American University of Beirut

EECE 501: Final Year Project

Final Report

A Configurable RISC-V Vector Accelerator for Machine Learning Applications

Authors:
Imad Al Assir
Mohamad El Iskandarani
Hadi Rayan El Sandid

Supervisor:
Dr. Mazen Saghir

Fall 2020

1 Executive Summary

Recently, the world has witnessed a massive increase in applications requiring Machine Learning and Deep Learning. These applications span multiple areas including healthcare, security, automotive industry, etc... A lot of research effort is currently going into building optimized hardware platforms for these applications and some of these solutions are algorithm-specific accelerators, optimizing applications on GPGPUs, TPUs, systolic arrays, neuromorphic computing and others. While these solutions successfully accelerate their assigned tasks, they are either too specific and thus quickly obsolete, require too much power or are fairly complex and are still in their early phases. One other solution is to use vector processors since these are designed to work on long vectors of data, making them ideal for ML and DL applications. Therefore, our solution is to build a vector accelerator for Machine Learning that is compliant with the recent RISC-V Vector Extension ISA.

The tasks needed to implement this solution are numerous and include: a VHDL description of the accelerator compliant with the RISC-V Vector ISA, an FPGA prototype of the accelerator integrated into a RISC-V core, a gem5 model of the accelerator, modifications to the LLVM/Clang cross-compiler to optimize code for our hardware.

During the fall semester, the team was able to achieve satisfactory progress by implementing, testing and combining the controller, the vector register file and ALU in hardware, as well as getting up to speed with the gem5 simulator and choosing LLVM/Clang as a cross-compiler.

Contents

List of Figures

2 Introduction

2.1 Motivation

Recently, there has been a massive increase in applications requiring Machine Learning and Deep Learning. However, these compute-intensive tasks cannot run efficiently on existing hardware. Even though these are currently executed on general-purpose hardware (e.g. CPUs and GPGPUs), they consume way too much power. Therefore, custom hardware needs to be designed for these specific applications. Other approaches have been to execute Machine Learning (ML) and Deep Learning (DL) tasks on tensor processing units (TPUs) or using neuromorphic computing. These solutions are great but are still in their early stages and are fairly complex. Another approach has been to design accelerators for specific algorithms. While these algorithm-specific accelerators are great, the algorithms they accelerate change very quickly, thus making the accelerator obsolete quickly. Note that we do not aim to run our accelerator at a faster frequency than algorithm-specific accelerators that are optimized, but our main goal is to support a wide range of ML algorithms, and outperform software and maybe even general-purpose platforms. Vector accelerators are particularly interesting because of their inherent exploitation of data parallelism, making them well suited for efficiently accelerating ML algorithms used in video, image and audio processing applications.

2.2 Desired Needs

Our vector accelerator's goal is to provide a design-time configurable platform for future ML and DL applications. Being compliant with the RISC-V vector extension, our design is compatible with existing tools designed for this ISA.

3 Requirements and Deliverables

In this section, we describe the requirements, deliverables of our project.

3.1 Requirements and Specifications

Our accelerator should comply with the the latest version of the RISC-V Vector Extension ISA (currently v0.9)[1]. It should be able to perform better than software on the MLPerf benchmark suite [2]. As for specific constraints such as timing, area, power consumption, they have not been discussed yet as our main focus currently is correctness. However, as a loose lower bound, we require the accelerator to run at a clock frequency of at least 150 MHz.

3.2 Deliverables

The expected deliverables at the end of the project are:

- Design and VHDL implementation of the vector accelerator hardware. This includes a controller, a vector register file, a vector Arithmetic and Logic Unit (ALU) and a memory unit. The datapath must consist of at least two lanes. The processor should also support chaining.
- Support for a subset of instructions specified in ISA. Details can be found in Appendix A.
- Support for special data types for Machine Learning: bfloat16 [3], posits [4].
- Field Programmable Gate Array (FGPA) prototype of the vector accelerator. The accelerator should be integrated within a RISC-V core.
- Implementation of a gem5 [5] model of the accelerator, which can be tested using the MLPerf benchmark suite.
- Customization of the LLVM/Clang cross-compiler [6] to optimize C code execution on the accelerator.
- Execution of ML programs on the platform with minor to no modifications.

4 Technical and non-Technical Constraints

In this section, we present the technical and non-technical constraints.

4.1 Technical Constraints

Technical constraints include:

- Compliance with the RISC-V Vector ISA specifications.
- Programmability of the accelerator in C using appropriate intrinsics.
- Operation of the hardware prototype at a clock rate of 150 MHz or better.
- Support for special data types for Machine Learning: bfloat16, posits.
- Utilization of Xilinx tools and prototyping on a Nexys Video FPGA [7].
- Achievement of better performance and less power-consumption in gem5 simulations of our hardware compared to software.
- Ability of the hardware design to scale to support more lanes and/or execution units.

4.2 Non-Technical Constraints

Non-technical constraints include:

- Availability of FPGA and software programs in ECE labs.

5 Literature Review

Supercomputers have long been synonymous to vector processors. As Valero notes in [8], vector processors have "reigned supreme" from 1975 to 1995. Their success can be attributed to their innovative idea of operating on complete vectors, exploiting data-level parallelism, as well as their ability to hide memory latency and overall energy efficiency –since they have less instructions to fetch and decode. Although superscalar processors took over due to their economic feasibility and increased research effort, current processors are increasingly starting to resemble classical vector architectures. In fact, many processors now support SIMD extensions (e.g. Intel AVX-512 [9]) with large vector-like execution units. Even GPUs, especially AMD's, closely resemble vector processors with their internal SIMD-vector units and their reliance on SIMT – a combination of simultaneous multithreading and SIMD. Moreover, just like current architectures borrowed ideas from vector architectures, the opposite is now true as there have been several recent attempts to revive vector processors by using current techniques such as Out-Of-Order execution. Finally, Valero et al. believe that vector processors could make a comeback in applications that work on long vectors such as Deep Learning and Gene Sequencing. However, they mention that for them to be successful, an ISA should support long vector instructions, and runtime systems and programming models should adapt to this new approach.

Another noteworthy novel approach to accelerate ML algorithms is Neuromorphic Computing, specifically Intel's Loihi: a Manycore Processor with On-Chip Learning [10]. The design is inspired by how the brain functions and, although some biological mechanisms are hard to replicate, they were able to achieve a preliminary design using simplified abstractions. Loihi is specifically designed for Spiking Neural Networks (SNN), which unlike Artificial Neural Networks "incorporate time as an explicit dependency in their computations". Conventional architectures do not support SNN models well. However, these models could be underappreciated, just like ANNs were before improvements to CPUs and GPUs. Results obtained show that although an ultra-low voltage processor (Intel Atom) outperformed Loihi for a small number of unknowns in a least absolute shrinkage and selection operator (LASSO) operation –which is a method used to enhance the prediction accuracy and interpretability of a statistical model [11]–, Loihi significantly shined for a large number of unknowns with almost 50x less energy, 120x less delay and 5000x less EDP. These preliminary results show the potential for SNNs and neuromorphic computing.

The need for faster customized hardware is a result of the exponentially increasing demand on machine learning applications. In [12], Jouppi et al. note that the increasing usage of voice recognition by consumers required a double in computation power by the servers! Adding more cores and servers wouldn't suffice to keep up with the demand, which made a case for using ASICs with the purpose of accelerating specific computations and algorithms. A recent prime example of such hardware is the tensor processing unit (TPU). The TPU was designed to run as a coprocessor to accelerate the inference stage of neural networks, where it is basically a multi pipeline stage matrix unit that relies on data coming from different directions instead of just one buffer. It yielded incredible performance against CPUs and GPUs in running neural network inference applications and models reaching 40X GPU performance and 80X CPU performance. On a large scale, cost-performance overshadows raw performance, and the TPU manages to achieve 14-34x better performance per watt versus the traditional CPU and GPU servers.

The usage of FPGAs in algorithm acceleration is not novel by any means. As seen in [13], Bai et al. managed to run a 48 node cluster of Zynq FPGAs to accelerate cryptographic algorithms which yielded impressive results against a traditional processor and a many-core server. A key benefit of utilizing FPGAs is exploiting parallelism and energy efficiency, for the cluster achieved x3.6 better energy efficiency than the server using the performance per watt metric.

Regarding our chosen machine learning benchmark MLPerf, it is slowly becoming the go-to

metric for machine learning accelerators [14]. Nowadays, MLPerf is being evaluated on traditional desktop processors, 2048 core architecture servers, and everything in between.There are several models that can be evaluated such as ResNet, SSD, and RCNN. Several optimization methods have been studied to further enhance the training and evaluation of these models on accelerators, such as model parallelism, weight update sharding, and gradient summation.

To assess the performance of our hardware design, we have looked for a simulator where we could model our hardware and benchmark it with enough accuracy on different workloads. Our selected hardware simulator is gem5 [5], which is a modular platform for computer-system architecture research. It encompasses system-level architecture, as well as processor microarchitecture. To get familiar with the gem5 simulator, we have mainly used the online book available here [15]. It contains information and examples to help getting started with the simulator.

One of the constraints of our project is the programmability of the accelerator in C. If we write C code, it should be compilable into RISC-V binaries making full use of the vector operations offered by our accelerator unit. At the start of our project, we have decided to use the RISC-V GNU/GCC cross-compiler [16] to generate RISC-V binaries. However, we have recently decided to transition from using GNU/GCC to using the LLVM/Clang cross-compiler [6]. The motivation behind this move is that the LLVM/Clang codebase is easier to work with than the GNU/GCC codebase. Using LLVM/Clang is a better choice, as we might modify the cross-compiler to implement features later on in the project. LLVM/Clang also offers experimental support for RISC-V Vector Instructions, but that should be researched further.

6 Methodology

In this section, we present the methodology we used to approach the design of the vector accelerator.

6.1 Hardware

Our first step was to get a firm grasp of the basics before we could dive into the design phase. We refreshed our memory on vector computing and studied comparable architectures in our literature review. Next, we studied the RISC -V vector instruction set architecture specifications sheet to start planning our design and pointing out any constraints we should abide by.
The design phase was ready to launch. We began coding each component in isolation and test benching it on its own. Once we made sure it was outputting the desired functionalities, we would combine it with another component in another entity. For example, once we made sure each of the register file and arithmetic unit were functional, we coupled them and tested the newly obtained entity. This bottom up procedure was our approach throughout the design phase. If the test benching did not yield the desired results, we would revisit the components and debug to find the root of the problem.
Once the VHDL description of the components is complete, the next step is to setup an Advanced eXtensible Interface (AXI) interface [17] whose purpose is to communicate with our processor, since the accelerator is designed to run as a coprocessor with MicroBlaze [18] –a soft-core microprocessor, i.e. that is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs– on FPGA. Once the AXI interface is ready, we will test it in software using the Xilinx Software Development Kit (SDK) [19]. If everything is functioning smoothly, we will transition to the integration of the design into a RISC-V sweRV core [20]. After rigorous testing, we will prototype the design on our FPGA again.

