Design and Implementation of a Massively Multi-threaded RISC-V Processor

Georgios-Michail Matzouranis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor:
Assistant Prof. *Vassilis D. Papaefstathiou*

This work has been performed at and supported by the Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH).

University of Crete
Computer Science Department

Design and Implementation of a Massively Multi-threaded RISC-V Processor

Thesis submitted by
Georgios-Michail Matzouranis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Georgios-Michail Matzouranis

Committee approvals: ______________________
Vassilis D. Papaefstathiou
Assistant Professor, Thesis Advisor

____
Manolis G.H. Katevenis
Professor, Committee Member

____
Polyvios Pratikakis
Associate Professor, Committee Member

Departmental approval: ______________________
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, July 2022

Design and Implementation of a Massively Multi-threaded RISC-V Processor

Abstract

The computational requirements of modern applications have out-scaled the typical capabilities that central-processing units (CPU) can provide. One of the reasons is the massive amount of data that needs to be processed in contrast to the processing power available, as well as the number of tasks and different contexts that the main CPU has to manage. Another reason is the technological obstacles, such as the power consumption needed to reach ambitious computational performance levels.

In order to address the later issues, the systems nowadays employ different types of specialized accelerators for various application domains, such as graphical processing units (GPUs), tensor processing units (TPUs), and reconfigurable hardware accelerators implemented in FPGAs. The accelerators are programmed, accessed, and used through different programming models and toolchains that try to maximize the utilization of the available resources. Currently some of the most popular programming models for accelerators are OpenCL and CUDA, where the main program runs on a host CPU and spawns computational kernels that are off-loaded for execution on the accelerator devices, such as GPUs. For several categories of applications, this results in substantial performance gains and energy savings, since the most important parts of the program are executed in the specialized and optimized accelerator devices.

This thesis contributes with the design and implementation of a massively multi-threaded, in-order superscalar core, Matzic , that acts as an accelerator capable of supporting OpenCL-like and CUDA-like programming models. Matzic uses the open RISC-V instruction set architecture (ISA) that is becoming increasingly popular in the recent years. The core maintains contexts for up-to 256 threads, can issue up-to 4 independent instructions per thread in each cycle and contains 7 execution clusters with different types of execution units. The core can also issue up-to 512 outstanding memory operations.

We implement and verify Matzic using SystemVerilog RTL and evaluate performance via RTL simulation and bare metal code that is compiled using the GNU RISC-V toolchain. Furthermore, we evaluate the resource utilization of the design on a Xilinx Kintex Ultrascale FPGA. Finally, we test and verify Matzic on an FPGA design that contains an CVA6 (Ariane) RISC-V CPU, that works as the main processor and runs the Linux operating system (OS), 2GBytes of DDR3 DRAM for main memory, our Matzic core and an additional memory delayer which is used to study various DRAM access latencies.

Σχεδίαση και Υλοποίηση Μαζικά Πολυνηματικού Επεξεργαστή RISC - V

Περίληψη

Οι υπολογιστικές απαιτήσεις των σύγχρονων εφαρμογών έχουν ξεπεράσει τις τυπικές δυνατότητες που μπορούν να παρέχουν οι μονάδες κεντρικής επεξεργασίας (CPU). Ένας από τους λόγους είναι ο τεράστιος όγκος δεδομένων που πρέπει να υποβληθούν σε επεξεργασία σε αντίθεση με τη διαθέσιμη επεξεργαστική ισχύ, καθώς και τον αριθμό των εργασιών και των διαφορετικών διεργασιών που πρέπει να διαχειριστεί η κύρια CPU. Ένας άλλος λόγος είναι τα τεχνολογικά εμπόδια, όπως η κατανάλωση ενέργειας που απαιτείται για την επίτευξη φιλόδοξων επιπέδων υπολογιστικής απόδοσης.

Προκειμένου να αντιμετωπιστούν τα προηγούμενα ζητήματα, τα συστήματα σήμερα χρησιμοποιούν διαφορετικούς τύπους εξειδικευμένων επιταχυντών για διάφορους τομείς εφαρμογών, όπως μονάδες γραφικής επεξεργασίας (GPU), μονάδες επεξεργασίας τανυστών (TPU) και επιταχυντές υλικού με δυνατότητα επαναδιαμόρφωσης που υλοποιούνται σε FPGA. Οι επιταχυντές προγραμματίζονται, προσπελάζονται και χρησιμοποιούνται μέσω διαφορετικών μοντέλων προγραμματισμού και αλυσίδων εργαλείων που προσπαθούν να μεγιστοποιήσουν τη χρήση των διαθέσιμων πόρων. Επί του παρόντος, μερικά από τα πιο δημοφιλή μοντέλα προγραμματισμού για επιταχυντές είναι το OpenCL και το CUDA, όπου το κύριο πρόγραμμα εκτελείται σε μια κεντρική CPU και δημιουργεί υπολογιστικούς πυρήνες που φορτώνονται για εκτέλεση στις συσκευές επιτάχυνσης, όπως οι GPU. Για πολλές κατηγορίες εφαρμογών, αυτό έχει ως αποτέλεσμα σημαντικά κέρδη απόδοσης και εξοικονόμησης ενέργειας, αφού τα πιο σημαντικά μέρη του προγράμματος εκτελούνται στις εξειδικευμένες και βελτιστοποιημένες συσκευές επιτάχυνσης.

Αυτή η εργασία συμβάλλει στη σχεδίαση και την υλοποίηση ενός μαζικά πολυνηματικού, υπερκλιμακωτού πυρήνα, τον Matzic , που λειτουργεί ως επιταχυντής ικανός να υποστηρίζει μοντέλα προγραμματισμού τύπου OpenCL και CUDA. Ο Matzic χρησιμοποιεί την ανοιχτή αρχιτεκτονική συνόλου εντολών (ISA) RISC-V που γίνεται όλο και πιο δημοφιλής τα τελευταία χρόνια. Ο πυρήνας διατηρεί περιεχόμενο κατάστασης για έως και 256 νήματα, μπορεί να εκδώσει έως και 4 ανεξάρτητες εντολές ανά νήμα σε κάθε κύκλο και περιέχει 7 συμπλέγματα εκτέλεσης με διαφορετικούς τύπους μονάδων εκτέλεσης. Ο πυρήνας μπορεί επίσης να εκδώσει έως και 512 εκκρεμείς λειτουργίες μνήμης.

Υλοποιούμε και επαληθεύουμε τον Matzic χρησιμοποιώντας SystemVerilog επιπέδου μεταφοράς καταχωρητών (RTL) και αξιολογούμε την απόδοση μέσω προσομοίωσης RTL και κώδικα που μεταγλωττίζεται χρησιμοποιώντας την αλυσίδα εργαλείων GNU RISC-V. Επιπλέον, αξιολογούμε τη χρήση πόρων του σχεδίου πάνω σε μία Xilinx Kintex Ultrascale FPGA. Τέλος, δοκιμάζουμε και επαληθεύουμε τον Matzic σε ένα σχέδιο για FPGA που περιέχει μια CVA6 (Ariane) RISC-V CPU, που λειτουργεί ως ο κύριος επεξεργαστής και εκτελεί το λειτουργικό σύστημα (OS) Linux, 2 GByte DDR3 DRAM για κύρια μνήμη, τον πυρήνα μας Matzic και μια πρόσθετη μονάδα

καθυστέρησης μνήμης που χρησιμοποιείται για τη μελέτη διαφόρων καθυστερήσεων πρόσβασης στη DRAM.

Acknowledgments

The work for this thesis was performed at the Computer Architecture and VLSI (CARV) Laboratory of the Institute of Computer Science (ICS) at the Foundation for Research and Technology Hellas (FORTH), from June 2020 to June 2022.

Firstly, I would like to thank my advisor, Professor Vassilis Papaefstathiou, for supporting, challenging and inspiring me in the duration of this thesis and the master's studies to continuously improve and gather more knowledge.

Secondly, I would like to express my gratitude to Professor Manolis Katevenis, for the insights he has provided me and the knowledge even from my undergraduate studies.

Thirdly, I would like to thank Giorgos Ieronimakis, Vassilis Flouris, Nikolaos Dimou and Theocharis Vavouris for the support, technical expertise and technical know-how mainly on the evaluation part of this thesis.

Furthermore, I would like to express my gratitude, appreciation and thanks to my family for all the years of support and encouragement they provided me, especially during the years of the coronavirus pandemic.

Moreover, I would like to thank my friends for their support especially in the duration of my university life.

Additionally, I would like to thank the members of the Institute of Computer Science, Foundation for Research and Technology - Hellas, Computer Architecture and VLSI Systems Laboratory (CARV) for always providing me with the equipment I needed and giving me an environment that inspired me to work more efficiently.

Lastly, I would like to thank the professors and members of the University of Crete, Computer Science department for their support and help during my studies.

Finally, I apologize if I forgot to explicitly thank someone I should have for supporting me in the pursuit of a MSc title.

dedicated to my family,
for their support through the years of the pandemic

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern applications have ever increasing computational demands and the problem sets are becoming substantially larger and more complex. For this reason, the typical capabilities and processing throughput offered by main central-processing units (CPU)s, as well as the performance scaling that is expected in the coming years, are being out-scaled[7] by the computational requirements of the applications. Another challenging problem that main CPUs are facing, apart from the need to provide more computational power, is that they are required to perform not only the computational work but also the administrative work to handle many different contexts and tasks – this often results in performance losses for the applications. Moreover, due to the limitations of the CPUs and the modern technological obstacles that arise, in order to reach ambitious computational performance levels they end up consuming increasingly higher power consumption, resulting in tremendous energy costs compared to the performance gains.

To address the latter issues, the computer architects are moving towards increasingly heterogeneous system architectures. The systems nowadays tend to contain and employ not only CPUs but also different types of specialized accelerators. Various application domains need the functionality offered by accelerators, such as floating-point intensive computations and machine learning (ML) tasks. Some of the popular accelerators used nowadays are:

- Graphical processing units (GPU)s. The GPUs provide large number of computational units, that are used for different floating-point arithmetic and ML tasks.
- Tensor processing units (TPU)s that are used to mainly accelerate ML and AI specific tasks.
- and configurable hardware accelerators that are implemented in FPGAs and their applicability is more general, due to the programmable nature of the FPGA.

Programming, access and use of the accelerators happens through different programming models and toolchains that try to maximize the utilization of the resources that accelerators provide. Currently, the most popular programming models and toolchains for accelerators are OpenCL[10] and CUDA[9]. The way of operation of those two programming models is to have the main program run on a host CPU and then spawn different computational kernels that are off-loaded for execution on the accelerator devices. The main program is responsible for managing the concurrent tasks and data movement such as: allocating memory, initializing data and transferring data between itself and the accelerator device(s). The task management activities are performed well by the CPU, as they don't require intensive computations that can lead to performance loss if the CPU is interrupted. The kernels that are off-loaded on the accelerators for execution contain mostly computationally heavy tasks that take advantage of the resources provided by the accelerators. For several classes of applications, the use of accelerators results in substantial performance gains and energy savings, since the most important parts of the programs are executed in the specialized and optimized accelerator units.

The rest of this chapter highlights the contributions of this thesis and presents a brief overview of the organization of this document.

1.1 Contributions

This thesis contributes with the design and implementation of a massively multi-threaded superscalar RISC-V core, Matzic , on real hardware. The implementation of our processor intends to be a basis for research studies on massively multi-threaded processors. This work also intends to further explore the capabilities of the open RISC-V ISA[2] and experiment with additional use cases. In summary, this thesis makes the following contributions:

- The micro-architecture for a massively multi-threaded superscalar RISC-V processor (Matzic).
- The implementation of Matzic 's design using synthesizable SystemVerilog[5] RTL and testing in simulation for compliance with the RISC-V ISA specification using the official RISC-V compliance test suite.
- An AXI4-Lite interface for the communication and control of Matzic .
- The detailed evaluation of the FPGA resource utilization of Matzic .
- Testing of Matzic on real hardware on a Xilinx FPGA together with the CVA6 (Ariane) [12] RISC-V CPU that runs a Linux kernel.
- Accelerating applications using Matzic as an accelerator capable of supporting OpenCL-like and CUDA-like programming models.

1.2 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 presents the background and basic concepts that are used by the design of this thesis.
- Chapter 3 presents the details of the design and implementation of the proposed RISC-V core, Matzic .
- Chapter 4 contains the evaluation of the design using both software and hardware, as well as a description of the environment and tools used.
- Finally, Chapter 5 concludes the thesis and presents perspectives for future work.

Chapter 2

Background

This chapter covers some of the background about RISC-V[2], which is the instruction set architecture (ISA) of our core, and some basics about processor multi-threading, AXI4 and AXI4-Lite protocols. Further technical details can be found in the original source for RISC-V [2], regarding multi-threading in [11] and about the AXI protocols in [1]. Moreover, we also present some related work relevant to Matzic . The structure of this chapter is as follows:

- RISC-V ISA
- Multi-Threading
- AXI Protocols

2.1 RISC-V

The name RISC-V (called "risk five") refers to the fifth major RISC instruction-set architecture (ISA) design from UC Berkeley (RISC-I, RISC-II, SOAR, and SPUR were the first four). The RISC-V ISA was originally designed to support computer architecture research and education. More specifically, some of the primary goals of the RISC-V design are:

- To be an open ISA that is freely available to both academia and industry.
- To be suitable for native hardware implementation.
- Not being specific for particular micro-architectures or implementations, but can be implemented in any of these.
- To have support for the revised 2008 IEEE-754 floating point standard.
- To support ISA extensions and specialized variants.
- Support for 32-bit and 64-bit address spaces for applications, operating system kernels, and hardware implementations.

