IMPLEMENTING VIRTUAL MEMORY SUPPORT AND BOOTING LINUX ON ARA: AN OPEN SOURCE VECTOR PROCESSOR

by

Gidha Iftikhar
2021-MS-EE-29

Research Supervisor:
Dr. Muhammad Tahir

2024

Department of Electrical Engineering,
University of Engineering and Technology, Lahore

IMPLEMENTING VIRTUAL MEMORY SUPPORT AND BOOTING LINUX ON ARA: AN OPEN SOURCE VECTOR PROCESSOR

by

Gidha Iftikhar

A THESIS

Presented to the University of Engineering and Technology, Lahore

In partial fulfillment of the requirements for the degree of

Master in Science

in

Electrical Engineering

APPROVED BY:

Dr. Muhammad Tahir

Dr. Muhammad Tahir

Dr. Muhammad Shoaib

Department of Electrical Engineering,
University of Engineering and Technology, Lahore

ABSTRACT

Virtual memory is essential for modern processor design, providing benefits like memory protection, efficient multitasking, and abstraction of physical memory management. It allows systems to handle larger applications and improve security by isolating processes. Virtual memory is crucial for booting modern operating systems like Linux. Linux relies on virtual memory to manage its multitasking environment, ensuring each process has its own memory space and preventing one process from interfering with another. Despite its importance, many Open source vector processors such as ARA from the PULP platform lack virtual memory support, limiting their applicability in complex, memory-intensive applications. This paper discusses the current limitations of ARA, which operates in bare-metal mode, and the research efforts aimed at making it compatible with Linux booting by adding virtual memory support.

By bridging this gap, the enhanced ARA vector processor will leverage the sophisticated memory management capabilities of the Ariane core, resulting in a more powerful and versatile processing unit. This integration is expected to expand the applicability of vector processors in high-performance and secure computing environments, opening new avenues for research and application.

An innovative and highly efficient design approach has been used to add virtual memory support in ARA by sharing Ariane's MMU. This results in running vector applications based on vector instructions, such as LLAMA, on the core and achieving high efficiency of the operations which takes more resources, power and time using only scalar instructions based applications.

Furthermore, the design has been completely verified by running RISC-V test suites and developing complex and efficient custom virtual memory tests and real time applications.

ACKNOWLEDGMENTS

I am deeply grateful to my supervisor, Dr. Muhammad Tahir, for his invaluable guidance, continuous support, and patience throughout my research. His expertise and encouragement have been instrumental in the completion of this work. I would also like to extend my sincere thanks to my committee members for their insightful feedback and suggestions, which greatly improved the quality of this thesis.

A special appreciation goes to my mentors from 10xEngineers, Fatima Saleem, Bilal Zafar, and Saad Waheed, for their mentorship, support, and encouragement. Their guidance has been crucial in navigating the challenges of this research.

I am profoundly thankful to my husband for his continuous support, patience, and understanding during the course of this work. His unwavering encouragement has been a source of strength for me.

And finally, I would like to thank GOD for giving me the strength, knowledge, and opportunity to undertake this research and complete this thesis. Without His blessings, this achievement would not have been possible.

STATEMENT OF ORIGINALITY

It is stated that the research work presented in this dissertation consists of my own ideas and research work. The contributions and ideas from others have been duly acknowledged and cited in the dissertation. This complete dissertation is written by me. If at any time in the future, it is found that the thesis work is not my original work, the University has the right to cancel my degree.

Gidha Iftikhar

TABLE OF CONTENT

LIST OF FIGURES

NOMENCLATURE

RISC-V	Reduced Instruction Set Computing
ISA	Instruction Set Architecture
VLEN	Vector Length
SEW	Selected Element Width
LMUL	Length Multiplier
FPU	Floating Point Unit
LSU	Load Store Unit
VLSU	Vector Load Store Unit
MMU	Memory Management Unit
PTW	Page Table Walk
TLB	Translation Lookaside Buffer
VRF	Vector Register File
AXI	Advanced eXtensible Interface
AGU	Address Generation Unit
SOC	System On Chip
VA	Virtual Address
PA	Physical Address
FPGA	Field Programmable Gate Arrays

1. INTRODUCTION

The RISC-V Instruction Set Architecture (ISA) has revolutionized the field of computer architecture by offering a flexible, open-source alternative to proprietary ISAs[1]. Its open nature allows for widespread adoption and customization, facilitating advancements in both academic research and commercial applications. The modularity and extensibility of RISC-V have made it a preferred choice for developing highly specialized processors, enabling innovations in fields ranging from embedded systems to high-performance computing[2]. The RISC-V ecosystem has grown to include comprehensive software toolchains, simulation environments, and extensive documentation, further promoting its use and exploration.It is expected that by 2025, 62.5 billion RISC-V processors will be used in various applications.

1.1. VECTOR PROCESSORS

The open RISC-V ISA specification is also leading an effort towards vector processing through its vector extension[1][4]. The vector extension is designed to handle large amounts of data in parallel, using a single instruction to perform operations on multiple data elements. Scalar processors can act on one piece of data at a time while vector processors can act on several pieces of data (multiple elements) at a time with a single instruction[3]. These multiple pieces of data are basically one-dimensional arrays of data called vectors and hence the need for a Vector ISA.

Vector ISA includes 32 vector registers and the number of bits in a single vector register (VLEN) is implementation dependent and a power of 2 (32, 64, 128, 256...)[3]. A vector instruction operates on all elements of a single vector register. The element size is selected through SEW(Selected Element Width) which can be 8(byte), 16(half word), 32(word) and 64(double word). So, the number of elements in a vector register is determined by VLEN/SEW. For example, if VLEN is 512 and SEW is 8 then in one vector register there will be 64 elements[3].

Furthermore, Multiple Vector Registers can be grouped (concatenated) together as shown in figure 1.1, so that a single vector instruction can operate on multiple vector registers which can be set through the LMUL which strips the elements across the vector registers[3]. LMUL has to be a power of 2 The maximum value of LMUL is 8 and can have fraction values with the minimum value of ⅛.

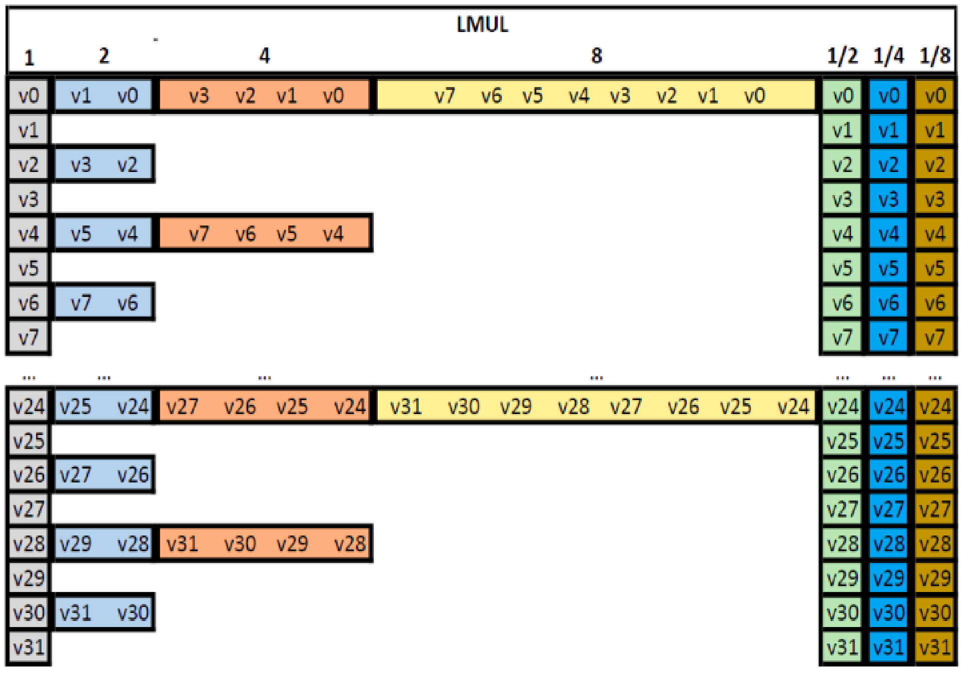

Figure 1.1 Vector register grouping according to LMUL[3]

The load/store operations of a vector processor moves a group of data between a vector register file and memory using a single instruction. The misaligned support is optional and implementation dependent. The RV specifications have four basic types of addressing for the load/store operations: Unit strided, Strided, indexed and segmented.

1.1.1. Unit strided operations

This operation accesses contiguous memory addresses as shown in figure 1.2.

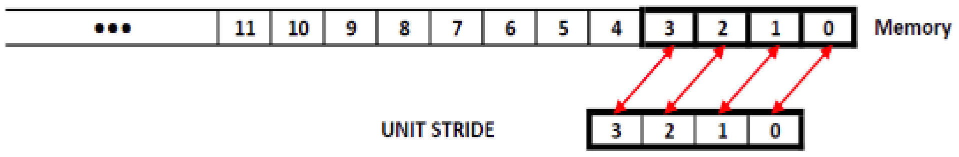

Figure 1.2 Unit Strided Operation[3]

1.1.2. Strided operations

This operation gathers/scatters data on a fixed stride. The stride can be negative and in case of zero stride, the same location of memory is read or written as shown in figure 1.3.