6.2 Software

We are currently using several software tools to complement our work on the vector accelerator hardware. These tools include the LLVM/Clang compiler toolchain, which is used to program our accelerator with a high-level programming language like C or C++, and the gem5 hardware simulator, which is used to model our accelerator and assess both its functionality and performance under specific workloads. We are also plan on using other tools, like the MLPerf benchmark suite.

6.2.1 The LLVM/Clang Compiler Toolchain

The LLVM/Clang toolchain is a set of compiler tools available under the LLVM project. Clang is the front-end of the toolchain, turning C/C++ source code into LLVM Intermediate Representation (IR) instructions[1]. These LLVM IR instructions are optimized using the LLVM optimizer, then turned into machine code for the target architecture using the LLVM static compiler.
To install the LLVM/Clang toolchain, we first followed the instructions on the LLVM-Project repository to download and build LLVM/Clang with support for RISC-V targets. Then, we have configured the toolchain to be able to cross-compile C code into RISC-V binaries. Finally, we tested if we have configured the cross-compilation process properly by compiling sample C code into RISC-V binaries, which were then used as simulation workload in a gem5 RISC-V system. A diagram describing the compilation of C/C++ code into RISC-V binaries is shown in Figure 1.

[1] An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. They are used to abstract away most details of the target hardware [21]

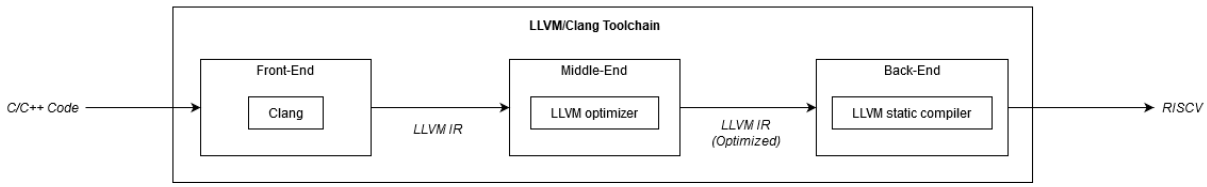

Figure 1: Cross-compiling C code into RISC-V binaries using the LLVM/Clang Toolchain

While we can cross-compile from C to RISC-V, this does not necessarily mean that we can write programs for use with our accelerator, as it only supports instructions from the unreleased RISC-V vector extension. LLVM/Clang currently offers experimental support for RISC-V vector instructions, which we will explore in the upcoming weeks to cross-compile code for our accelerator without extending LLVM. Extending LLVM/Clang for RISC-V Vector support would involve adding intrinsic functions, which would allow us to call specific RISC-V vector instructions directly in our C code using programming constructs.

6.2.2 The gem5 Simulator

The gem5 simulator is a modular platform used to simulate a computer system. It offers parameterized models for many hardware components (i.e. DRAM, CPUs, Caches ...), and an API to model custom hardware for use in the simulator. A diagram of the simulator's architecture is shown in Figure 2.

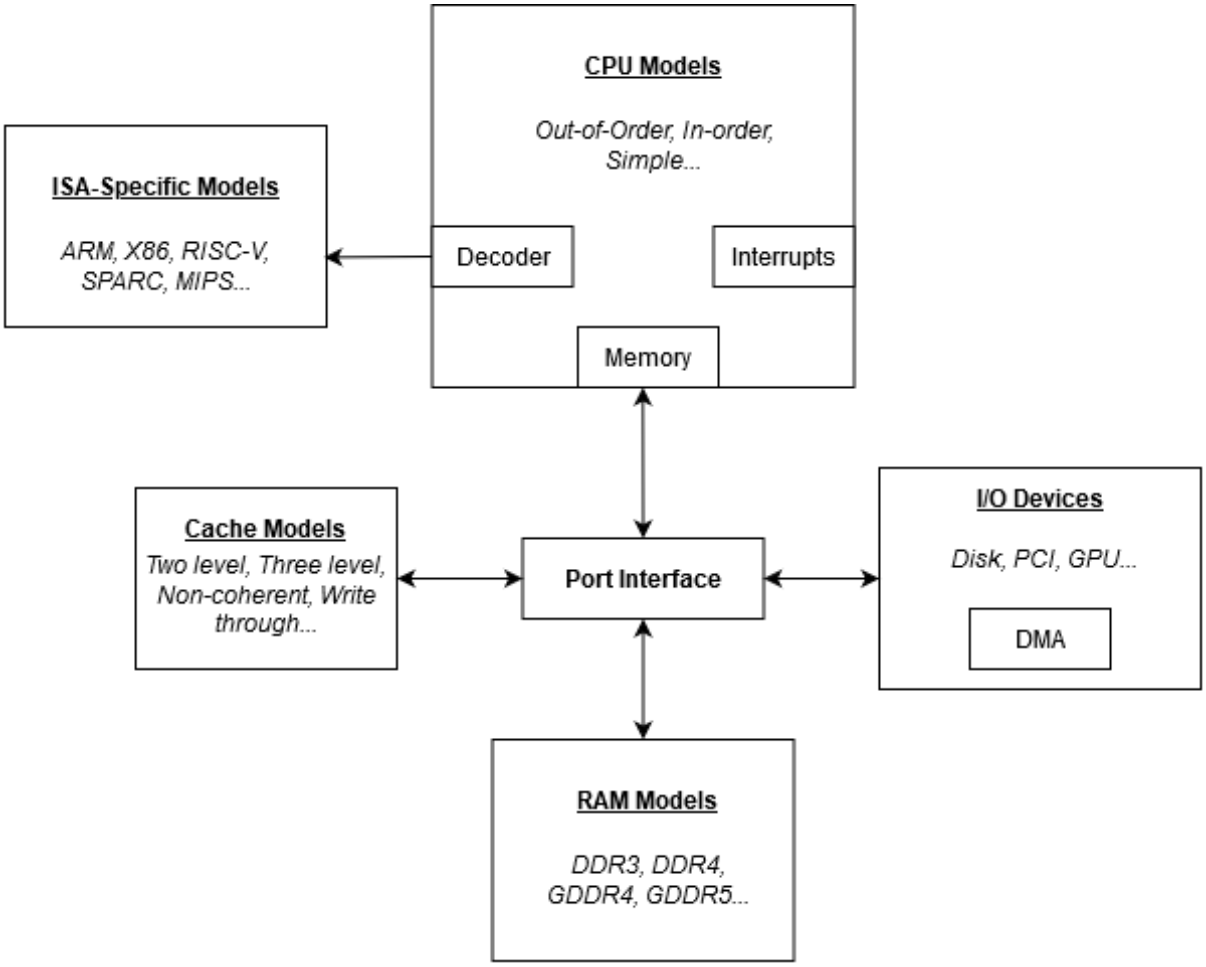

Figure 2: Overview of gem5's architecture

To get started with gem5, we first cloned its source files from the official repository and built

it with RISC-V ISA support. We then followed documentation and examined different example projects included in the source files to learn how to use the simulator. There are two main aspects to consider when working with the gem5 simulator : configuration files and hardware models.
Configuration files for gem5 are included in the '/cfg' directory of the repository. They are used to specify the structure of our simulated computer-system (i.e. which hardware components to use, how they are connected), and also to specify the general characteristics of the computer-system we would like to simulate (i.e. CPU frequency, memory size, etc.).
Hardware models in gem5 are included in the '/src' directory of the repository. Hardware models in gem5 inherit from a base type 'SimObject', which contains all main interfaces and parameters used in gem5 simulations. Extending gem5 by adding hardware models requires us to rebuild the simulator every time we modify or add a hardware model to implement the changes. A sample system simulation is illustrated in Figure 3.

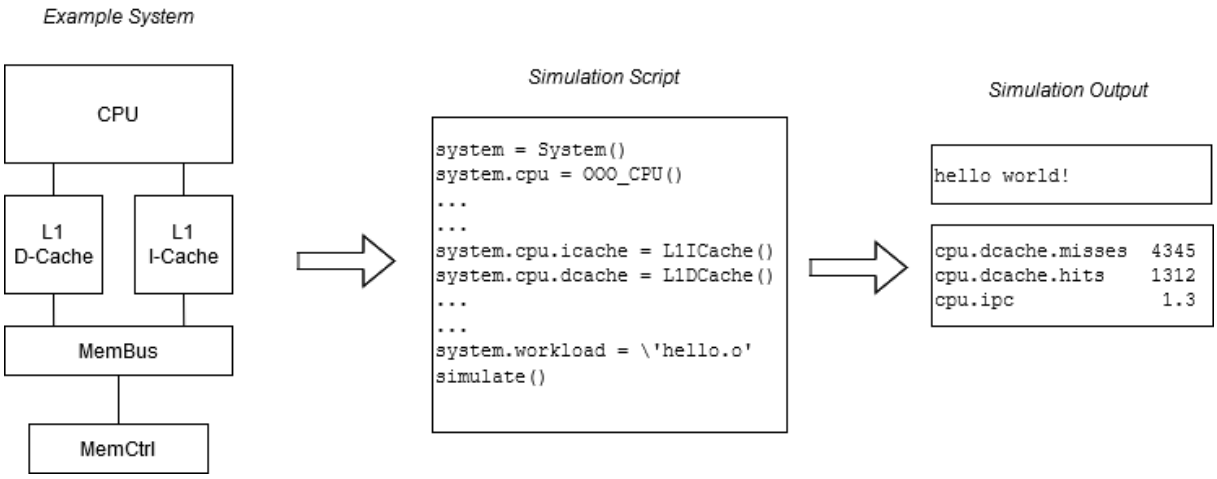

Figure 3: Sample system simulation using gem5

To assess the functionality and performance our vector accelerator, we intend to model it in gem5 by designing an equivalent hardware model. This hardware model will then be used to run simulations using our target workloads (i.e. MLPerf Inference benchmarks). It will also be used as a reference to evaluate the hardware implementation.