- Support for highly-parallel multi-core or many-core implementations, including heterogeneous multiprocessors.

There are also more design goals of this ISA, as well as, extensions to the base integer ISA, including, custom extensions that we use in this thesis and will describe after describing the RV32I integer base ISA.

2.1.1 The RISC-V 32I ISA

The RISC-V 32I (RV32I) ISA is designed sufficiently enough to be a compiler target and support modern operating system environments, as well as, reducing required hardware for a minimal implementation. Also, RV32I can be used to emulate the other extensions, except the A extension which requires additional hardware support for atomic operations.

The RV64I architecture has 32 registers with width of 64 bits, from those register 0 is hardwired to equal 0 and the other 31 registers are general purpose registers. In addition to those 32 registers there is also the pc register that holds the address of the current instruction.

For the instructions of the ISA, there are four base formats(R/I/S/U), that can be seen in Figure 2.1. Every instruction is aligned to a four-byte boundary in memory and if the address is not aligned then an instruction misaligned exception is generated. For conditional jumps generate an exception only if the branch is successful.

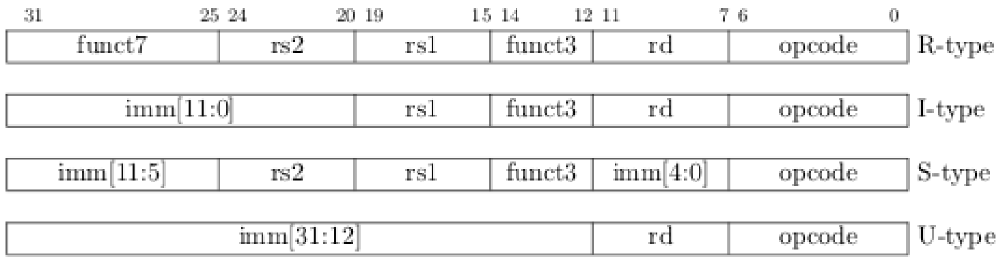


Figure 2.1: RV32 I R, I, S, U formats. source:[2]

As Figure 2.1 shows, the destination register and source registers address field remain at fixed positions, as does the position of funct3, in the instruction. This convention simplifies the decoding process. The only part that changes between the different formats is the bits of the immediate value, as Figure 2.2 shows. A sign-extension to 32-bits is always performed to the generated immediate value, by replicating bit 31 of the instruction, thus allowing for a faster immediate generation. In Figure 2.1 there are also two more formats, in addition to the previous four. The two new formats (B/J) differ from the S and U formats in the position of the immediate bits and also that the generated immediate value is a multiple of

2. Figure 2.3 shows the immediate value each instruction format generates from the instruction bits it has.

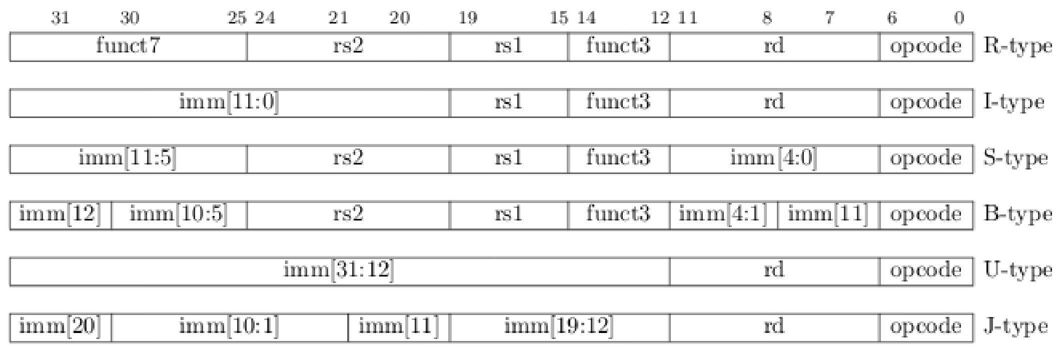

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 2.2: RV32I instruction formats with immediate positions. source: [2]

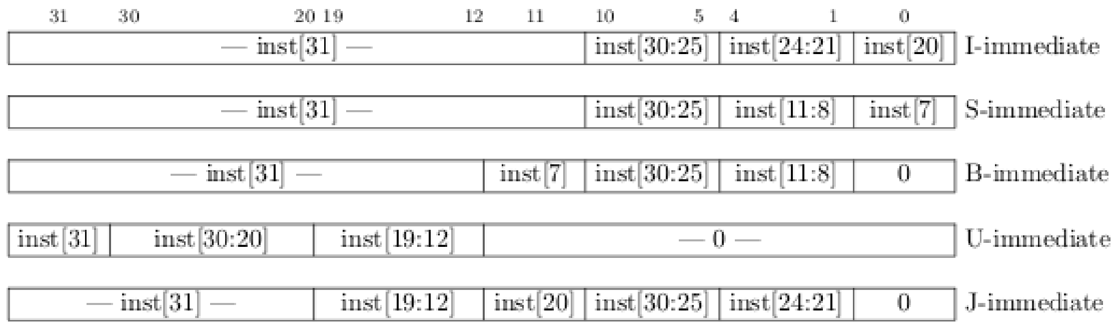

31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

Figure 2.3: Generated immediates by each format. source: [2]

2.1.2 RISC-V 64I

The RISC-V 64I extension of the RISC-V ISA builds upon the RISC-V 32I instruction set. One of the extensions it adds is to widen the registers from 32-bits to 64-bits. It also widens the user address space that is supported to 64-bits. All the instructions that the 32I base extension defines, now operate on 64-bit registers and immediates. This also means that immediates now sign-extend to 64-bits instead of 32-bits that they were before.

Since all the instructions of the 32I base extension now operate between 64-bit immediates and registers, the 64I extension introduces additional instructions that operate between 32-bit values. The result of those instructions are sign-extend to 64-bit to be compatible with the base registers of the 64I extension. The extension also adds load and store instructions to write and read atomically 64-bit from the memory, as well as a load operation where it loads a 32-bit value from memory without sign-extending it.

2.1.3 RISC-V 64E

The RISC-V 64E extension reduces the number of available registers from 32 to 16. More specifically, it only permits the use of the registers $x0 - x15$. The extension reserves the rest of the registers ($x16 - x31$). In all other aspects, this extension has the same instructions as the 64I extension.

2.1.4 RISC-V M

The M extension of the RISC-V ISA is used to add standard integer multiplication and division capability. If it is used together with the 64I extension it also adds multiplication and division instructions for 32-bit integer values. The result of the 32-bit instructions of course is sign-extended to 64-bit.

The multiplication instructions the extension adds, also allow to get the upper bits of the result, which would overflow when the result of the multiplication doesn't fit in 64-bits. Those multiplication instructions take into consideration the sing of the operands when the upper bits, of the multiplication, are requested, as it can result in different numbers than the ones intended for multiplication.

The division instructions provided are for the remainder of a division and not only the quotient of the division. The division instructions also provide a choice for the sign of the operands the operation will use.

2.1.5 RISC-V F and D

The F extension of the RISC-V ISA adds support for floating point operations. It also adds 32 additional registers, the $f0 - f31$ registers, to the ones of the base 32I ISA, which are exclusive for floating point operations. Those registers are 32-bits wide. The extension mostly has instructions that operate between floating point registers, but it also provides instructions that load and store floating point values to memory, as well as, capabilities to transfer and convert data between the integer and floating registers.

While for the load and store instructions the extension uses the I and S instruction format, it generally uses the R format for all others where it further defines things as Figure 2.4 shows. In the Figure the funct3 field of the R-format

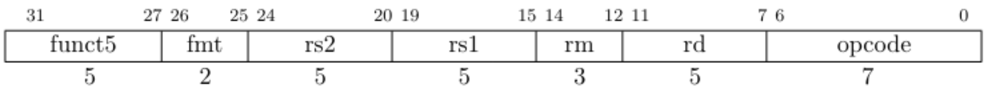


Figure 2.4: Instruction format for float operations. source: [2]

changes to rm which is the type of rounding mode to use when there is a need for rounding, the operands, or the result, for the operation that occurs. The funct7 part of the instruction also divides to funct5 and fmt. While funct5 serves the same purpose as funct7, the fmt part of the instruction indicates the format of the floating-point values the operation uses. In the F extension, fmt is always set to indicate that the operands and result are 32-bit single-precision floating point numbers.

The F extension also adds fused multiply instructions, that require 3 source operands as it does 2 operations at the same time. The first operation is to multiply the first 2 operands and then add the third operand to the result of the multiplication. For those instructions the F extension adds a new type of instruction format, the R4-type format, that Figure 2.5 shows. This instruction

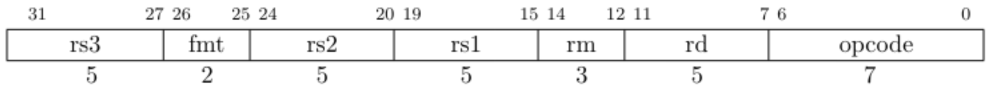


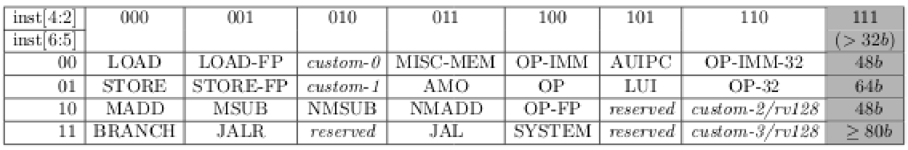
Figure 2.5: R4 instruction format. source: [2]

format uses the 5-bit wide func5 field that is seen in Figure 2.4 as the 3^{rd} source operand register. All the other aspects of the instruction remain the same.

The D extension works similarly to the 64I extension for the 32I base ISA. The extension widens the float registers to 64-bits, as well as introduces the value for double-precision 64-bit floats to the fmt format field where it is used to define the size of the operands. Moreover when a 32-bit float needs to be represented in the 64-bit float registers then the upper 32 bits must all be set to 1. Finally the D extension also adds instructions for converting floating values between different floating point formats.

2.1.6 The RISC-V Custom Opcode and Instructions

The RISC-V ISA defines the opcodes in a way that allows for custom ISA extensions to be made. In Figure 2.6 we see how the ISA defines the opcodes and how it leaves opcodes open for custom instructions to be implemented. In this thesis we use the custom-0 opcode, which equals 0001011, to implement a custom barrier and get thread id instructions that the programs can use for synchronization and other reasons when the processor runs them.



inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

Figure 2.6: RISC-V ISA opcode definition table. source:[2]

2.2 Multi-threading

There is a limit to the instruction level parallelism (ILP) in conventional instruction streams. That means there is also a limit to the instructions we can have in a typical pipeline. A multi-threaded processor tries to solve this by concurrently executing instructions of different threads of control within a single pipeline. Because the threads of control are different that means the instructions between those threads are also independent and can be executed concurrently. Through the different architectural approaches, multi-threading can either:

- increase the performance of a single program by implicitly utilizing more coarse-grained parallelism than ILP. We call that implicit multi-threading.
- increase the performance of workload that is either multiprogramming or multi threaded. We call that explicit multi-threading.

A thread for a multi-threaded processor is different from the software threads of multi-threaded operating systems. We always view a thread of a multi-threaded processor as a hardware-supported thread which, depending on the specific form of multithreaded processor, can be a full program (single-threaded UNIX process), a light-weight process (e.g. a POSIX thread) or a compiler- or hardware-generated thread (subordinate microthread, microthread, nanothread, etc.). The consequences for designing a multi-threaded processor are:

- In case the processor wants to execute multiple processes in parallel, it needs to maintain different address spaces for each instruction stream in execution.
- In case of executing multiple threads from a single application, usually that implies that the threads can have a common address space. Which means we can have threads share some structures, such as caches or even registers. That also depends on the application.

Therefore we define a multi-threaded processor to be able to interleave the execution of instructions of different threads of control in the same pipeline, as well as have multiple program counters available in the fetch unit and store multiple contexts in different register sets on the chip. Its execution units also multiplex between thread contexts that are loaded in the register sets. As a result, the latency that may arise in the computation of a single instruction stream is filled by computations of another thread. In that way, multi-threaded processors can tolerate memory latencies by overlapping the long-latency operations of one thread with the execution of other threads.

2.2.1 Implicit Multi-threaded Processors

The term implicit multi-threaded architecture refers to any architecture that can concurrently execute several threads from a single sequential program. The threads may be obtained with or without the help of the compiler. To get threads a higher than normal degree of speculation is used in combination with functional partitioning of the processor, which allows finding contiguous regions of the static or dynamic instruction sequence. So a thread in such architectures refers to those instruction sequences.

2.2.2 Explicit Multi-threading

The multi-threaded processors that are designed to simultaneously execute threads of the same or of different processes, are called explicit multi-threaded processors in contrast to the implicit multi-threaded processors mentioned above. The Explicit multi-threaded processors can to increase the performance of a multi-programming or multi-threaded workload. However, the single-thread performance may slightly decrease when compared to a single-threaded processor. Notice that explicit multi-threaded processors aim at a low execution time of a multi-threaded workload, while implicit multi-threaded processors aim at a low execution time of a single program.

2.2.3 Multi-threaded Approaches

The minimal requirement a multi-threaded processor should have is the ability to pursue two or more threads of control in parallel within the processor pipeline and a mechanism that triggers a thread switch. With that we have the following main approaches to multi-threading processors:

- The Interleaved multi-threading technique, which is to fetch the instruction of another thread and feed it into the execution pipeline at each processor cycle.

