based VPU, that is parameterized and designed for FPGA. My design can accommodate other functional units to support different vector instructions.
- I have shown that my distributed VRF-based architecture provides flexibility to configure the number of lanes and the MVL. My VPU performance is scalable with a number of lanes and MVL.
- I present an efficient reduction sum operation in conjunction with a vector multiplier to reduce the latency of integer dot/convolution operation.
- I implement the VPU in FPGA for different parameters and show that FPGA resource utilization is much lower than the other proposed designs, and the operating frequency of my VPU has a negligible drop with the number of lanes.

4.2 Vector Processing

A key element of vector processing is that arithmetic, logic, and memory (load/store) instructions operate on a set of elements of vector instead of individual data items. Vector architectures pipeline the operations to obtain a good performance. The vector ISA defines the Vector Register File(VRF) as a number of vector registers. The important feature of vector architecture is that each vector register can hold a large number of bits, and vector register contents can be used as different widths of elements depending on the user's selected element width (SEW). For example, if a vector register length is 128-bit, then it can be used as four 32-bit elements (e0 to e3), or eight 16-bit elements (e0 to e7) and sixteen 8-bit elements (e0 to e15) as depicted in figure 35.

Vector architectures can include multiple parallel lanes (data paths), which can produce multiple results in parallel. Adding multiple vector processing lanes is advantageous for performance with scalability. In multiple-lane vector architecture, the VRF can be divided into multiple lanes, or there could be a centralized VRF from which the vector register contents are delivered to the lanes. Furthermore, multiple-lane vector architectures

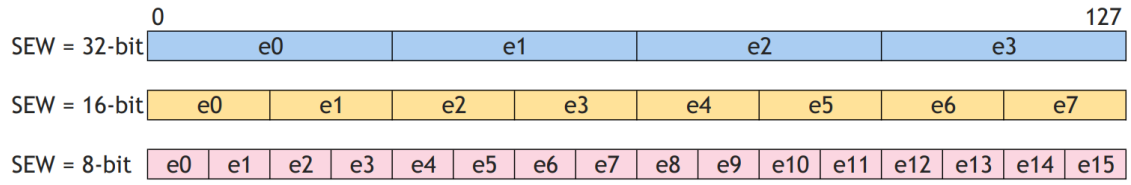


Figure 35: Vector register elements with different widths

may have extra hardware to control the data movement between all lanes along the lane interconnection network.

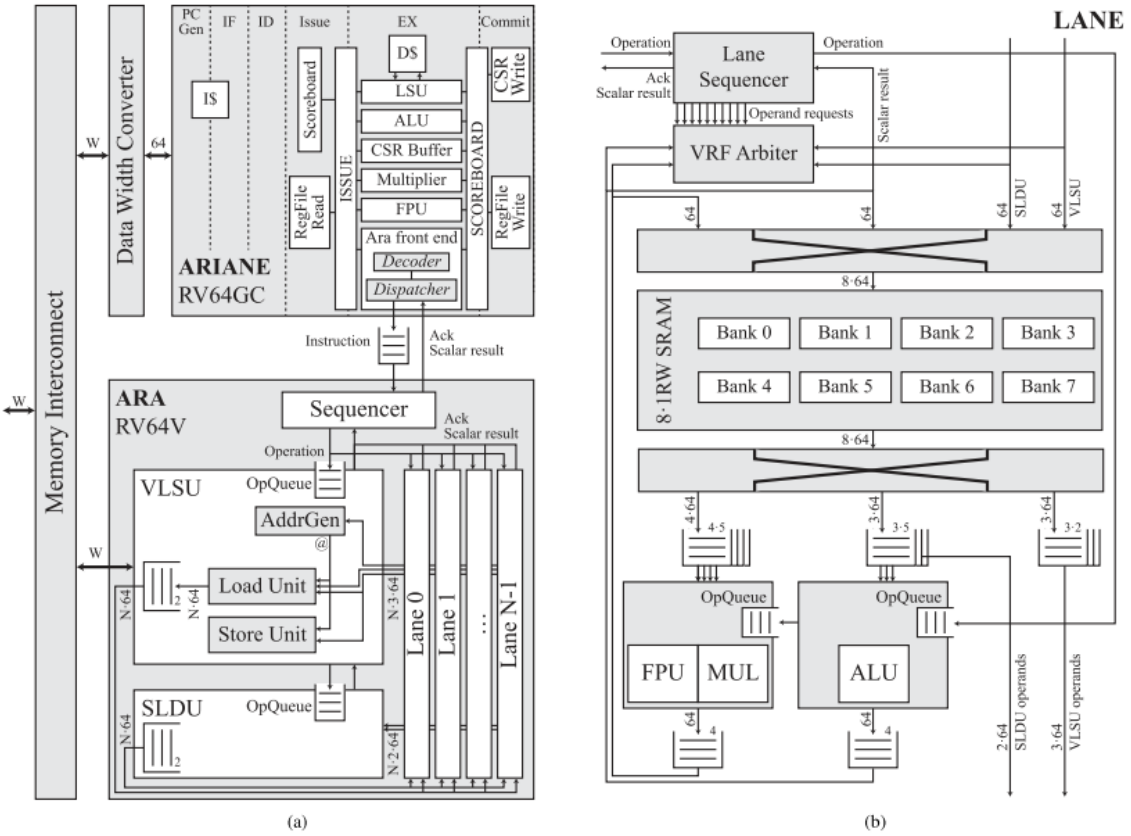


Figure 36: (a) Block diagram of an Ara [17] instance with N parallel lanes (b) Block diagram of one lane of Ara © 2020, IEEE

Figure 36 shows the block diagram of the vector processor Ara proposed in [17]. Ara receives its commands from 64-bit RISC-V scalar core Ariane. The vector unit has the main sequencer to keep track of the instructions and dispatch them to different execution units. It has N parallel lanes, a sliding unit (SLDU) that accesses all VRF banks at once, and a vector load store unit (VLSU). Each lane contains a lane sequencer, a VRF, an integer ALU, an integer multiplier (MUL), and a floating-point unit (FPU). VRF is

organized as eight banks, and this VRF structure is replicated at each lane, and all inter-lane communication is concentrated at the VLSU and SLDU.

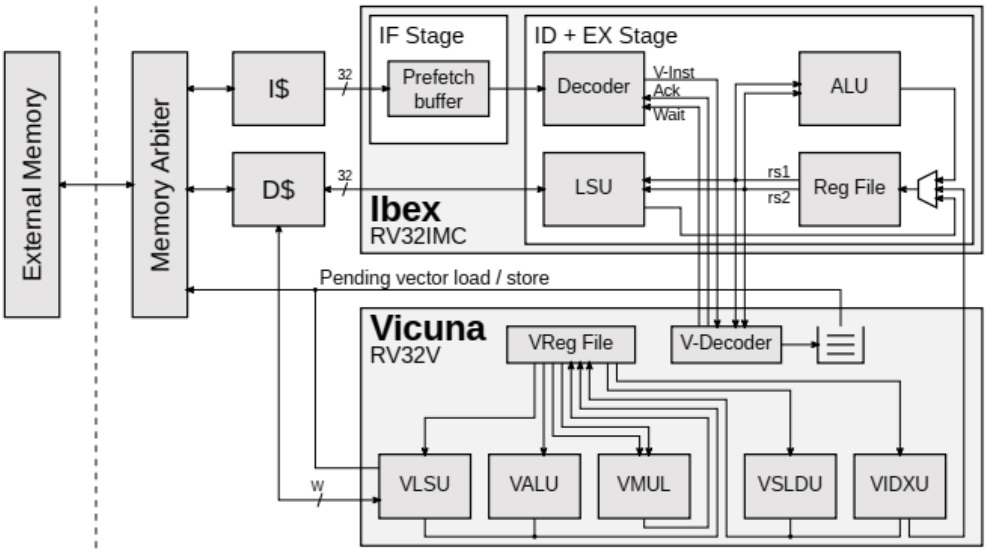


Figure 37: Architecture of Vicuna vector co-processor [18]

The architecture of the Vicuna[18] vector co-processor is shown in figure 37. Vicuna is integrated with a 32-bit RISC-V scalar core. It has a centralized VRF and multiple execution units such as vector ALU (VALU), vector multiplier (VMUL), vector load store unit (VLSU), vector sliding unit (VSLDU), and vector indexing unit (VIDXU). All functional units read the whole source vector registers into shift registers. The contents are then shifted by the number of elements that can simultaneously be processed by the unit each cycle.

4.3 Proposed Architecture

I implement a subset of the RVV[30] v1.0 ISA extension (Zve32) for integer computation which is commonly used for acceleration such as convolution, matrix multiplication, etc.[3] It does not support widening and permutation instructions. However, it supports integer scalar move instructions. RVV defines the vector element length as *ELEN*, which is the maximum size of a single vector element in bits. Users can select element width *SEW*

[3]This work is published in the following ACM proceedings, Md Ashraful Islam and Kenji Kise. 2023. Resource-efficient RISC-V Vector Extension Architecture for FPGA-based Accelerators. In Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '23). Association for Computing Machinery, New York, NY, USA, 78–85. https://doi.org/10.1145/3597031.3597047

from 8-bit to ELEN bits. The number of bits in a single vector register is defined as vector length (*VLEN*). Depending on the implementation, VLEN can be 32-bit to 65,536-bit. The number of elements in a single vector register can be found by dividing the VLEN by SEW. In this architecture, ELEN is 32-bit, and SEW can be 8-bit, 16-bit, or 32-bit. The number of lanes can be configured from 1 to 16, and the maximum vector length (*MVL*) can be configured from 128-bit to 4096-bit.

RVV also defines the vector control status register (CSR), which is used by the software to control the execution and check the status. For example, if application software wants to process the vector register contents as an 8-bit element, the user should write the vector type (*vtype*) CSR's SEW field to 0 by executing vector configuration instruction.

Figure 38 shows the block diagram of the proposed vector microarchitecture. my proposed vector processing unit (*VPU*) executes the vector instructions irrespective of the scalar RISC-V core architecture. VPU can have single or multiple lanes of up to 16 lanes. Each lane has a 32-bit data path consisting of its own part of the vector register file, an execute and write back unit. Each lane can perform 4/2/1 operations in parallel if the selected element width (*SEW*) is 8/16/32 bit, respectively.

Vector instructions are executed in the 5-stage pipeline - decode (DEC), sequencer (SEQ), vector register file (VRF), execute (EXE), and writeback (WB). Once the scalar processor fetches and decodes the vector instruction, it pushes that vector instruction to the vector extension queue, given that branch prediction and other hazards and exceptions are resolved. For some vector instructions, a scalar operand from the scalar core register file is required, which is also pushed into the same queue along with the vector instruction. Few instructions require the single vector register element value from the VPU to the scalar core. For such instructions, the write back unit of lane 0 will return the value directly to the scalar core. While executing such instructions, the scalar core should be stalled and wait for the return value from the VPU. For other vector instructions, the scalar core can execute in parallel to the VPU, given that there are no exceptions or race conditions between the scalar core and the VPU. The functional description of major blocks is given in the following sections.

4.3.1 Decoder

Vector Decoder reads 32-bit instructions from the instruction queue, which are pushed by the scalar RISC-V core. Then it decodes the instruction into the appropriate control signals and required source and destination addresses of the vector registers for the subsequent

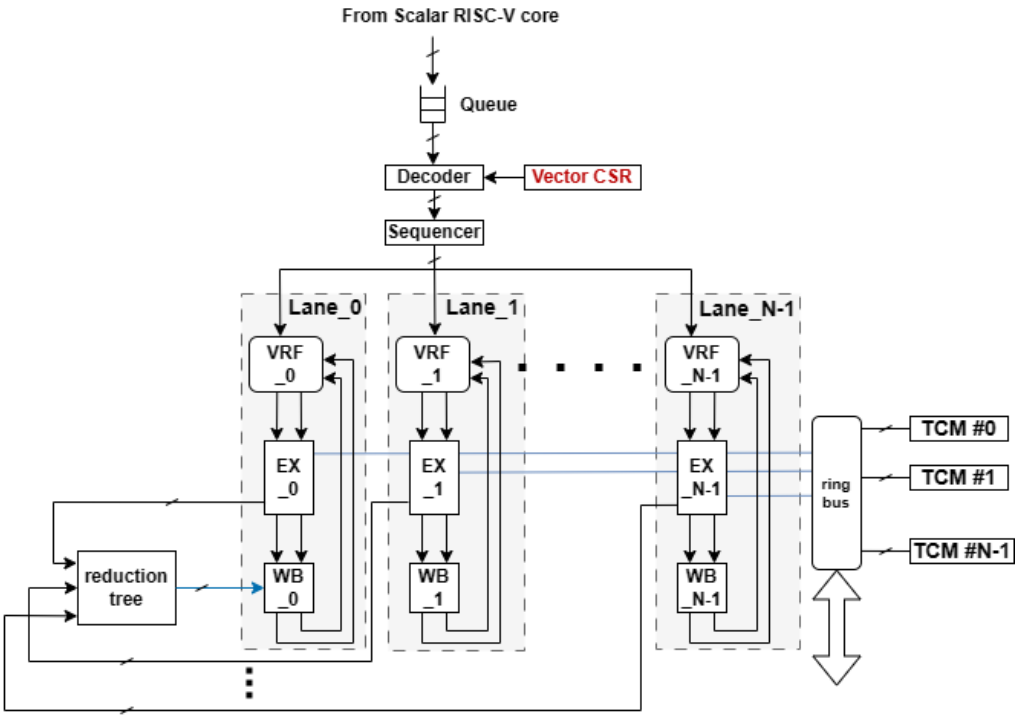


Figure 38: Proposed Vector Processing Unit's microarchitecture

pipeline stages. If the instruction requires a scalar value from the scalar core register file, then that value is also popped from the queue.

Decoder output is passed to the sequencer stage and waits for the sequencer to be ready if the previous instruction in the sequencer is not finished. It then reads the next instruction from the queue if the queue is not empty. For vector configuration instruction (e.g. vsetvli), the decoder reads and sets the vector CSR values and returns the new value of the vector length to the scalar core. The updated vector configuration values are provided by the decoder to the sequencer for the forthcoming instructions executions.