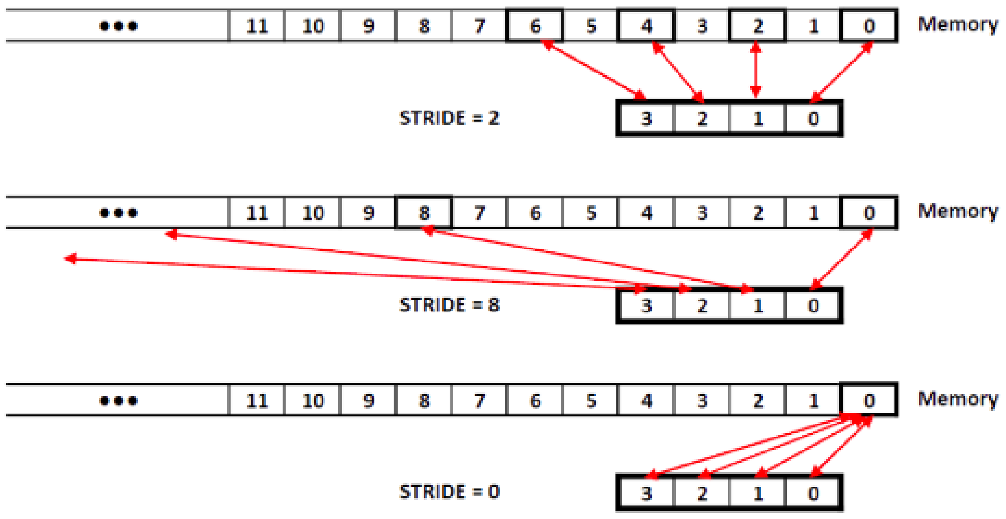

Figure 1.3 Strided Operations[3]

1.1.3. Indexed operations

In case of indexed operations, the data is scattered/gathered on a variable stride. The stride value for each element is provided in a vector register. On send a request for data operation, indexed load/store reads VRF for stride value and add it to the base value to generate a memory address as shown in figure 1.4.

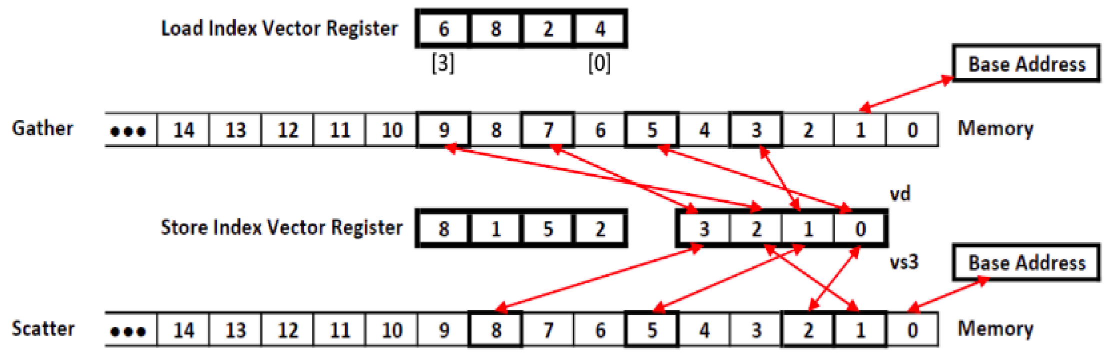

Figure 1.4 Indexed Operations[3]

Segmented operations are one of the most complex operations of RV ISA. It is further divided into unit stride segmented, stride segmented and indexed segmented operation. The research of vector core selected for this thesis doesn't support segmented operations so we have excluded the details of these operations.

Recognizing the efficiency of vector processors and the simplicity, compatibility, and ease of the RISC-V Vector ISA, numerous research groups and companies have begun developing RISC-V compliant vector cores. Many of these organizations have also open-sourced their RISC-V processor implementations, contributing to the growing ecosystem. One of the most cited among them is the ARA Vector Unit working as a co-processor for the Ariane core from OpenHW Foundation[5][6].

1.2. ARIANE

CVA6, also known as Ariane, is a 64-bit application-class RISC-V core developed by the OpenHW Group[7]. Ariane is a wrapper of CVA6 which instantiates CVA6 and leaves space to integrate extensions and co-processors. The CVA6 features a six-stage pipeline and is compatible to boot Linux[8]. It is designed for high-performance computing tasks. It has support for multiply/divide and atomic operations as well as IEEE-compliant FPU[11]. It is manufactured in GLOBALFOUNDR IES 22FDX FD-SOI by Zaruba and Benini[8]. The Six stages pipelined architecture consists of Program Counter Generation, Instruction Fetch, Instruction Decode, Issue stage, Execution stage and commit stage as shown in figure 1.5. The first two stages are Ariane's frontend responsible for fetching new instructions and housing the branch prediction unit while the rest of the stages are its backend and have scalability of integrating co-processors[7].

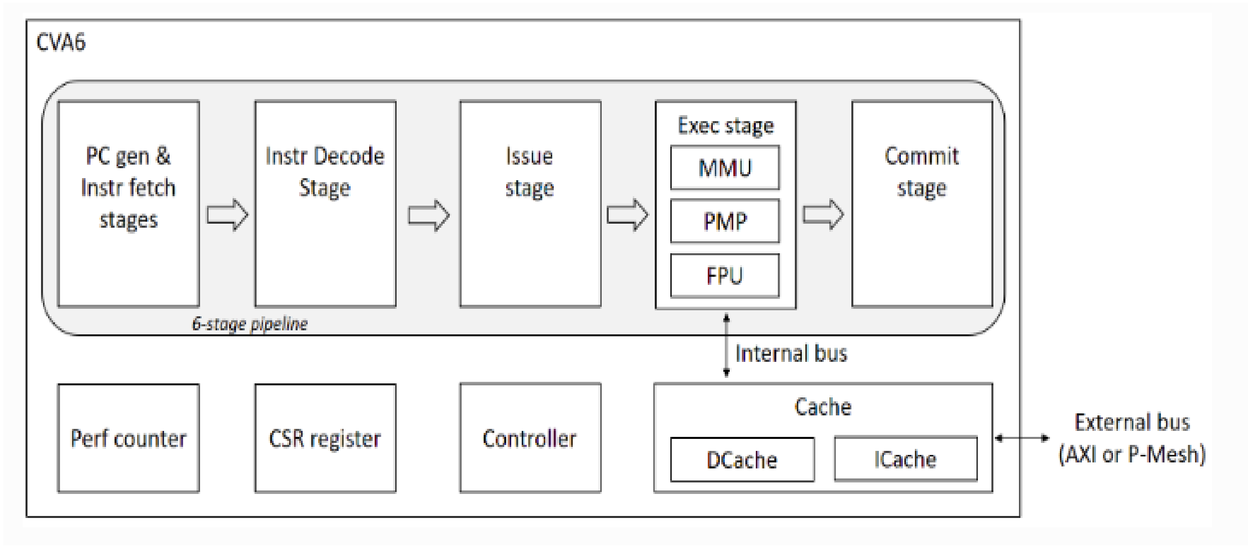

Figure 1.5 CVA Core[7]

The load store unit(LSU) of CVA6 is responsible for reading and writing the memory. The internal structure of LSU is shown in figure 1.6. It is located in the execution stage and can configure MMU based on sv32 and sv39 architecture. Additionally, it has to manage the interface to the data memory (D$). In particular, it houses the DTLB (Data Translation Lookaside Buffer), the hardware page table walker (PTW), and the memory management unit (MMU). It also arbitrates the access to data memory between loads, stores, and the PTW - giving precedence to PTW lookups[7]. This is done in order to resolve TLB misses as soon as possible. A high-level block diagram of the LSU is shown below