6.2.3 The MLPerf Benchmark Suite

The MLPerf Benchmark Suite is used to assess the performance of hardware specifically designed for machine learning applications. It offers a training benchmark suite, which is used to measure how fast systems can train models to a target quality metric, and an inference benchmark suite, which is used to measure how fast systems can process inputs and produce results using a trained model. We have decided to use the inference benchmark offered by MLPerf benchmark suite to assess the performance of our accelerator.
Benchmarks in the MLPerf inference suite are defined by a task, a model, and a dataset. There are three sets of benchmarks in the MLPerf inference suite we would like to use: the datacenter systems suite, the edge systems suite, and the mobile systems suite. These are described respectively in Figures 4, 5 and 6.

Area	Task	Model	Dataset
Vision	Image Classification	Resnet50-v1.5	ImageNet (224*224)
Vision	Object Detection	SSD-ResNet34	Coco (1200*1200)
Vision	Medical Image Segmentation	3D UNET	BraTS (224*224*160)
Speech	Speech-to-text	RNNT	Librispeech
Language	Language Processing	BERT	SQuAD v1.1
Commerce	Recommendation	DLRM	1B click logs

Figure 4: Benchmarks in the MLPerf inference suite for datacenter systems

Area	Task	Model	Dataset
Vision	Image Classification	Resnet50-v1.5	ImageNet (224*224)
Vision	Object Detection	SSD-ResNet34	Coco (1200*1200)
Vision	Object Detection	SSD-MobileNets-v1	Coco (300*300)
Vision	Medical Image Segmentation	3D UNET	BraTS (224*224*160)
Speech	Speech-to-text	RNNT	Librispeech
Language	Language Processing	BERT	SQuAD v1.1
Commerce	Recommendation	DLRM	1B click logs

Figure 5: Benchmarks in the MLPerf inference suite for edge systems

Area	Task	Model	Dataset
Vision	Image Classification	MobileNetEdgeTPU	ImageNet (224*224)
Vision	Object Detection	SSD-MobileNets-v2	Coco (300*300)
Vision	Segmentation	DeepLabv3+	ADE20K
Language	Language Processing	Mobile-BERT	SQuAD v1.1

Figure 6: Benchmarks in the MLPerf inference suite for mobile systems

The reference implementations of the MLPerf benchmarks are currently only available in Python, and cannot be directly cross-compiled to a RISC-V target. Thus, we will be investigating methods to port the MLPerf reference implementations to C/C++ in the upcoming weeks.

7 Design

In this section, we provide details on the hardware and software design of our vector accelerator.

7.1 Hardware

The vector accelerator design consists of various components. It consists of 3 pipeline stages (for simplicity): Decode (Control signals generation and banked Register File access), Execute (ALU and memory) and Write-Back. The processor supports chaining[2] between ALU lanes themselves, and between memory and ALU. This technique aims to reduce computational speed by taking the result directly from the generating unit, thus skipping additional register file bank accesses and avoiding Read-After-Write (RAW) hazards. We discuss below some of the design decisions taken.

7.1.1 Design Alternatives

Hardware design alternatives include:

- **Number of lanes**: Although 1 lane would have been a lot simpler to implement. One of our main goals was to support chaining. Therefore,for chaining to apply, we need to have at least 2 execution lanes. We chose to have this minimum of 2 execution lanes (for both ALU and memory) for simplicity.
- **Number of register file banks** To be able to service both lanes efficiently, 2 register file banks were designed with 2 read ports and 1 write port each. The 32 registers required by the ISA were split into 2, with each bank containing respectively the lower and upper 16 vector registers. There could have been other ways to split the banks including an odd/even distribution or other more complex methods, but this method was chosen for its simplicity. The implication here is that the compiler has to choose registers that map to banks in an alternating manner. In other words, if a vector add instruction is to be executed, the compiler maps the operands to registers from the lower bank. Next, if a vector subtract instruction is to be executed, the compiler should map the operands to registers from the other bank to avoid structural hazards. Note that there could have been a method to arbitrate that in hardware on-the-fly, but it requires more complex hardware that is not currently our priority.
- **Dedicated register file banks/lanes**: Dispatching instructions to register file banks and subsequently providing the register outputs to the lanes can be designed in various ways. It can be arbitrary, i.e. any lane can read from any bank, or it can be dedicated, i.e. each lane reads/writes to 1 bank only. The implication here is that if 1 lane and 1 bank are busy and a subsequent instruction wants to read from that same bank, it should stall due to a structural hazard. However, as mentioned above, the compiler can prevent this from happening.
- **Choice of FPGA**: Zedboard [22] vs Nexys 4 [23] vs Nexys Video [7]. The Zedboard is an FPGA with an on-chip Zynq processor. This FPGA could have been used for testing purposes early on. However, since we will be integrating our accelerator into a RISC-V core, we will not need the Zynq processor and we will be unable to use the on-chip memory that is connected to the Zynq. The Nexys 4 is an FPGA with only logic fabric and no processor. However, this FPGA does not have an external DDR memory. An 'upgraded' version of the Nexys 4 FPGA is the Nexys Video FPGA which contains both a larger logic fabric and an external DDR3 memory.
- **MicroBlaze vs Zynq vs RISC-V processors**: On FPGA, various processors could be used to communicate with our accelerator. These include a MicroBlaze softcore processor,

[2]Chaining is not yet supported but will be later on.

a Zynq processor and a RISC-V processor. For testing purposes, both the MicroBlaze and the Zynq processor [22] could have been used and our accelerator would be a co-processor that executes vector instructions. However, we chose to go for MicroBlaze during testing because we can implement it on any FPGA, unlike the Zynq which is only available on the Zedboard. Moreover, our goal is to integrate the accelerator into a RISC-V core, so we will have to migrate from MicroBlaze after validating the design. We chose to use the SweRV core by Western Digital [20] because of its availability to students and numerous features.

- **Instruction scheduling**: To avoid dealing with complex scheduling, we are assuming that no 2 subsequent instructions will access the same register file bank/lane. For now, we will have to write Assembly code to be able to control that. Later on, we will add scheduling logic, and the compiler will take care of addressing the correct banks.

- **Striping Distance**: The RISC-V vector spec defines an implementation-specific parameter which is the striping distance. Striping is used to distribute register accesses and operations in a vector unit, thus allowing the unit to access elements non-contiguously. For simplicity, we enforce that the striping distance should be equal to the maximum vector length, thus only supporting contiguous access to elements.

7.2 Software

Our vector accelerator unit will be modeled in gem5 and added to a functioning system model so we can run simulations and assess the unit's performance in ML-related workloads.

7.2.1 Design Alternatives

Software design alternatives include:

- **Modeling communication between a host CPU and our vector accelerator**: Currently, we are working on a gem5 system model where the host CPU communicates with our vector unit using memory mapped I/O. For our final design, we intend to have the vector unit tightly coupled to the host CPU's instruction pipeline.

- **LLVM/Clang vs GNU/GCC**: To generate RISC-V binaries, we have decided to transition from using the GNU/GCC cross-compiler to using the LLVM/Clang cross-compiler. This transition is motivated by the lack of modularity in the GNU/GCC codebase when compared to LLVM/Clang, as well as the experimental support for RISC-V instructions currently offered by LLVM/Clang.

- **LLVM/Clang cross-compiler support for RISC-V Vector Instructions**: The current release of LLVM/Clang offers experimental support for RISC-V Vector Instructions. We still have not tested this feature, but plan to do so in the upcoming weeks.

8 Implementation

In this section, we discuss implementation details related to our hardware and software. Our code can be found on Github [24].

8.1 Hardware

We have chosen to decouple the hardware entities as much as possible to facilitate debugging and having room for future optimization (e.g. adding pipeline stages to increase clock frequency). For that reason, the register offset generation –and memory addressing which will be implemented in the near future– are performed independently by the offset generator. A noteworthy detail is the need for register v0 as an input to the offset generator. According to the spec, register v0 stores the mask bits. These are used to selectively operate on part of the register. Therefore the offset generator needs the mask bits to provide the appropriate offsets to read from and write to. The full hardware design, excluding the memory, can be seen in Figure 7.

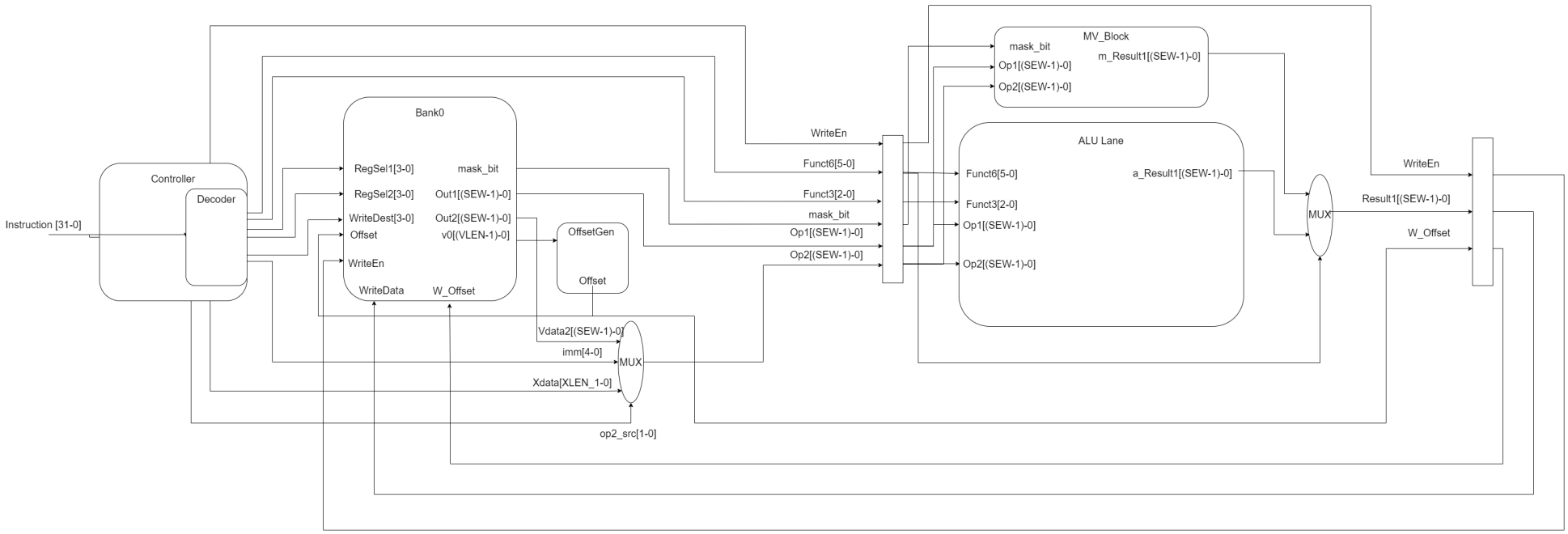

Figure 7: Datapath without Memory

8.1.1 Register File + Offset Generator

Next, we coupled the offset generator with the register file, where each bank has a separate generator. The offset generator provides the read and write offsets (i.e. the element number within a vector register). Note that the write offset is the same as the read offset but it is passed (delayed) through the pipeline registers, to make sure that the write offset enters the register file at the same time as the Write_Enable. This way, the bank code is only responsible for reading and writing from/to the registers without any additional logic. The register file design can be seen in Figure 8.