- The Blocked multi-threading technique, which is to execute the instructions of a thread successively until an event occurs that may cause long latency. When the event occurs, it induces a context switch.

- The Simultaneous multi-threading technique, which combines the wide superscalar instruction issue with the multiple-context approach. Therefore, it simultaneously issues instructions from multiple threads to the execution units of a superscalar processor, in-order to utilize as many of them as possible.

Figure 2.7 shows how the techniques of interleaved multi-threading (b) and blocked multi-threading (c) work by showing how the traditional pipeline (a) would work. It also shows how a superscalar pipeline (d) works when it implements the interleaved multi-threading (e) and blocked multi-threading (f) techniques as well. The A, B, C and D are the contexts of different instruction streams that can be viewed as threads.

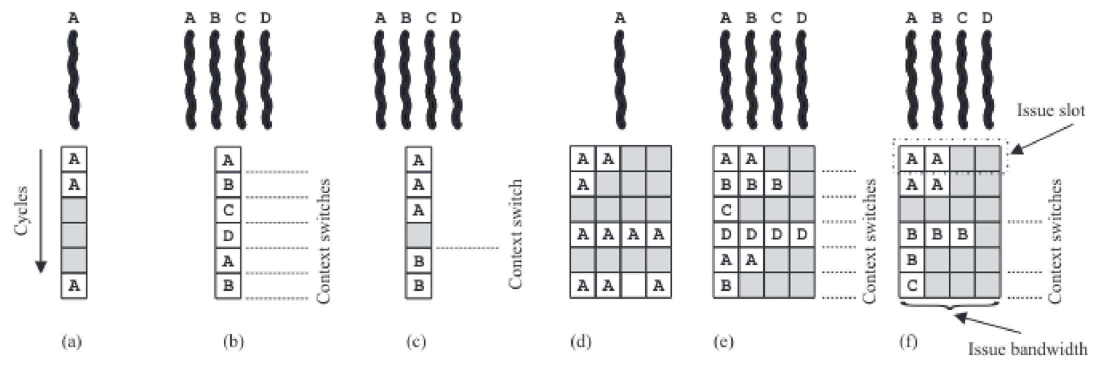


Figure 2.7: Different approaches possible with scalar and superscalar processors: (a) single-threaded scalar; (b) interleaved multi-threading scalar; (c) blocked multi-threading scalar; (d) single-threaded superscalar; (e) interleaved multi-threading superscalar; (f)blocked multi-threading superscalar. source [11]

2.2.3.1 Interleaved Multi-threading

The interleaved multi-threading model, also called as fine-grain multi-threading, allows the processor to switch to a different thread after each instruction fetch. In principle, we feed an instruction of a thread into the pipeline after the retirement of the previous instruction of that thread. Since interleaved multi-threading eliminates control and data dependencies between instructions in the pipeline, the known pipeline hazards don't arise and the processor pipeline is easily built without the complex forwarding paths. Because no pipeline hazards can occur and we don't calculate forwarding, this leads to a very simple and therefore potentially very fast pipeline. Moreover, the context-switching overhead is zero cycles. Memory latency is tolerated by not scheduling a thread until the memory transaction has been completed. This model requires at least as many threads as pipeline stages in the processor. Interleaving the instructions from many threads limits the processing power accessible to a single thread, thereby it degrades the single-thread performance.

2.2.3.2 Blocked Multi-threading

The approach of blocked multi-threading, which is also called coarse-grain multi-threading, executes a single thread until it reaches a situation that triggers a context switch to another thread. Usually, a situation for the context switch appears when the instruction execution reaches a long-latency operation or a situation where latency may appear. An example of such a situation is when a memory operation is executed or an interrupt arrives at the core. Compared to the interleaved multi-threading technique, a smaller number of threads is needed and a single thread can execute at full speed until the next context switch. The single-threaded performance of this scheme is similar to the performance of a comparable processor without multi-threading.

2.2.3.3 Simultaneous Multi-Threading

The approach of simultaneous multi-threading (SMT) combines the wide superscalar instruction issue with the multi-threading approach by providing several register sets on the processor and issuing instructions from threads simultaneously. Therefore, operations of several threads can fill the issue slots of the wide-issue processor. The latencies occurring in the execution of a single thread are bridged by issuing operations of the other remaining threads loaded on the processor. In principle, a thread combination of instructions can utilize the full issue bandwidth. The SMT fetch unit can take advantage of the interthread competition for instruction bandwidth in two ways

1. First, it can partition the instruction bandwidth among the threads and fetch from several threads each cycle. In this way, it increases the probability of fetching only non-speculative instructions.

2. Second, the fetch unit can be selective about which threads it fetches. For example, it may fetch those that will provide the most immediate performance benefit.

Because an SMT processor exploits both coarse- and fine-grained parallelism, it uses its resources more efficiently and thus achieves better throughput and speedup than single-threaded superscalar processors for multi-threaded workloads. The trade-off is a more complex hardware organization.

2.3 AXI4 and AXI4-Lite

The AXI protocols design is to support high-performance and high-frequency system designs [1]. More accurately the AXI protocol:

- is suitable for high-bandwidth and low-latency designs.

- provides high-frequency operation without using complex bridges.

- meets the interface requirements of a wide range of components.

The way it accomplishes the design goals is through the key features the protocol provides. Those are:

- to have separate phases for the control/address of a transaction and the data of the transaction.

- to have separate read and write channels, that provide low-cost direct memory access.

- the ability to issue multiple outstanding transactions and also have support for out-of-order completion of those transactions.

Those are some of the futures the protocol provides that made us consider and decide to use it in the implementation of our thesis for more information you can read the source material[1].

2.3.1 AXI4

The AXI4 is a master-slave protocol that has five different transaction channels. The transaction channels it has are:

- The read address (AR) channel. This channel is responsible for sending the address of a read operation as well as different control signals from the master to the slave that will handle the request.
- The read data (R) channel. This channel sends from the slave to the master the data corresponding to the address the master provides from the AR channel. The R channel also generates a response for the validity of the address and control combination the master provides. In case of an invalid combination, an error response returns to the master alongside the data.
- The write address (AW) channel. This channel does the same job as the AR channel but for write transactions. So it provides the address and control for a write transaction to the slave from the master.
- The write data (W) channel. This channel is responsible for sending the data for a write transaction that started or will start from the AW channel to the slave interface.
- The write response (B) channel. This channel's responsibility is to notify the master about the success or failure of an issued Write request. If the action of the request results in unsuccessfulness, an error response generates and reaches the master.

Each of the above channels is independent and consists of a set of information signals, as well as a $VALID$ and $READY$ signal pair to provide a two-way handshake between master and slave. The information source asserts the $VALID$ signal to show that it is issuing a valid address, data, or control on the channel. The destination asserts the $READY$ signal to show that it is available to receive the information from the source. A $VALID$ signal must not depend on the $READY$ signal to avoid possible deadlocks. It is possible to assert the $READY$ signal before the $VALID$ signal, to show the readiness of the destination to receive the information. The W and AR channels that transfer data also include a $LAST$ signal that signals the dispatch of the final data of a transaction.

Taking advantage of the different independent channels, the AXI protocol:

- permits the issue of the address and information ahead of the actual data transfer.
- allows for multiple outstanding transactions to happen.
- allows for out-of-order transaction completion (the completion of the transactions can be in a different order from the one they were issued).

2.3.1.1 AXI read transaction

A single read transaction uses the AR and R channels in the way that Figure 2.8 shows.

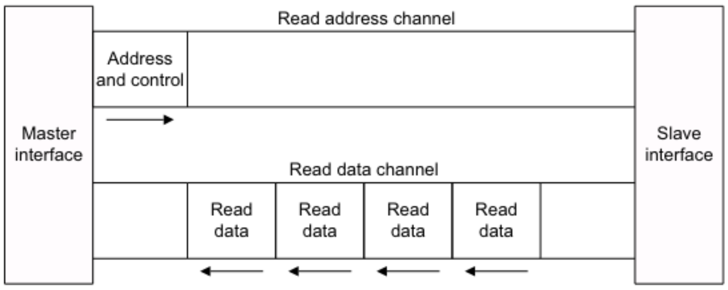


Figure 2.8: An AXI read transaction. source:[1]

In the Figure, we see that the master first sends the address and control for a read transaction, and then the slave sends back all the data for that request. That defines a relation between the Read address channel (AR) and the read data channel (R), which is that there must be a request in the AR channel before the R channel sends the data and the response.

The signals the AR channel uses when issuing a request are:

- *ARID* signal: Since AXI can have multiple outstanding requests and also supports out-of-order completion, this signal is the way to differentiate between active requests. If two or more active requests have the same ARID value, then those requests must complete in the order they are issued. Otherwise, there isn't a guarantee for the completion of requests.
- *ARADDR* signal: This is the address from which we want to read data.
- *ARVALID* signal: Indicates when the master sends valid address and control signals.
- *ARREADY* signal: Slave asserts it when he is ready to receive an address and associated control signals.

The signals the R channel uses when a slave returns data to the master are:

- *RID* signal: This is the signal that differentiates between the active requests this transaction replies to. It has the same value as the ARID value of the request the reply refers to.
- *RDATA* signal: It is the signal that carries the data of the read request, that was issued earlier in the AR channel. This signal is a bus with a width of 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide. In this thesis, we define it as 64-bit.
- *RRESP* signal: It is the signal that indicates if the transaction was successful or an error occurred.
- *RVALID* signal: The slave asserts it when it sends back (to the master) valid data and response.
- *RREADY* signal: The master asserts when he can receive the data and information a slave sends.

The signals we describe on the channels above are not all the signals that the AXI protocol provides. The mentioned signals are the ones this thesis mostly uses. The signals not mentioned, like the *RLAST* signal, have to mainly do with the burst nature of the protocol that this thesis doesn't use, so they have a default value defined by the spec. The *RLAST* signal signifies when the slave sends the last data of a read operation. For more information, see source[1].

2.3.1.2 AXI write transaction

A write transaction on the AXI protocol has the form that Figure 2.9 shows. In

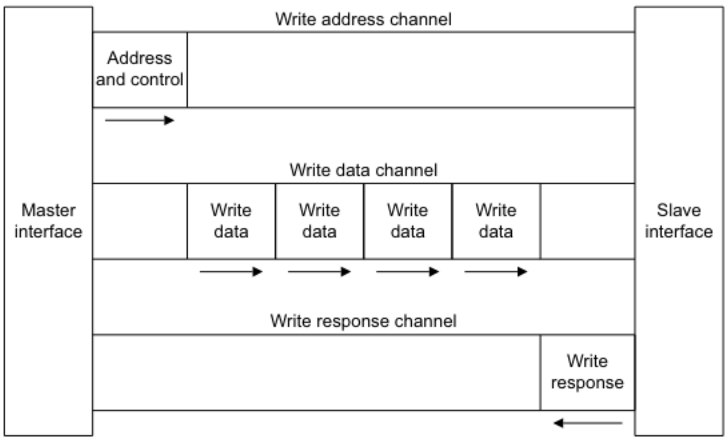


Figure 2.9: An AXI write transaction. source:[1]

the Figure, we see that a write transaction uses the remaining three channels of the protocol (the Write address channel (AW), the Write data channel (W), and the

Write response channel (B)). The AW channel is responsible for transferring (from the master to the slave) the information (address and control) needed to complete the transaction. The W channel handles the transfer of the transaction data the slave uses to complete the transaction. The data correspond to the address and control information that passes through the AW channel. In the Figure, we see a lot more transactions on the W channel because of the burst nature of the protocol, which this thesis doesn't use. After the slave receives all the data and information (address and control), it performs the write operation and sends a response to the master, through the B channel, about the operation's success.

Although the Figure shows the master sending the data (through the W channel) after sending the address and control (through the AW channel) for simplicity, the protocol doesn't define such an ordering constraint. The dispatch of information (address and control) and data can happen in any order. The only ordering existing is that the slave must send the response after both data and information arrive at the slave.

The signals the AW channel uses when issuing a Write request are

- $AWID$ signal: Since AXI can have multiple outstanding requests and also supports out-of-order completion, this signal is the way to differentiate between active write requests. If two or more active requests have the same $AWID$ value, then those requests must complete in the order the master issues them. Otherwise, there isn't a guarantee about the order the requests complete. Write requests have no relation to read requests, so there is no relation between $AWID$ and ARID.

- $AWADDR$ signal: This is the address that the slave uses to write the data of the request.

- $AWVALID$ signal: Indicates when the master sends valid address and control signals.

- $AWREADY$ signal: Indicates when the slave is ready to receive the write address and associated control signals.

The signals the W channel uses when issuing a Write request are:

- $WDATA$ signal: This is a bus signal with a width of 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide. It carries the data the slave needs to perform and finish the write request.

- $WSTRB$ signal: This signal indicates which byte lanes (from the $WDATA$ bus) contain valid data for the write request. It has a validity bit per 8-bits in the $WDATA$ bus. An asserted bit means the slave should write the corresponding byte in the memory.

- $WVALID$ signal: Indicates when the master sends valid data to the slave.

- $WREADY$ signal: Indicates when the slave is ready to receive the data from the master.

As we see, the W channel has no ID field that can connect the data it sends to an address and control previously the AW channel sends. For that reason, the master must send the data in the same order as the address and control transactions they correspond to (in the AW channel).