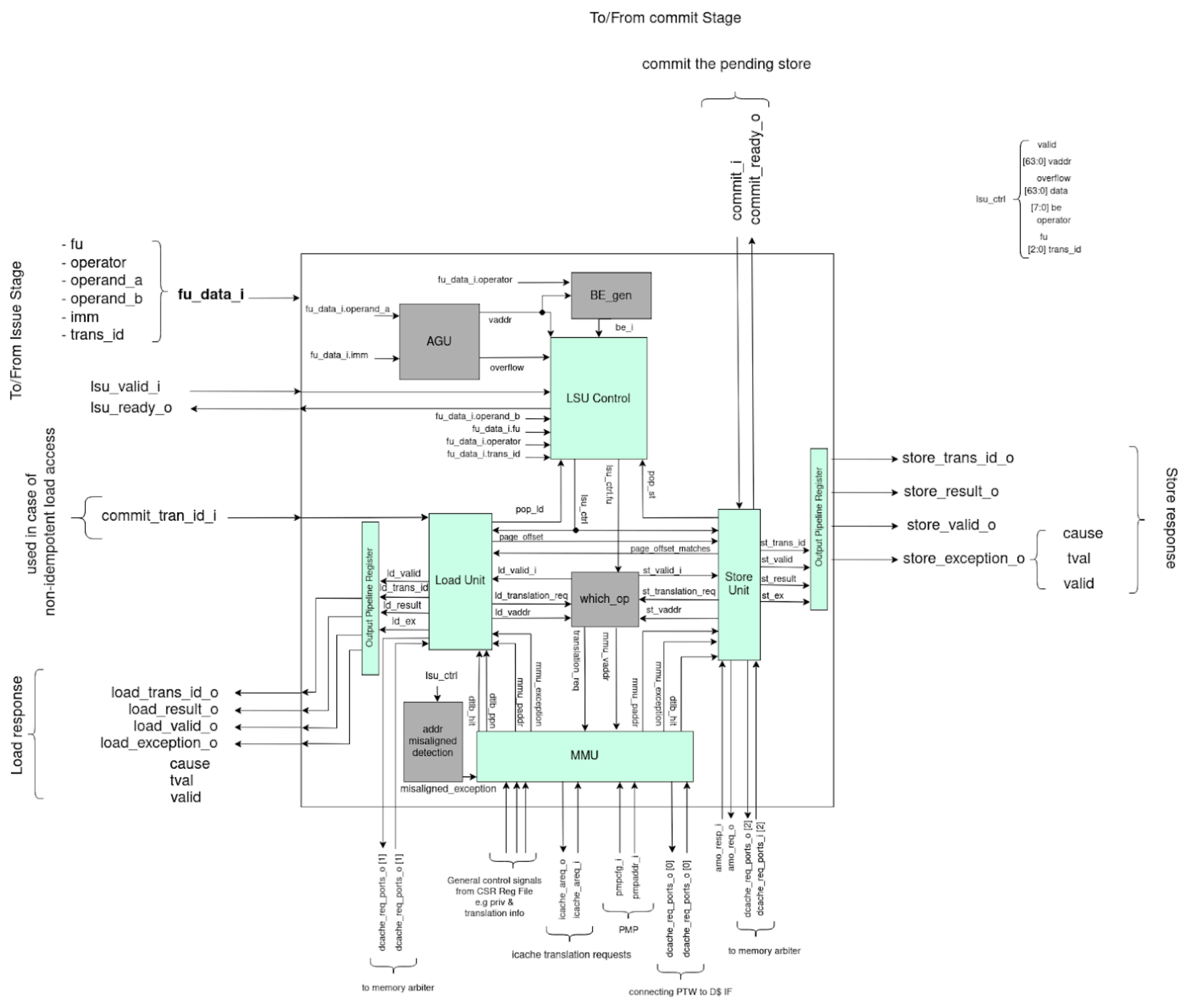

Figure 1.6 CVA6 Load Store Unit

The LSU has a separate load unit and store unit to manage the data bus. Once an instruction is entered to LSU, it is sent to the load unit or store unit based on their opcodes. The corresponding unit sends a translation request to MMU along the virtual address. If translation is enabled and pmp is properly configured, MMU sends the physical address back. The load unit or store unit sends the request to memory for relevant operation.

1.3. ARA

ARA is a high-performance 64-bit vector unit designed by PULP Platform for data-parallel applications, compliant with the RISC-V Vector 1.0 Extension[5][3]. It supports mixed-precision arithmetic and integrates within the PULP platform to work as an extension of the Ariane core. This processor aims to deliver extreme energy efficiency and scalability for computational tasks that benefit from vectorized operations.

ARA consists of first pass decoder, Dispatcher, Sequencer VLSU, Slide unit and Lanes. ARA can have up to 4 lanes configuration with 1024 VLEN. ARA is integrated at the backend of Ariane using a first pass decoder as shown in figure 1.7. The vector instructions are partially decoded and passed to ARA first pass decoder which dispatches it to ARA.

The dispatcher of ARA is responsible for the interface between Ara and Ariane's dedicated scoreboard port. The sequencer keeps track of the instructions running in Ara, dispatching them to different execution units and acknowledging them to Ariane. The slide unit is responsible for handling instructions that need access to the vector register file. The VLSU performs memory operations. It uses AXI protocol and generates AXI requests upon receiving requests of memory load and store operation from sequencer.

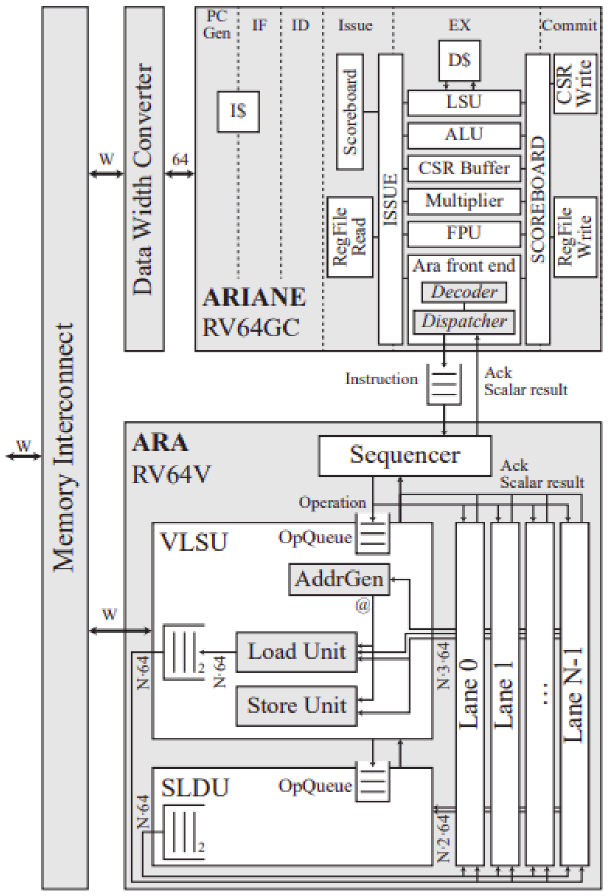

Figure 1.7 ARA Internal Structure[5]

Ara can be configured with N number of identical lanes. Each lane has its own lane sequencer which can keep track of up to eight vector instructions running in parallel. Each lane is almost independent and contains ALU, MUL, FPU and part of vector register file[5].

The VLSU of ARA also has a separate load and store unit. On receiving a request from sequence, the address generator unit generates the address and sends the address on AXI read address channel for load operations and AXI write address channel in case of store operation. It signals the load and store units about the request so that they can receive data or write data on data bus respectively as shown in the figure 1.8.

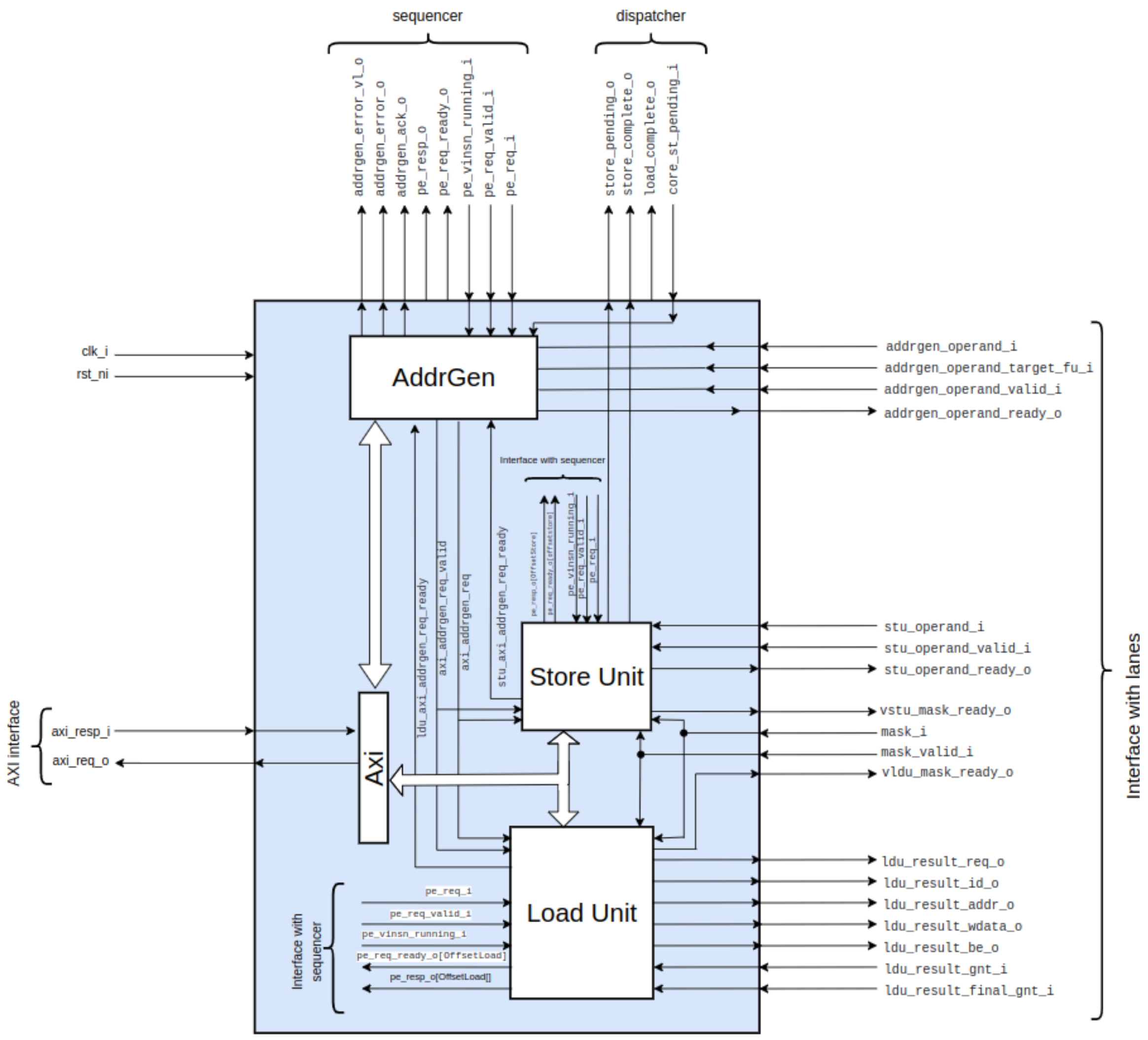

Figure 1.8 ARA's Vector Load Store Unit(VLSU)

Currently, ARA operates in a bare-metal mode without virtual memory support, limiting its ability to run complex operating systems like Linux[5][9]. My research focuses on making ARA compatible with Linux booting by integrating virtual memory support[10]. This endeavor aims to bridge the gap, although resource constraints have so far prevented a successful Linux boot on the vector processor. Overcoming these challenges will significantly enhance the capabilities of ARA and open new avenues for its application in high-performance and secure computing environments[12].