4.3.2 Vector Register File and Sequencer

In my architecture, each lane has a partial set of the vector register file. The vector register's maximum vector length (MVL) is parameterized and equally divided among the lanes. Since the VRF is distributed among the lanes and there are no interconnections among VRF in different lanes, this architecture does not support instructions that require inter-lane VRF data exchange. Owing to that fact, this architecture does not support widening and permutation instructions. Because those instructions need to access the VRF elements over multiple lanes. Figure 39 shows the vector register file organization

for MVL=512 and the number of lanes NLANE=4 and SEW=32 (i.e., each element e in this figure is 32-bit). Elements of each vector register are indexed as e0, e1, e2, and so on. My motivation for this VRF organization is the following.

- VRF is implemented using block RAM (BRAM), which is available in the FPGA as a memory resource. BRAM is usually having a few kilobytes of RAM (in modern Xilinx FPGA devices, it is a minimum of 4KB). According to the RISC-V Vector ISA, the VRF has 32 registers. For 32-bit element width, it requires only 128 Bytes, which underutilizes the available BRAM memory. To increase memory utilization, I have allocated more bits to a single vector register by increasing the vector length (VLEN). Thus for any given SEW, the number of elements in a single vector register will increase.
- Increasing the number of elements per vector register increases the MVL, which is beneficial for application acceleration due to reduced software loop iteration and may hide memory latency for long memory bursts.
- Implementation of a monolithic register file with banked memory for multiple lanes is inefficient for FPGA as it requires the interconnection of memory banks, and area increases exponentially with the number of lanes. Each lane has its own VRF allocated with a partial set of elements from 32 vector registers and does not require any interconnection for inter-lane data transfer from VRF. Thus implementing multiple lanes is straightforward and can be configured according to the application requirements.

The VRF is implemented as a 2-read and 2-write port (2R2W) register file using the live value table (LVT) technique[63]. Two read ports to perform two register operands read (vs1/vs3 and vs2) operation. The scalar value from the scalar core is multiplexed with the vs1/vs3 vector register read data. One register file write port is used for a vector load operation, and another write port is used for ALU and multiplication operations. There is no bypassing data path from the write port to the read port.

As the vector registers have multiple elements, the vector execution has to iterate over this number of 32-bit elements, I named it SEQLEN (MVL/(NLANE X 32)). For example, according to figure 39, lane 0 has to process elements e0, e4, e8, and e12 for the given instruction, thus requiring four iterations. The sequencer performs this iterative execution over multiple lanes in parallel.

Sequencer also checks for the data hazards of read after write (RAW), write after read

	Lane_0	Lane_1	Lane_2	Lane_3
V31	e12	e13	e14	e15
	e8	e9	e10	e11
	e4	e5	e6	e7
	e0	e1	e2	e3
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
V1	e4	e5	e6	e7
	e0	e1	e2	e3
V0	e12	e13	e14	e15
	e8	e9	e10	e11
	e4	e5	e6	e7
	e0	e1	e2	e3

Figure 39: Vector register file organization

(WAR), and write after write (WAW) before dispatching it to VRF for subsequent execution. The sequencer tracks the state of 32 vector registers of VRF using a state table to detect data hazards. The vector register file state of the destination register (*vd*) is updated by the sequencer, and during the write back corresponding *vd* register state is restored by the write back module. If the hazard is detected, the sequencer inserts a bubble for the subsequent pipeline stages.

The pipeline diagram of my VPU is illustrated in figure 40. In my design, the execution stage takes two clock cycles, and the other stages are one clock cycle. While the sequencer is doing iterative execution, the decoder is stalled by the sequencer. In figure 40, I have shown the example for SEQLEN=2. Therefore the sequencer has to iterate two times (SEQ 0 and SEQ 1) to execute the instruction. In this example, instruction K+2 has a read-after-write (RAW) data hazard with instruction K+1. As a result, the sequencer stalls two clock cycles. From this figure, it can be easily verified that if the SEQLEN is greater than or equal to 4, there will be no stall due to data hazard except for the reduction operation, which will be explained in section 4.3.3.

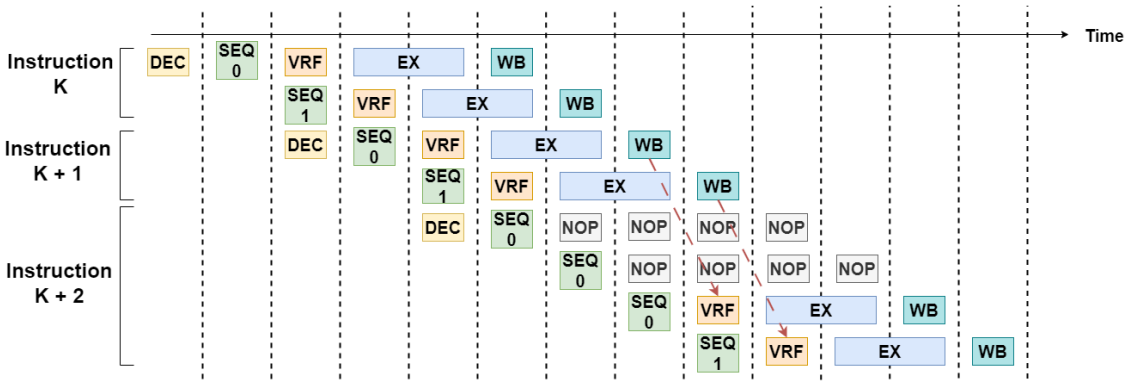


Figure 40: Pipeline execution for two elements per vector register per lane

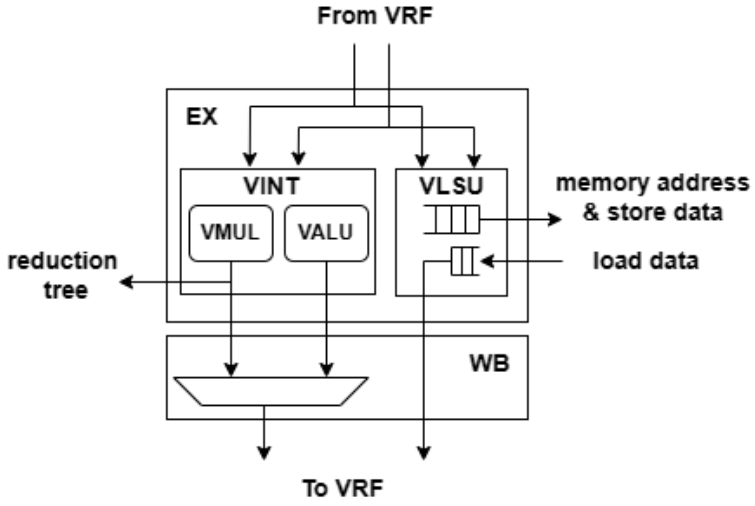


Figure 41: Execution and Write back block diagram

4.3.3 Execution and Write back

Figure 41 shows the block diagram of the execution (EX) and writeback (WB) stages. The execution unit has a vector integer unit (VINT) and a vector load store unit (VLSU). VINT executes the vector integer arithmetic (e.g., vadd, vsub, vmin, vmax, vsll, vsrl, etc.) and multiplication instruction (vmul, vmulh, etc.) and VLSU performs vector load-store instructions(vle, vse). Vector multiplication instructions are executed in the VMUL module, and the VALU module performs vector integer arithmetic and logical instructions. The vector division instructions are not implemented, and shift instructions are optional, as those are less frequently used in many applications.

VINT takes two operands from the VRF and takes two clock cycles to compute the result in the pipeline. It performs four operations for SEW=8-bit, two operations for

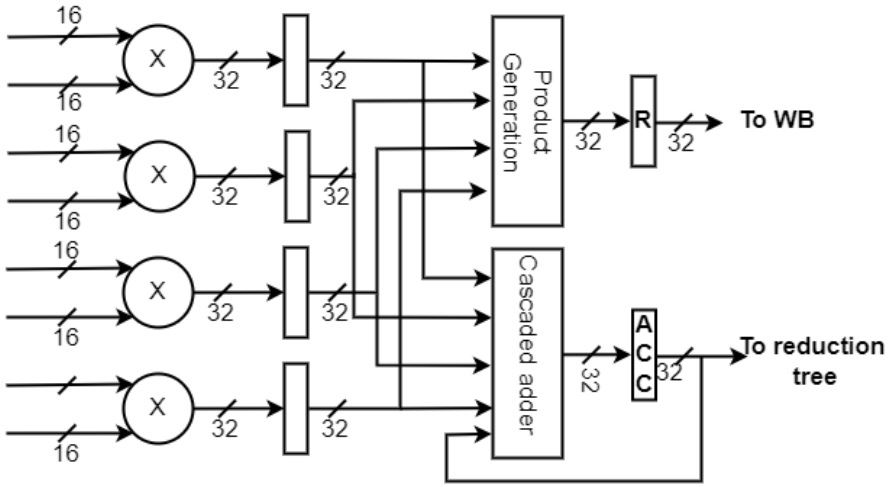


Figure 42: Block diagram of vector multiplier (VMUL)

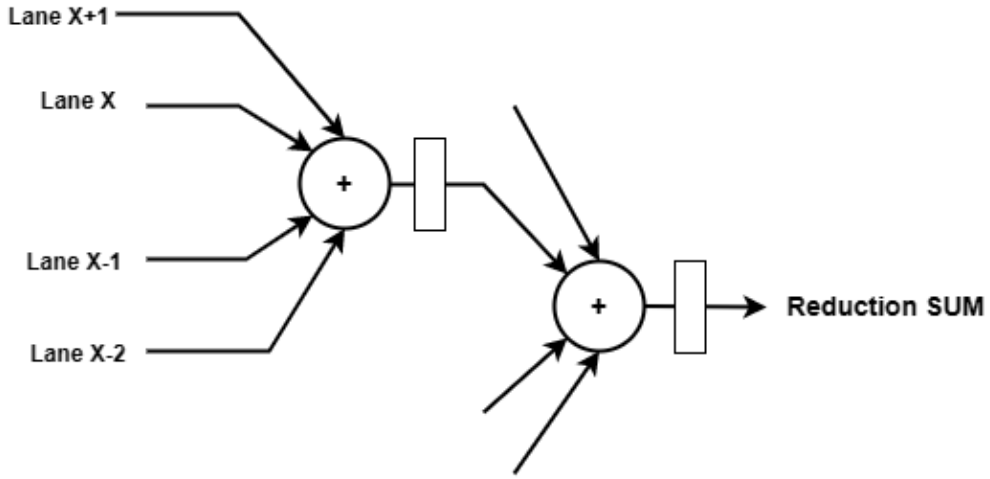


Figure 43: reduction tree diagram

SEW=16-bit, and one operation for SEW=32-bit. For variable precision (8/16/32-bit) vector multiplication, I have used a similar approach presented in [64]. The block diagram of VMUL is shown in figure 42. In the first clock cycle, it calculates the partial product from the 16-bit input multiplicand and multiplier (for 8-bit multiplication, it is signed/unsigned extended to 16-bit) and stored in registers. In the second clock cycle, based on the multiplier precision, it performs a summation of the partial products and calculates the multiplication result put into a register (R).

Since the VRF can produce only two operands, some instructions that require three inputs, such as multiply-accumulate instruction, will require two operations, for example, multiply operation and then add operation. Implementing three read ports (3R2W) for

VRF operand read will use up two more BRAMs compared to 2R2W. However, if the target application is multiply-accumulate intensive, one can configure the VRF as 3R2W without any architectural changes.

Since many accelerators manipulate multiplication and convolution functions, I have optimized the dot operation for these functions. I do not support widening operation, but I fuse widening multiplication (VWMUL) and widening reduction (VWREDSUM) instructions internally to support dot operation. The VWMUL instruction calculates the multiplication results and stores them in the internal accumulator (ACC) as shown in figure 42. This accumulates the summation over the elements of the destination vector register of its own lane. For example, if there are eight elements per vector register per lane (SEQLEN=8), the accumulator will store the sum of 8 vector element multiplication results. During VWREDSUM instruction, the reduction tree module does a summation of these accumulators from all the lanes.

When the number of lanes increases, the addition of these accumulators over multiple lanes becomes a timing-critical path because of long wire delay and LUT chaining delay. To overcome this issue, I have used a quad-tree graph-based reduction tree presented in figure 43. Accumulators from multiple lanes are the input nodes of this graph, and intermediate nodes calculate the summation of 4 inputs from the previous layer nodes and register the results. These node registers act as pipeline registers to break down the critical path of reduction operation over multiple lanes.

This reduction tree aims to reduce the required number of clock cycles for reduction sum instruction while achieving higher operating frequency. For example, in 16-lane VPU, if I do the iterative execution, it will take eight clock cycles for SEQLEN=8, while the reduction tree will take only two cycles. Again for iterative execution, there will be 16 addition per clock cycle, which will slow down the maximum operating frequency. On the other hand, the proposed quad-tree graph-based reduction tree will take only two clock cycles for 16 lanes. Since the reduction instruction will execute on the vector elements at once, there will be RAW data hazard, and the sequencer will insert two bubbles after the multiply instruction and then will execute the reduction sum instruction.

For instance, if MVL is 4,096 bits and SEW is 8-bit, there are 512 8-bit integers (INT8) elements in a vector register. And if NLANE is 16, then there are (512/16=) 32 elements per vector register per lane. It will take eight clocks to multiply two vector registers and produce 16-bit results. During the multiplication operation, it will also accumulate the multiplication results along the lane. Reduction sum instruction immediately after the

multiplication instruction will be stalled for two clock cycles due to RAW data hazard with the multiplication result's vector register. After stalling for two clock cycles, the sequencer will issue the reduction instruction. Then the reduction sum operation will take two clock cycles to do the summation in the quad-tree over the 16 lanes. A total of 12 clock cycles will be required to perform a dot/convolution operation on 512 INT8 elements to produce the result.

Vector load store unit (VLSU) has a memory address queue for load and store operation. Write-data queue for the store operation and the read-data queue for the load operation. Address and write-data queues have 16-depth, so if the SEQLEN=8, it can queue two vector loads or store instructions. Each location in the queue holds the address of the given iteration of load or store instruction from the sequencer. The read data queue has only a single depth because the load data from the TCM will be written back to the VRF immediately. RVV has defined unit stride, constant, and index strided addresses for vector load and store instruction. My architecture implements unit stride as the main feature, and other strided load and store instructions are optional. However, the VLSU does not support the segmented vector load and store instructions.