The signals that are used when the slave sends a response to the master about a write request are:

- BID signal: It indicates for which active write request this write response refers. For that reason, it has the same value as the $AWID$ signal of the request.
- $BRESP$ signal: It is the signal that indicates if the transaction was successful or an error occurred.
- $BVALID$ signal: The slave asserts it when it sends back a response to the master.
- $BREADY$ signal: The master asserts it when he can receive information that the slave interface sends.

The above signals in the channels are the ones this thesis uses. The protocol defines more signals for the support of the burst functionality it has. For example, the protocol defines the $WLAST$ signal in the W channel, which the master asserts when he sends the last needed data of a write request, else the master will send more data. For more information about that functionality and the signals refer to source[1].

2.3.2 AXI4-Lite

Simpler control register style interfaces that don't need to provide the full functionality of AXI4 (for communication) can use the AXI4-Lite protocol (a subset of the AXI4 protocol). Some of the main features the AXI4-Lite protocol provides are:

- It has no burst transactions.
- All data accesses use the full bus width. Also, the width data busses are either 32- or 64-bit.

The AXI4-Lite protocol (according to the above) still has the same five channels as the AXI protocol, but some signals and functionality have been changed or removed. More accurately, AXI4-Lite:

- Removes the $AWID$, BID, $ARID$, and RID signals. That means the requests in the interface are complete in the order of issue. However, read and write actions remain independent still.

- Doesn't guarantee support of the $WSTRB$ signal by the slave.
- Defines the $WDATA$ and $RDATA$ busses to have a width of either 32- or 64-bit.
- Contains (only) all the other signals we describe for the AXI4 protocol in each channel, plus the $AWPROT$ signal in the AW channel and $ARPROT$ in the AR channel. The use of these two additional signals is to protect against illegal transactions.

The changes (the AXI4-Lite protocol makes to the AXI protocol above) still allow for multiple outstanding transactions. Although multiple active transactions can exist, the transactions will complete in the same order as the order the master issues them.

Chapter 3

Design & Implementation

In this chapter, we present the design of Matzic . Specifically the chapter starts with the analysis of the pipeline, continues with the scheduling of the threads and completes with the controller of the core.

The structure of this chapter is as follows:

- Matzic Pipeline
- Thread scheduling
- Controller

3.1 Pipeline

The pipeline of Matzic is a 6-stage superscalar pipeline that supports up-to 256 threads, as Figure 3.1 shows. The following stages make up the pipeline:

1. PC Fetch
2. Instructions Fetch
3. Decode
4. Operands Fetch
5. Execute
6. Commit

Matzic also allows the schedule of up-to 4 instructions and has 7 execution clusters to service them. The execution clusters are:

1. Multiply and ALU. Able to execute multiply or ALU operations.
2. Div and ALU. Able to execute divide and ALU operations.
3. FPU. Able to execute floating point operations.
4. Load1. Able to execute Load operations.
5. Load2. Able to execute Load operations.
6. Store. Able to execute store operations.
7. Branch and ALU. Able to execute branch, jump, and ALU operations.

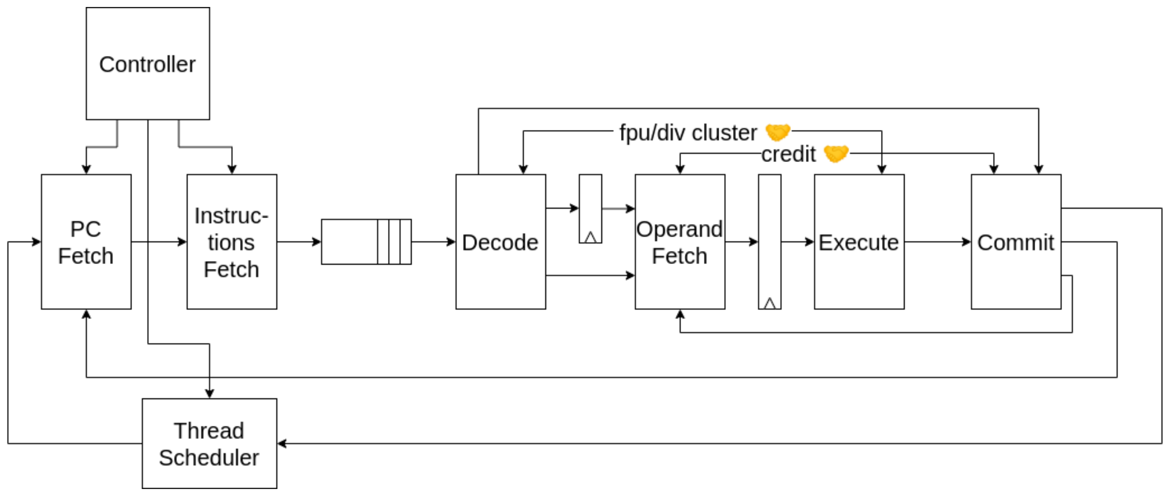


Figure 3.1: Execution Pipeline and Matzic Overview

Each instruction that passes through the pipeline needs a minimum of 8 Cycles until it reaches the commit stage of the execution.

3.1.1 PC Fetch

The PC Fetch section of the pipeline is responsible for keeping the PC values for each thread. The implementation of the section can be seen in Figure 3.2 .

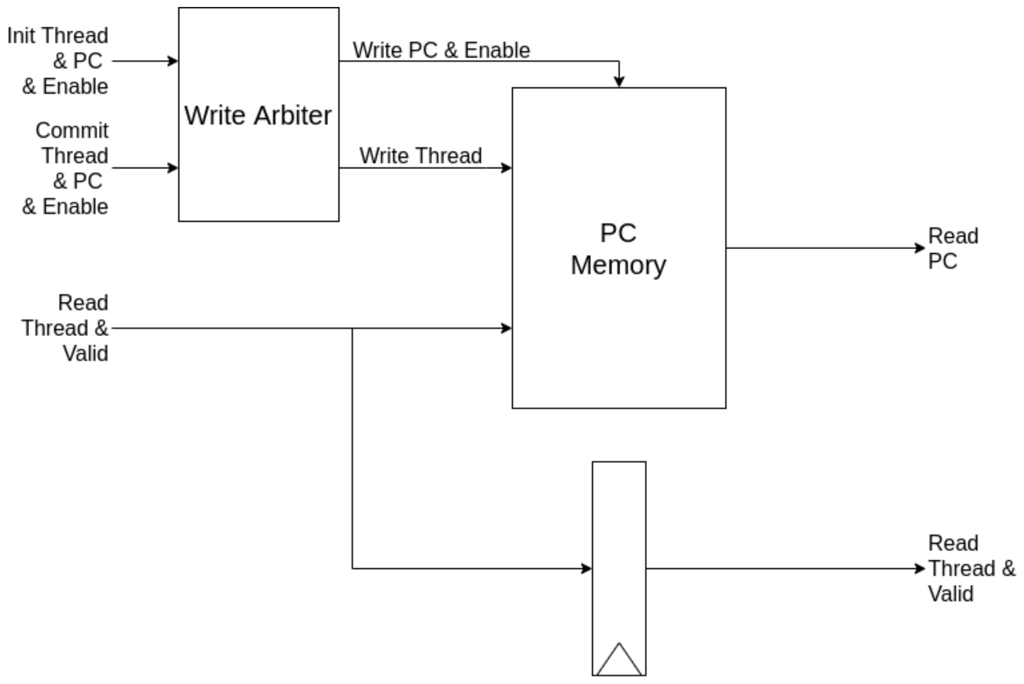


Figure 3.2: PC Fetch Top

For the task of keeping the PC values, we use a 256×64-bit two-port memory named PC Memory in Figure 3.2. Each address maps to a distinct thread-ID. That allows for the ability to read and write the value of two different threads at the same time. In the case the reading and the writing thread are the same, then the memory returns the old value to the reading thread.

The write arbiter selects which thread should write in the memory. The options for writing threads are:

1. A thread that's ready to commit its PC value coming from the Commit stage of the pipeline.

2. A thread that wants to initialize its PC value coming from the Controller.

The scheduling policy of the arbiter is to schedule the committing thread when available. Otherwise, it schedules the initializing thread. This policy assumes that an initialization operation is a rare occurrence, and it's best to be ready as soon as possible in case the committing thread starts execution in the next cycle.

3.1.2 Instruction Fetch

The Instruction Fetch section is responsible for getting 4 instructions from the Instruction Memory. The implementation of the Instruction Fetch can be seen in Figure 3.3.

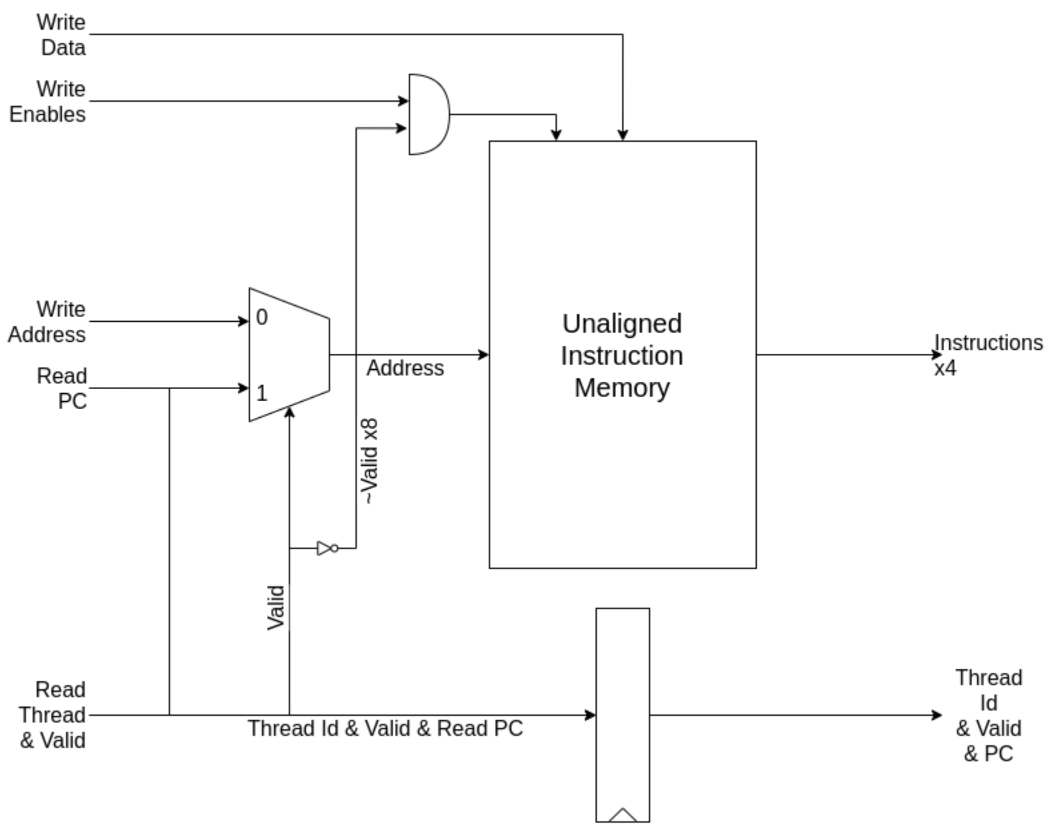


Figure 3.3: Instruction Fetch Top

We read the 4 instructions starting from PC. That means we get the instructions from PC, PC+4, PC+8 and PC+12 . For that reason we use an unaligned memory of width 16 Bytes. In case of using an aligned memory that would not allow to always read 4 instructions from memory because there is no guarantee that PC would always be aligned at 16 Bytes. As a result we would need 2 reads from the memory. By using an unaligned memory we can get the full 16 Bytes in 1 cycle without having the concern of PC to be aligned at 16 Bytes.

The unaligned memory has a single address for reading and writing, as we deem a write operation while we execute code to be an uncommon case. For that reason, we prioritize reading (from the Read PC) when a valid reading thread is present and only write when there is a bubble at this stage. The unaligned memory consists of 8 2-Byte wide single ported memories. Each memory has a possibility of 2 addresses, except the last memory, which has only 1 possible address. We create the first address by discarding the 4 LS bits of the input address and the second by adding 1 to the first address. If the value of the 4 LS discarded bits is greater than the id of that memory, we select the second address. Otherwise, we select the first address. The value of the discarded bits can never be greater than the id of the last memory, so we always address it using the first address.

3.1.3 Decode

The Decode section of the pipeline is responsible for decoding and scheduling the instructions to the different execution clusters. It also detects the hazards between instructions. The implementation can be seen in Figure 3.4.