The main focus of this research revolves around the address generator unit in VLSU of ARA as it is generating addresses to send the requests to AXI Bus for load/store operation[5]. Currently, Ara supports unit strided, strided and indexed operations. For unit strided operations, burst is generated aligned with 4k pages. When the burst crosses the page boundary, new requests with the remaining elements are generated. For strided and indexed operations, a request is generated for each calculated address.

2. METHODOLOGY

To implement virtual memory support in ARA and make it compatible with booting Linux, a systematic approach is essential. The process begins with a thorough analysis of ARA's and Ariane's microarchitecture and configurations to determine the most suitable method for our implementation. Understanding the existing structure and performance characteristics is crucial to ensure that the new virtual memory support will integrate seamlessly and operate efficiently. Below are the steps for the implementation process:

2.1. STABILIZING THE ARA

Ara is currently in its development phase, which means it still has many bugs and errors that are reported from time to time. Among the reported issues, two critical bugs related to exception handling were identified as particularly relevant to ongoing research and implementation efforts.

Exception handling is crucial, especially when booting Linux on a core, as it ensures the system can gracefully handle unexpected situations and maintain stability. Recognizing the importance of this, a thorough investigation was conducted to address the identified bugs. The bugs were successfully fixed, establishing a stable path for exception handling within Ara, and ensuring smoother and more reliable operations during core booting processes.

After implementing the fixes, a Merge Request (MR) was created to integrate these changes into the main codebase. The MR was reviewed, and the changes have since been mainstreamed into Ara's GitHub repository. This contribution not only aids ongoing research but also enhances the overall stability of Ara, benefiting other developers and researchers working on the project.

2.2. SELECTED CONFIGURATIONS OF ARA AND ARIANE

Considering the available resources and existing limitations, the configuration chosen for ARA is lane 2 with a VLEN of 512. On the Ariane side, the Floating Point

Unit (FPU) and Bit Manipulation extension have been disabled to streamline the compilation process and manage resource constraints effectively.

2.3. CHOOSING THE SUITABLE ENVIRONMENT

Ara has its own System On Chip(SOC) environment and its own testbench[5]. In ara soc, it instantiates ara_syatem which have Ara and Ariane integrated in parallels shown in figure 2.1. This setup can be used to implement the proposed design but the issue is that ara_soc doesn't include ariane's capability of booting linux.

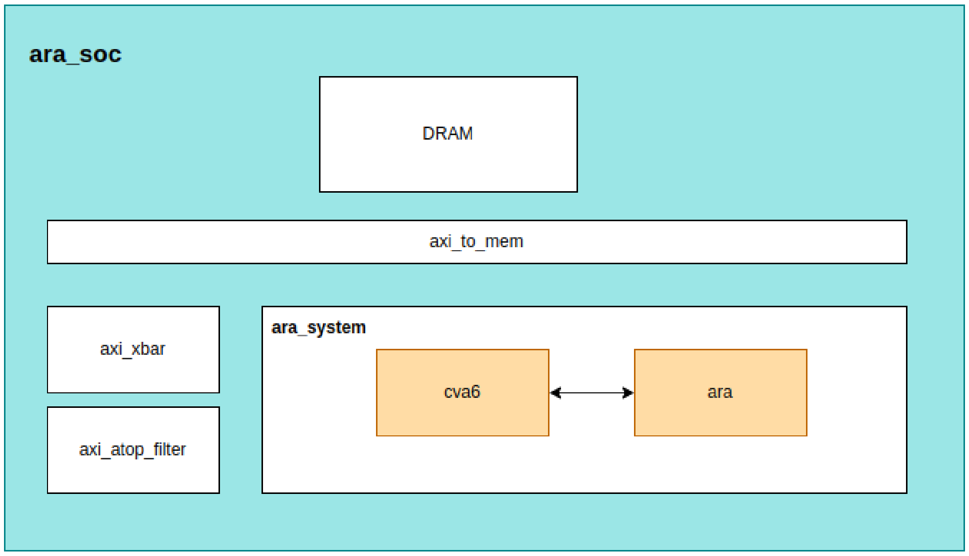

Figure 2.1 ARA System On Chip Design

On the other hand, Ariane's environment is much more stable and already capable of booting linux[2]. It provides a robust environment ideal for designing and testing due to its comprehensive integration of various master and slave modules connected via an AXI crossbar. This setup includes components such as DRAM, GPIO, Ethernet, SPI, Timer, UART, PLIC, CLINT, BootROM, and SRAM, facilitating extensive testing and debugging as shown in figure 2.2 . With clear signal assignments and modular structure, this environment supports seamless communication and interaction among components, making it highly suitable for advancing research and implementation in system-on-chip designs.

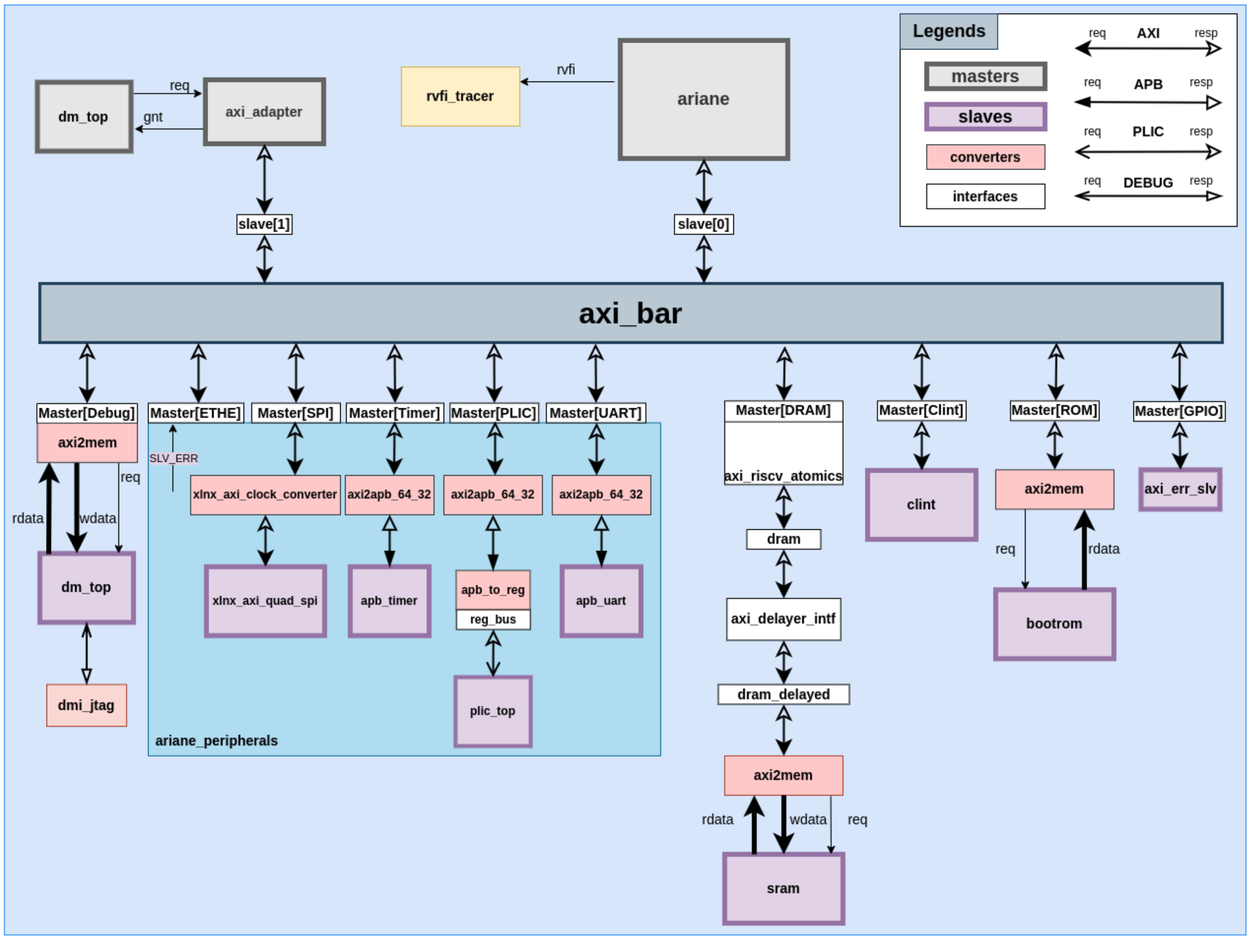

Figure 2.2 Ariane's System On Chip Design[7]

2.3.1. Integration of Ara and Ariane in Ariane's SOC

To integrate Ara and CVA6 in ariane wrapper, an invalidation filter and AXI Multiplexer is added. This wrapper is designed specifically for this research work. The axi request from ara passes through the inval_filter to generate an invalidation request for CVA6 caches. The AXI Mux multiplexed the axi requests between CVA6 and Ara as shown in the figure 2.3. Cva6 and Ara communicated with the cvxif port of cva6, specifically designed to integrate the vector processor.