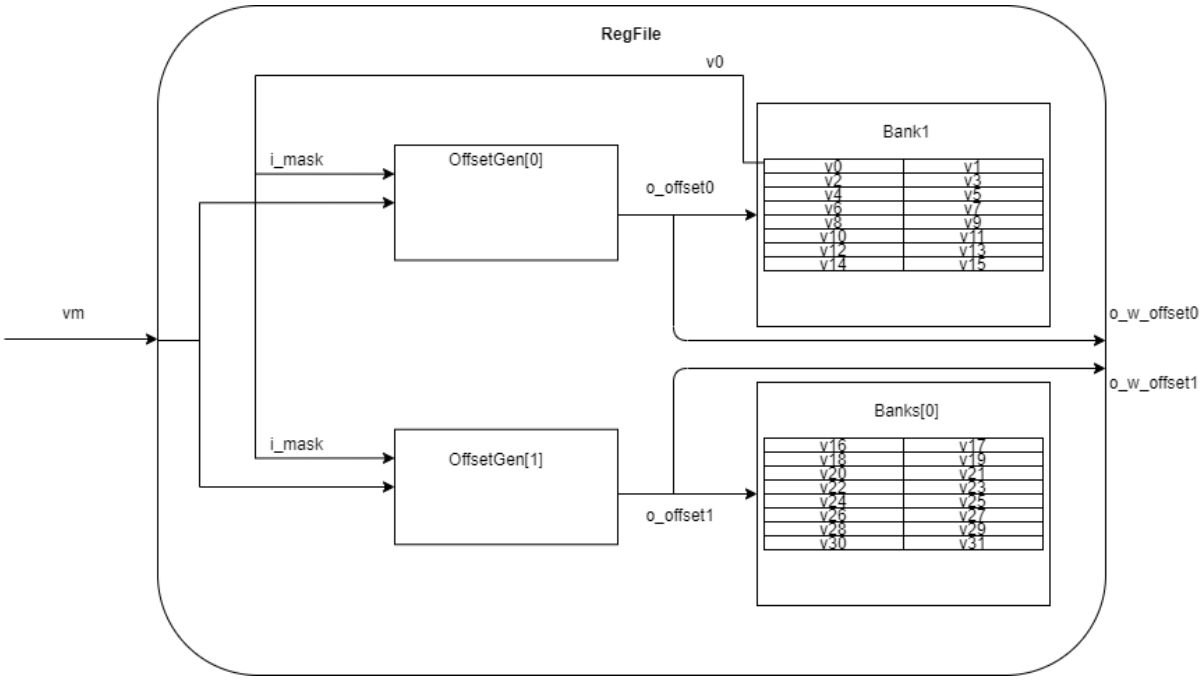

Figure 8: Register File Design

8.1.2 Scalability

We updated the project code in general to support scalability, where the input port widths are now dependent on the VHDL generic[3] NB_LANES (number of lanes). We also utilized VHDL "for generates" – which are basically loops used to instantiate components, instead of performing computational operations– to generate the register file banks, their respective offset generators and ALU lanes. This way, change of the generic NB_LANES alone is sufficient to scale the design completely. This was implemented on the register file, ALU unit, and controller. We are yet to apply it to the memory unit. An example showing how to implement scalability for the register file can be seen in Figure 9.

[3]VHDL generics provide a way to provide static information to blocks, similar to constants. However, unlike constants, their values can be supplied externally. They thus allow us to reconfigure our design at implementation-time without making changes to the code.

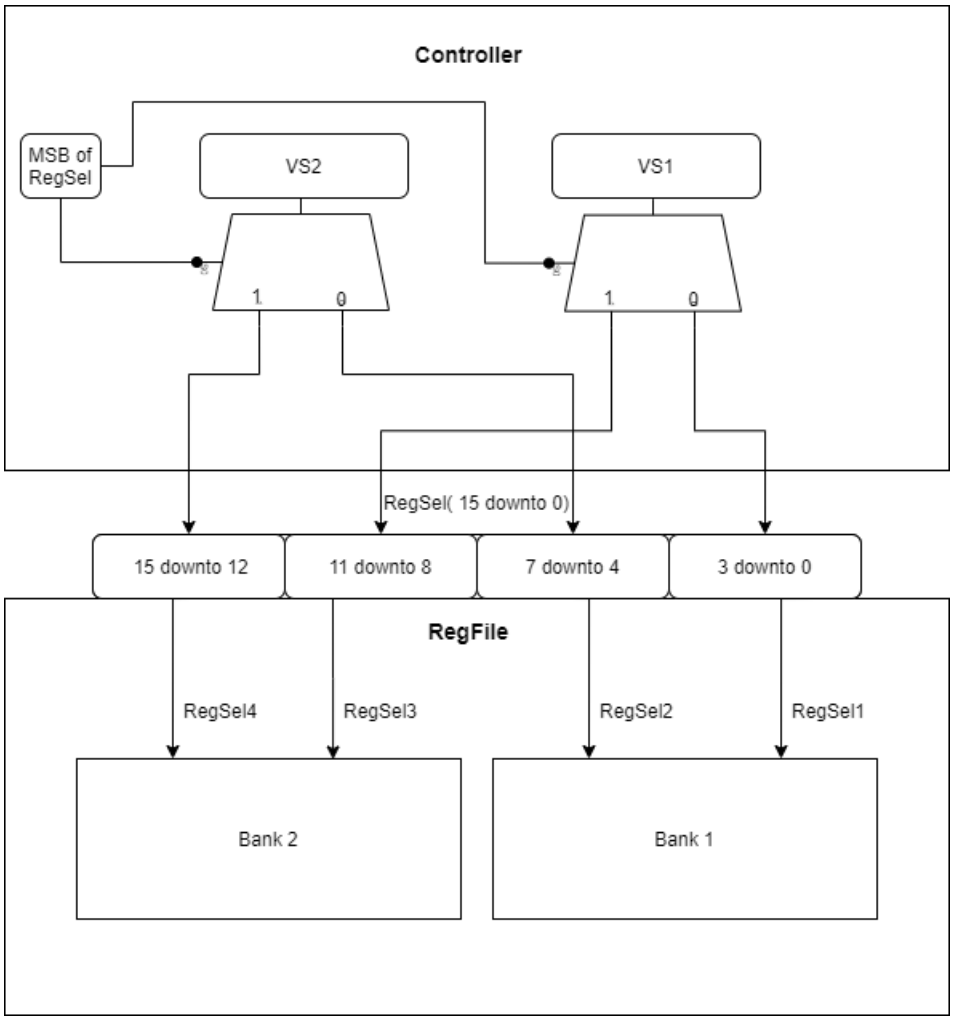

Figure 9: Support for scalability in the vector register file

8.1.3 Controller

We updated the control signals, exiting instruction fields, and encodings in compliance with the new RISC-V Vector ISA v0.9 Spec [1]. We then updated the controller to support scalability as mentioned previously and test benched the design.

8.2 Software

8.2.1 gem5 Model

To become familiar with the gem5 hardware simulator, we have started working on a system model containing a host processor and co-processor, where the host processor communicates with the co-processor through memory-mapped I/O. We designed gem5 'SimObjects' to model our hardware, and wrote custom Python configuration files to set simulation parameters and connections between hardware components. In the later stages of our project, we plan to have a working gem5 model of our vector accelerator to assess its functionality and performance.

8.2.2 Programmability in C

To generate RISC-V binaries from C code, we have been using the RISC-V GNU/GCC toolchain, and more recently the LLVM/Clang toolchain. We are currently testing LLVM/Clang's experimental support for RISC-V vector instructions, which would allow us to generate RISC-V binaries making full use of vector instructions supported by our accelerator unit.
In Figure 10, we can see a block of C code, and its equivalent LLVM IR instructions obtained

after compiling the code with Clang. The code has been vectorized using vectors of four 32-bit integer values, which are represented as ¡4xi64¿ elements in the LLVM IR code block, effectively reducing the total number of iterations required (i.e. 100 iterations required in the C code block vs 25 iterations required in the LLVM IR code block). Vector types used in LLVM IR are listed in Figure 11.

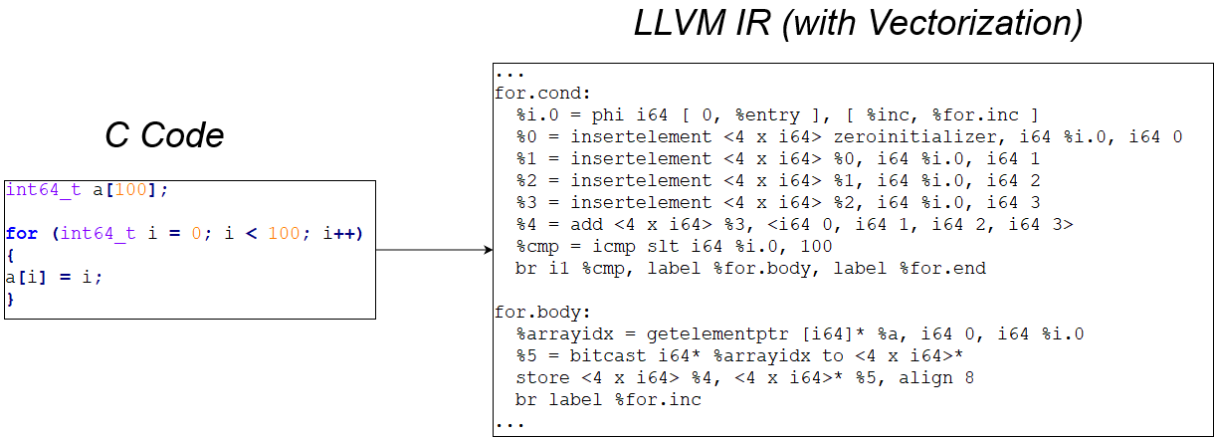

Figure 10: C code compiled into optimized LLVM IR with the LLVM/Clang toolchain

<4 x i32>	Vector of four 32-bit integer values.
<8 x float>	Vector of eight 32-bit floating-point values.
<2 x i64>	Vector of two 64-bit integer values.
<4 x i64*>	Vector of four pointers to 64-bit integer values.