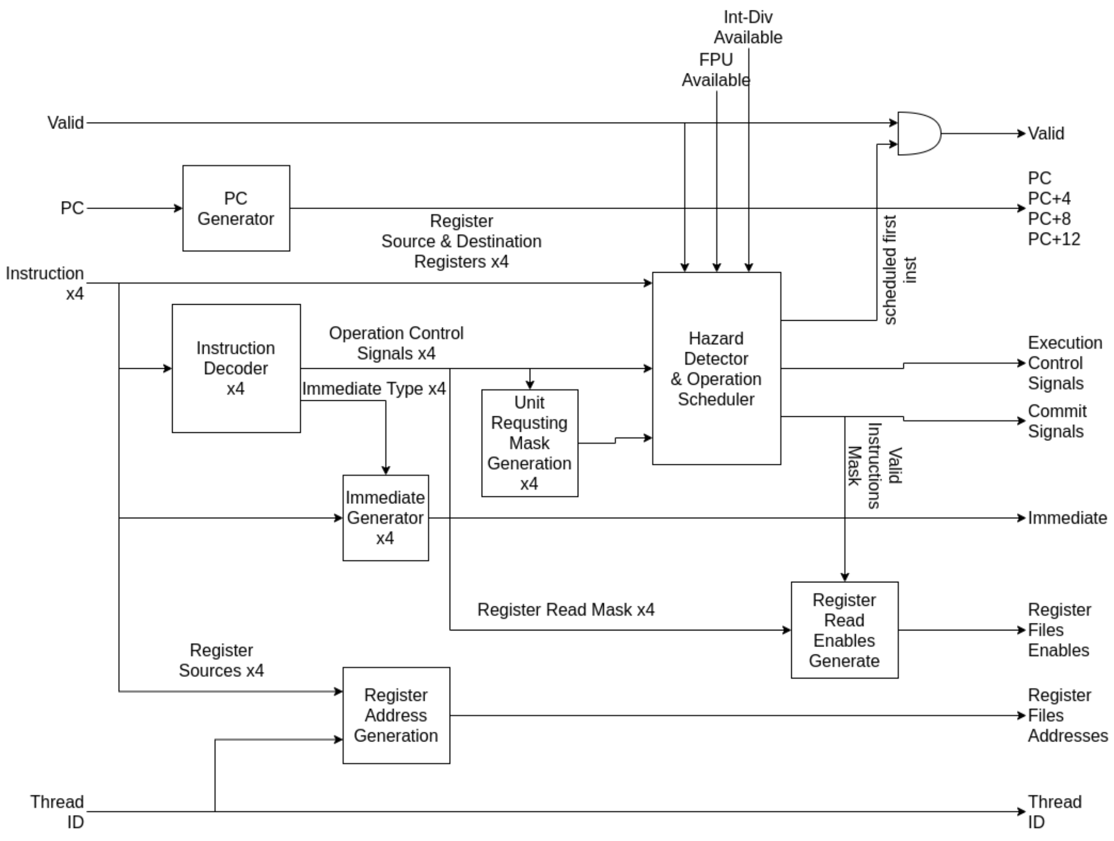


Figure 3.4: Decode

We use 4 instruction decoders and immediate generators, 1 for each instruction, as each instruction generates differently an immediate value and has different requirements for execution. Also, each instruction generates a mask with asserted bits to the units it can use to execute. The PC Generator has 3 adders that create the PC values for all the instructions after the first. The first instruction has the same PC value as the one used for reading the instructions. Because the instructions are sequential, the PC values corresponding to the second, third, and 4^{th} instructions are PC+4, PC+8, and PC+12.

The Register Address Generator creates the register addresses (for the register files) to read the correct source register values the instructions need. We do this by concatenating the thread ID and the source register field in the instructions. As a result, it creates 8 addresses for the integer register file and 12 for the float register file.

The module (hazard detector and operation scheduler) detects and schedules all the instructions that are possible to execute in program order. The checks that the scheduler does are for:

- Read-After-Write (RAW) hazards. By checking the source and destination registers of the instructions.
- Memory hazards. To avoid the memory disambiguation problem, we avoid issuing together load and store (or two store) instructions for execution, from the same thread, in the same issue window. As a result, that avoids the potential write-after-read (WAR), write-after-write (WAW), and RAW hazards that can happen when two memory operations access the same memory address.
- Control hazards. We define control hazards being the branch, jump, and barrier instructions. Branch and jump instructions change the normal flow of the program, which is to increase the PC counter by 4, and set the PC value to some other address. As a result, we don't schedule (to units) the instructions after a branch (or jump) instruction, as we deem them invalid. Although barrier instructions maintain the program flow, they also work as a point of synchronization between threads. For the thread to continue to issue and execute the instructions succeeding the barrier instruction, all other threads must also reach the synchronization point.
- Structural hazards. This hazard occurs when the processor has insufficient units to service all the instructions.

After detection of the hazards, the detector schedules as many sequential instructions as possible, as long as they don't violate the program order and there aren't any hazards between them. The pipeline stalls when it's impossible to schedule even the first instruction (due to structural hazards), until it becomes possible.

The Register Read Enables Generate module creates the corresponding read enables required to read from the register file. The register file performs a read to a register address only for the corresponding asserted enable signals. Thus, the scheduled instructions can read only the registers they require from the register file and reduce power consumption.

3.1.4 Operands Fetch

The Operands Fetch stage is responsible for keeping the register values for each thread and reserving credits for the clusters that the execution can complete in a different order than the one issued. Figure 3.5 shows the implementation of this section.

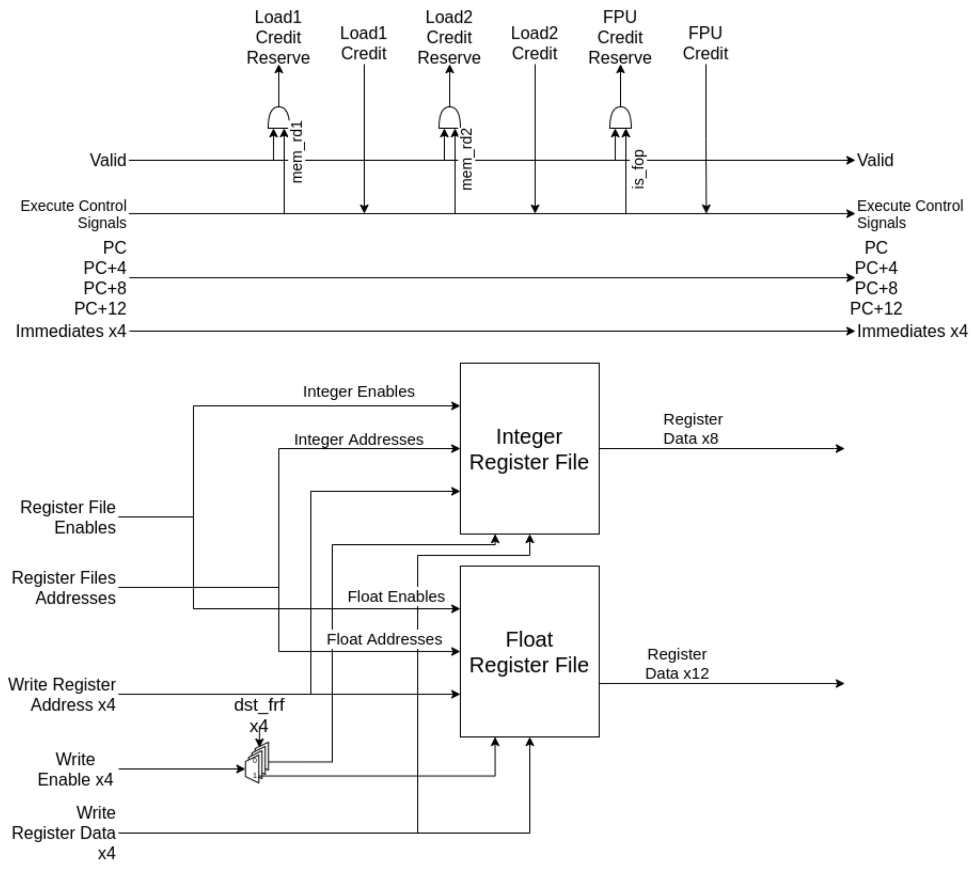


Figure 3.5: Operand Fetch

We keep two register files 1 for the integer registers and 1 for the float registers. Each register file has the same number of write-back ports but a different number of read ports. More accurately, the float register file has 12 read ports, and the integer register file has 8. The number of ports a register file needs is given by the function: $f_{rd\_ports} = InstructionIssueWidth * RegisterSources$. Where $RegisterSources$ is the maximum possible number of sources the instructions can have, which resolves to 2 for integer instructions and 3 for float instructions due to the fused multiply (-add and -sub) instructions. Although we can reduce the number of ports in the float register file, as it's impossible to schedule 4 fused multiply instructions, the 12 ports simplify the logic for the register source selection in the Execute Section.

The write-back is the same for both register files, as the mapping for the float registers corresponds to the mapping for the integer registers. The number of write-back ports is given by: $f_{wr\_ports} = InstructionIssueWidth * DestinationRegs$. The $DestinationRegs$ parameter is the maximum possible number of destinations registers in the ISA, which for the case of RISC-V is 1. So the number of write-back ports is 4. Although it is impossible to utilize all the read ports on the float register file, that is not the case for write ports, as 4 instructions can perform a write-back in the float register file i.e. 2 fld, 1 fadd, and 1 fmv.

3.1.4.1 Integer Register File

Figure 3.6 shows the implementation of the integer register file.

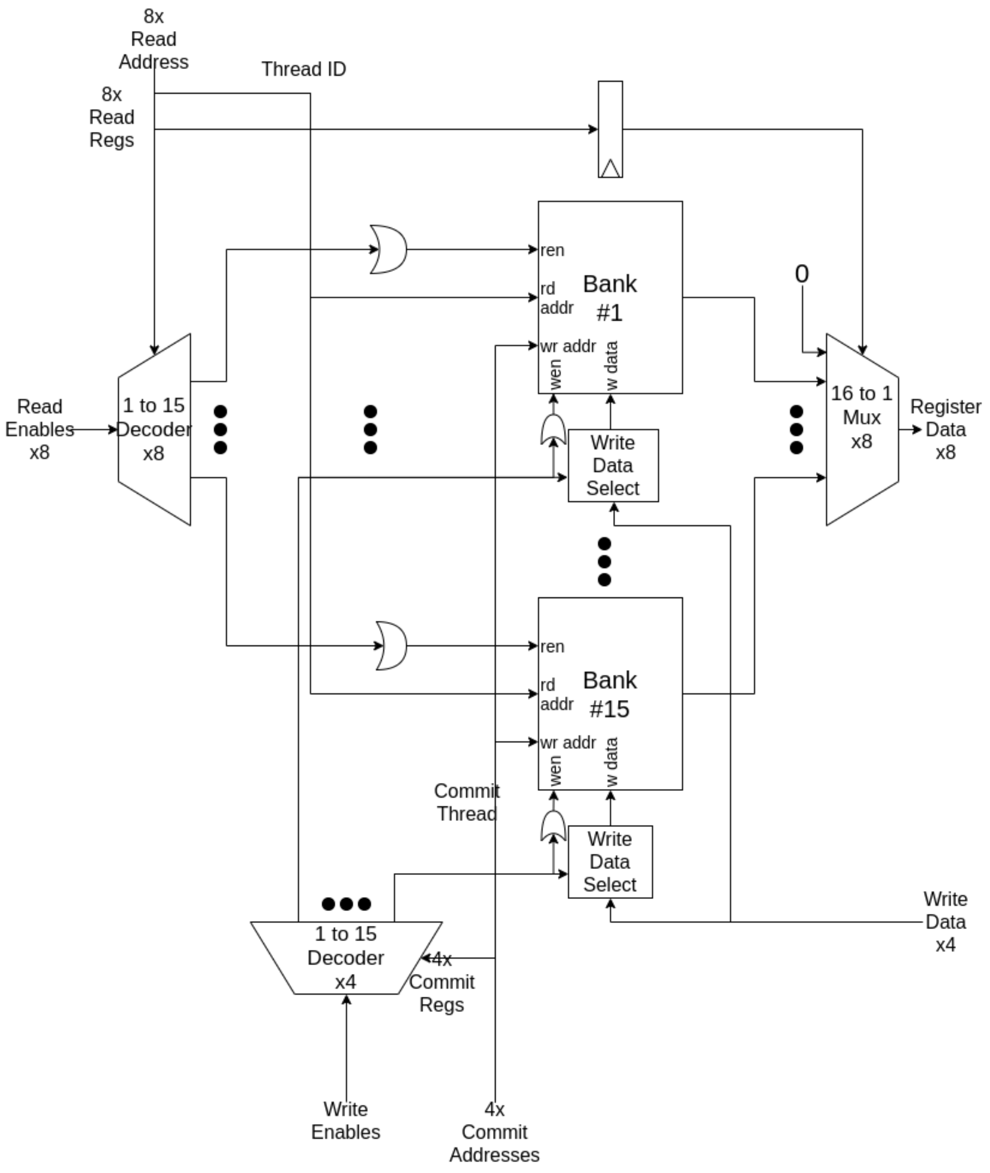


Figure 3.6: Integer Register File

We implement the integer register file as a banked memory. For that reason, it has 15 256 × 64-bit two-port memories. Each memory bank corresponds to an ISA register from $x1$ to $x15$. There is no need to have a bank for the register $x0$ as its value is always 0, and a write to that register has no effect. Each memory has as many words as existing threads, so each word in the bank maps to the register value of a thread's register. That way, we create the address for each memory without wasting resources and only provide the executing thread's ID when we want the value of that thread's register. Each memory also has a read enable signal (ren), which disables the read port of the memory and thus reduces power consumption, as reads happen only to the banks (registers) the thread needs. Each of the 8 $(1 - to - 15)$ decoders produces a ren signal for every bank. All the ren signals for a bank then pass from an *or* gate to determine if that bank should perform a read operation. The *or* gates also solve the problem when we need to read a register more than once. We use the 8 read regs signals to decide which bank to read and also pipeline them to set the output value for each read port of the register file to the correct read register value that the instructions need.

When committing data to the register file, we follow the same method as for reading. We use 4 decoders to decide the bank(s) where a write operation must occur, and we use an *or* gate for each bank in case multiple ports want to write at the same bank(register) – this should not typically happen by efficient code sequences but it is permitted. The write address for all the banks we want to write follows the same philosophy as the read but this time we use the committing thread's ID. When the commit bank(register) is the same for more than one port, the write operation uses the data of the last port in the register file, this is because the last port is used by the last instruction issued in program order for the specific thread. For example, if both port 0 and port 2 want to write at Bank#4(register $x4$), then the register file uses the data of port2.