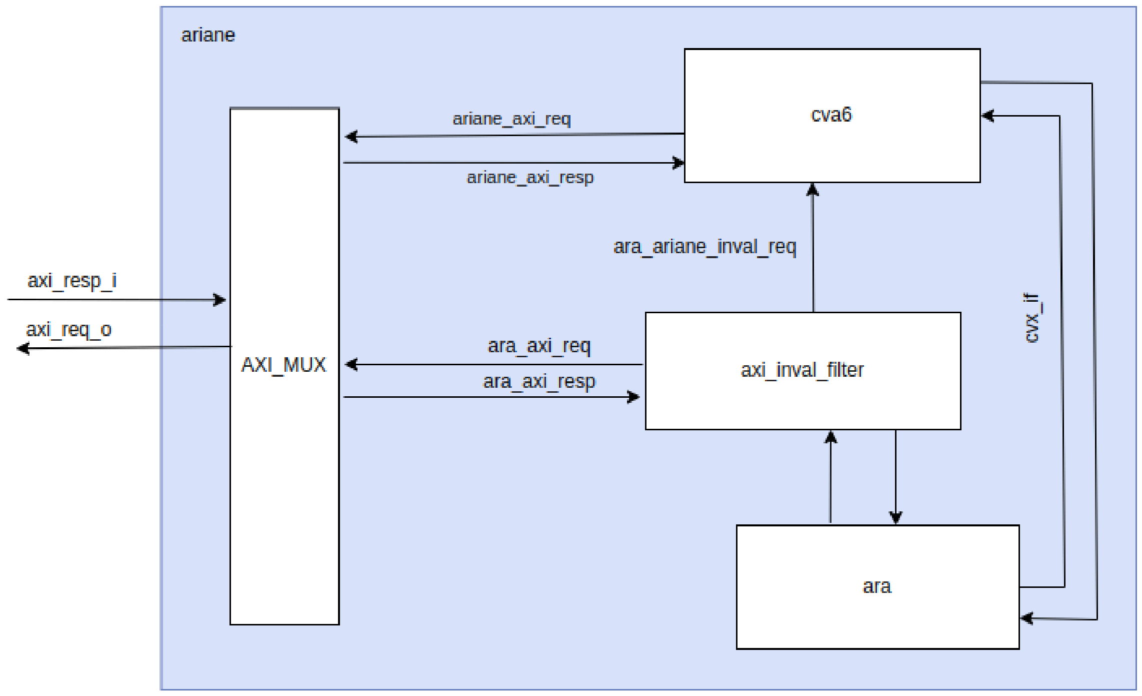

Figure 2.3 Integrating Ara in Ariane's test bench

2.4. DESIGN APPROACH

The design approach includes sharing the Ariane's Memory Management Unit (MMU) with ARA. ARA is integrated with Ariane using Ariane's cvxif port. To create a communication path from ARA's Vector Load/Store Unit (VLSU) to the MMU, additional signals are incorporated into the same port. The main design modifications focus on the Address Generator unit of ARA's VLSU. This module, responsible for sending AXI requests to the memory, is altered to incorporate Ariane's MMU path for address translation. By leveraging the shared MMU, the Address Generator unit can route AXI requests through this newly established path, ensuring efficient address translation and memory access. This approach optimizes resource utilization by reducing redundancy, simplifies address translation, and enhances system integration, leading to improved performance and efficiency. A clear path from Ara's address generator unit to cva6's MMU can be shown in figure 2.4.

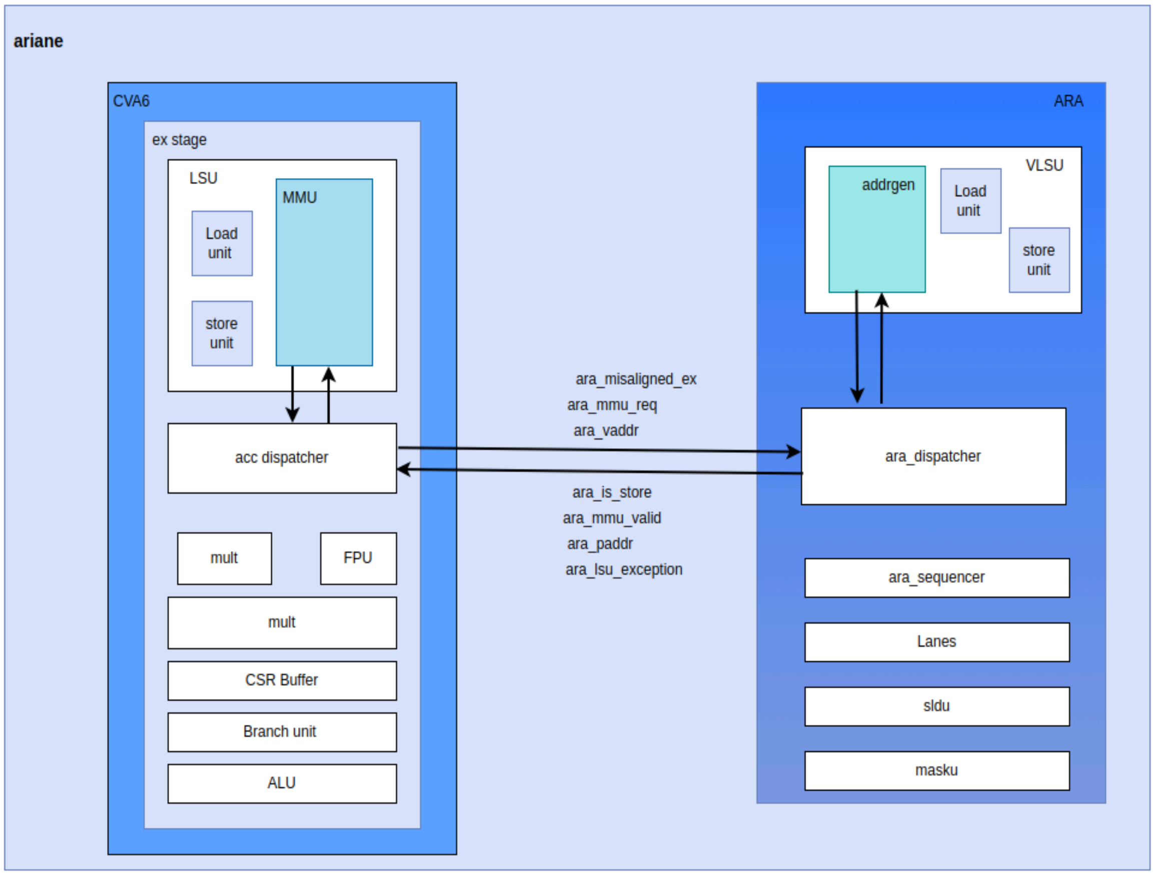

Figure 2.4 Added path from ARA's VLSU to Ariane's MMU

The virtual address is sent to the cva6's MMU and a physical address is received. The load store unit of both the cores are responsible for handling AXI data transactions.

2.4.1. CVA6 Modifications

An arbiter is integrated into the Load/Store Unit (LSU) of the CVA6 to manage the requests from both the scalar core and the vector core. When a request is directed to the LSU by the issue stage, the LSU generates a virtual address using the operands received from the decode stage and the register file. This virtual address is then sent to the Memory Management Unit (MMU) for translation. If virtual memory is enabled, the MMU translates the virtual address into a physical address.

In order to share the MMU with Ara, a round-robin arbiter is implemented within this process, with an additional port introduced in the LSU specifically for handling ARA's translation requests as shown in figure 2.5. Requests from the vector and scalar units are fed into the arbiter through a 2-bit input, with scalar requests assigned to index zero and vector requests to index one. The arbiter grants one request at a time, registering the index of the granted request and forwarding it to the MMU.