Figure 11: Reference table for vector types used in LLVM IR

8.2.3 Using the MLPerf Inference Benchmark Suite

We are currently exploring tools to port the Python-based reference implementation of the MLPerf Inference benchmark suite to C/C++ so we could use the LLVM/Clang toolchain to cross-compile the benchmarks to a RISC-V target and use them in our gem5 simulations.

8.2.4 Testing Hardware Performance

We will use the gem5 model of our accelerator unit to validate its functionality, and evaluate its performance compared to a general-purpose scalar processor. The set of workloads we would like to use in our gem5 simulations are C/C++ ports of benchmarks present in the reference implementation of the MLPerf inference benchmark suite.

9 List of Applicable Standards

The standards that we abide by include:

- **ARM AMBA AXI4** [17]
- 1076-2019 - **IEEE Standard for VHDL** Language Reference Manual [25]
- **MLPerf Benchmark Suite** [2]
- **RISC-V Vector Extension ISA** [1]

10 List of Resources Needed

The resources used include:

- **Xilinx Vivado Design Suite**: used to write and simulate VHDL code. Imad and Mohammad were familiar with this tool. [26]
- **Xilinx Software Development Kit (SDK)**: used to write C/C++ code and execute them on FPGA. [19]
- **Nexys Video FPGA**: used for hardware prototyping. The FPGA was acquired from the ECE labs.
- **gem5 Simulator**: used for simulating the vector accelerator in a system, benchmarking it and comparing it to other designs. The gem5 simulator is open-source, and its codebase is publicly available. [5]
- **LLVM/Clang Toolchain**: used to compile C/C++ code into RISC-V binaries for our gem5 simulations. The LLVM/Clang Toolchain is open-source, and its codebase is publicly available. [6]
- **MLPerf** : used to benchmark the performance of our hardware with ML-related workloads. The MLPerf benchmark suite is open-source, and its set of repositories are publicly available. [2]

11 Timeline planning and schedule

In this section, we present Gantt charts to illustrate our progress and upcoming tasks. Please zoom in to see the charts and tasks clearly.

11.1 Fall

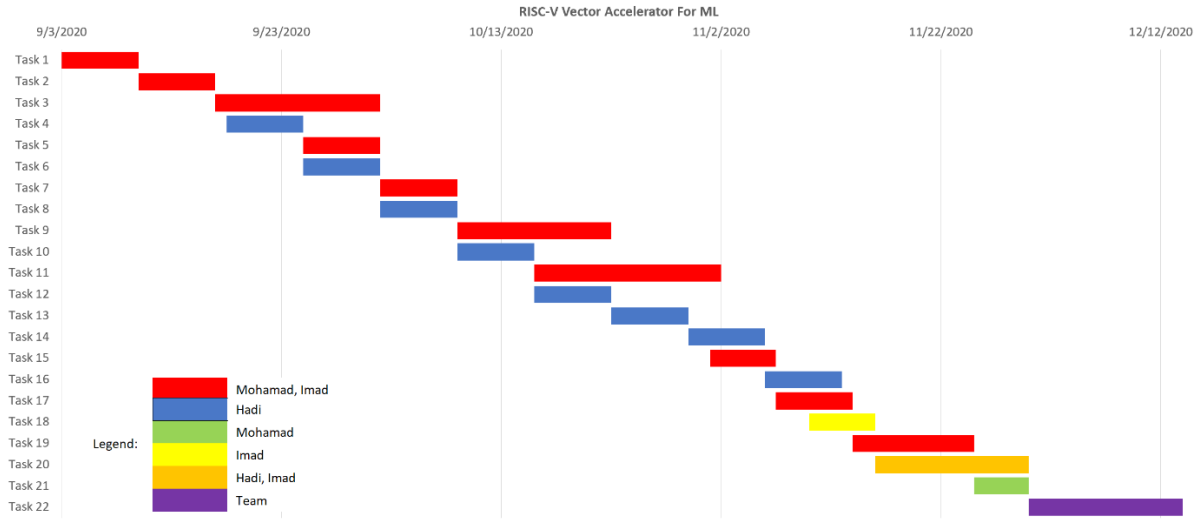

Figure 12: Gantt Chart showing our schedule for the Fall semester

TASK NAME	ASSIGNED TO	START DATE	DUE DATE	DURATION	% DONE	DESCRIPTION
Task 1	Mohamad, Imad	9/3/2020	9/10/2020	7	100	Review 499 code and recall bugs
Task 2	Mohamad, Imad	9/10/2020	9/17/2020	7	40	Implement masked operations in address generator
Task 3	Mohamad, Imad	9/17/2020	10/2/2020	15	100	Document problems
Task 4	Hadi	9/18/2020	9/25/2020	7	100	Set up website to track FYP progress & start learning about compilers
Task 5	Mohamad, Imad	9/25/2020	10/2/2020	7	100	Connect register file to ALU and debug
Task 6	Hadi	9/25/2020	10/2/2020	7	100	Debug FYP website
Task 7	Mohamad, Imad	10/2/2020	10/9/2020	7	100	Review vector spec sheet 0.9 and implement changes
Task 8	Hadi	10/2/2020	10/9/2020	7	100	Get started on Gem5 simulator
Task 9	Mohamad, Imad	10/9/2020	10/23/2020	14	80	Resolve the documented problems (address gen & register file)
Task 10	Hadi	10/9/2020	10/16/2020	7	40	Build gem5 with support for x86 ISA
Task 11	Mohamad, Imad	10/16/2020	11/2/2020	17	100	Update design for scalability (register file, ALU, controller)
Task 12	Hadi	10/16/2020	10/23/2020	7	100	Look into CPU models provided in Gem5
Task 13	Hadi	10/23/2020	10/30/2020	7	100	Download & build RISC-V GNU Toolchain & model basic ALU coprocessor
Task 14	Hadi	10/30/2020	11/6/2020	7	50	Learn how to work with tools in the RISC-V GNU Toolchain
Task 15	Mohamad, Imad	11/1/2020	11/7/2020	6	100	Renovate controller and debug
Task 16	Hadi	11/6/2020	11/13/2020	7		Look into a possible transition to from GNU/GCC to the LLVM/Clang cross-compiler
Task 17	Mohamad, Imad	11/7/2020	11/14/2020	7		Connect controller to Regfile - ALU
Task 18	Imad	11/10/2020	11/16/2020	6		Work on AXI interface
Task 19	Mohamad, Imad	11/14/2020	11/25/2020	11		Update and debug memory unit
Task 20	Hadi, Imad	11/16/2020	11/30/2020	14		FPGA prototype and software test
Task 21	Mohamad	11/25/2020	11/30/2020	5		Look into MLPerf algorithm implementations in hardware
Task 22	Team	11/30/2020	12/14/2020	14		Finalize Fall 2020 deliverables

Figure 13: Tasks completed during the Fall semester

11.2 Winter and Spring

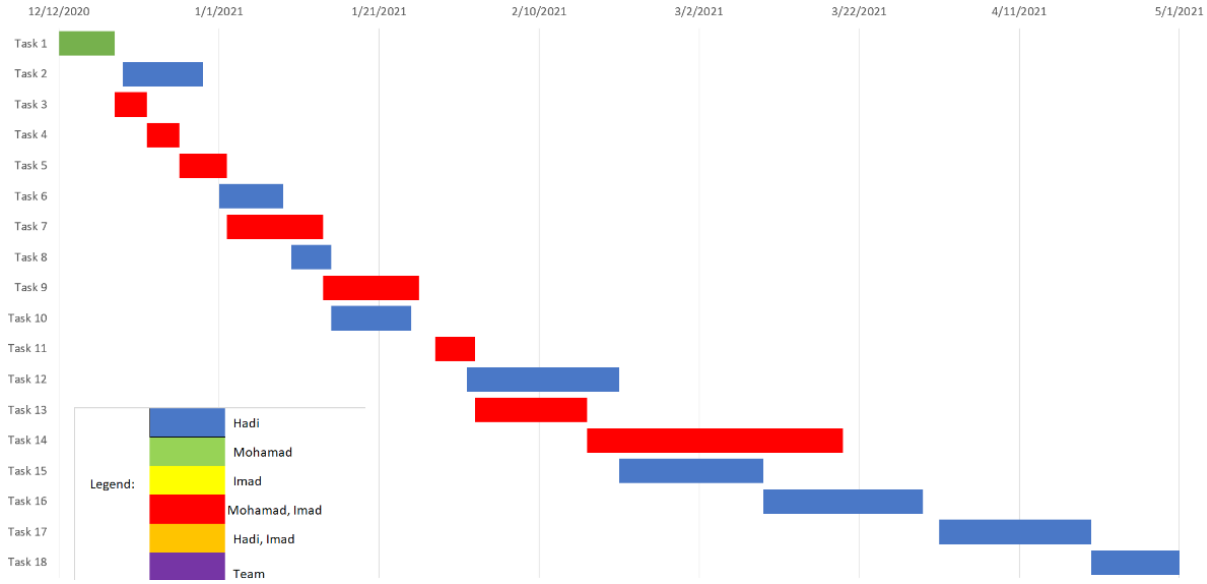

Figure 14: Gantt Chart showing our schedule for the Winter and Spring semesters

TASK NAME	ASSIGNED TO	START DATE	DUE DATE	DURATION	% DONE	DESCRIPTION
Task 1	Mohamad	12/12/2020	12/19/2020	7		Revise and clean up code
Task 2	Hadi	12/20/2020	12/30/2020	10		Explore options (e.g. tools like Cython) to port some of the reference MLPerf tests to C.
Task 3	Mohamad, Imad	12/19/2020	12/23/2020	4		Finalize controller design
Task 4	Mohamad, Imad	12/23/2020	12/27/2020	4		Test all cases and instructions rigorously
Task 5	Mohamad, Imad	12/27/2020	1/2/2021	6		Set up AXI interface and implement on FPGA
Task 6	Hadi	1/1/2021	1/9/2021	8		Complete gem5 model for ALU unit with memory-mapped I/O
Task 7	Mohamad, Imad	1/2/2021	1/14/2021	12		Design and testbench modified memory unit
Task 8	Hadi	1/10/2021	1/15/2021	5		Resolve some of the issues encountered when cross-compiling C code to RISC-V with vector instruction experimental support enabled in Clang/LLVM
Task 9	Mohamad, Imad	1/14/2021	1/26/2021	12		Implement memory on FPGA
Task 10	Hadi	1/15/2021	1/25/2021	10		Cross-compile workloads (i.e. ported MLPerf Inference tests) to RISC-V target with Vector Instructions support.
Task 11	Mohamad, Imad	1/28/2021	2/2/2021	5		Implement chaining
Task 12	Hadi	2/1/2021	2/20/2021	19		Implement RISC-V vector instructions in gem5
Task 13	Mohamad, Imad	2/2/2021	2/16/2021	14		Build system with swerV core
Task 14	Mohamad, Imad	2/16/2021	3/20/2021	32		Integrate vector unit to RISC-V core
Task 15	Hadi	2/20/2021	3/10/2021	18		Complete gem5 model for ALU unit tightly coupled to instruction pipeline
Task 16	Hadi	3/10/2021	3/30/2021	20		Model Vector Accelerator unit in gem5
Task 17	Hadi	4/1/2021	4/20/2021	19		Port a complete set of MLPerf Inference Benchmarks to our Target System
Task 18	Hadi	4/20/2021	5/1/2021	11		Run MLPerf Inference benchmark tests on our gem5 Vector Accelerator model and summarize performance results

Figure 15: Tasks to do during the Winter and Spring semesters

References

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

[2] "MLperf." [Online], Nov. 2020. Available: https://mlperf.org/.