There are tightly coupled data memories (TCM) connected to each lane. TCM consists of dual port BRAM and has local and global ports as presented in [65]. TCMs are organized as word (32-bit) interleaving memory among the lanes (lane 0 TCM have byte 0 to 3, lane 1 TCM have byte 4 to 7, and so on). Each lane has a dedicated read and write port to its TCM (local port), and another TCM port (global port) is connected to the ring interconnect. For instance, if each lane is accessing its own TCM, then there is a fixed 2-clock cycle latency. It will go through the ring bus if it accesses another lane's TCM.

Since memory operation latency can vary with the memory location, the VINT can execute in parallel while the VLSU queue is not empty, given that there is no data dependency between them. Write back unit multiplexes data from VALU and VMUL and write it back to one of the write ports of VRF in that lane. The load data from the VLSU unit is written to another port of that VRF.

4.4 Evaluation

I implemented my vector extension core in SystemVerilog and then integrated it with RVCoreP-32IM [65], which is a 5-stage pipeline 32-bit RISC-V soft processor core with RV32IM extension. I have configured my vector processing unit's number of lanes (NLANE) as 1, 4, 8, and 16 lanes for evaluation.

4.4.1 FPGA Implementation

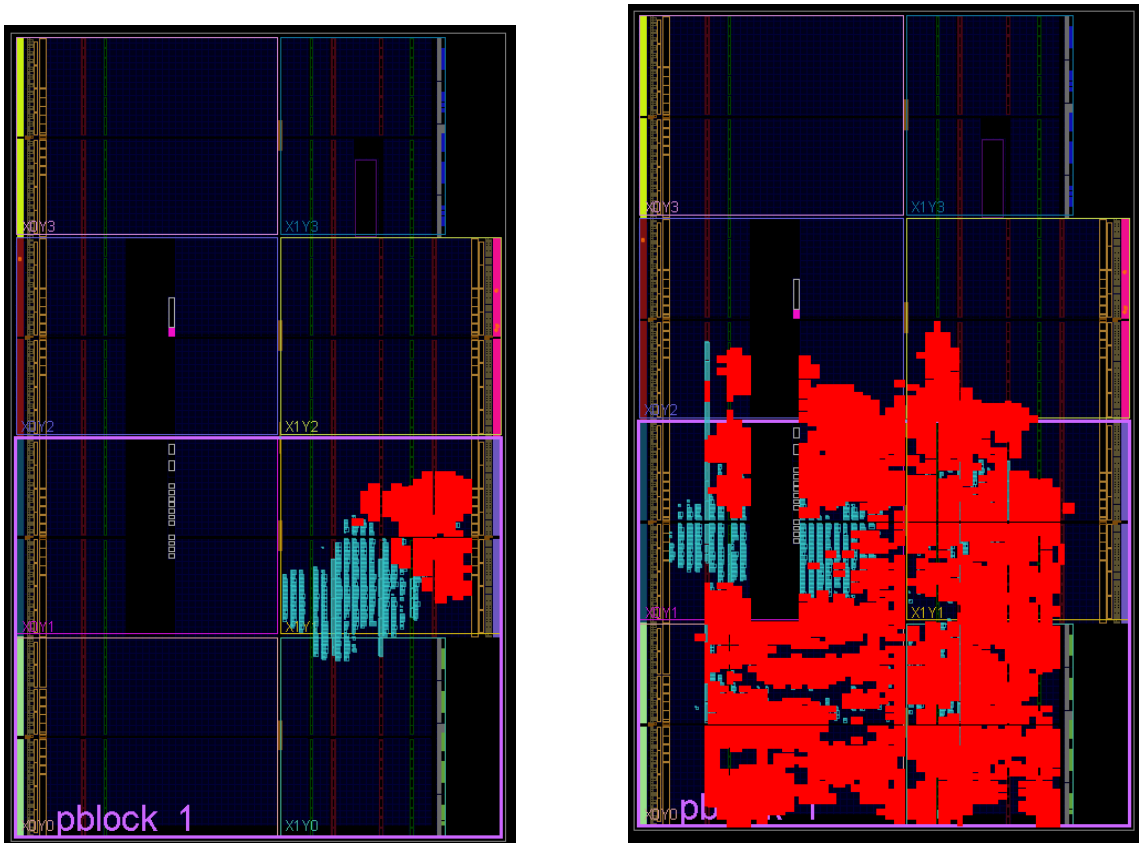
I evaluate the maximum frequency and hardware resource utilization for Xilinx Artix-7 FPGA and Zynq UltrScale+ FPGA devices. Artix-7 is a relatively cost-optimized smaller FPGA than Zynq UltrScale+ FPGA. Sometimes, smaller and cost-effective FPGA devices like Artix-7 are preferred for edge computing. I use the Nexys A7 FPGA board, which contains xc7a-100tcsg324-1 from the Artix-7 FPGA device family, speed grade -1. Vivado 2021.1 is used for Synthesis, placement, and routing with a default strategy to evaluate the operating frequency and hardware resource utilization. For this evaluation, I have used 4KB of tightly coupled data RAM for each vector lane. The RISC-V core RVCoreP-32IM is connected to these lanes using a ring interconnect. RVCoreP-32IM has its own instruction and data tightly coupled memory of 32KB for program execution and a UART as a peripheral.

Table 1: FPGA (Artix-7) resource utilization for vector processing unit

NLANE	1		4		8		16	
MVL	128	256	512	1024	1024	2048	2048	4096
LUT	1,236	1,209	3,750	3,741	7,119	7,187	13,978	13,852
Register	737	743	2,059	2,068	3,871	3,853	7,375	7,382
BRAM	2	2	8	8	16	16	32	32
DSP	5	5	20	20	40	40	80	80
FMAX (MHz)	125	125	125	125	125	125	120	120

FPGA implementation result is shown in table 1. From the table 1, we can see that the look-up table (LUT) and register utilization do not increase linearly with the increment of NLANE, as the vector instruction decoder and sequencer are fixed overhead for all lane configurations. Since the vector register file uses the BRAM and VMUL uses the DSP, BRAM and DSP utilization go up straightly with the NLANE. In the FPGA implementation, two BRAMs are used as four 2KB RAMs for the 2R2W port for the vector register file. In all configurations, RVCoreP-32IM uses around 1,500 LUTs. The FPGA post place and route layout are shown in figure 44 for a single lane with MVL=256 VPU and 16-lane with MVL=4,096.

The maximum operating frequency (FMAX) drops only 5 MHz for the 16-lane configuration, while other configurations have 125 MHz. In my design, the critical path is partial product sum generation through vector multiplier units. For the same NLANE, increasing the MVL (that is, the number of elements per vector register per lane) does not increase the FPGA logic utilization. It does not impact the maximum operating frequency.



(a) NLANE=1, MVL=256

(b) NLANE=16, MVL=4,096

Figure 44: Artix-7 FPGA implementation layout for VPU with NLANE=1, MVL=256 and NLANE=16, MVL=4,096. VPU is highlighted in red color

I have presented the comparison of resource usage with other soft vector processors in table 2. Since those publications mostly used different Xilinx Zynq UltrScale+, I have also synthesized the Zynq UltrScale+ FPGA device with the configuration of 8 Lane and MVL=1024. VPU1[10] has a dedicated unit for permutation, which could be a reason for higher resource usage. From the table 2, we can see that my proposed architecture has achieved more than four times higher operating frequency than VPU3 and more than three times higher operating frequency than other VPUs. And considering the NLANE, my proposed VPU uses significantly fewer FPGA resources than other VPUs.

VPU2[66] and VPU3[67] have the same architecture, but the configurations are different. VPU3[67] has similar logic utilization to the proposed architecture, though their operating frequency is much lower than the proposed architecture. Compared to my proposed 5-stage pipeline architecture, VPU3 has a 2-stage pipeline, which could be the reason for this frequency difference. Vicuna[18] has 2 stage pipeline though the execution can take

Table 2: Comparison of FPGA resource utilization (for Zynq UltrScale+ device)

VPU	MVL	LUT	Register	FMAX (MHz)
Vicuna[18]	2048, NLANE=32	80k	40k	80
VPU1[10]	512	136.5k	37.9k	75
VPU2[66]	256, NLANE=8	20.5k	9.5k	N/A
VPU3[67]	128, NLANE=2	7.3k	4.85k	62.5
Proposal	2048, NLANE=8	6.5k	3.8k	270

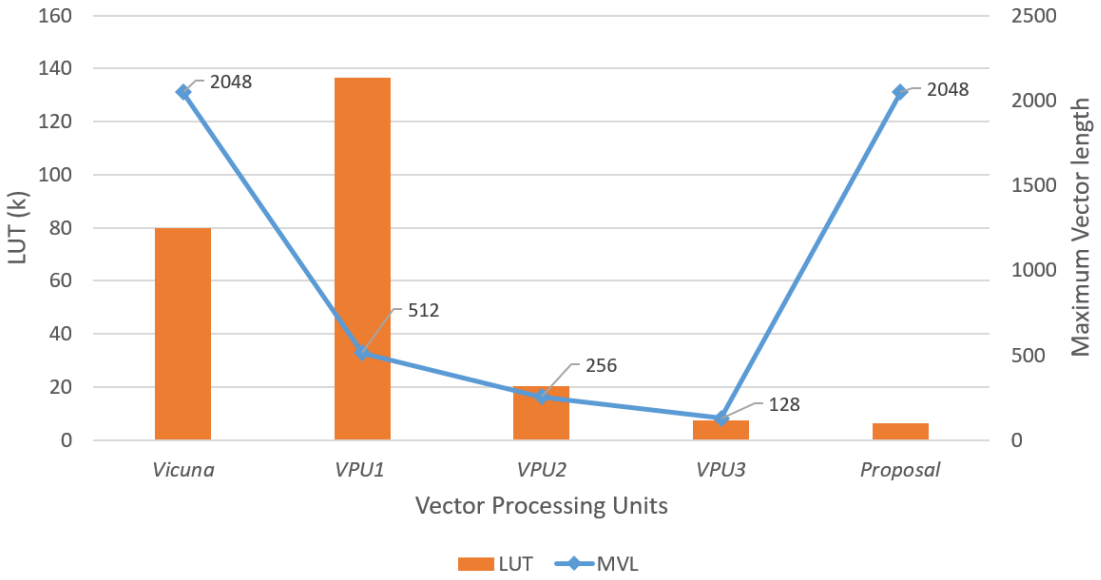


Figure 45: LUT usage compared to the vector length

multiple clock cycles. But architecture takes much more logic resources compared to the proposed VPU, which might negatively impact the Vicuna operating frequency. Figure 45 shows the LUT usage compared to the vector length. This figure shows that my proposed VPU architecture requires substantially less LUT to implement the larger vector length.

4.4.2 Performance Evaluation

I have used convolution for performance evaluation because it is widely used in signal processing, image processing, and convolutional neural network. I perform cycle-accurate RTL simulation for the convolution operation with kernel size 3x3x256 (=2,304) weight and data. I have used the convolution filter and input data type of 8-bit(INT8), 16-

Table 3: Required number of clock cycles to perform 3x3x256 convolution operation

VPU Configuration	INT8	INT16	INT32
NLANE 1, MVL 128	682,276	1,359,652	2,714,404
NLANE 1, MVL 256	513,716	1,021,748	2,037,812
NLANE 4, MVL 512	174,244	343,588	682,276
NLANE 4, MVL 1024	132,692	259,700	513,716
NLANE 8, MVL 1024	89,572	174,244	343,588
NLANE 8, MVL 2048	69,188	132,692	259,700
NLANE 16, MVL 2048	47,236	89,572	174,244
NLANE 16, MVL 4096	40,964	69,188	132,692

bit(INT16), and 32-bit(INT32) integers, and the output is a 32-bit(INT32) integer. I use the vector load, widening multiplication, reduction sum, and move instructions to perform the convolution operation.

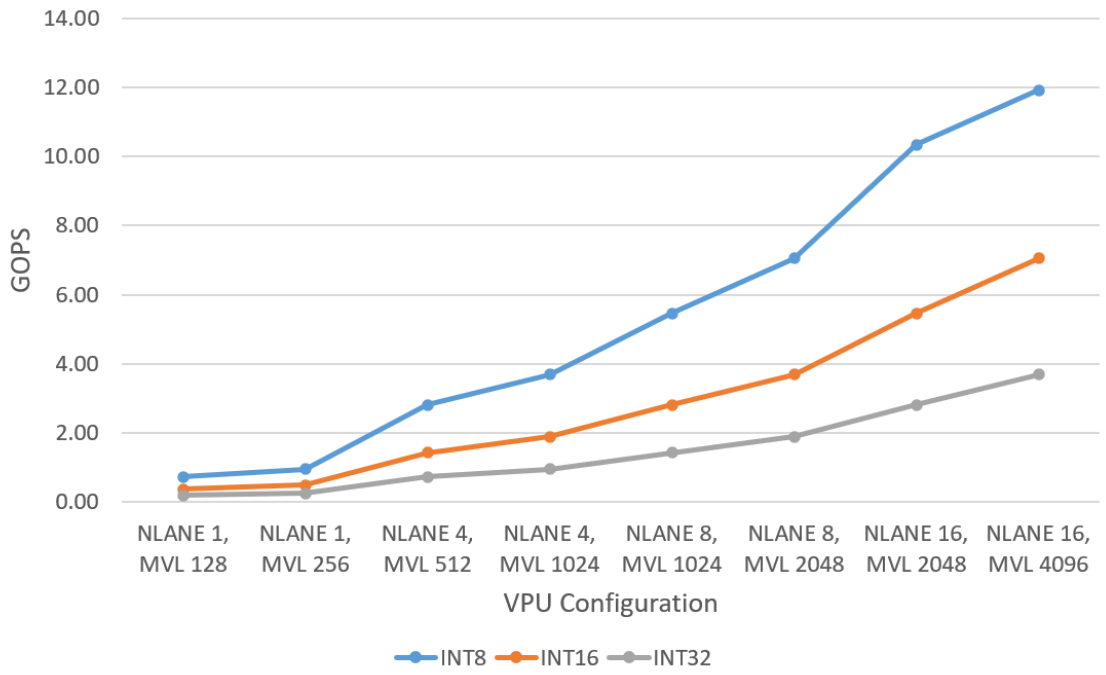


Figure 46: Giga operations per second (GOPS) for different VPU configuration

The cycle count required to perform the convolution operation is shown in table 3. As shown in table 3, the performance improved with a higher MVL for a given number of lanes. This is because of less instruction overhead. Since my architecture supports 4 INT8, 2 INT16, and 1 INT32 operation per lane, the execution time for INT8, INT16, and INT32 also increases linearly. Table 3 also illustrates that adding more lanes significantly improves the performance, owing to higher parallel operations.