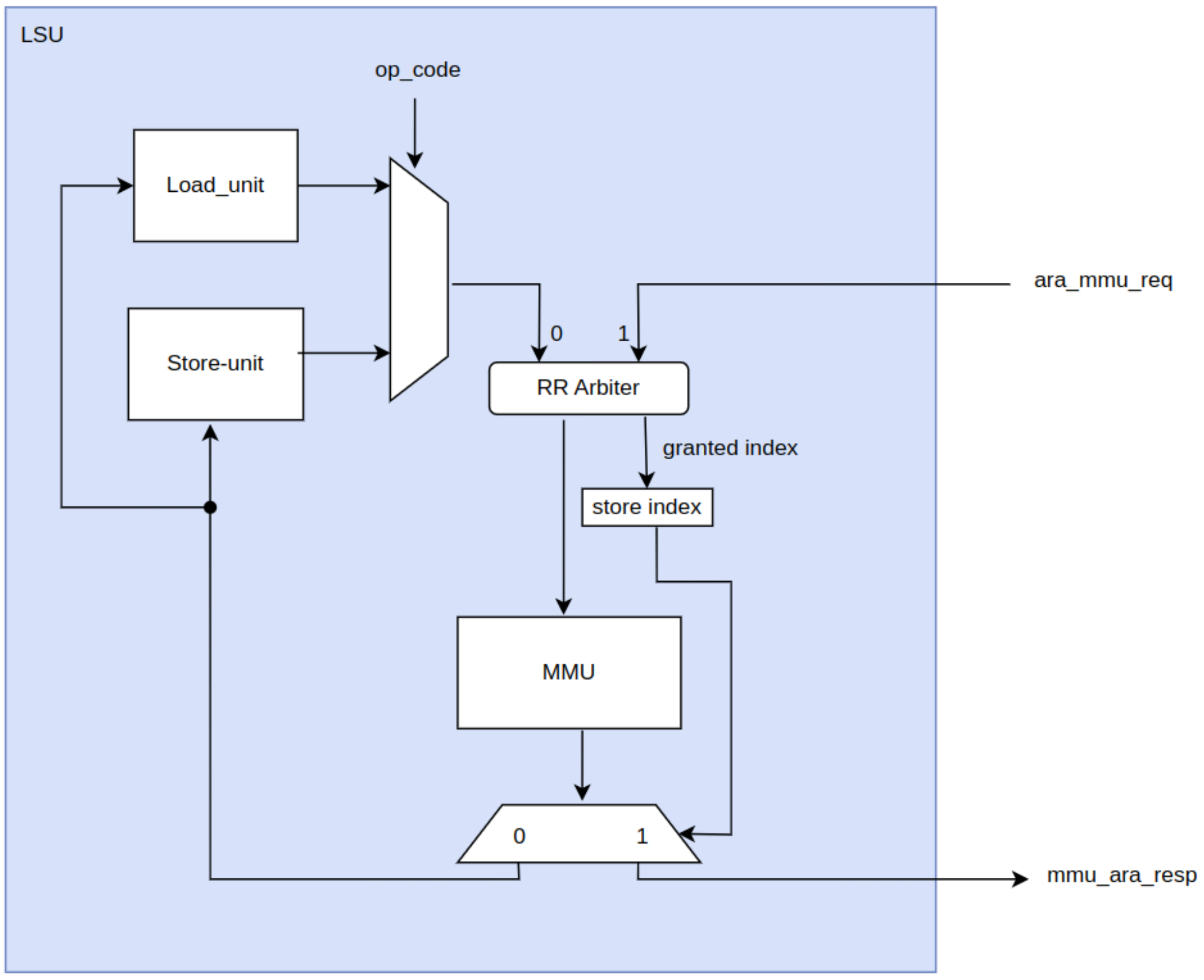

Figure 2.5 Design modification in CVA6’s LSU

Once the MMU completes the address translation, the stored index is used to route the response back to the appropriate unit. This is crucial because the MMU may take multiple cycles to translate the address, especially in the case of a Translation Lookaside Buffer (TLB) miss. By using this mechanism, the arbiter ensures that the translated address is correctly returned to either the scalar or vector unit, maintaining efficient and orderly processing of memory requests.

2.4.2. ARA’s Address generator Unit Design

In the VLSU's Address Generator Unit (AGU), a crucial redesign is necessary to implement address translation. The AGU is responsible for generating addresses for load and store operations, sending them over the AXI AR channel for loads and AW channel for stores[5]. To achieve this, the AGU must be modified to interface with the CVA6's Memory Management Unit (MMU).

As illustrated in the figure, the original AGU consists of two primary FSMs as shown in figure 2.6 :

1. **Address Generator FSM**: This FSM receives requests from the sequencer and processes them accordingly. For unit-strided and strided operations, it forwards the requests to the AXI Request Generation FSM, accompanied by the stride value. In the case of indexed operations, the Address Generator FSM reads the Vector Register File (VRF), calculates the address by adding the base address and index, and stores it in the spill registers.
2. **AXI Request Generation FSM:** This FSM is responsible for managing the AXI channel interfaces. For unit-strided operations, it sends burst requests over the AXI channel. For strided operations, it sends consecutive requests, incrementing the address by the stride value each cycle. In the case of indexed operations, it reads the spill registers and sends the corresponding requests over the AXI channel.

A critical design decision was made to integrate the Memory Management Unit (MMU) interface into either the Address Generator FSM or the AXI Request Generation

FSM. After careful consideration, it was determined that the latter would be the optimal choice, as it is responsible for forwarding addresses to the AXI channels. Consequently, the AXI Request Generation FSM was redesigned to accommodate an additional port with the MMU, ensuring efficient address translation and memory request management.

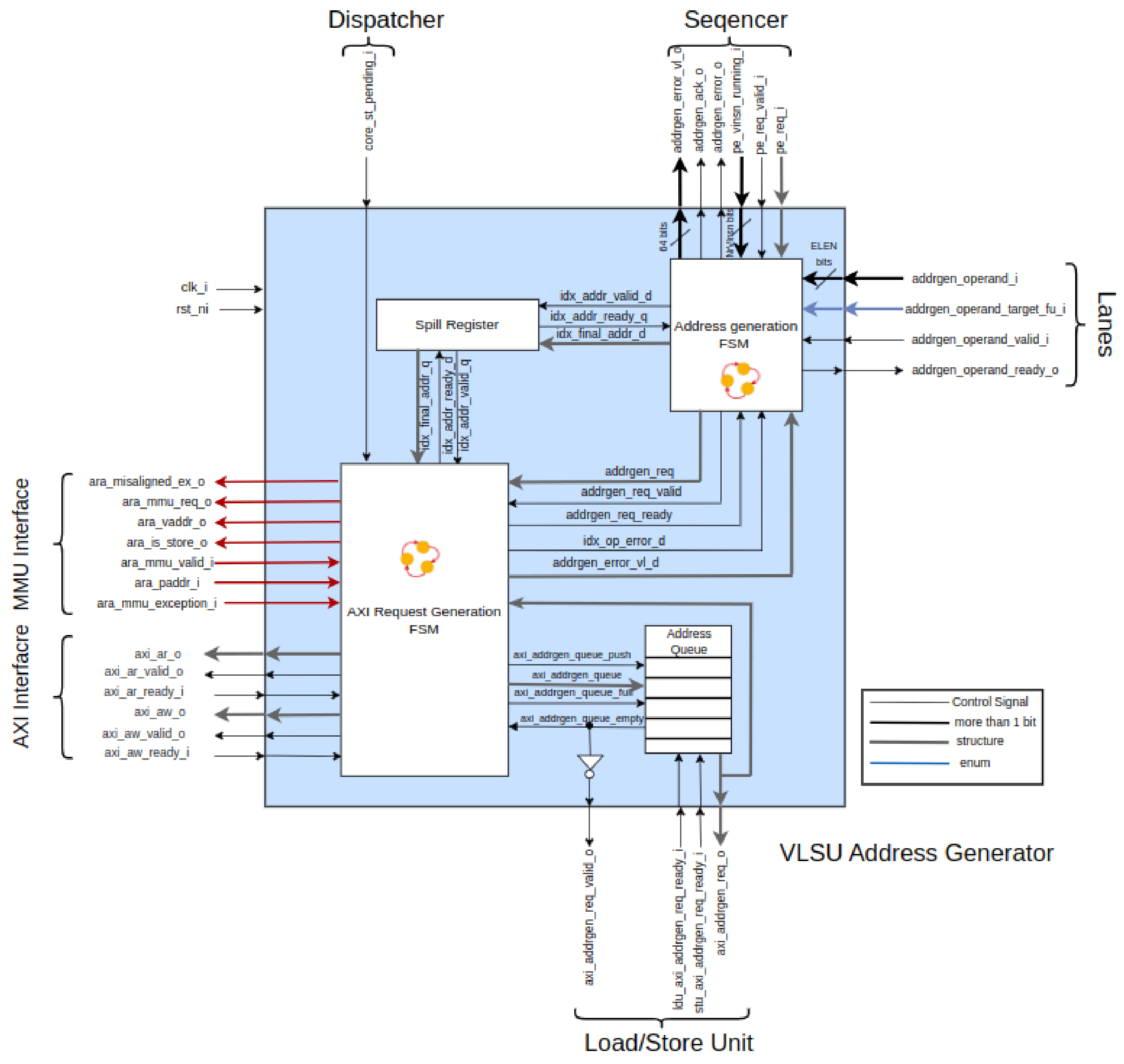

Figure 2.6 ARA’s Address Generator Unit(AGU)

The modified AXI Request Generation FSM is shown in the figure 2.7. The states of the FSM are

- AXI_ADDRGEN_IDLE
- AXI_ADDRGEN_WAIT_MMU
- AXI_ADDRGEN_SEND_REQ
- AXI_ADDRGEN_MISSALIGNED
- AXI_ADDRGEN_WAITING

The primary concept involves sending the base address for translation in both unit-strided and strided operations. The AXI request aligns with 4KiB pages. The boundaries of the current page and the next page are calculated and stored. So, If a transaction crosses a page boundary, a new request is sent to the MMU with the next 4KiB page base address for translation. Counters are used to add the offset based on stride in each cycle, monitoring the transaction length. When the transaction length reaches zero, the operation ends.

For indexed operations, a translation request must be sent to the MMU for each transaction because the index value cannot be determined without reading it from the VRF. To minimize the latency of waiting for the MMU, the next address is read from spill registers and sent to the MMU simultaneously, while the physical address is sent on the AXI channel in parallel. This process eliminates a cycle wait for each transaction under ideal conditions (TLB hit).