3.1.4.2 Float Register File

Figure 3.7 shows the implementation of the float register file. The implementation is similar to the integer register file, with the difference that we use 16 256×64-bit two-port memories, which correspond to registers $f0$ to $f15$. The float register set doesn't have special cases as the $x0$ register in the integer register set.

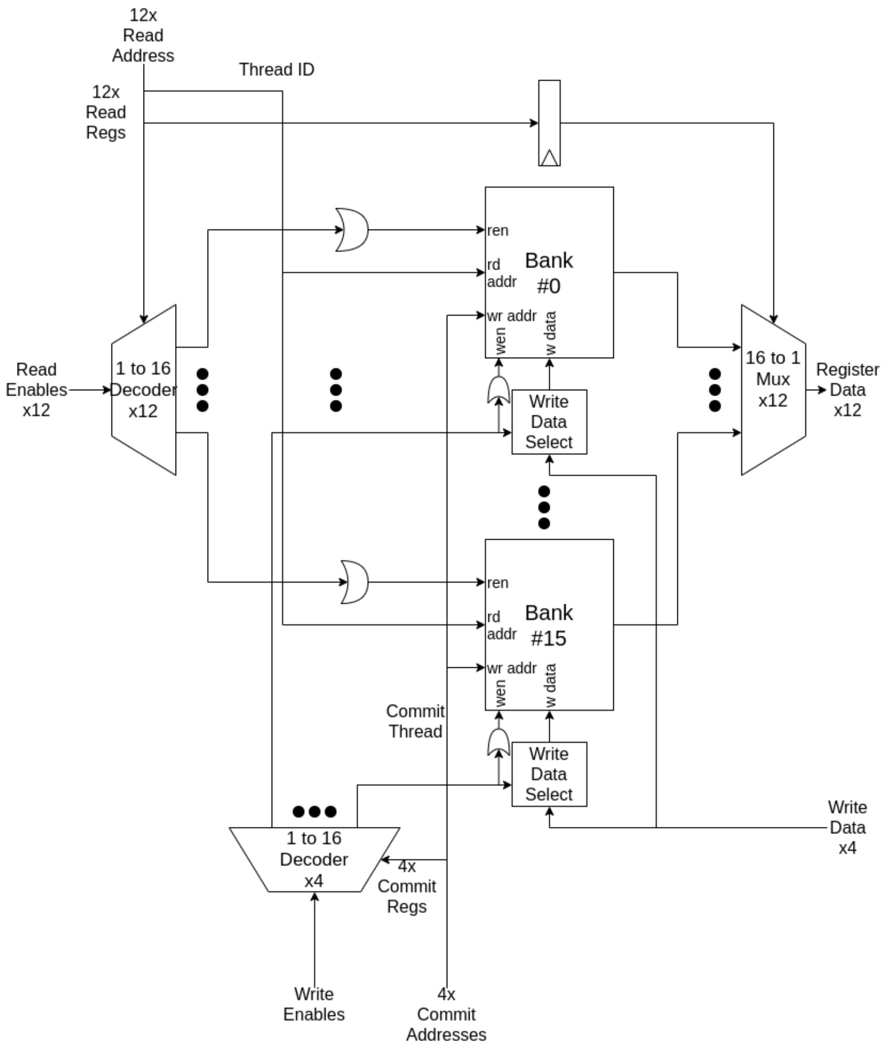


Figure 3.7: Float Register File

3.1.5 Execute

The Execute section performs the operations that the scheduled instructions need. Figure 3.8 shows the implementation of this section.

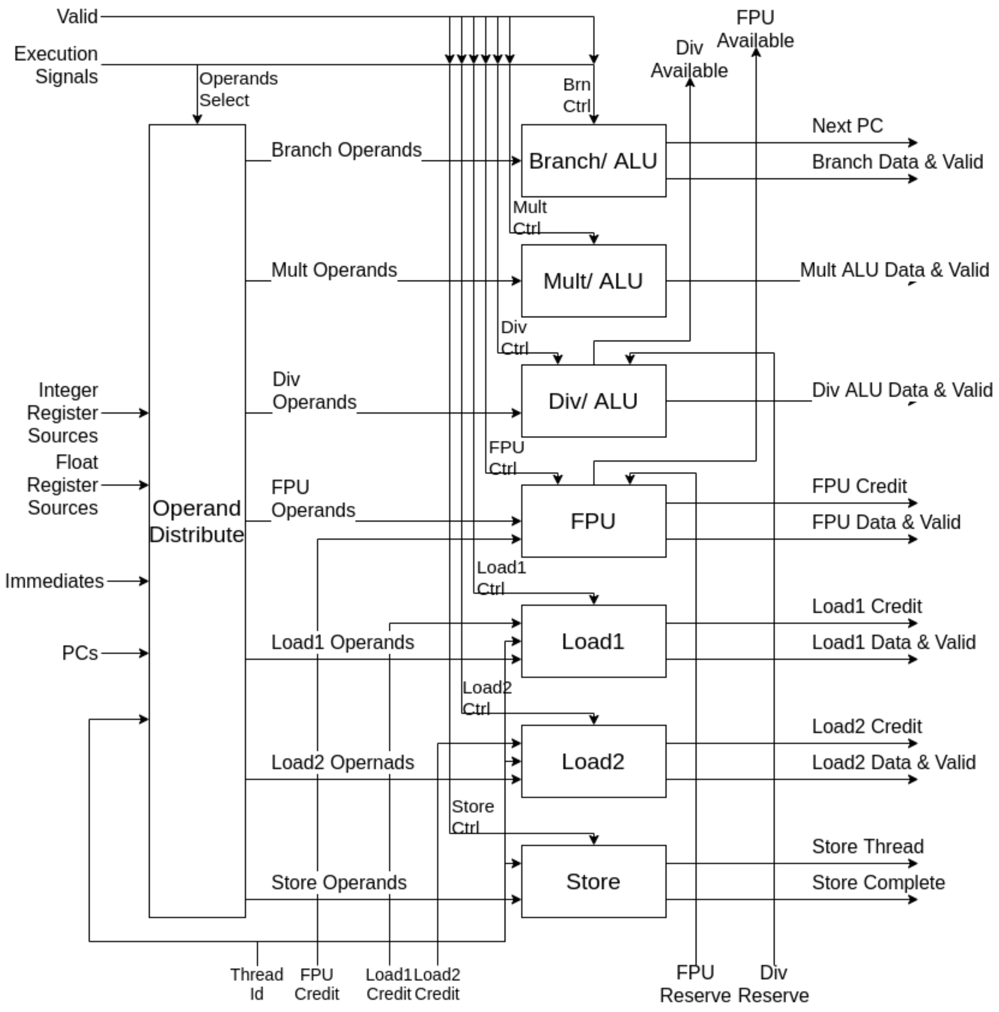


Figure 3.8: Execute

All the operands (which the previous stages of the pipeline create or read) go into the Operand Distribute module so that it can distribute them to the appropriate clusters. Each of the seven clusters also receives the control signals it needs for the execution. When the FPU and Div/ ALU clusters have space to receive a new operation, they assert the $DivAvailable$ and $FPUAvailable$ output signals. When the Decode stage schedules an instruction to the FPU or Div/ ALU clusters, it also asserts the appropriate signal from $FPUReserve$ and $DivReserve$. The scheduler from the Decode stage of the pipeline shouldn't schedule an operation to a cluster when the corresponding available signal is deasserted.

The FPU Credit, Load1, and Load2 Credit that go into FPU, Load1, and Load2 clusters, respectively, are the credits that accompany the resulting values the clusters produce since they can complete the operations in a different order than that which they accept them. For example, if two threads with id 0 and id

1 enter the Load1 cluster with that order, then it is possible the output order to be id 1 and then id 0. The credits assure that the data will commit in the correct order when the time for commit comes.

Although the Store Cluster has the same trouble as the Load1, Load2, and FPU clusters, which is the completion of out-of-order store operations, we don't need something elaborate for that because store operations don't commit any data to the register files, and we only need to know that the store operation for a thread is complete.

Each cluster has a different set of operations it can perform, and we can see most of them below.

3.1.5.1 Branch/ ALU

The Branch/ ALU execution cluster can perform either ALU or branch type operations. Its implementation can be seen in Figure 3.9.

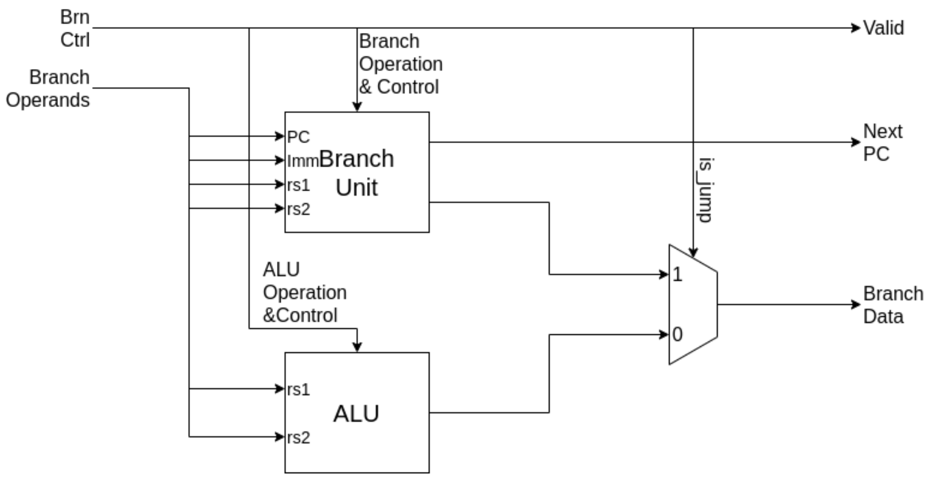


Figure 3.9: Branch/ ALU

The Branch Unit receives the PC and immediate values corresponding to the instruction scheduled in that cluster and the 2 source operands when the instruction is a conditional branch instruction. The output of the Branch Unit is the next PC value (Next PC) that the thread should next fetch instructions from, as well as the return address of a jump instruction.

The Branch Unit always provides the next PC value to fetch instructions from, even when it has no branch or jump operation to execute. If it has no operation to execute, then the Next PC value is the PC of the last instruction Decode stage scheduled, plus 4.

The ALU allows the Decode stage to schedule more instructions in the pipeline. It commits its data when there is no operation for the Branch Unit. Also, since the Branch Unit needs to write back in the register file only when it executes an unconditional branch instruction, this cluster outputs the data of the ALU by default. The ALU Operation & Control signals mostly contain the operation that the ALU has to execute, as well as if the result is a 32-bit integer and has to sign-extend to 64-bit or it is a move operation to the float register file, and the result may be NaN boxed in case of single precision float numbers.

3.1.5.2 Mult/ ALU

The Mult/ ALU cluster operates for ALU and Multiply operations. The cluster implementation can be seen in Figure 3.10.

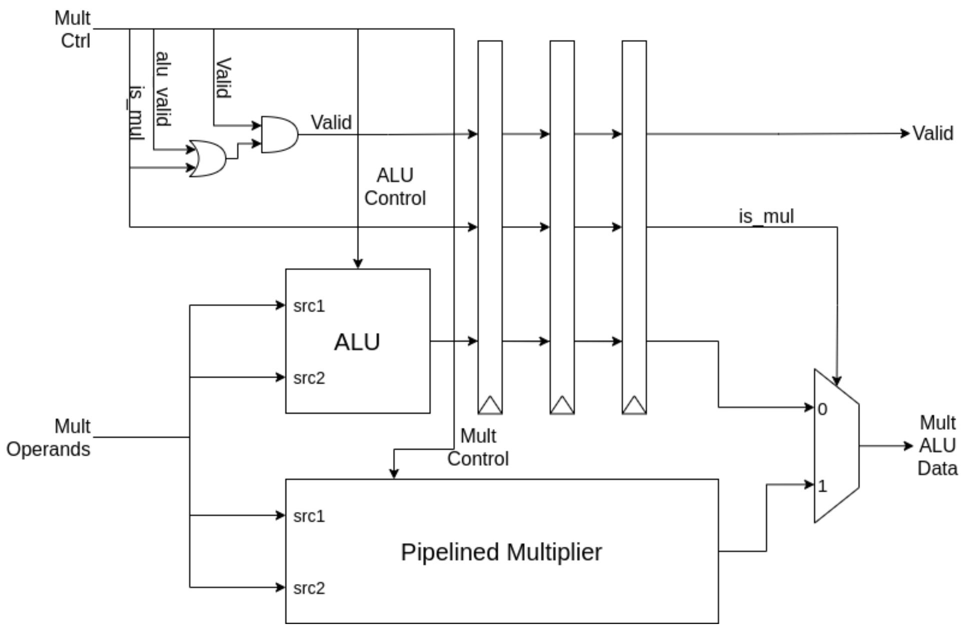


Figure 3.10: Mult/ ALU

This cluster contains an ALU and a 4-stage pipelined multiplier. This cluster pipelines the output of the ALU to be synchronous with the output of the multiplier. It also generates a $Valid$ output signal to avoid committing any erroneous results. The cluster also pipelines the *is_mul* signal that determines which unit produces the operation's result value. An asserted $Valid$ signal generates when a valid input and operation are present in the cluster. To identify a valid operation, the cluster checks the *alu_valid* and *is_mul* signals.