I have calculated the Giga operations per second (GOPS) from the simulation result (using FMAX for Zynq Ultrasacle+ FPGA) and presented it in figure 46. To calculate GOPS, I took the total number of operations by multiplying the number of vector instructions executed by the VPU with the number of elements processed by a single vector instruction. I then measure the execution time for the convolution operation by multiplying the clock period of FMAX for the Zynq Ultrasacle+ FPGA device with the cycle count as given in table 3. After that, I get the GOPS from the total number of operations divided by the execution time.

For INT8 with NLANE=16 and MVL=4096, I achieved the maximum GOPS of 11.9. I have compared the GOPS values with a few other VPUs in table 4. It shows that my 16-lane VPU achieved remarkable GOPS compared to the other VPUs. The critical factors of my VPU performance are substantially higher operating frequency due to pipeline execution, reduced inter-lane communication because of distributed VRF, and an optimized reduction tree for lower latency.

Table 4: Comparison of GOPS with other VPUs

VPU	NLANE	GOPS
Vicuna[18]	32	10
VEGAS[8]	32	15
Proposal	16	11.9

4.5 Related Works

In the past, several soft vector processing units and co-processors have been presented, such as VESPA[68], which is a vector co-processor coupled to a MIPS processor, VIPERS[69], a vector processing unit. Both VESPA and VIPERS have poor support for the variable-width integer data type. While VEGAS[8] supports the variable width data type, it used scratchpad memory instead of the vector register file and did not have load-store instruction. VENICE[70] or MXP[71] has reduced the area of VEGAS, but still, it relies on scratchpad memory, and custom instructions need to be issued by the scalar core.

Initial RISC-V vector extension-based architecture was proposed in Ara[17] based on RVV version 0.5. This architecture is proposed for ASIC implementation and has a vector sliding unit (SLDU) that performs inter-lane data transfer involving data from all the vector register file (VRF) banks at once. Such banked VRF and inter-lane connection to SLDU will be a significant overhead for implementing multiple lanes for the FPGA soft vector processing unit. Because the number of elements to be transferred among the lanes will

increase with the number of lanes. Thus interconnecting these elements over the multiple lanes VRF banks will require a larger crossbar or other interconnect with various ports. Another critical point is that Ara architecture does not have support for vector reduction sum operation, which is essential for the dot/convolution operation.

Vicuna[18] is a timing-predictable vector processor implemented on FPGA. It has implemented the vector register file as monolithic multi-ported RAM and vector functional units read the whole vector register into a large shift register, which will overwhelm the FPGA resource with increasing vector length. SIMD-based vector processing unit was proposed in [66][67], which has a 2-stage pipeline that might limit FPGA operating frequency and does not support reduction instruction.

A pluggable vector processing unit is presented in [10] with banked SRAM as VRF. To reduce the cross-lane communication for vector permutation instruction, VRF banks are consolidated as a single unit and maintain the locking protocol to avoid hazards over different functional units. However, such centralized VRF with multiple bank arbitrators requires much FPGA logic, and complexity grows with the number of functional units and vector length. Spatz[72] is a vector processing unit with a shared L1 cache. Again it is based on centralized VRF and implemented using the latch for ASIC.

4.6 Summary

Exploiting data-level parallelism (DLP) is one of the effective ways to achieve high performance and efficiency. Parallel datapath architectures like vector processing can perform well with lower instruction overhead. In a Vector Architecture, the application can use any vector length that does not exceed the maximum vector length of that architecture. On the other hand, the vector lanes define the number of parallel datapaths that exist in that architecture. The performance also depends on the number of vector lanes.

In this research, I presented a vector processing unit with the ability to configure the MVL, thus allowing more elements to be processed with lower instruction overhead. I support a parameterized NLANE, which benefits a longer vector to execute in a shorter time using parallel lanes. Performance increases with MVL for a given NLANE. Again the performance also increases with NLANE for a given MVL. Maximum performance is obtained for higher MVL and NLANE.

While for a given NLANE, the FPGA resource utilization does not increase with MVL. And the resource utilization of FPGA also does not grow linearly with increasing NLANE.

My FPGA implementation has demonstrated 4x frequency improvement. And the FPGA implementation result shows my design has the lowest FPGA resource utilization. Considering the resource utilization, my proposed VPU has achieved higher performance than other VPUs. Given those factors, my architecture provides a resource-efficient implementation for FPGA while maintaining higher frequency.

5 Heterogenous Multicore Architecture with Vector Processing Unit

5.1 Motivation

Hardware development using FPGA needs knowledge of hardware description language (HDL), and design verification. Design and verification of hardware require a long development time and are much effort. The soft processor core is commonly employed to take care of regular and sequential tasks which saves the time and effort to develop a hardware controller for those tasks. Soft processor cores can be used for many different tasks without redesigning new hardware. When the system needs to manipulate multiple tasks, the multicore processor can be built by utilizing the soft processor to achieve task-level parallelism.

In many embedded applications, it processes an array of data, such as image processing, signal processing, etc. However, the performance of soft-core processors is limited as they process the data sequentially. When the application needs to process a large number of data, where data can be processed in parallel, the accelerator can be used to take the opportunity of data-level parallelism (DLP). Utilizing a vector processor as an accelerator saves the time and effort to develop the custom accelerator specific to the applications. Vector architectures can exploit the data level parallelism from its two important aspects, one is vector length which defines how many elements can be processed by a single vector instruction, and another is the number of lanes that produces multiple results per clock cycle. Thus Vector architecture provides the advantage of performance scalability while shortening the hardware development time.

5.1.1 Purpose

The purpose of this research is to propose a heterogeneous multicore architecture for FPGA. The architecture should accommodate multiple soft processor cores and a soft vector processing unit. Multiple processor cores provide the flexibility of task-level parallelism and the vector processing unit provides the flexibility to take advantage of data-level parallelism. A novel resource-efficient multicore architecture with higher performance and scalability.

5.1.2 Contribution

The contribution of this research is summarized below:

- I have proposed a microarchitecture of a 5-stage RISC-V processor (RVCoreP-32IMV) to utilize the vector processing unit, which decouples the scalar and vector execution, thus increasing the efficiency of the processor.
- I have presented the heterogeneous multicore architecture with hardware logic sharing among the cores.
- I implemented the multicore for FPGA and study the resource utilization with and without logic sharing and for multiple vector lanes and vector lengths.
- I evaluated the heterogeneous multicore performance by simulating a benchmark program. And compare the result with the homogeneous multicore processor and show the comparison of FPGA resources and performance.

5.2 Heterogeneous Architecture

The heterogeneous multicore systems customarily incorporate general-purpose processors with domain-specific accelerators. GRVI Phalanx [19] is such a heterogeneous multicore. GRVI is a 32-bit RISC-V soft processor implementing RV32I instructions. A shared memory cluster is composed of 8 GRVI processors and an accelerator as sown in figure 47.

Each cluster contains 12 BRAMs, and four BRAMs are used as 4 KB instruction memories (IRAMs). Each pair of processors share one IRAM. The other eight BRAMs form a 32 KB cluster shared memory (CRAM). It has twelve 32-bit wide ports. Four ports provide a 4-way banked interleaved memory for processor cores (PE). The eight PEs connect to the CRAM via four 2:1 concentrators and a 4x4 crossbar. Clusters are interconnected with each other using Hoplite NoC. Hoplite is a configurable directional 2D torus router that efficiently implements NOCs on FPGA. The Phalanx is a framework of an array of parallel processors and accelerators.

HERO[20] is a Heterogeneous system-on-chip platform that integrates a high-performance multicore host processor with programmable manycore accelerators (PMCAs). Figure 48 presents an overview of the system components and their connections. The host processor consists of ARM Cortex-A cores attached to a coherent interconnect. Each host core includes private hardware-managed caches and an MMU.

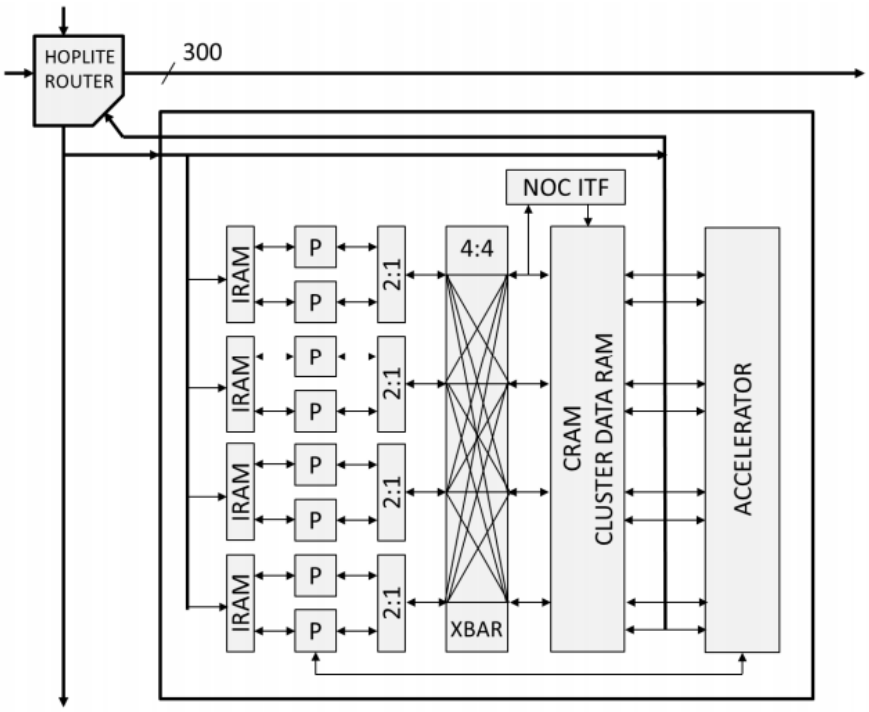


Figure 47: Eight GRVI Cluster and 288-bit payload Hoplite router[19] © 2016, IEEE

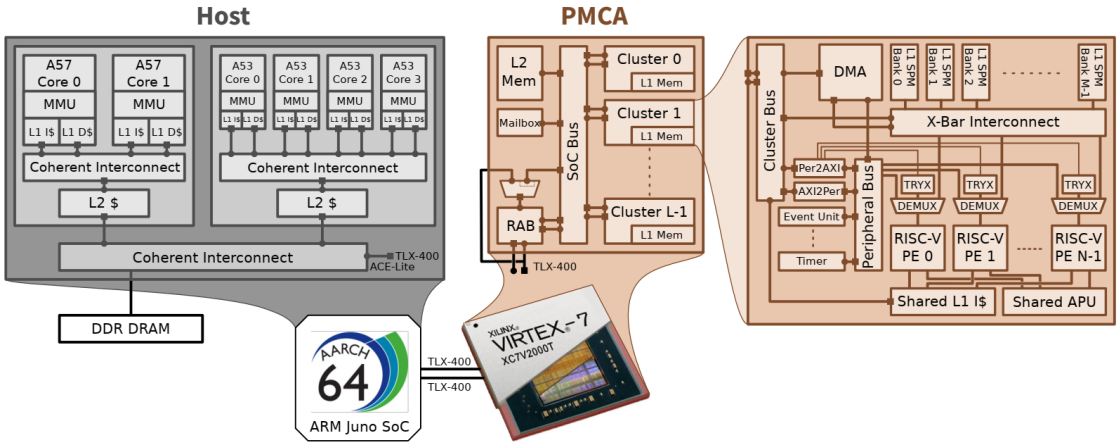


Figure 48: HERO's hardware architecture[20]

The host shares the main memory with the PMCAs through the system interconnect, which is coherent with the caches of the host. Instead of caches, a PMCA employs software-managed Scratchpad memories (SPM), together with DMA engines to move data. To access virtual memory, each PMCA is attached to a hybrid MMU, which is managed by the PMCA. The 32b RISC-V processing elements (PEs) within a cluster primarily operate on data present in the shared L1 SPM to which they connect through a low-latency,

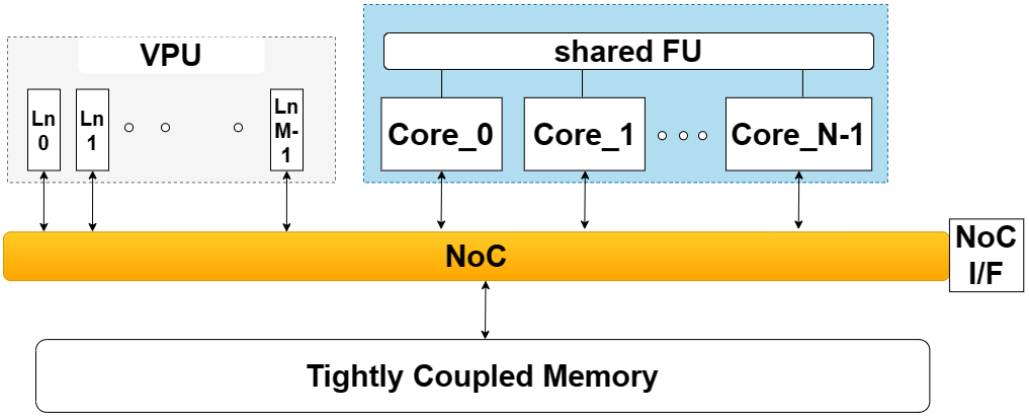


Figure 49: Abstract diagram of heterogeneous multicore architecture

logarithmic interconnect. The PEs use the DMA engine to copy data between the local L1 SPM among clusters and shared main memory.

5.3 Heterogeneous Multicore Architecture with VPU

The key idea of heterogeneous multicore architecture is presented in figure 49. There is N number of RISC-V scalar cores from core_0 to core_N-1 to execute different tasks/programs. To reduce the FPGA logic utilization, the cores will share functional units among them which are less frequently used. There is a vector processing unit (VPU) that will be used to accelerate the programs that contain data level parallelism (DLP). The VPU contains M number of parallel lanes to enhance the performance. The scalar cores and VPU are connected to the tightly coupled memory (TCM) through the network-on-chip (NoC). The NoC interface (NoC I/F) will allow us to connect such other multicores or the other peripherals and external memories.