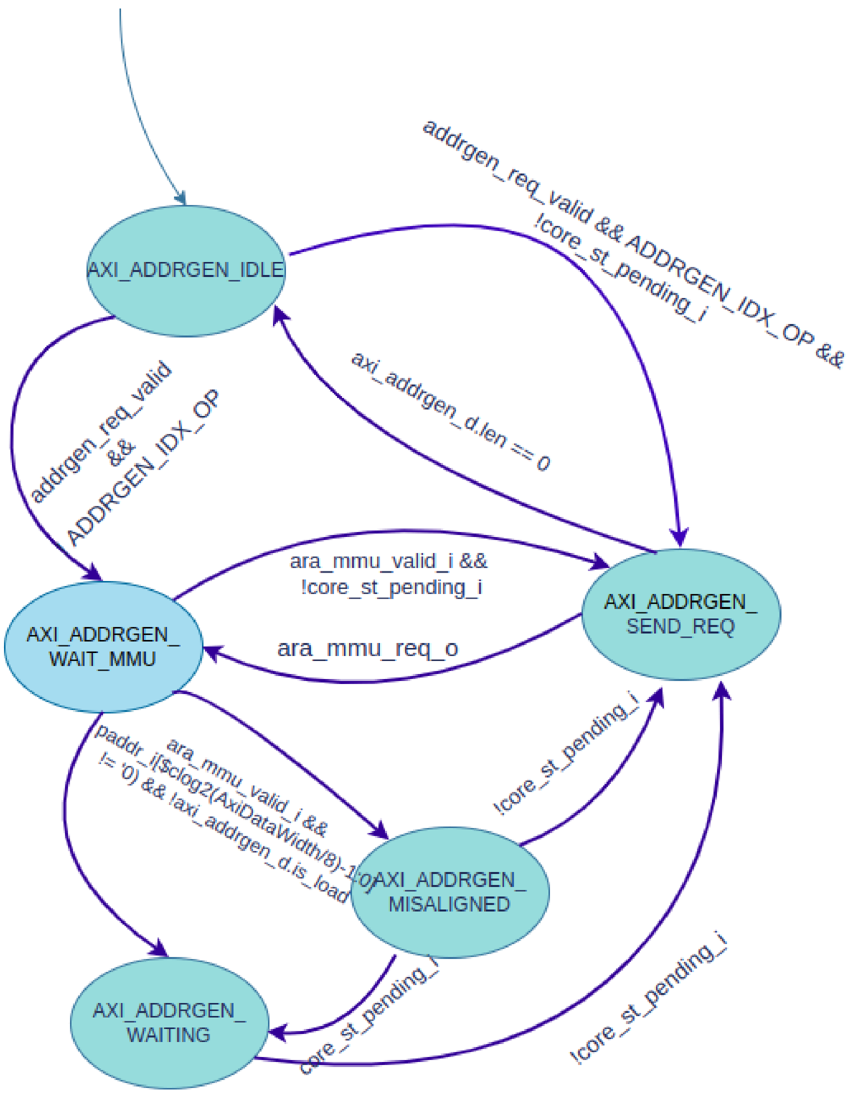
AXI_ADDRGEN_IDLE
addrgen_req_valid && ADDRGEN_IDX_OP && !core_st_pending_i
axi_addrgen_d.len == 0
addrgen_req_valid && ADDRGEN_IDX_OP
ara_mmu_valid_i && !core_st_pending_i
AXI_ADDRGEN_ SEND_REQ
AXI_ADDRGEN_ WAIT_MMU
ara_mmu_req_o
!core_st_pending_i
ara_mmu_valid_i && paddr_i[$clog2(AxiDataWidth/8)-1:0] != '0) && !axi_addrgen_d.is_load
AXI_ADDRGEN_ MISALIGNED
AXI_ADDRGEN_ WAITING
core_st_pending_i
!core_st_pending_i

Figure 2.7 AGU’s FSM

Unit strided and strided operation: The FSM starts in the IDLE state, awaiting a request from the Address Generation FSM. Upon receiving a valid request, it forwards a request to the MMU along with the virtual base address, transitioning to the WAIT_MMU state. It remains in this state until the MMU provides a physical address.

Once a physical address is obtained, if it meets alignment requirements, the FSM progresses to the SEND_REQ state. Here, it continuously generates addresses by incrementing the physical address with stride value or unit stride until it reaches a page boundary. Upon crossing the page boundary, it requests the next page base address from the MMU and re-enters the WAIT_MMU state to await a response.

Indexed Operations: For indexed operations, the FSM doesn't initiate a translation request while in the IDLE state since the address isn't yet available in the spill registers. Instead, it transitions directly to the WAIT_MMU state, where it requests translation from the MMU before moving to the SEND_REQ state. In this state, it retrieves the virtual address of the next transaction from the spill registers to request translation from the MMU, while simultaneously forwarding the current physical address to the AXI channel.

The implementation of this innovative technique has yielded significant efficiency gains. Under ideal conditions, it takes only two additional cycles for unit-strided and strided operations, and just one extra cycle for indexed operations. Importantly, this address translation enhancement has been achieved without introducing additional latency.

3. VERIFICATION EFFORTS

An additional challenge in this research was the verification of the implemented design, which posed a significant obstacle. Notably, no prior efforts had been made to develop tests for vector cores with virtual memory enabled, and no relevant vector tests were available in open-source repositories for use with the Ariane testbench and environment. To overcome this hurdle, custom tests were specifically designed and developed for this research, and run on the proposed design to validate its functionality.

Furthermore, a customized application was created using only scalar instructions and then vector instructions to compare the efficiency of a vector core capable of virtual memory and booting Linux. This endeavor significantly added to the effort and complexity of this thesis.

3.1. CUSTOMIZED TESTS

To ensure the design's accuracy, self-checking vector tests were developed in both bare metal and user modes, with virtual memory enabled. These tests were written within the Ariane environment and successfully executed using Ariane's testbench. The comprehensive test suite covers a wide range of scenarios, including unit strided, strided, and indexed operations for various element widths (SEW) of 8, 16, 32, and 64 bits, thereby exhaustively testing all possible cases.

3.2. CUSTOMIZED APPLICATION

A highly customized self-checking algorithm was developed to efficiently manage a series of load and store operations on extensive data chunks. This algorithm was initially implemented using only scalar instructions. The scalar version of the application was executed on the core operating in user mode, with virtual memory enabled. This environment simulates a realistic operational context, ensuring the algorithm's performance is evaluated under conditions similar to actual use cases.

In this implementation, the application sequentially processed the data using scalar instructions, which handle one data element per instruction cycle. Despite the straightforward nature of scalar processing, it tends to be less efficient for operations involving large datasets due to its inability to exploit data-level parallelism.

To address this inefficiency, the application was reimplemented using vector instructions. Vector instructions can process multiple data elements simultaneously, thereby significantly improving the throughput of data-intensive operations. This vectorized version was also run on the core in user mode with virtual memory enabled, maintaining the same operational context to ensure a fair comparison. The operations of application is shown in figure

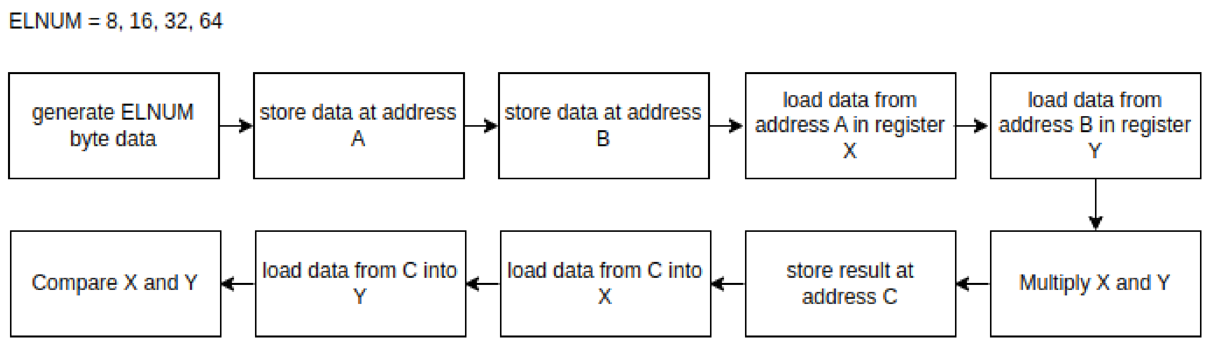

Figure 3.1 Customized App data flow

The results demonstrated significant efficiency gains: the scalar instruction-based application's cycle consumption progressed exponentially while the vector based application's cycle count almost remained the same to complete the same amount of operations compared to the vector instruction-based application.