[3] "BFloat16: The secret to high performance on cloud TPUs — google cloud blog." [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[4] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017.

[5] J. Lowe-Power, "Main page: gem5." [Online]. Available: http://gem5.org/Main_Page.

[6] "The LLVM compiler infrastructure." [Online], 2020. Available: https://llvm.org/.

[7] Digilent, "Nexys video reference manual." [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-video/reference-manual.

[8] J. Dongarra, V. Getov, and K. Walsh, "The 30th anniversary of the supercomputing conference: Bringing the future closer—supercomputing history and the immortality of now," *Computer*, vol. 51, pp. 74–85, oct 2018.

[9] Intel, "Intel® advanced vector extensions 512 (intel® AVX-512) overview." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[10] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, pp. 82–99, jan 2018.

[11] "Lasso (statistics)." [Online], Oct 2020. Available: https://en.wikipedia.org/wiki/Lasso_(statistics).

[12] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, jun 2017.

[13] X. Bai, J. Yang, Q. Dai, and Z. Chen, "A hybrid ARM-FPGA cluster for cryptographic algorithm acceleration," *Concurrency and Computation: Practice and Experience*, vol. 31, aug 2019.

[14] S. Kumar, V. Bitorff, D. Chen, C. Chou, B. Hechtman, H. Lee, N. Kumar, P. Mattson, S. Wang, T. Wang, Y. Xu, and Z. Zhou, "Scale MLPerf-0.6 models on google TPU-v3 pods," Oct. 2019.

[15] J. Lowe-Power, "Learning gem5." [Online]. Available: http://learning.gem5.org/.

[16] "GCC, the GNU compiler collection." [Online], Oct. 2020. Available: https://gcc.gnu.org/.

[17] ARM, "AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite." [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/.

[18] Xilinx, "Xilinx MicroBlaze processor reference guide." [Online], 2020. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf.

[19] Xilinx, "Xilinx software development kit (SDK)." [Online], 2020. Available: https://www.xilinx.com/products/design-tools/embedded-software/sdk.html.

[20] W. Digital, "RISC-V SweRV core family: Western digital." [Online]. Available: https://www.westerndigital.com/company/innovations/risc-v.

[21] "Intermediate representation." [Online], Nov 2020. Available: https://en.wikipedia.org/wiki/Intermediate_representation.

[22] Digilent, "Zedboard reference." [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/zedboard/start.

[23] Digilent, "Nexys 4 reference manual." [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4/reference-manual.

[24] I. A. Assir, M. E. Iskandarani, H. E. Sandid, and M. Saghir, "A configurable RISC-V vector accelerator for machine learning applications." [Online]. Available: https://github.com/imadassir/RISC-V_Vector_Accelerator_For_ML.

[25] "IEEE standard for VHDL language reference manual," *IEEE Std 1076-2019*, pp. 1–673, 2019.

[26] Xilinx, "Xilinx Vivado design suite." [Online], 2020. Available: https://www.xilinx.com/products/design-tools/vivado.html#overview.

Appendix A: Supported Instructions

Since the project is still in its early stages, only part of the RISC-V instruction set was implemented. These instructions are listed below, with a quick description. Please refer to the spec sheet for more details.
Note that .v stands for vector only instructions (e.g. memory instructions do not need multiple vector operands), .vv stands for vector-vector instructions, .vx vector-scalar instructions, and .vi vector-immediate instructions. Also note that the section number of each group of instructions refers to the section number in the RISC-V vector spec.

7.4 Vector Unit-Stride Instructions
vlw.v: vector load word signed
vlwu.v: vector load word unsigned
vle.v: vector load element
vsw.v: vector store word
vse.v: vector store element

7.5 Vector Strided Instructions
vlsw.v: vector load word strided signed
vlswu: vector load word strided unsigned
vlse.v: vector load element strided
vssw.v: vector store word strided
vsse.v: vector store element strided

7.6 Vector Indexed Instructions
vlxw.v: vector load word indexed signed
vlxwu.v: vector load word indexed unsigned
vlxe.v: vector load element indexed
vsxw.v: vector store word indexed
vsxe.v: vector store element indexed
vsuxw.v: vector store word unsigned indexed
vsuxe.v: vector store element unsigned indexed

12.1 Vector Single-Width Integer Add and Subtract
vadd.vv: element-wise vector addition
vadd.vx
vadd.vi
vsub.vv: element-wise vector subtraction (vs2[i]-vs1[i])
vsub.vx
vrsub.vx element-wise vector reverse subtraction (vs1[i]-vs2[i])
vrsub.vi

12.4 Vector Bitwise Logical Instructions
vand.vv: element-wise vector and
vand.vx
vand.vi
vor.vv: element-wise vector or
vor.vx
vor.vi
vxor.vv: element-wise vector xor
vxor.vx
vxor.vi

12.5 Vector Single-Width Bit Shift Instructions vsll.vv: shift left logical

vsll.vx
vsll.vi
vsrl.vv: shift right logical (zero-extended)
vsrl.vx
vsrl.vi
vsra.vv: shift right arithmetic (sign-extended)
vsra.vx
vsra.vi

12.7 Vector Integer Comparison Instructions

vmseq.vv: vd[i]= vs2[i]==vs1[i]
vmseq.vx:
vmseq.vi
vmsne.vv: vd[i]= vs2[i] != vs1[i]
vmsne.vx
vmsne.vi
vmsltu.vv: vd[i]= vs2[i] < vs1[i] unsigned
vmsltu.vx
vmslt.vv: vd[i]= vs2[i] < vs1[i] signed
vmslt.vx
vmsleu.vv: vd[i]= vs2[i] <= vs1[i] unsigned
vmsleu.vx
vmsleu.vi
vmsle.vv: vd[i]= vs2[i] <= vs1[i] signed
vmsle.vx
vmsle.vi
vmsgtu.vx: vd[i]= vs2[i] > vs1[i] unsigned
vmsgtu.vi
vmsgt.vx: vd[i]= vs2[i] > vs1[i] signed
vmsgt.vi

12.8 Vector Integer Min/Max Instructions

vminu.vv: element-wise minimum unsigned: vd[i]= min(vs2[i],vs1[i])
vminu.vx
vmin.vv: element-wise minimum signed: vd[i]= min(vs2[i],vs1[i])
vmin.vx
vmaxu.vv: element-wise maximum unsigned: vd[i]= max(vs2[i],vs1[i])
vmaxu.vx
vmax.vv: element-wise maximum signed: vd[i]= max(vs2[i],vs1[i])
vmax.vx

12.9 Vector Single-Width Integer Multiply Instructions

These instructions return the same number of bits as the operands (i.e. SEW)
vmul.vv: returns lower SEW bits of element-wise multiply signed
vmul.vx
vmulh.vv: returns higher SEW bits of element-wise multiply signed
vmulh.vx
vmulhu.vv: returns higher SEW bits of element-wise multiply unsigned
vmulhu.vx
vmulhsu.vv: returns higher SEW bits of element-wise multiply signed-unsigned
vmulhsu.vx

12.10 Vector Integer Divide Instructions

vdivu.vv: vd[i]= vs2[i]/vs1[i] unsigned
vdivu.vx
vdiv.vv: vd[i]= vs2[i]/vs1[i] signed
vdiv.vx
vremu.vv: vd[i]= vs2[i] rem vs1[i] unsigned
vremu.vx
vrem.vv: vd[i]= vs2[i] rem vs1[i] signed
vrem.vx

12.15 Vector Integer Merge Instructions

vmerge.vvm: vd[i] = v0[i].LSB ? vs1[i] : vs2[i]
vmerge.vxm
vmerge.vim

12.6 Vector Integer Move Instructions

vmv.v.v
vmv.v.x
vmv.v.i

Appendix B: Project Description and Agreement Form

American University of Beirut

Department of Electrical and Computer Engineering

Project Description and Agreement Form

Final Year Project

Faculty Supervisor	**Dr. Mazen Saghir**
Co-Supervisor [optional]	N/A
Sponsor [optional] ***Is there industry support or funding the project?***	No
Project Title ***Descriptive title not necessarily the final title that will be adopted by the team***	A Configurable RISC-V Vector Accelerator for Machine Learning Applications
Project Description and Design Aspects ***What is the main motivation for the project?*** ***Specify the desired needs that the final product is expected to meet.***	With increased reliance on artificial intelligence and machine learning there is a growing need for efficient computing platforms to support emerging applications at suitable levels of performance and energy efficiency. Vector processing is a computing style that exploits high levels of data parallelism, making it well suited for efficiently accelerating machine learning algorithms used in video, image, and audio processing applications. The aim of this project is to design, implement, and evaluate a configurable RISC-V vector accelerator using a Xilinx FPGA device. The accelerator should communicate with a host processor using a suitable AXI4 interface. It should also be capable of transferring data between a vector register file and a DDR memory chip. Finally, the vector accelerator should be programmable in C using appropriate intrinsics. The latency and throughput of vectorized machine learning kernels will be measured using appropriate Xilinx Vivado tools and compared against non-vectorized software implementations.
Expected Deliverables ***Required deliverable(s) from the team at the conclusion of the design project***	1- A design-time confifurable VHDL model of the vector accelerator. 2- Support for a subset of instructions specified in ISA. Details can be found in appendix 3- A customized LLVM cross-compiler that optimizes code for the accelerator