For this study, I used a 4-core multicore processor with resource sharing[65]. However, the same architecture can be extended for other numbers of cores. 5-stage RVCoreP-32IM cores are reused for the design, which supports RV32IM instruction sets. In my design, only one core (I have chosen core_0) can handle the vector instructions. I have used the VPU design published in [73]. For that reason, core_0 should support the RV32IMV extension. To support the vector extension, the RVCoreP-32IM has been modified, and I have named the new core RVCoreP-32IMV.

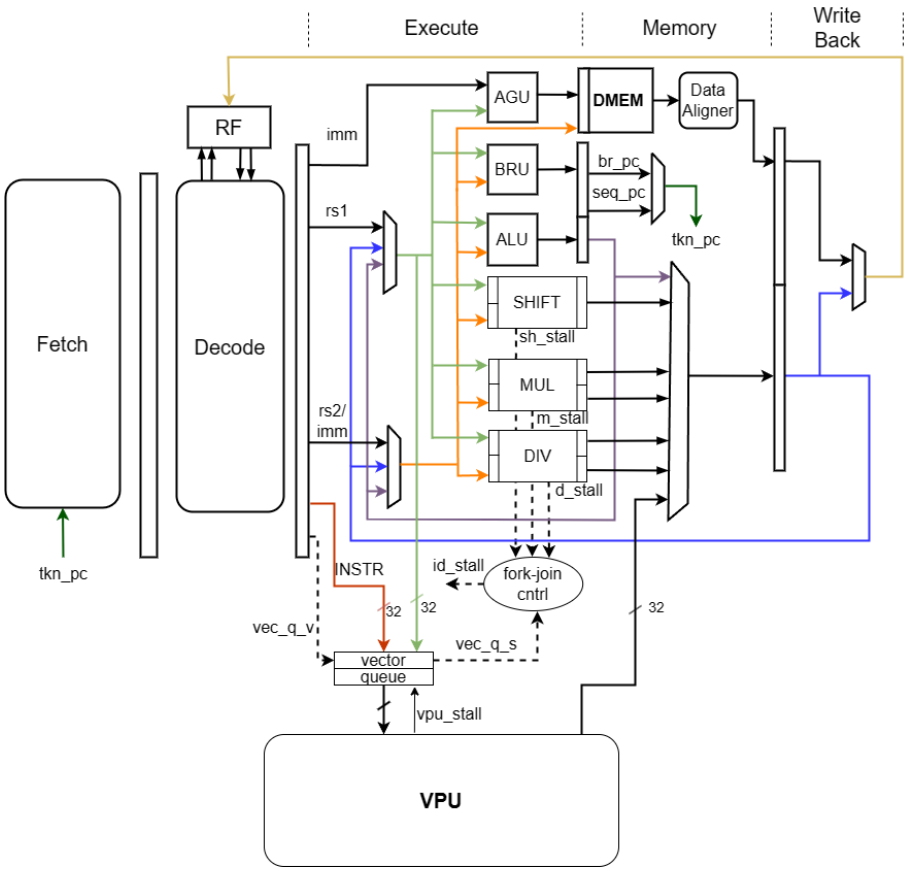


Figure 50: Microarchitecture of RVCoreP-32IMV processor core

5.3.1 RVCoreP-32IMV Microarchitecture

The detailed microarchitecture of RVCoreP-32IMV is shown in figure 50. It executes instructions in a 5-stage pipeline - instruction fetch (IF), instruction decode (ID), execute (EX), memory (ME), and write back (WB), the same as RVCoreP-32IM. Vector instructions are fetched from the instruction memory as it fetches other RV32IM instructions. Decoder can partially decode the vector instruction - whether the fetched instruction is a vector instruction and if there is any value that will be returned from the VPU, which will be written back to the register file (RF). When the decoder decodes the vector instructions, the execution stage pushes the whole instruction to the vector instruction queue. There are several vector instructions that require the scalar register value from the processor register file for vector instruction execution. The required register for vector instruction comes from the rs1 register. The rs1 register value after data forwarding is also pushed together with the vector instruction.

Vector instruction queue work as an interface between the scalar processor core and a

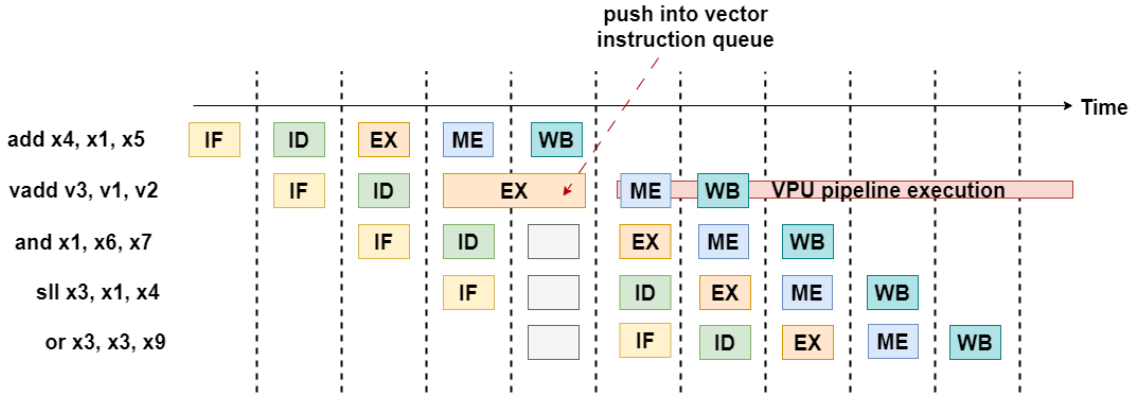


Figure 51: Pipeline flow diagram of scalar instruction execution and vector instruction en-queue

vector processing unit. For this design I have used the queue size 2, that is it can store 2 vector instructions. The scalar core can push to this queue when there is any available slot in the queue. When VPU pops an instruction from the queue it will assert "vpu_stall" signal to indicate that it is processing a vector instruction. However, if there is another vector instruction comes in and the queue is not full then execute stage will be able to push the vector instruction to the queue. When the vector instruction queue is full the "vec_q_s" signal is asserted by the vector instruction queue to indicate the queue is full.

Execute stage takes 2 clock cycles to push vector instruction into the queue, as a result, the pipeline is stalled for 1 clock cycle. The pipeline flow diagram of the scalar core is shown in figure 51. Once the vector instruction is pushed, the scalar core can execute other scalar instructions while VPU is executing the vector instruction. It decouples the scalar and vector execution.

Few vector instruction execution returns a single value to the scalar core, for example, the vsetvli instruction returns the vector length. The returned value from the VPU is written back into the scalar register file into the rd register. When such instruction is decoded the decoder signals the execution stage that the returned value is required from the VPU. Such vector instruction is regarded as multi-cycle execution and handled by the fork-join controller. The original RVCoreP-32IM has latency-insensitive execution controlled by the fork-join controller, which takes an input of stall signals from the shifter (sh_stall), multiplier (m_stall), and divider (d_stall) to stall the decode and fetch stage. In addition to that the RVCoreP-32IMV fork-join controller takes stall inputs from the VPU (vec_q_s) when the VPU will return data. The scalar core will be stalled until the VPU executes the instruction and returns the value. Since the VPU executes the instructions in-order,

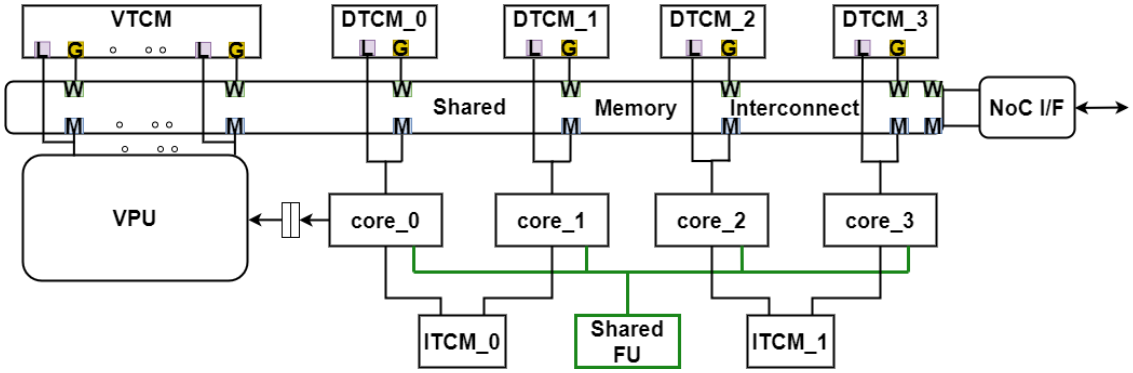


Figure 52: Heterogeneous multicore architecture

when the VPU will return the value it can be assumed that all the previous instructions pushed into the queue have been executed. Such kind of instructions can be used for synchronization between the scalar core and VPU.

5.3.2 Heterogeneous Multicore Architecture

Figure 52 depicts my proposed heterogeneous multicore architecture of 4-cores. Only core_0 is RVCoreP-32IMV and the other cores are RVCoreP-32IM (5-stage). Thus only core_0 can execute the vector instructions. The programs that can be accelerated by the vector processing will be assigned to the core_0. Other sequential programs can be assigned to the rest of the cores. For example, while core_0 is executing data-parallel applications, other cores can handle the data pre-processing, data post-processing, peripheral controls, security, and others. The shifter, multiplier, and divider are shared among the cores in the shared functional unit (FU).

Tightly coupled memories are used for scalar core instruction (ITCM) and data memory (DTCM) and for VPU memory (VTCM). All the memory's data width is 32-bit. ITCM, DTCM, and VTCM are configured as dual-port RAM, which utilizes the FPGA block RAM (BRAM). It does not have a memory management unit (MMU) as it is designed for bare-metal firmware execution and memory addresses are directly mapped. the dual port of ITCM is shared between 2 cores. DTCM's dual port is configured as a local (L) and global (G) port. When cores access its local memory it goes through the L port. And when cores access the shared memory it goes through the shared memory interconnect. The shared memory interconnects the master (M) port is connected to the cores and the worker (W) port is connected to the G port of DTCM.

The VTCM is organized as banks, where the number of banks is equal to the number of

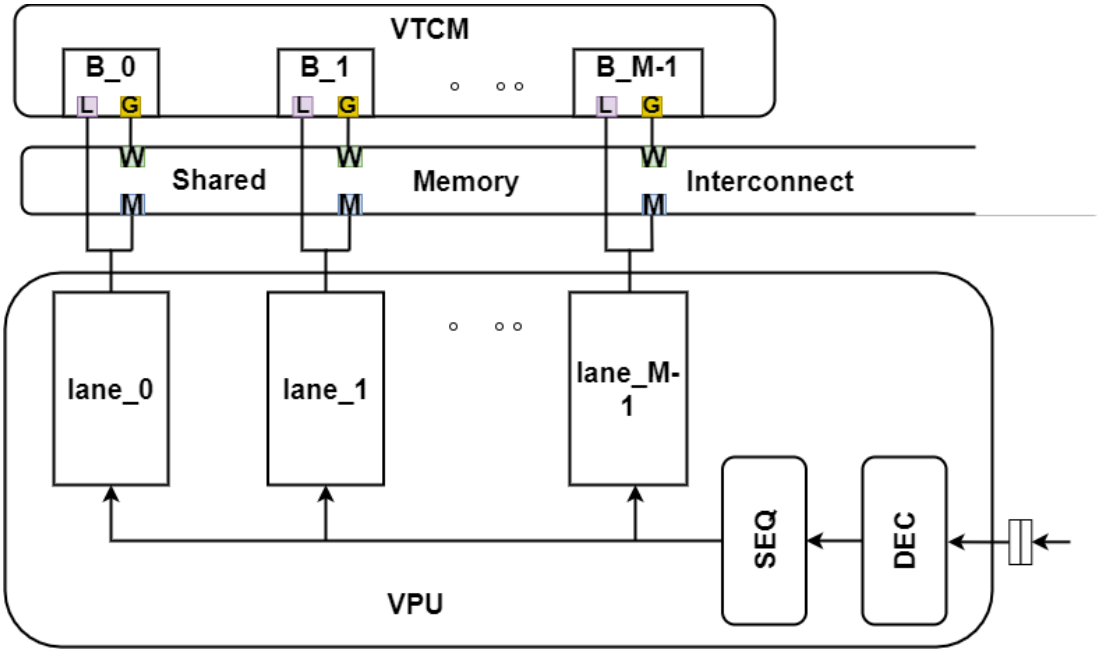


Figure 53: Vector processing unit and vector TCM diagram

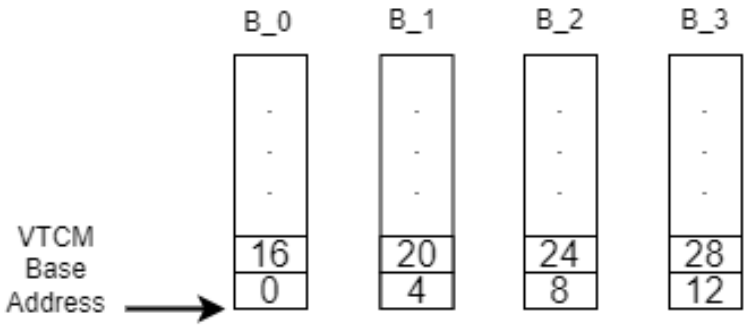


Figure 54: Vector TCM organization for 4 banks

lanes in VPU. The organization of VTCM and VPU is shown in figure 53, where VPU has M number of lanes and VTCM has M number of banks from B_0 to B_M-1. The vector instructions are popped from the queue is done by the vector decoder (DEC), then the sequencer (SEQ) dispatches the instructions to lanes. The lanes are comprised of vector register file (VRF), execute and write back stage. The vector load store unit from each lane has individual memory addresses and data channels.