3.1.5.3 Div/ ALU

The Div/ ALU cluster operates for ALU and Multiply operations. The cluster implementation can be seen in Figure 3.11.

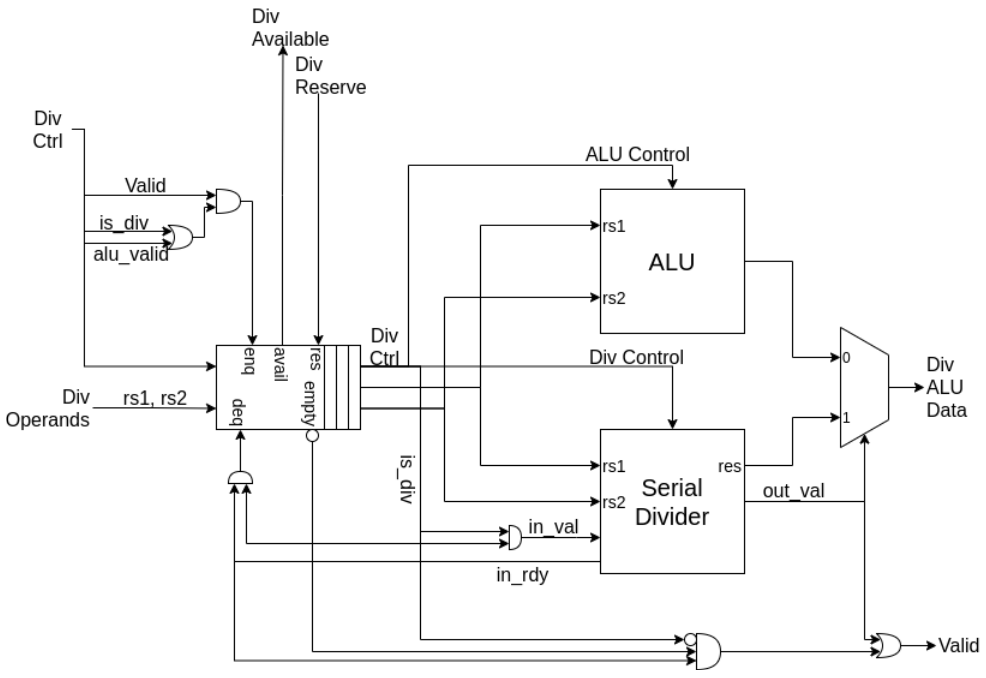


Figure 3.11: Div/ ALU

The cluster uses a credited FIFO to buffer operations in case of slow division, as well as an ALU and the *SerialDivider* from the Ariane CPU [12] that handles integer division operations. We use an output *Valid* signal that signifies an operation's completion. The cluster serves the operations on a first come first served basis, which means that the executing threads that enter the cluster will finish and exit in the same order. We couple the divider with an ALU to increase the possible amount of instructions scheduled.

The cluster regulates how many operations it can handle by using a credited FIFO that acts as a buffer. The proper order of operations to enqueue data to the credited FIFO is to allocate a position and then enqueue the data. To allocate a position the Decode stage asserts the *DivReserve* signal (which also asserts the *res* FIFO input signal) only when the FIFO has the *DivAvailable* signal asserted (through the *avail* signal). If the FIFO has available space for allocation it asserts the *avail* signal, else the deassertion of the signal signifies the lack of free space in the FIFO. The FIFO provides the *enq* input signal for when data need to enqueue. The FIFO deasserts the *empty* signal only after an enqueue operation and leaves it deasserted as long as it still contains data. When we assert the *deque* input signal the FIFO frees (and makes available for allocation) the oldest enqueued position, and in the next cycle, the output data change to the ones directly enqueued after the current. We enqueue data in the FIFO only when a valid thread is in this stage of the pipeline, and either we have either of the input *is_div* or *alu_valid* signals asserted. Those signals can never be asserted simultaneously, as doing both operations isn't possible. The Hazard Detector & Operation Scheduler, seen in Figure 3.4, ensures the exclusivity of those signals.

The *SerialDivider* (divider) accepts operations and outputs via handshaking. A divide operation happens when the FIFO is not empty, and an asserted *is_div* signals from the head output of the FIFO. The divider asserts the *in_rdy* signal when idle. We use the ALU when *is_div* and *empty* signals are deasserted. The *DivControl* signals contain information about the type of division operation the *SerialDivider* will perform. We assert the *Valid* signal when either the divider asserts the *out_val* signal, which signifies that the divider outputs the result of the latest operation it started, or when the FIFO is not empty, the divider is idle, and *is_div* is deasserted. The default data output of the cluster is from the ALU and changes to the divider's only when it asserts *out_val*. Although Figure 3.11 does not depict this, the divider has an input *out_rdy* signal that notifies that the result it outputs is accepted, and can return to an idle state, since we can always accept the result of the divider we always assert the *out_rdy* signal.

3.1.5.4 FPU

The FPU cluster is responsible for performing all of the floating-point operations, except for the floating-point move operations, which we handle with ALUs. Figure 3.12 shows the implementation of this cluster.

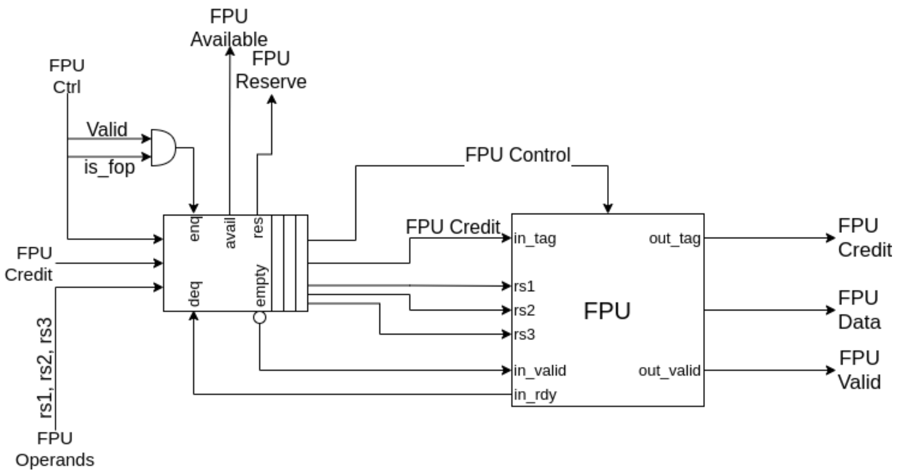


Figure 3.12: FPU

This cluster contains a floating-point unit (FPU) created from ETH Zurich [8] and a credited FIFO. The FPU operates via handshaking to receive operations and return results. When the FPU can't accept an operation, we use a credited FIFO to buffer them and not stall the pipeline.

The FIFO is used to buffer up to 8 operations if the FPU can't absorb them. It works the same as the one used in the Div/ALU cluster, where the *avail* output signal signifies that the cluster has available space, and the *res* input signal allocates space in the FIFO. To enqueue data a valid thread must be present in the Execute stage, and there must also be an operation for the FPU. An asserted $Valid$ signal signifies the presence of a valid thread, while an asserted is_fop signal means there is a valid operation for the ALU.

The FPU has different configurations depending on the different needs that exist. We use the configuration for the RISC-V 64-bit D extension, and also change the pipeline depth of the units it contains to reach higher frequencies. The FPU can execute different operations concurrently. An input tag is given together with an input operation to differentiate between the active operations in the FPU. The FPU can have a lot of active operations because the operations in the FPU have different execution times. That means the operations can finish executing in a different order from that which the FPU starts processing them, because of their execution time differences. The unit accepts and starts an operation when it has both the *in_valid* input signal and *in_rdy* output signals asserted. The in_tag input signal is the tag that accompanies the operation the FPU accepts. As the tag value, we use the credit that this thread will use when its result reaches the Commit stage of the pipeline, because the credit is already unique, and it allows us to conserve resources we would use to track it. The FPU Control signals are the signals that contain information about the operation the FPU will perform, such as the operation it will execute, the rounding mode, and others. When the FPU wants to output the result data of an operation, it asserts the *out_valid* signal and sets the *out_tag* signal value to the *in_tag* (credit) of the operation the result corresponds. The FPU also provides an *out_rdy* signal which Figure 3.12 does not depict, as it is always asserted. That signal signifies to the FPU that the current output is received and can continue to the next.

3.1.5.5 Load1 & Load2

The Load 1 and Load 2 clusters have the same structure, as they only handle memory read operations. Figure 3.13 shows the implementation of these clusters.

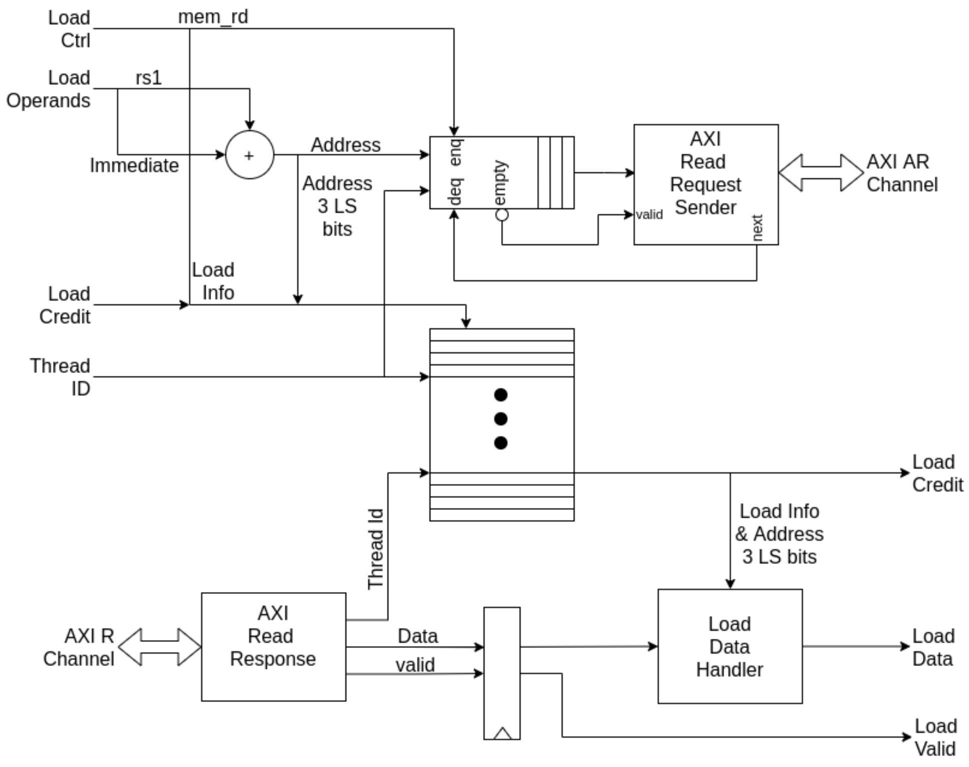


Figure 3.13: Load Unit

Each cluster contains an adder that adds the first operand read from the integer register file and adds the immediate value to create the final destination address. The cluster stores the *Thread ID* and the address from which the read operation will take place in the memory in a FIFO. The FIFO has 256 available slots. We use the FIFO to buffer operations when a slave device's AXI AR Channel either can't handle many different requests or accepts requests slower than the pipeline issues them. Thus the use of the buffer avoids introducing stalls in the pipeline. The value we use for the AXI *ARID* of the AXI AR Channel is the same as the $Thread_ID$. That allows each cluster to issue up to 256 outstanding requests. That means both clusters can issue up to 512 outstanding read requests together if the environment can handle them. The reason for having so many requests is to be tolerant of the delay of reading from a DRAM memory, which is about 100 nanoseconds. The AXI Read Request Sender module creates and handles the AXI transaction that a read operation needs. The *valid* input signal signifies a valid memory read request, which happens when the FIFO is not empty. The *next* output signal notifies about the acceptance of the request and that in the next clock cycle the AXI Read Request Sender can issue a new request.

In the clusters, there is also a 256 entry two-port memory, to keep information for each active load request. The data the memory keeps for each load is the memory credit it needs when the result goes to the Commit stage of the pipeline, as the order in which the replies to the load requests arrive isn't guaranteed and can be different from the order of issue. The other info we have is the size (how many Bytes we read from memory) of the load operation, the sign of the operation, and the 3 LS bits of the address for shifting the received data if the address is not 8-Byte aligned.

The AXI Read Response module receives the replies to the read requests issued by the AXI AR Channel. The module outputs the $Thread\ ID$ (which it knows from the AXI RID signal that the AXI AR Channel provides), the data the memory sends, and a *valid* signal to signify when the output of the module is valid. We pipeline the data and *valid* signal as we need a cycle delay to retrieve the $Load\ Info$ (from the array) that we need to create the correct data for the load operation and the credit the Commit stage needs to commit the data. The Load Data Handler module is responsible for creating the final (result) data by doing the sign extensions and ordering the operation needs on the pipelined data.

These clusters only handle aligned memory operations. In case of an unaligned memory operation the behaviour is undefined.

3.1.5.6 Store

The Store cluster is responsible for the memory store operations. Figure 3.14 shows the cluster's implementation.