	4- A functional hardware prototype implemented on a Nexys Video FPGA and capable of executing vectorized machine learning kernels with minor to no modifications. 5- A gem5 simulation model of the accelerator and comparison to existing solutions. Also benchmark design using MLperf.
Technical Constraints ***A preliminary list of multiple realistic technical constraints, e.g. power, accuracy, real-time operation , … The technical constraints included should be detailed and specific to the design project not generic.***	1- The accelerator should be compliant with the RISC-V Vector ISA. 2- The accelerator should be programmable in C using appropriate intrinsics. 3- The hardware prototype should operate at a clock rate of 150 MHz or better.
Non-Technical Constraints ***A preliminary list of multiple realistic non-technical constraints, e.g. cost, environmental friendliness, social acceptance , political, ethical, health and safety, etc… The non-technical constraints included should be detailed and specific to the project not generic.***	1- Availability of target FPGA and software programs in ECE labs. 2- Working from home due to COVID-19.
Contemporary Issues ***Cite one or more recent articles pertaining to the project or project area: news articles, blog discussions, academic articles, conference topics, etc.***	Machine learning hardware accelerators; vector architectures; RISC-V.
Resources and Engineering Tools ***Identify resources and engineering tools needed and whether they are available or need to be acquired (if known), e.g. software licenses, instruments, facilities, components, …***	1-Xilinx Vivado and SDK 2-RISC-V toolchain 3-gem5 simulator 4-Nexys Video FPGA . .
Possible Applicable Standards ***List potential standards directly or indirectly used or involved in the project***	1- RISC-V Vector Extension ISA 2- ARM AXI4 3- 1076-2019 - IEEE Standard for VHDL
List of Disciplines	-Hardware, Computer Architecture, and Digital Systems -Software Engineering

Identify at least THREE engineering disciplines (within or outside ECE)	-Machine Learning
Number of Students ***Please consider the number of disciplines checked above***	3 students
Required Courses [Optional] ***List the courses that are essential for the successful execution of the project (especially advanced courses)***	1- EECE 420 2- EECE 421 3- EECE 423 4- EECE 430
Date Submitted	September 4 2020

Appendix C: Meeting Minutes

Final Year Project Meeting Minutes

Meeting #1		
Date: 9.30.2020	Time: from 12:00 to 13:00	Location: Zoom
Meeting called by	Group	
Attendees	Imad Al Assir, Hadi El Sandid, Mohammad El Iskandarani, Dr. Mazen Saghir	
Minutes taker	Imad Al Assir	

Agenda Item: Description of FYP	
Discussion	**Scope of FYP**
Dr. Saghir went over the scope and the end-goal of the FYP.	
Conclusions	

- Design a RISC-V vector processor as per the RISC-V Vector Extension ISA.
- Integrate the processor into a RISC-V core (sweRV core), but first into MicroBlaze.
- Build hardware on Nexys Video FPGA
- Create a model on the GEM5 simulator to simulate the complete system with cache and memory in software. Can help debug hardware issues and compare to other existing solutions
- Ideally, would be able to run ML algorithms with none to few changes.
- Benchmark using MLPerf
- Low priority: Configure accelerators and generate VHDL code through a Python script to facilitate setup.

Action Items	**Person Responsible**	**Deadline**
Finalize and test hardware of RISC-V vector processor	Imad Al Assir, Mohammad El Iskandarani	End of Fall
Support evaluation and validation of the hardware by developing APIs for MicroBlaze in software	Hadi El Sandid, Imad Al Assir	End of Fall
Migrate from MicroBlaze to RISC-V sweRV core	Hadi El Sandid, Imad Al Assir	End of Spring
Model the vector processor in GEM5	Hadi El Sandid	End of Fall
Study how to run ML algorithms on vector processor	Mohammad El Iskandarani	End of Spring
Benchmark using MLPerf	Mohammad El Iskandarani	End of Spring
Low priority: Write a Python script to configure accelerators and generate VHDL code automatically.	Hadi El Sandid	End of Spring

Agenda Item: Choice of Processor	
Discussion	
Discussed which processor to use: MicroBlaze softcore processor vs RISC-V sweRV core	
Conclusions	

Final Year Project Meeting Minutes

- MicroBlaze:
  - Pros: accessible, have prior experience with it.
  - Cons: vector processor does not integrate very well with it. It will be a co-processor, not autonomous (e.g. able to access memory on its own) as it should be.
- RISC-V sweRV:
  - Pros: compatibility with RISC-V Vector Processor, better integration, can use native RISC-V compiler to vectorize code.
  - Cons: no prior experience, harder to get started

Action Items	Person Responsible	Deadline
Implement platform with MicroBlaze softcore processor	Hadi El Sandid, Imad Al Assir	End of Fall
Migrate to RISC-V sweRV	Hadi El Sandid, Imad Al Assir	End of Spring

Agenda Item: Tasks for the near future

Discussion	

Discussed where we should start, what to do in the near future

Conclusions	

- Hadi should familiarize himself with MicroBlaze and development for FPGA using Xilinx Vivado, by building a simple application. Imad can help with that since he has significant experience.
- Hadi should start looking into GEM5 and get it running on his machine.
- Since Imad and Mohammad had previously started the project in EECE 499, need to revisit previously faced problems and document them. Dr. Saghir will then help solve them.
- Imad, Mohammad and Dr. Saghir should meet again to review design choices made previously (e.g. number of lanes, scheduling mechanism, ...)

Action Items	Person Responsible	Deadline
Build simple MicroBlaze application on FPGA	Hadi El Sandid	By next meeting
Setup GEM5 on personal machine and get up to speed	Hadi El Sandid	In 2 weeks
Review and report problems faced previously	Imad Al Assir, Mohammad El Iskandarani	By next meeting
Review previous design choices	Imad Al Assir, Mohammad El Iskandarani	By next meeting

FYP Meeting Minutes

Meeting #:		
Date: [Pick the date]	Time: from 12-PM to 1-PM	Location: Zoom
Meeting called by	Dr. Mazen Saghir	
Attendees	Dr. Mazen Saghir – Imad Al Assir – Mohammad El Iskandarini – Hadi Rayan Al Sandid	
Minutes taker	Hadi Sandid	

Agenda Item: Addressing Issues related to the RISC-V Vector Processor Unit design	
Discussion	
We have addressed issues present in our Vector Processor Unit design.	
Conclusions	

- Discussed the scalability aspect of our design, and what design changes could be made to have better scalability.
- Addressed Issues concerning address generation and memory issues.
- Addressed issue concerning reservation stations and compiler constraints in our design.
- Updated Vector Processor design to support the latest version of RISC-V vector spec (v0.9).

Action Items	**Person Responsible**	**Deadline**
Address the rest of the design issues present in our Vector Processor before the next meeting.	Imad Assir & Mohammad Iskandarini	In the Next two Weeks

Agenda Item: Designing and Simulating Systems in Gem5	
Discussion	
We have discussed more about the Gem5 simulator, and how we should use it to simulate our target vector processor unit	
Conclusions	

- Hadi has been working with the Gem5 simulator over the last week. He has experimented with both the x86 and RISC-V ISAs in the simulator.
- Hadi has also learned to write basic Gem5 configuration scripts to both model hardware and initialize complete systems.
- Hadi should learn to model micro-architecture designs (i.e. co-processor…)

Action Items	**Person Responsible**	**Deadline**
Model a basic co-processor in Gem5	Hadi Sandid	Before Next Week
Write benchmark programs to test the basic co-processor design in Gem5	Hadi Sandid	Before Next Week
Read more about Gem5 simulation outputs, and how these can be used for development and debugging	Hadi Sandid	Before Next Week

Final Year Project Meeting Minutes

Meeting #:		
Date: 10.14.2020	Time: from 12-PM to 1-PM	Location: Zoom
Meeting called by	Dr. Mazen Saghir	
Attendees	Dr. Mazen Saghir – Imad Al Assir – Hadi Rayan Al Sandid	
Minutes taker	Hadi Sandid	

Agenda Item: Addressing Issues related to the RISC-V Vector Processor Unit design	
Discussion	
We have addressed issues present in our Vector Processor Unit design.	
Conclusions	
• Mohammad could not attend the meeting due to a conflict with his GRE exam. • Due to their midterms and the GRE, Imad and Mohammad could not address last week's tasks related to the Vector processor unit. They will address them for the next meeting. • Discussed some issues with the address generator unit of the RISC-V Vector Processor unit. • Discussed some issues related to the overall scalability of the RISC-V Vector Processor unit.	

Action Items	**Person Responsible**	**Deadline**
Address the rest of the design issues present in our Vector Processor before the next meeting.	Imad Assir & Mohammad Iskandarini	In the Next two Weeks

Agenda Item: Designing and Simulating Systems in Gem5	
Discussion	
We have discussed more about the Gem5 simulator, and what system should we aim to simulate	
Conclusions	
▪ Hadi continued experimenting with the different features of Gem5. He still needs to learn more about modeling hardware, and more specifically CPUs. A good exercise would be to simulate a system containing a host processor and a co-processor, which would communicate through memory-mapping ▪ The current systems we are simulating are using x86 binaries for simulation. We'd like to use RISC-V binaries starting next week, which requires us to get a RISC-V cross-compiler up and running.	

Action Items	**Person Responsible**	**Deadline**
Setup a RISC-V cross-compiler, and build RISC-V binaries	Hadi Sandid	Before Next Week
Design on Gem5 a system with a host-processor & co-processor	Hadi Sandid	Before Next Week

Final Year Project Meeting Minutes

Meeting #4		
Date: 23/10/2020	Time: from 12:00 to 13:00	Location: Zoom
Meeting called by	Group	
Attendees	Imad Al Assir, Hadi El Sandid, Mohammad El Iskandarani, Dr. Mazen Saghir	
Minutes taker	Imad Al Assir	

Agenda Item: Hardware updates

Discussion

Mohammad and Imad discussed their updates on the hardware design

Conclusions

- Mohammad and Imad explained their approach to the design scalability: making one large port whose size can change based on a generic. The port is then split into different signals and distributed to each component. This is necessary because, since it will be connected to the controller, it should match it. Dr. Saghir approved the design.
- Mohammad and Imad also mentioned that they will be working on the controller updates for next week. Dr. Saghir also approved.
- Dr. Saghir mentioned that he started working on bug fixes in the address generator masking logic but did not solve them completely yet. He said that he should finalize it by next week.