VTCM banks store the word (32-bit) in an interleaved way. An example of VTCM memory organization for 4 banks (i.e. 4 lanes) is shown in figure 54. In this figure 54, the numbers represent the byte offset related to the VTCM base address. Since each back stores 32-bit (4-byte) the offsets are incremented in 4 among the banks. When a given lane accesses

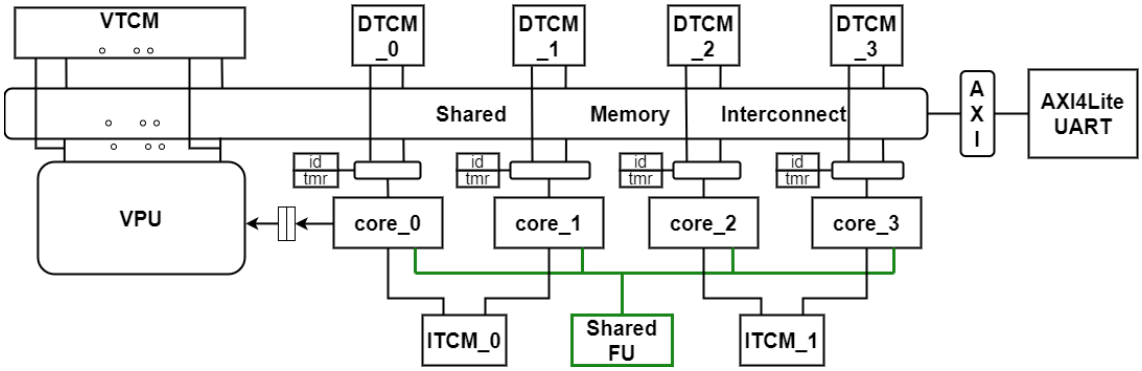


Figure 55: The simple SoC for evaluation in FPGA

the memory it goes through the VTCM local port if the requested memory address match to the VTCM memory region and the byte offset of the memory address matches with the corresponding VTCM bank offset. For example, according to the figure 54, if lane_1 accesses the memory in the VTCM region and address offset is 4, 20, 36 and so on it will be accessed through the VTCM B_1 local port. Other than that it will be routed through the memory interconnect. In this design, I have a ring bus as a shared memory interconnect.

5.4 Evaluation

RVCoreP-32IMV and RVCoreP-32IM both use a two-stage pipelined gshare branch predictor with a pattern history table (PHT) of 8,192 entries and a branch target buffer (BTB) of 512 entries. I prepare a simple SoC to evaluate the FPGA implementation of heterogeneous multicore processors. The SoC block diagram is shown in Figure 55, where an AXI UART peripheral is connected to a multicore by an AXI4 Lite bus. Each core has its own hardware timer (tmr) and core id register (id). The core id register is read by the cores to identify the unique id of individual cores in the multicore, like 0 for core_0, 1 for core_1, and so on.

I use registers as a buffer to store the memory requests from the cores. This double-depth buffer is in the interface of shared memory interconnect to CPU cores. The minimum latency for load and store instruction execution is 3 clock cycles (1 clock for buffering at shared memory interconnect, 1 clock to decide whether to route to local memory port or route to ring bus and 1 clock for BRAM access, and 1 clock for memory stage execution). Programs are preloaded into the ITCMs.

5.4.1 FPGA Results

I evaluate the operating frequency and hardware resource utilization for Nexys A7 board [38] containing xc7a-100tcsg324-1 FPGA, which is a family of Xilinx Artix-7 FPGA of speed grade -1. In this FPGA for 32-bit data width, BRAM size is 4KB each. Xilinx Vivado 2021.1 is used to evaluate operating frequency and hardware resource utilization. Synthesis, placement, and routing strategy were set by Vivado default.

I performed the logic synthesis and placement, and routing by incrementally stepping the clock constraint in the 5MHz scale. The highest frequency that achieves the positive timing slack is taken as the operating frequency. For hardware resource evaluation, I used the result of placement and routing at the maximum operating frequency. For all the multicore configurations the achieved maximum operating frequency is 120MHZ.

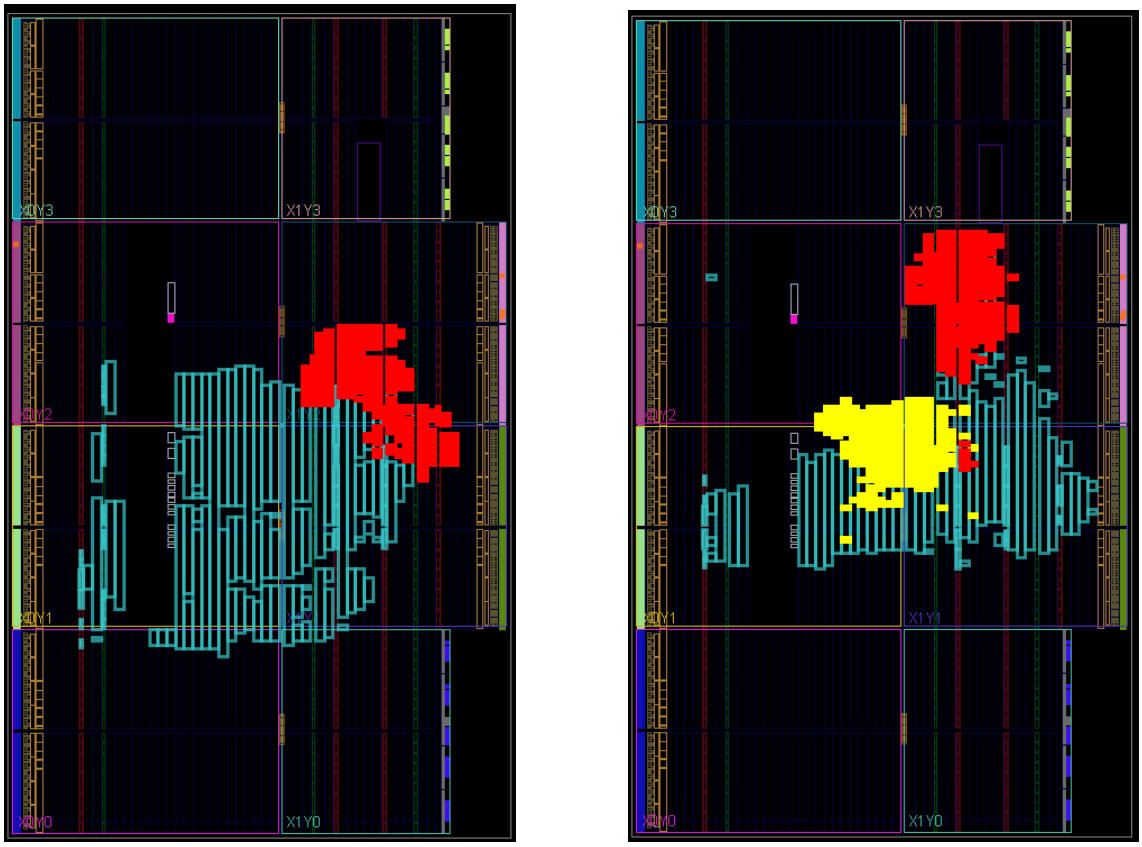
FPGA implementation result is shown in Table 1, where I take LUT, register, and DSP as FPGA resources. I have presented the resource usage for the multicore when no functional unit is shared (no_shared) and when the shifter, multiplier, and divider are shared (shared). In Table 1, the first entry shows the logic resources required for 4 RVCoreP-32IM (without VPU) and shared functional units (FU). In the other entries, it represent the combined logic resources required by the 4 cores, VPU, and shared functional units (FU). I have shown four VPU configurations, where the number of lanes (NLANE) is 1, 4, 8, and 16, and the maximum vector length (MVL) is 256, 1024, 2048, and 4096.

Table 1: FPGA implementation of heterogeneous multicore processor

	LUT		Register		DSP	
	no_shared	shared	no_shared	shared	no_shared	shared
RVCoreP-32IM only	**5,967**	4,407	3,685	3,538	**16**	4
NLANE 1, MVL 256	7,717	**6,330**	4,835	4,508	21	**9**
NLANE 4, MVL 1024	11,013	9,734	6,139	5,752	36	24
NLANE 8, MVL 2048	15,432	14,092	7,977	7,650	56	44
NLANE 16, MVL 4096	**24,377**	**23,153**	11,538	11,210	**96**	**84**

The table 2, shows the resource-saving for the multicore when shifter, multiplier, and divider are shared. To calculate the resource-saving I compared the same VPU configuration of no_shared with the shared configuration. Table 1 depicts that, with the increasing number of lanes, the VPU resource utilization becomes dominant. Figure 56 shows the FPGA layout after place and route for single lane VPU without and with sharing configuration. And figure 57 presents the layout for 16 lanes VPU no_shared and shared configuration.

Sharing resources in the multicore reduces the LUT utilization to 18% and the DSP



(a) no_shared configuration

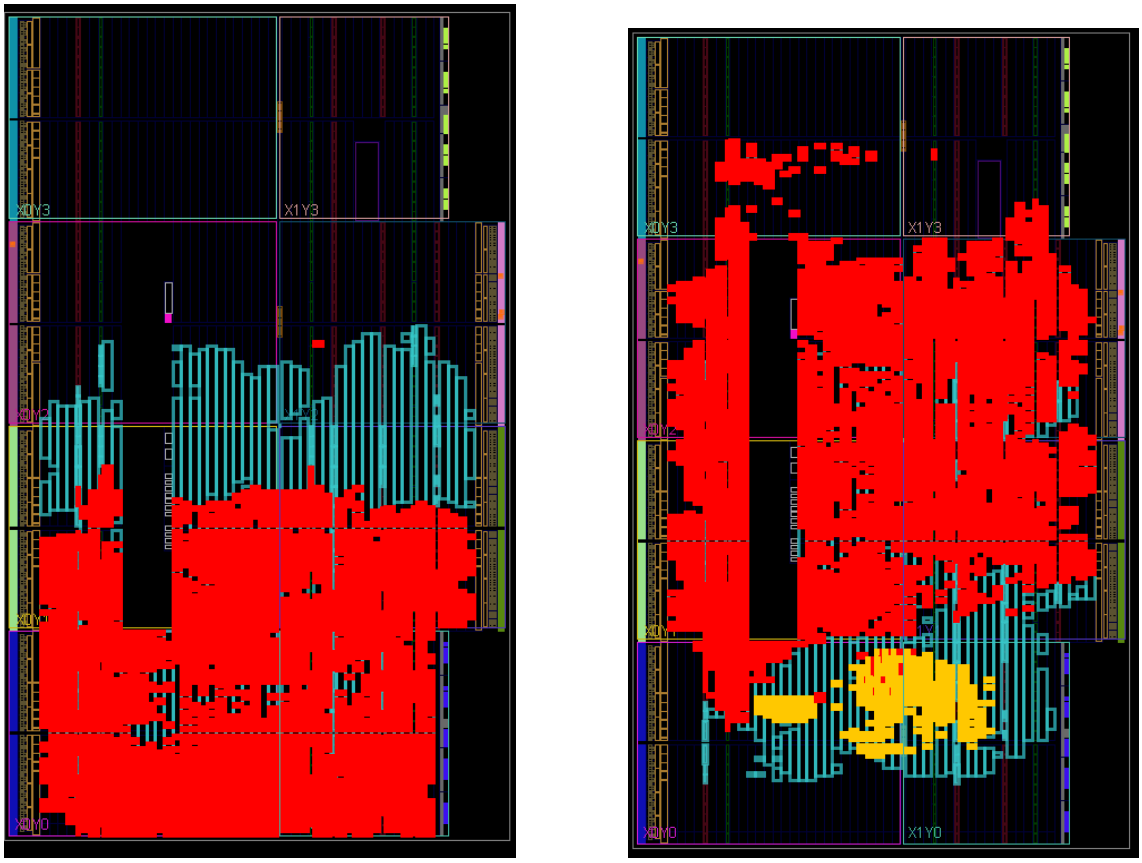
(b) shared configuration

Figure 56: Artix-7 FPGA implementation layout for 4-core and VPU with NLANE=1. The shared functional unit is highlighted in yellow color and VPU is highlighted in red color

utilization to 57% for single lanes VPU. Interestingly, sharing functional units reduces the logic utilization for the increasing number of lanes as well. For example, the LUT usage was reduced to 5% for the 16 lanes compared to single lane VPU has savings of 18%. That is even though the lanes increased to 16 times but logic savings does not reduce 16 folds. Thanks to the VPU architecture, its resource does not grow linearly with the number of lanes.

Table 2: FPGA resource saving for heterogeneous multicore processor

	LUT	Register	DSP
NLANE 1, MVL 256	**18%**	7%	**57%**
NLANE 4, MVL 1024	12%	6%	33%
NLANE 8, MVL 2048	9%	4%	21%
NLANE 16, MVL 4096	**5%**	3%	**13%**



(a) no_shared configuration

(b) shared configuration

Figure 57: Artix-7 FPGA implementation layout for 4-core and VPU with NLANE=16. The shared functional unit is highlighted in yellow color and VPU is highlighted in red color

In table 3, I have compared the resource-shared 4-core processor which integrates single lane, MVL=256-bit VPU with other multicore processors/clusters. The Artix-7 is a relatively low-speed FPGA compared to the Ultrascale family device. For comparison, I have shown the FPGA operating frequency of the proposed design for both Artix-7 and Zynq Ultrascale+ devices. This table 3 shows that except for GRVI[19], compared to all other multicore processors my proposed architecture consumes significantly lesser resources and has a higher operating frequency. GRVI[19] implements only RV32I ISA extension which does not have the shifter, multiplier, and divider. On the other hand, my proposed architecture supports RV32IM ISA extension and it also includes a VPU. Given those architectural features, my proposed architecture is more resource efficient and with higher operating frequency compared to the GRVI[19].

Table 3: FPGA resource and frequency comparison with other multicore architectures

Reference	No. of cores	ISA	LUT	Frequency	FPGA
OpenPiton+Ariane[12]	1	RV64GC	20K	30MHz	Artix-7
GRVI[19]	8	RV32I	7.1K	150MHz	Kintex UltraScale
Many core[13]	4	RV32IMC	36.5K	120MHz	Virtex Ultrascale+
Agiler[74]	4	RV32IMC	30K	120MHz	Virtex Ultrascale+
Proposal	4	RV32IM(V)	6.3K	120MHz/ 260MHz	Artix-7/ Zynq Ultrascale+

5.4.2 Simulation Results

To study the performance of the heterogeneous multicore, I have simulated the benchmark program compiling with the RV32IM extension and RV32IMV extension. Since while writing this thesis paper, the GCC compiler support for vector extension is still under development and kept in a separate branch rvv_next [75]. I used the GCC compiler version 12.0.1 20220505 (prerelease).