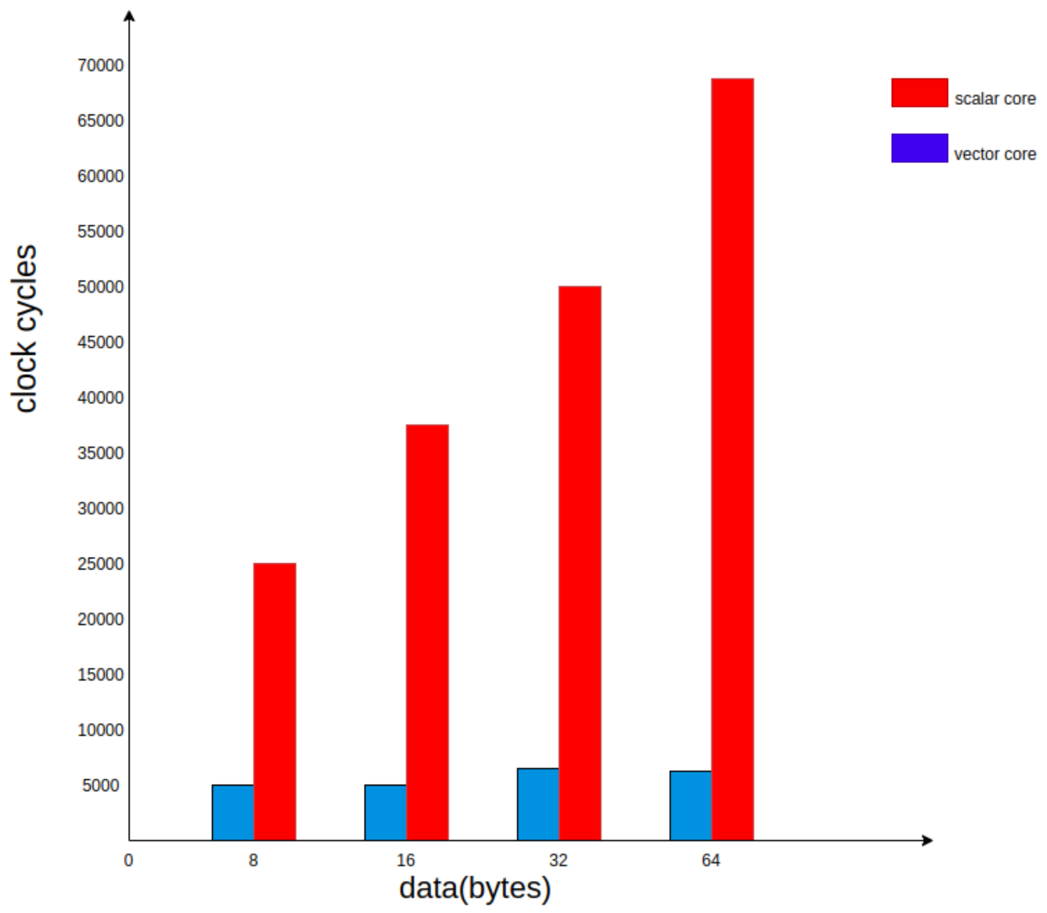

Figure 3.2 clock cycles vs. data

The instructions per cycle comparison is:

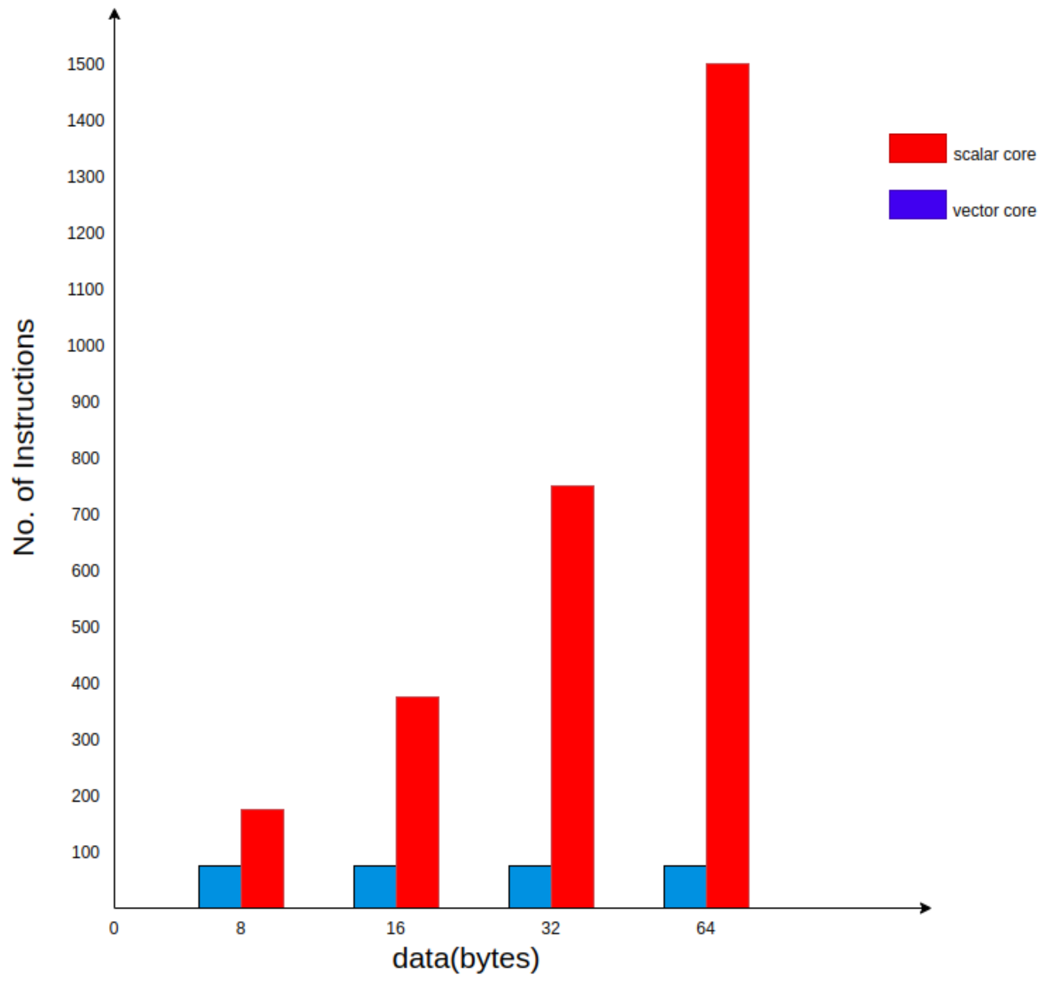

Figure 3.3 No. of instruction vs. data

In summary, running a vector application on a core significantly enhances its speed and efficiency. For a core to run vector applications, it must have vector extensions enabled, and the vector coprocessor must support booting Linux to facilitate communication with the hardware. Additionally, the vector processor needs to support virtual memory.

In this research, we successfully enabled virtual memory for the vector coprocessor ARA, making it capable of booting Linux and running vector applications. This marks a significant milestone, as ARA is now fully equipped to handle advanced computing tasks with improved performance.

However, due to resource constraints, we were unable to boot Linux fully because of the limited availability of larger FPGAs required to map both Ariane (CVA6) and ARA. The combined size of CVA6 and ARA exceeds the capacity of the available FPGAs in Pakistan. We attempted to use the following boards from UET:

- Xilinx Kintex-7 FPGA KC705
- Xilinx Zynq-7000 ZC702
- Nexys A7

Despite these constraints, we were able to successfully boot Linux by stubbing ARA, including its wrapper, on these boards. This success confirms that our design is functioning correctly and lays the groundwork for future implementations on more capable hardware.

BIBLIOGRAPHY

[1] RISC-V Foundation, "Volume 2, Privileged Specification version 20240411," [Online]. Available: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

[2] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7GHz 64bit RISC-V core in 22nm FDSOI technology," arXiv e-prints, Apr. 2019.

[3] "Working draft of the proposed RISC-V V vector extension," 2019, accessed on March 1, 2023. [Online]. Available: https://github.com/riscv/riscv-v-spec

[4] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual: User-Level ISA," CS Division, EECS Department, University of California, Berkeley, CA, USA, Jun. 2019, version 20190608-Base-Ratified.

[5] Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., & Benini, L. (2021). Ara: A Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI. IEEE Transactions on Computers, 70(7), 1173-1187

[6] OpenHW Group, "CORE-V Family of Open Source RISC-V Cores," 2024. [Online]. Available: https://github.com/openhwgroup

[7] OpenHW Group, "CVA6: Application-Class RISC-V CPU Core," 2024. [Online]. Available: https://github.com/openhwgroup/cva6.

[8] Zhang, Y., Gu, Y., & Zhang, W. (2020). Design and Implementation of CVA6: A High-Performance RISC-V CPU Core for Embedded Applications. Journal of Systems Architecture, 107(1), 101725.

[9] S. F. Beldianu and S. G. Ziavras, "ASIC design of shared vector accelerators for multicore processors," in 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, Oct. 2014, pp. 182–189.

[10] Y. Lu, S. Rooholamin, and S. G. Ziavras, "Vector coprocessor virtualization for simultaneous multithreading," ACM Trans. Embed. Comput. Syst., vol. 15, no. 3, pp. 57:1–57:25, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2898364

[11] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, "A transprecision floating-point architecture for energy-efficient embedded computing," in 2018 IEEE International Symposium on Circuits and Systems (ISCAS), May 2018, pp. 1–5.

[12] Smith, J. E., & Nair, R. (2005). Virtual Memory: Issues, Architectures, and Implementations. ACM Computing Surveys (CSUR), 37(3), 357-403.

VITAE

I, Gidha Iftikhar, earned my Bachelor’s degree in Electrical Engineering from CEME NUST in 2016. Following graduation, I spent a year teaching physics to FSc students at Brook Field College in Rahim Yar Khan. Transitioning to the professional sphere, in 2018, I began my role as a Teaching Assistant at Information Technology University in Lahore.

In 2021, I pursued my studies by enrolling in the MS EE program at UET and securing a prestigious SemiConductor Industry Fellowship from 10xEngineers. During the second semester of my MS EE studies in February 2022, I joined 10xEngineers as a Graduate Intern. By August 2022, I had advanced to the position of Design Engineer, where I engaged in various client projects.

Throughout my tenure, I've had the privilege of working on diverse projects, including both client-based and open-source initiatives focused on RISC-V compliant scalar and vector cores. Additionally, I've gained valuable experience collaborating on projects involving superscalar cores and Very Large Instruction Word (VLSI) architectures.