Action Items	**Person Responsible**	**Deadline**
Update controller design to enable scalability and control of multiple instructions in pipeline	Imad Al Assir, Mohammad El Iskandarani	Next week
Finalize address generator masking logic	Dr. Saghir	Next week

Agenda Item: Design and Simulation in GEM5

Discussion

Discussed the next steps we should take to simulate our RISC-V Vector Processor Unit in gem

Conclusions

- Hadi has modeled a basic system containing a host processor and an ALU co-processor, which communicate together through memory mapped I/O.
- There were some issues setting up the RISC-V cross-compiler to obtain proper RISC-V binaries. Hadi has agreed with Dr. Saghir to focus on getting a working installation of the RISC-V GNU Toolchain by next week.

Action Items	**Person Responsible**	**Deadline**
Setup the RISC-V GNU Toolchain, including the cross-compiler and simulator	Hadi El Sandid	Next week
Use the cross-compiler to obtain RISC-V binaries, and run them on a RISC-V compatible gem5 system	Hadi El Sandid	Next week

Final Year Project Meeting Minutes

Meeting #5		
Date: 29/10/2020	Time: from 12:30 to 13:30	Location: Zoom
Meeting called by	Group	
Attendees	Imad Al Assir, Hadi El Sandid, Mohammad El Iskandarani, Dr. Mazen Saghir	
Minutes taker	Imad Al Assir	

Agenda Item: Hardware updates

Discussion

Mohammad and Imad discussed their updates on the hardware design

Conclusions

- Mohammad and Imad presented a problem related to the address generator and register file: delay of 1 cycle between provided offset and actual value read at that offset. Dr. Saghir said that problem might be: inferred latch (no else statement or default value set), else might need to pipeline address generation and regfile (shorter critical path than including address generation in regfile).
- Dr. Saghir proposed adding a register that counts the number of cycles taken by an instruction (to measure performance because we cannot insert probe into our hardware design). Useful for later comparing to GEM5 simulation results and validating hardware.
- Dr. Saghir also said that to mark it as a milestone and avoid future surprises, when we connect the controller to other components of the design (ALU, register file, etc...), we should synthesize and test it on FPGA.

Action Items	**Person Responsible**	**Deadline**
Debug regfile+address gen	Imad Al Assir, Mohammad El Iskandarani	Next week
Update controller design to enable scalability and control of multiple instructions in pipeline	Imad Al Assir, Mohammad El Iskandarani	Next week
Finalize address generator masking logic	Dr. Saghir	Next week
Prepare AXI interface for vector processor and test it on FPGA with MicroBlaze	Imad Al Assir	In 2 weeks

Agenda Item: Simulation and compilers

Discussion

Hadi discussed progress done in simulators (GEM5 and SPIKE) as well as running the GCC compiler on his machine.

Conclusions

Final Year Project Meeting Minutes

- Hadi successfully ran the Spike simulator, and RISC-V cross-compiler. Dr. Saghir suggested that we use the Spike simulator later on as a reference to validate our hardware. Hadi needs to look at possible statistics given by the Spike simulator.
- Hadi looked at how to add intrinsics (pragmas) to cross-compiler. Tried to develop some basic ones. Still not very successful. Dr. Saghir clarified that intrinsics are used to give hints to compiler e.g. use intrinsics to tell compiler to unroll loop 10 times => Give compiler info that is not available at compile-time. Info that does not necessarily translate into code.
- Dr. Saghir asked if we can do assembly inlining, Hadi responded we can. Saghir suggested creating an Assembly function. Hadi suggested: create header file and functions implemented using inline assembly then call these functions from C code.
- Dr. Saghir said that we need to do a comparison between LLVM and GNU compilers to view which is better. GNU might be adequate but need to do our homework. Specifically, check support for vector instructions.
- Hadi showed output of running a simple multiply co-processor in GEM5. Dr. Saghir noticed a high tick count for simple multiplication code. Need to look into that because it is important to have correct performance measurements, since this is the goal of our vector processor.

Action Items	**Person Responsible**	**Deadline**
Finish original co-processor code and verify tick count	Hadi El Sandid	Next week
Compare LLVM and GNU	Hadi El Sandid	Next week

Final Year Project Meeting Minutes

Meeting #6		
Date: 04/11/2020	Time: from 12:30 to 13:30	Location: Zoom
Meeting called by	Group	
Attendees	Imad Al Assir, Hadi El Sandid, Mohammad El Iskandarani, Dr. Mazen Saghir	
Minutes taker	Imad Al Assir	

Agenda Item: Hardware updates

Discussion	

Mohammad and Imad discussed their updates on the hardware design

Conclusions	

- Mohammad and Imad first mentioned their solution to the address generator problem: flipping reading/writing edges in register file. Also they showed their simulation results to Dr. Saghir and he was satisfied.
- Dr. Saghir asked a few questions related to generics in the address generator design. Mohammad and Imad answered them and Dr. Saghir said that he should finish the address generator very soon.
- Mohammad and Imad mentioned that there were few undocumented changes in the v0.9 of the spec, so they updated the design to support them. Dr. Saghir requested that they go over the spec sheet again to make sure there are no other changes.
- Dr. Saghir reminded the team that they should synthesize the design and test it on FPGA to mark a milestone.

Action Items	**Person Responsible**	**Deadline**
Review spec v0.9 and make sure there are no unimplemented changes	Mohammad El Iskandarani, Imad Al Assir	Next week
Finalize address generator masking logic	Dr. Saghir	Next week
Prepare AXI interface for vector processor and test it on FPGA with MicroBlaze	Imad Al Assir	Next week

Agenda Item: Exploring different toolchains to generate RISC-V binaries

Discussion	

Dr. Saghir had proposed we investigate other options that the GNU/GCC toolchain to generate RISC-V binaries. He recommended we investigate the LLVM toolchain.

Conclusions	

Final Year Project Meeting Minutes

- Hadi presented the advantages of LLVM over GNU: LLVM uses modern ideas and is easier to use than GNU. However, to run Linux, he said that GNU is still required. Also, to generate RISC-V binaries, GNU (RISC-V toolchain) is required.
- Dr. Saghir mentioned that Hadi needs to look at the Mlperf benchmark suite, and research how to compile it using LLVM. He also requested that Hadi looks at APIs for LLVM and gets a running example for next week.
- Concerning compilation, Dr. Saghir mentioned that we do not necessarily need automatic code vectorization at this point; we just want the code to run. Therefore, we should use Assembly inlining to test our hardware.
- Dr. Saghir recommended we start looking at the support for RISC-V vector instructions in the current release of the LLVM compiler

Action Items	**Person Responsible**	**Deadline**
Become familiar with the LLVM/Clang compiler toolchain	Hadi El Sandid	Next week
Look into LLVM/Clang support for RISC-V vector instructions	Hadi El Sandid	Next week

Final Year Project Meeting Minutes

Meeting #8		
Date: 11/18/2020	Time: from 12:30 to 13:30	Location: Zoom

Meeting called by	Group
Attendees	Imad Al Assir, Hadi El Sandid, Mohammad El Iskandarani, Dr. Mazen Saghir
Minutes taker	Imad Al Assir

Agenda Item: Hardware updates

Discussion	

Mohammad and Imad discussed their updates on the hardware design

Conclusions	

- The newInst hot one encoding worked, but the same procedure couldn't be performed on the other control signals
- The offset generator code by Dr. Saghir worked, but Mohamad and Imad need to support the writing offset
- Dr. Saghir provided a great solution for the writing offset later that night by mail, which his to ripple the reading offsets through the pipeline registers and utilize them as writing offsets when they reach the Write ports of the Register file.

Action Items	**Person Responsible**	**Deadline**
Work on Writing Offset ripple, and debug controller signals	Mohamad El Iskandarani, Imad Al Assir	Next week

Agenda Item: Researching more about the MLPerf Benchmark Suite

Discussion	

Dr. Saghir asked us to investigate the MLPerf Benchmark suite, to assess if we could use it to test both our gem5 model and an actual hardware implementation

Conclusions	

- We conducted a brief overview of the ML Training and Inference tests offered by MLPerf.
- More details of the ML Inference tests were discussed : They are separated into three sets, each targeting a different type of device (i.e. datacenters, edge-systems, mobile devices)
- We have talked about the recommended implementation of MLPerf in Python, and how we could port it for use with our gem5 model/hardware implementation

Action Items	**Person Responsible**	**Deadline**
Prepare a comparative table of all ML Inference tests available in the MLPerf benchmark suite	Hadi Sandid	Before Next Week
Work on obtaining a C/C++ version of the MLPerf codebase, so we could test it on our gem5 model/hardware implementation	Hadi Sandid	Before Next Week

Final Year Project Meeting Minutes

Meeting #9		
Date: 11/25/2020	Time: from 12:30 to 13:30	Location: Zoom

Meeting called by	Group
Attendees	Imad Al Assir, Hadi El Sandid, Mohammad El Iskandarani, Dr. Mazen Saghir
Minutes taker	Imad Al Assir

Agenda Item: Hardware updates

Discussion	

Mohammad and Imad discussed their updates on the hardware design

Conclusions	

- Mohamad and Imad discussed their revised generics
- Mohamad and Imad discussed design decisions regarding MLEN (in the offset generator)
- Mohamad, Imad, and Dr. Saghir discussed the plans for the Winter break and Spring Semester, mainly implementing the design on an FPGA, using a swerV core, and migrating the design to a RISC-V core

Action Items	**Person Responsible**	**Deadline**
Finish Final report and presentation hardware content	Mohamad El Iskandarani, Imad Al Assir	Next week

Agenda Item: Porting MLPerf Inference tests to C

Discussion	

Hadi must work on porting the reference implementation of the MLPerf Inference benchmark tests to C/C++, so we are able to cross-compile these tests for a RISC-V target using LLVM/Clang.

Conclusions	

- Python Reference implementation of the MLPerf Benchmark Suite is available online. We would like to find an appropriate tool to port it to C/C++.
- One of tools we have proposed using is Cython, an optimizing static compiler that allows us to generate C code from a Python codebase. Hadi has tried to use Cython but ran into time constraints this week (i.e. exams) which did not allow him to test the tool properly.

Action Items	**Person Responsible**	**Deadline**
Use Cython to import tests from the MLPerf reference implementation	Hadi Sandid	During the Winter break
Explore other methods to port the MLPerf reference implementation	Hadi Sandid	During the Winter break