At first, I compare the single-core RVCoreP-32IMV performance with respect to the RVCoreP-32IM. I have used the aXPY (a*X+Y) and general matrix-vector multiplication (GEMV) program written in C language. For both programs, I used a 32-bit integer data type. For the aXPY program, I choose the X and Y vector size of 512. And for the GEMV, the number of columns is 512, and the number of rows is 64. For the vector extension-based program, I have used the assembly sub-program.

Table 4: Simulation result of single core RVCoreP-32IM with RVCoreP-32IMV

	aXPY		GEMV	
	Cycles	gain	Cycles	gain
RVCoreP-32IM	7,736	1.0	524,989	1.0
NLANE 1, MVL 256	3,480	1.70	164,605	2.43
NLANE 4, MVL 1024	888	6.64	41,725	9.59
NLANE 8, MVL 2048	456	12.94	21,245	18.84
NLANE 16, MVL 4096	240	24.58	11,005	36.37

Table 4 shows the simulation result for the single core. For the RVCoreP-32IMV, I use 4 VPU configurations. The cycles column represents the number of clock cycles required to simulate the program. I then calculated the performance gain of RVCoreP-32IMV with respect to RVCoreP-32IM and presented it in the gain column. To calculate the performance gain I compare the execution time of RVCoreP-32IMV with RVCoreP-32IM. The execution time is determined by multiplying the simulation cycle with the clock

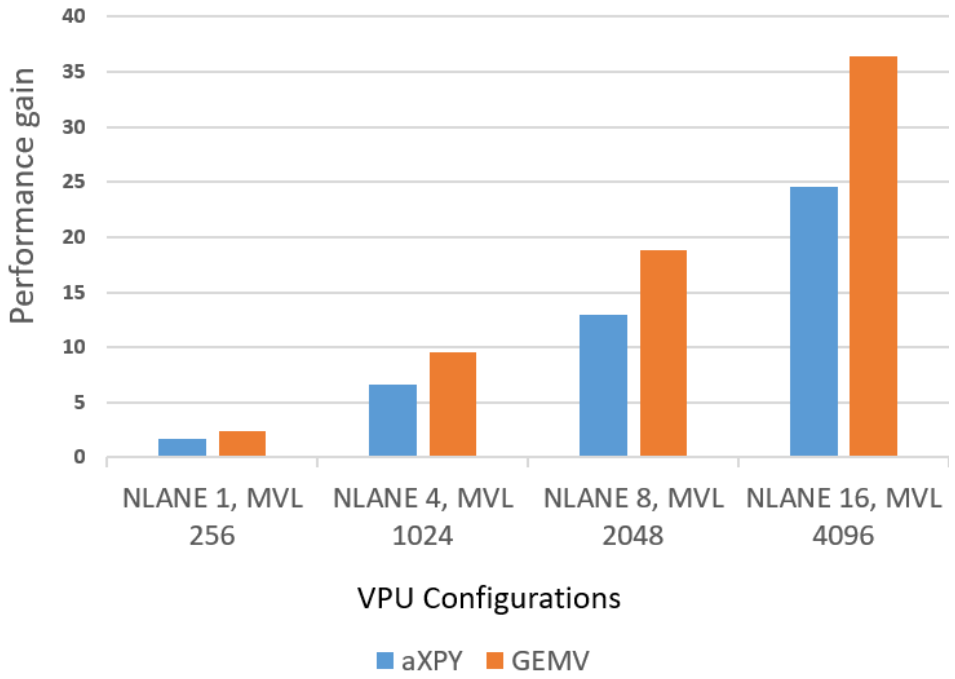


Figure 58: Performance comparison of single core RVCoreP-32IMV with RVCoreP-32IM

period obtained from the corresponding core's evaluated maximum operating frequency. The maximum operating frequency for the RVCoreP-32IM is 164MHz, and for RVCoreP-32IMV is 125MHz.

Table 4 demonstrates that RVCoreP-32IMV performs better than RVCoreP-32IM. For a single lane (NLANE=1), the performance gain for aXPY is 1.7 times, and GEMV is 2.43 times compared to RVCoreP-32IM. Figure 58 shows the performance gain in the bar diagram. From this figure 58, we can see that the performance increases linearly with the number of lanes as well as vector length. Since the GEMV is more computationally intensive than the aXPY, all vector configuration of RVCoreP-32IMV achieves better performance for the GEMV program than the aXPY program. For 16 lanes and 4,096-bit vector length, the RVCoreP-32IMV achieves 24.58 and 36.37 times, respectively for the aXPY and GEMV programs.

I have used the GEMV program to evaluate the multicore without logic sharing and with shifter, multiplier, and divider sharing. In this study, I used the core_0 for executing the GEMV program as a parallel application. And for sequential execution, the core_1 executes string search, core_2 executes rsa encryption, and core_3 executes hash function infinitely. I took the required number of simulation cycles to execute the GEMV. The simulation result is shown in table 5. The RVCoreP-32IM no_shared and shared row

Table 5: GEMV simulation result for the multicore

	Cycle	gain
RVCoreP-32IM no_shared	591,739	1
RVCoreP-32IM shared	608,123	0.97
NLANE 1, MVL 256	166,251	3.56
NLANE 4, MVL 1024	42,142	14.04
NLANE 8, MVL 2048	21,500	27.52
NLANE 16, MVL 4096	11,148	53.08

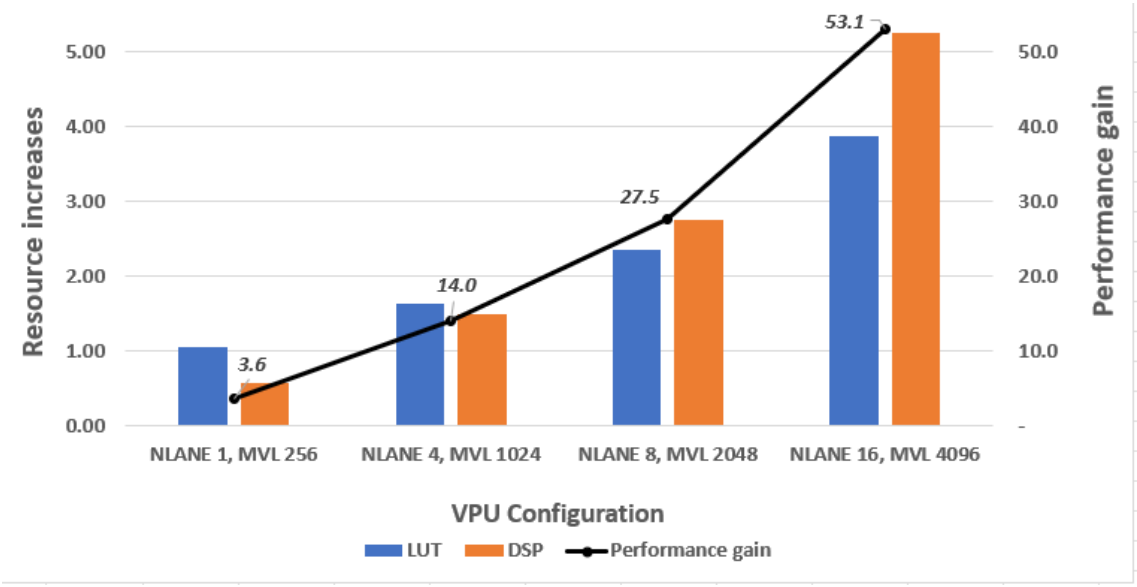


Figure 59: Comparison of FPGA resource increment versus performance gain for heterogeneous multicore

represents the homogeneous multicore using only scalar cores, i.e., four RVCoreP-32IM. For the heterogeneous multicore, I took the simulation cycles for the RVCoreP-32IMV execution with a shared FU configuration. I have then calculated the performance gain with respect to multicore RVCoreP-32IM no_shared configuration and presented it in the gain column.

Since both the scalar homogeneous multicore and heterogeneous multicore with RVCoreP-32IMV achieves the same operating frequency, the performance gain is calculated directly by comparing the simulation cycle with homogeneous multicore no_shared configuration. Table 5 presents that heterogeneous multicore with 16-lane RVCoreP-32IMV achieves 53 times better performance than homogeneous multicore.

The LUT and DSP are key resources of FPGA devices as more registers are available in the

FPGA than others. I compared the LUT and DSP resource increment for the heterogeneous shared multicore to the homogeneous multicore no_shared configuration. At the expense of FPGA resources, how much performance gain is obtained is demonstrated in figure 59. Interestingly heterogeneous shared multicore use similar LUT resources and around half of the DSP resources of the homogeneous no_shared configuration, yet the performance gain is 3.6 times. For 16 lanes and 4096-bit vector length heterogeneous multicore, it achieved a significant performance gain of 53 times though LUT usage increased to 3.9 times and DSP usage increased 5.25 times.

5.5 Related Works

Heterogeneous multicore with SIMD/MIMD accelerator was proposed in [76], which uses a custom address generation unit (AGU) for memory accesses, 1D and 2D processing elements as SIMD and MIMD accelerator. It needs to program the AGU and PE separately, making it difficult to develop programs for this system. GRVI Phalanx[19] has proposed a heterogeneous array of parallel processors and accelerator framework called Phalanx. In Phalanx, groups of processors and accelerators form shared memory clusters. However, the implemented RISC-V core only supports RV32I extension, and the developer needs to implement a custom accelerator for the cluster.

CoreVA-MPSoC[77] proposes a hierarchical architecture-based many-core processor. Few CPU creates a cluster, and several clusters are connected through NoC. Clusters use software-managed scratchpad memories. It shows that utilizing scratchpad memories as shared memory increases the application performance. However, the CPUs are custom VLIW processors and do not support accelerators. HERO[20] is a RISC-V-based heterogeneous system on chip architecture that integrates programmable manycore accelerators (PMCAs) with ARM host processor. But the scalar RISC-V cores only perform the acceleration, which will be limited.

Memphis [78] framework to generate scalable heterogeneous many-core SoCs. Single Processor cores with local memories are connected through the NoC. A set of peripherals connected at the boundaries of this multi-core. Therefore, it does not support the tightly coupled integration of accelerators directly with the core. Heterogeneous multicore architecture for edge computing is proposed in [79]. It considers the heterogeneity at the processor microarchitecture-level design, such as supporting or not supporting floating point instruction and in-order or out-of-order computing capabilities. Such heterogeneous architecture is unsuitable for FPGA as the higher microarchitectural level feature will

consume much more logic and limit the number of cores.

A scalable FPGA-based RISC-V many-core platform using a lightweight NoC is presented in [13]. It features a homogeneous multi-core compute tile based on 32-bit RISC-V cores. The cores, shared, and local memory are connected through an AXI interconnect within the tile. Agiler[74] also has a similar architecture but introduces heterogeneity by multi/single-core compute tiles that support 32-/64-bit RISC-V ISAs with different memory hierarchies. , The Agiler architecture supports design/run-time re-configurations to change numbers and types of compute tiles for several many-core configurations.

To avoid the complexity of the cache coherency in the RISC-V based multicore system, [80] proposed temporary caching (TC). It only caches data that will not be shared for a certain period of time. However, TC and non-TC data are on the same cache line, which can cause a fatal system error. To avoid such conditions, users should carefully develop the program and use the API to handle data relevant to the TC. HEROv2 [81] is a heterogeneous computing research platform that provides all hardware and software required to develop, compile, and run single-source, single-binary heterogeneous applications. It offloads and shares data from an application-class 64-bit host processor to a programmable 32-bit parallel accelerator. However, it uses significant FPGA resources, and operating frequency is also limited.

5.6 Summary

I presented the microarchitecture of a 5-stage RISC-V core supporting vector extension. The scalar and vector instructions are executed in parallel until there is any data dependency from the VPU to the scalar core. Even a single-lane VPU achieves more than 2.4 times higher performance than a scalar processor because the hardware performs the iteration over the vector elements compared to loop iteration by the scalar core. VPU enhances the single-core performance more than 36 times.

I propose the heterogeneous multicore architecture based on the proposed resource-efficient VPU and 5-stage scalar processor. The architecture supports hardware resource sharing among the scalar cores, which further reduces the logic resources of the multicore processor. The FPGA implementation results demonstrate that sharing can save LUT up to 18% and DSP utilization up to 57%. Scalar cores are suitable for different sequential tasks that can be executed in parallel, and VPU is devoted to data-parallel execution. VPU can boost the multicore performance more than 53 times.

My proposed heterogeneous architecture provides the acceleration without developing any custom hardware accelerator and only eases the application development with software. Moreover, my proposed architecture is more resource-efficient and performs better than the other state-of-the-art RISC-V based multicore processors. My proposed architecture is also scalable to performance.

6 Conclusion

6.1 Concluding Remarks

The Internet-of-Things (IoT) movement pushes the computation from cloud to edge devices, which insistence on more processing capabilities to meet the task requirements at the edge. Multicore or MPSoC with scalable architectures is a promising solution for this computation demand. Furthermore, heterogeneous multicore solutions consisting of different core types can offer a performance advantage[82].

Such heterogeneous multicore can be implemented in the FPGA. There are benefits of utilizing the FPGA for such applications, like FPGAs are re-configurable devices; thus, users can reconfigure the system and adapt the architectures according to the requirement. Moreover, the FPGA provides a faster path to realize the system than the ASIC development. Since multicore architectures consume many resources in FPGA, I have proposed microarchitectures to accomplish the heterogeneous multicore in FPGA.

In Chapter 3, I propose 4-stage and 5-stage pipeline architectures for FPGAs with hardware resource sharing. I study the impact of resource usage on shared functional units (such as shifters, multipliers, and dividers) versus non-shared units. Sharing reduced DSP usage significantly, regardless of the pipeline architecture. A considerable amount of LUT usage was reduced. A 4-stage pipeline is slightly smaller than a 5-stage pipeline for all multicore shared configurations. Nonetheless, FPGA logic usage is still an important design factor due to fixed and limited resources. Simulation of benchmark shows that all kinds of workloads are not affected by the hardware sharing; mainly, it depends on the instruction stream and ratio of instructions that utilize the shared functional units. The application developer can choose the proper sharing configuration and pipeline stages according to their requirements. My proposed multicore architecture maximizes the BRAM utilization.

Chapter 4 introduced a vector processing unit with vector length and lane configuration capability. Increasing vector length enables the processing of more elements with less instruction overhead. I design with a parameterized number of lanes which uses parallel datapaths to accelerate the execution of a longer vector. For a given number of lanes, performance improves with vector length, but FPGA resource usage does not increases. Additionally, the resource use of FPGA does not increase linearly as the number of lanes increases. FPGA implementation shows that my design has the lowest FPGA resource use and a significant boost in frequency. Given those elements, my design offers a resource-

efficient FPGA implementation while retaining greater frequency and scalability.

In Chapter 5, I propose a heterogeneous multicore architecture based on the resource-efficient vector processing unit from Chapter 4 and the 5-stage RISC-V scalar processor proposed in Chapter 3. I present the efficient integration of the vector processing unit with the scalar core. The multicore architecture supports the sharing of hardware resources between scalar cores, thus further reducing the logic resources of multicore processors. FPGA implementation results show that sharing can save up a considerable amount of LUTs and DSP usage, even with the vector processing unit coupled to the multicore. Scalar cores are advantageous for various sequential tasks; in multicore, these sequential tasks can be executed in parallel, while vector processing units are dedicated to data-parallel execution, boosting the multicore performance for the data level parallelism. My proposed heterogeneous architecture provides acceleration without the need to develop custom hardware accelerators and simplifies application development using only software.

6.2 Future Works

In this thesis paper, I have studied the shared functional unit containing only one shifter, multiplier, and divider depending on the sharing configuration. However, the shift instructions occur more repeatedly than the multiplication and division instructions. Future studies which take the different sharing granularity, such as having two shifters in the sharing functional unit, will need to be performed. Moreover, other RISC-V ISA extension's usability and applicability to the shared functional unit can be investigated.

My proposed multicore is inherently suitable for asymmetric multiprocessing (AMP) from a software development perspective; AMP is the most popular and practical approach for heterogeneous embedded multicore systems. The two essential features required to develop the AMP are process management (including boot sequence) and inter-core communication. My proposed multicore has both features, as the process is managed by selectively loading programs in the instruction TCM for each core, and inter-core communication is possible through the shared memory interconnection.

The dissertation does not focus on the development of an AMP framework to manage and execute tasks in the proposed multicore architecture. Such a framework also needs to investigate the task scheduling algorithm with an optimal balance of workloads, such that sharing functional units does not provide much impact on the performance. For example, selectively assigning division instruction-rich tasks to a single core may improve the performance as it will not be much arbitrated with other cores for divider operation.

My proposed vector processing unit does not support permutation and widening instruction. How distributed vector register files can still be used while supporting those instructions at reduced FPGA logic need further research. Furthermore, sharing vector lanes among the cores could be compelling for FPGA heterogeneous multicore research.

References

[1] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4, 2016.

[3] Ben Keller. Risc-v, spike, and the rocket core. URL https://inst.eecs.berkeley.edu//~cs250/fa13/handouts/lab2-riscv.pdf#13.

[4] Openhw group core-v cv32e40p risc-v ip. URL https://github.com/openhwgroup/cv32e40p.

[5] Hiromu Miyazaki, Takuto Kanamori, Md Ashraful Islam, and Kenji Kise. Rvcorep: An optimized risc-v soft processor of five-stage pipelining. *IEICE TRANSACTIONS on Information and Systems*, 103(12):2494–2503, 2020.

[6] The berkeley out-of-order risc-v processor. URL https://github.com/riscv-boom/riscv-boom.

[7] Susumu Mashimo, Akifumi Fujita, Reoma Matsuo, Seiya Akaki, Akifumi Fukuda, Toru Koizumi, Junichiro Kadomoto, Hidetsugu Irie, Masahiro Goshima, Koji Inoue, et al. An open source fpga-optimized out-of-order risc-v soft processor. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 63–71. IEEE, 2019.

[8] Christopher H Chou, Aaron Severance, Alex D Brant, Zhiduo Liu, Saurabh Sant, and Guy GF Lemieux. Vegas: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 15–24, 2011.

[9] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202. IEEE, 2014.

[10] Vincenzo Maisto and Alessandro Cilardo. A pluggable vector unit for risc-v vector extension. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1143–1148. IEEE, 2022.

[11] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, et al. Intel alder lake cpu architectures. *IEEE Micro*, 42(3):13–19, 2022.

[12] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. Openpiton+ ariane: The first open-source, smp linux-booting risc-v system scaling from one to many cores. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, pages 1–6, 2019.

[13] Ahmed Kamaleldin, Salma Hesham, and Diana Göhringer. Towards a modular risc-v based many-core architecture for fpga accelerators. *IEEE Access*, 8:148812–148826, 2020.

[14] Romain Dolbeau and André Seznec. *CASH: Revisiting hardware sharing in single-chip parallel processor*. PhD thesis, INRIA ⟨inria-00071925⟩, 2002.

[15] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, 2011.

[16] Peter N. Glaskowsky. Nvidia's fermi: The first complete gpu computing architecture, 2009. URL https://www.nvidia.com/content/pdf/fermi_white_papers/p.glaskowsky_nvidia%27s_fermi-the_first_complete_gpu_architecture.pdf.

[17] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):530–543, 2019.

[18] Michael Platzer and Peter Puschner. Vicuna: a timing-predictable risc-v vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[19] Jan Gray. Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20. IEEE, 2016.

[20] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. Hero: Heterogeneous embedded research platform for exploring risc-v manycore accelerators on fpga. In *Proc. Computer Architecture Research on RISC-V Workshop (CARRV)*, pages 1–7, 2017.

[21] Number of iot connected devices worldwide 2019-2021, with forecasts to 2030. URL https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[22] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

[23] History of risc-v. URL https://riscv.org/about/history/.

[24] Chisel/firrtl hardware compiler framework. URL https://www.chisel-lang.org/.

[25] Berkeley hardware floating-point units. URL https://github.com/ucb-bar/berkeley-hardfloat.

[26] Rocket chip generator. URL https://github.com/chipsalliance/rocket-chip.

[27] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[28] Scott McFarling. Combining branch predictors. Technical report, Digital Western Research Laboratory, 1993.

[29] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. Boomv2: an open-source out-of-order risc-v core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[30] RISC-V International. Working draft of the proposed risc-v vector extension, 2022. URL https://github.com/riscv/riscv-v-spec.

[31] Pong P Chu. *Embedded SOPC design with NIOS II processor and VHDL examples*. John Wiley & Sons, 2011.

[32] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8. IEEE, 2017.

[33] Peter Yiannacouras, Jonathan Rose, and J Gregory Steffan. The microarchitecture of fpga-based soft processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 202–212, 2005.

[34] Rakesh Kumar, Norman P Jouppi, and Dean M Tullsen. Conjoined-core chip multiprocessing. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 195–206. IEEE, 2004.

[35] David Sheldon, Rakesh Kumar, Frank Vahid, Dean Tullsen, and Roman Lysecky. Conjoining soft-core fpga processors. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 694–701, 2006.

[36] Luca P Carloni. From latency-insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151, 2015.

[37] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 657–662, 2006.

[38] Digilent Inc. Nexys a7 board, part no. xc7a100t-1csg324c. URL https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/start.

[39] The fastest verilog simulator. URL https://www.veripool.org/verilator/.

[40] RISC-V BOOM. URL https://boom-core.org/.

[41] Rocket Chip Generator. URL https://github.com/chipsalliance/rocket-chip.

[42] Pulp platform. URL https://pulp-platform.org/.

[43] RiscyOO: RISC-V Out of Order Processors. URL https://github.com/csail-csg/riscy-OOO.

[44] Riscy processors - open-sourced risc-v processors. URL https://github.com/csail-csg/riscy.

[45] The lizard core. URL https://github.com/cornell-brg/lizard.

[46] SpinalHDL. Vexriscv: A fpga friendly 32 bit risc-v cpu implementation. URL https://github.com/SpinalHDL/VexRiscv.

[47] Shakti, open source processor development ecosystem. URL https://shakti.org.in/.

[48] C. Wolf. Picorv32 - a size-optimized risc-v cpu. URL https://github.com/cliffordwolf/picorv32/.

[49] Veer eh1 risc-v core. URL https://github.com/chipsalliance/Cores-SweRV/.

[50] Rsd risc-v out-of-order superscalar processor. URL https://github.com/rsd-devel/rsd.

[51] Rvsoc project, a portable and linux capable risc-v computer system on an fpga. URL https://www.arch.cs.titech.ac.jp/wk/rvsoc/doku.php?id=start.

[52] A winning processor portfolio. URL https://www.sifive.com/risc-v-core-ip.

[53] Codasip risc-v processors. URL https://codasip.com/products/codasip-risc-v-processors/.

[54] Risc-v@andes. URL http://www.andestech.com/en/risc-v-andes/.

[55] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, et al. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64. IEEE, 2020.

[56] The scr family of customizable processor ip. URL https://syntacore.com/page/products/processor-ip.

[57] Eric Matthews and Lesley Shannon. Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[58] Abbas Rahimi, Andrea Marongiu, Rajesh K Gupta, and Luca Benini. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[59] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Transactions on Computers*, 70(11):1845–1860, 2020.

[60] ARM Limited. Introducing neon development article, 2009. URL https://developer.arm.com/documentation/dht0002/a/?lang=en.

[61] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual, 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[62] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE micro*, 37(2):26–39, 2017.

[63] Charles Eric LaForest and J Gregory Steffan. Efficient multi-ported memories for fpgas. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 41–50, 2010.

[64] Stefania Perri, Pasquale Corsonello, Maria Antonia Iachino, Marco Lanuzza, and Giuseppe Cocorullo. Variable precision arithmetic circuits for fpga-based multimedia processors. *IEEE Transactions on very large scale integration (VLSI) systems*, 12 (9):995–999, 2004.

[65] Md Ashraful Islam and Kenji Kise. An efficient resource shared risc-v multicore architecture. *IEICE TRANSACTIONS on Information and Systems*, 105(9):1506–1515, 2022.

[66] Muhammad Ali, Matthias von Ameln, and Diana Goehringer. Vector processing unit: a risc-v based simd co-processor for embedded processing. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 30–34. IEEE, 2021.

[67] Muhammad Ali and Diana Göhringer. Application specific instruction-set processors for machine learning applications. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–4. IEEE, 2022.

[68] Peter Yiannacouras, J Gregory Steffan, and Jonathan Rose. Vespa: portable, scalable, and flexible fpga-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70, 2008.

[69] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):1–34, 2009.

[70] Aaron Severance and Guy Lemieux. Venice: A compact vector processor for fpga applications. In *2012 International Conference on Field-Programmable Technology*, pages 261–268. IEEE, 2012.

[71] Aaron Severance and Guy GF Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[72] Matheus Cavalcante, Domenic Wüthrich, Matteo Perotti, Samuel Riedel, and Luca Benini. Spatz: A compact vector processing unit for high-performance and energy-efficient shared-l1 clusters. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[73] Md Ashraful Islam and Kenji Kise. Resource-efficient risc-v vector extension architecture for fpga-based accelerators. In *Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART '23, page 78–85, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700439. doi: 10.1145/3597031.3597047. URL https://doi.org/10.1145/3597031.3597047.

[74] Ahmed Kamaleldin and Diana Göhringer. Agiler: An adaptive heterogeneous tile-based many-core architecture for risc-v processors. *IEEE Access*, 10:43895–43913, 2022.

[75] Risc-v gnu compiler toolchain. URL https://github.com/riscv-collab/riscv-gnu-toolchain.git.

[76] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Masanori Hariyama, and Michitaka Kameyama. Fpga implementation of heterogeneous multicore platform with simd/mimd custom accelerators. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1339–1342. IEEE, 2012.

[77] Johannes Ax, Gregor Sievers, Julian Daberkow, Martin Flasskamp, Marten Vohrmann, Thorsten Jungeblut, Wayne Kelly, Mario Porrmann, and Ulrich Rückert. Coreva-mpsoc: A many-core architecture with tightly coupled shared and local data memories. *IEEE Transactions on Parallel and Distributed Systems*, 29(5): 1030–1043, 2017.

[78] Marcelo Ruaro, Luciano L Caimi, Vinicius Fochi, and Fernando G Moraes. Memphis: a framework for heterogeneous many-core socs generation and validation. *Design Automation for Embedded Systems*, 23:103–122, 2019.

[79] Abdoulaye Gamatié, Guillaume Devic, Gilles Sassatelli, Stefano Bernabovi, Philippe Naudin, and Michael Chapman. Towards energy-efficient heterogeneous multicore architectures for edge computing. *IEEE Access*, 7:49474–49491, 2019.

[80] Hyeonguk Jang, Kyuseung Han, Sukho Lee, Jae-Jin Lee, Seung-Yeong Lee, Jae-Hyoung Lee, and Woojoo Lee. Developing a multicore platform utilizing open risc-v cores. *IEEE Access*, 9:120010–120023, 2021.

[81] Andreas Kurth, Björn Forsberg, and Luca Benini. Herov2: full-stack open-source research platform for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4368–4382, 2022.

[82] Tulika Mitra. Heterogeneous multi-core architectures. *Information and Media Technologies*, 10(3):383–394, 2015.

Publications

- "An Efficient Implementation of a TAGE Branch Predictor for Soft Processors on FPGA," Katsunoshin Matsui, **Islam Md Ashraful**, and Kenji Kise; IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2019), pp.108-115
- "RVCoreP: An optimized RISC-V soft processor of five-stage pipelining," Hiromu Miyazaki, Takuto Kanamori, **Islam Md Ashraful**, and Kenji Kise; IEICE Transactions on Information and Systems, Vol.E103-D, No.12, pp.2494-2503 (December 2020).
- "Efficient resource shared RISC-V multicore processor," **Islam, Md Ashraful**, Kenji Kise; International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021) **Best paper runner up award**
- "An Efficient Resource Shared RISC-V Multicore Architecture," **Islam, Md Ashraful**, and Kenji Kise. IEICE TRANSACTIONS on Information and Systems 105.9 (2022): 1506-1515. doi: 10.1587/transinf.2021EDP7248
- "Resource-efficient RISC-V Vector Extension Architecture for FPGA-based Accelerators," **Islam, Md Ashraful**, and Kenji Kise. 2023 The International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART 2023), June 14–16, 2023, Kusatsu, Japan. ACM