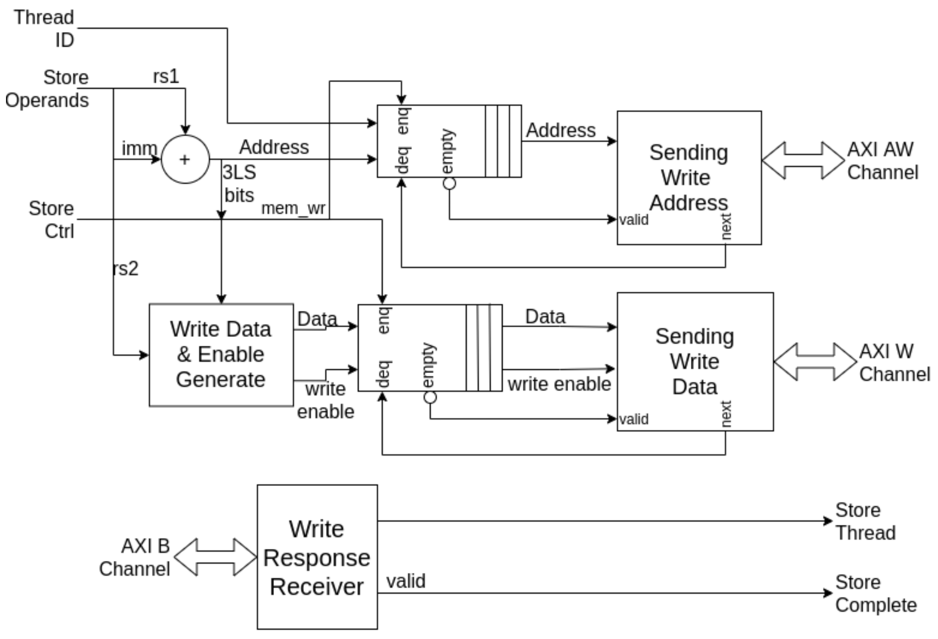


Figure 3.14: Store

This cluster has an adder to create the address where the store should take place. It also has 2 256 entries FIFOs for buffering requests in case any of the AXI

AW or W Channels accepts requests slower than the pipeline issue rate. In the first FIFO, we enqueue together the address and the $Thread\ ID$. The data we will store in the memory pass through the Write Data & Enable Generate module, which creates the final data that will submit in the memory, as well as a write-enable mask that has the asserted bits for the exact bytes (from the data) that the memory should update, and discard the others. The second FIFO stores the data and write-enable mask.

The Sending Write Address module is responsible for sending the address where the memory should store the data and opening a write transaction on the AXI AW Channel. The $AWID$ of the transaction is the same as the $Thread\ ID$. Through that, we know which thread completes his store operation when a response arrives. The *valid* input signal signifies that a store request is valid, which corresponds to the FIFO having data. The *next* output signal hints that the module can accept a new input request in the next cycle.

The Sending Write Data module is responsible for sending the data the memory needs to store, and the write strobe (write-enable mask) that has asserted bits for the data bytes the memory should update. The memory keeps the old value for the bytes that have deasserted bits in the write strobe. The module sends the data and write strobe through the AXI W Channel. Because this channel doesn't have an ID, the transactions on this channel must follow the same order as the ones AW Channel starts. For example, if a thread with id 0 wants to write at address A0 the data D0 and a thread with id 1 to address A1 the data D1, then if thread 0 sends the address (A0) through the AW channel first it must also send first the data (D0) in the W channel. If thread 1 sends its data (D1) first, then the memory would write the D1 data to the A0 address, which is different from the intended action.

The Write Response Receiver receives the responses for the write operations that start through the AW channel and outputs for which thread the write operation in the memory was successful. It can extract that information through the BID field of the B Channel, which corresponds to the $AWID$ of a started transaction for which we use the $Thread\ ID$ of a thread.

This cluster only handles stores for aligned addresses. In case of an unaligned address the behavior is undefined.

3.1.6 Commit

The Commit Stage of the pipeline is responsible for committing the results of a thread when they are ready and updating the state of a thread. Figure 3.15 shows the implementation of this pipeline stage.

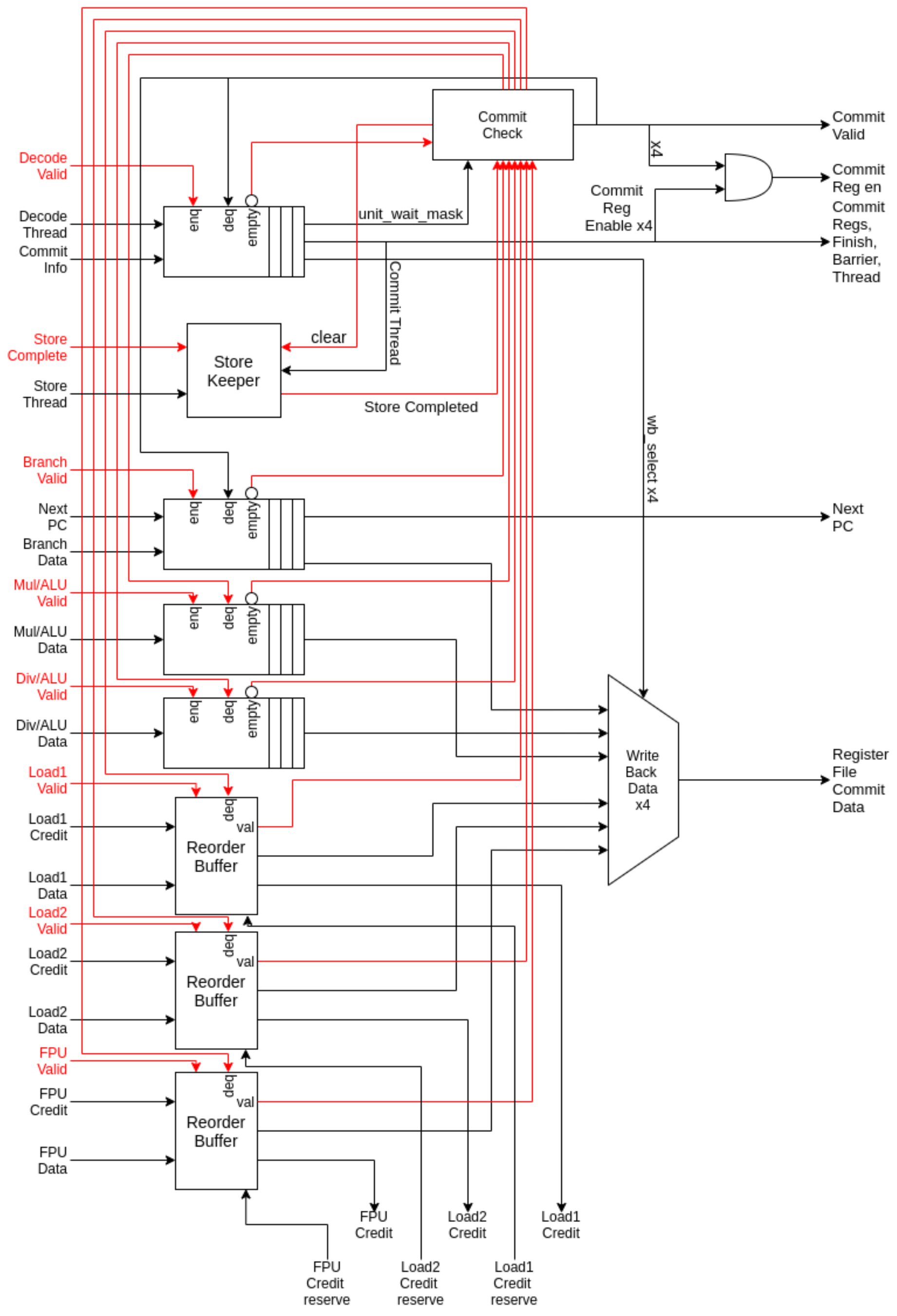


Figure 3.15: Commit

This stage has a 256 entry FIFO that contains the order in which the threads will commit their results and information about the commit. The commit information the FIFO contains for each entry are:

- the destination register file (integer or float) where each instruction wants to commit its results.
- a bit for each instruction that wants to update a register.
- the register each instruction wants to update.
- a bit to signal that the thread has to enter a barrier state and wait for others.
- a bit to signal that the thread has finished execution.
- a mask ((the *unit_wait_mask*) with asserted bits for the clusters the thread uses.

We use the *unit_wait_mask* mask to know when all the instructions executing for the thread have finished, so we can commit their result data and update the PC of the thread. The FIFO that keeps the commit information enforces the order the threads will commit. The order the threads commit is the same as the order they enter the pipeline because the FIFO fills when the Decode stage successfully schedules a thread, and the Decode stage doesn't reorder the threads.

We use 3 FIFOs to keep the result data from the Branch/ ALU cluster, Mul/ ALU cluster, and Div/ ALU cluster. We use FIFOs for those clusters because the thread execution follows the same execution order as the order they enter the commit FIFO. So the output data of those FIFOs corresponds to the output commit info the commit FIFO outputs. When those FIFOs are not empty, it means they hold the result value for a thread that uses the corresponding cluster. We always enqueue data to the Branch FIFO as each entry holds two values, the next PC of the thread (that we always use to update the threads PC) and a value we can commit to the register file. The value we commit to the register file is either the return address from a jump and link instruction or the result of the ALU in the cluster.

The Load1, Load2, and FPU clusters, that can reorder the threads' finish order, have 3 Reorder Buffers to keep their results. The reorder buffers give a credit that allows reordering the data in the order the credits are reserved. So it puts the data in the correct position for commit. An asserted *val* signal means the buffer outputs valid data. The Operand Fetch stage reserves the credits when a scheduled instruction heads to one of the Load1, Load2, and FPU execution clusters. When a cluster commits to the reorder buffer it uses the reserved credit and asserts a valid signal to signify that the buffer should store the data. It also provides the result data for the thread. When we assert the *deq* input signal, the reorder buffer continues to fetch the next data in the next cycle.

The Store Keeper tracks the threads with a finished store operation. The Store Keeper asserts the *Store Completed* signal when the Commit thread has finished its store operation. The input *clear* signal resets the state for the Commit thread so it can track another store operation of the Commit thread in the future.

The Commit Check module checks if all clusters thread uses have produced results. When all the clusters produce the results the thread needs, it then asserts the *Commit Valid* signal to signify that the thread is now committing. It also asserts the appropriate *deq* signals for the units the *thread* uses, so it doesn't erase data of other *threads*. The 4 Write Back Data multiplexers select the appropriate data to commit from one of the 3 FIFOs or 3 Reorder Buffers, using the 4 *wb_select* signals. The *Commit Reg en* signal is a mask that has asserted bits only for the instructions that need to commit data and only when the commit is valid to avoid unneeded commits in the register files. We always dequeue both the FIFO for the Branch Cluster and the commit info FIFO, as the first always contains useful data for the thread (which is the next PC) and the second to advance to the next thread to commit.

3.2 Thread Scheduling

The thread scheduling is one of the most important aspects of the processor. The scheduling we follow is a simple round-robin. The scheduler always goes to the next thread from the current one, even when the next thread is not available for execution until it reaches a specific thread ID which is the maximum allowed ID for a thread. When the scheduler reaches the max ID, it starts again from the first thread. More accurately, the function f_{next} gives the next thread that will enter the pipeline. We describe the function as follows:

$$f_{next}(thread) = \begin{cases} thread + 1 & ,\text{if } 0 \leq thread < MAX\_THREAD \\ 0 & ,\text{if } thread \geq MAX\_THREAD \end{cases}$$

Moreover someone can generally say that f_{next} is a function that has a definition domain, as well as, a result domain to be [0, MAX_THREAD], or more mathematically correct $f_{next} : [0, MAX\_THREAD] \mapsto [0, MAX\_THREAD]$.
The f_{next} input argument *thread* is the current thread that the scheduler sends to the pipeline for execution, while MAX_THREAD is the maximum thread ID the scheduler can reach. In the design, MAX_THREAD can only be a value in the range [0, 255] as there are 256 threads. The behavior of the scheduler is undefined for values outside that range.

As stated above the scheduler always advances to the next thread, even when the next thread is not ready for execution. The reason for that decision is that in the common case of operation, which is to have all 256 threads active, each thread should finish execution in 256 cycles until the scheduler decides to schedule it again. The latency of 256 cycles is enough to cover a round-trip to the DRAM, which in modern processors is about 100 processor cycles. The reason a thread is not available for execution can be any of the following:

- It reaches the end of execution. After a thread executes all the available code it needs to execute, it should stop, as further execution may corrupt the data and state of the program. A thread finishes execution when it executes an *ecall* instruction.

- It's in a barrier state and all the other threads have yet to arrive at the synchronization point specified by the barrier. A barrier is a point in the code where every thread should reach and wait for all the other threads. If a thread continues to execute before all threads reach that point, then the result may differ from the intended behavior by the programmer.

- It still executes the instructions from a previously scheduled time. That can be because of different reasons in the pipeline, like having unavailable the unit the first fetched instruction needs or the memory latency being greater than the time the scheduler needs to reach the ID again.

If any of the above cases is true then the scheduler doesn't schedule the thread and inserts a bubble in the pipeline to replace the slot of the thread.

3.3 Controller

The controller provides communication with the processor pipeline. It is responsible for initializing the PC values of each thread, initializing the instruction memory, setting the MAX_THREAD variable where the thread id wraps around, and starting the execution of the threads. It also contains different performance counters that we use for evaluation.

The controller communicates via the AXI-Lite protocol, with a 64-bit bus for address and data. The controller implements the functionality of an AXI-Lite slave. Since we use the AXI-Lite protocol, we map operations to different addresses. More accurately, we use the 18 LS bits of the address which the AW and AR Channels of the protocol provide to determine the operation the controller needs to execute to the processor. The controller can handle read and write requests simultaneously because of the AXI protocol.

3.3.1 Write Operations

The controller handles write requests one at a time. The way the controller handles a write request is by using an FSM. Figure 3.16 shows the FSM.

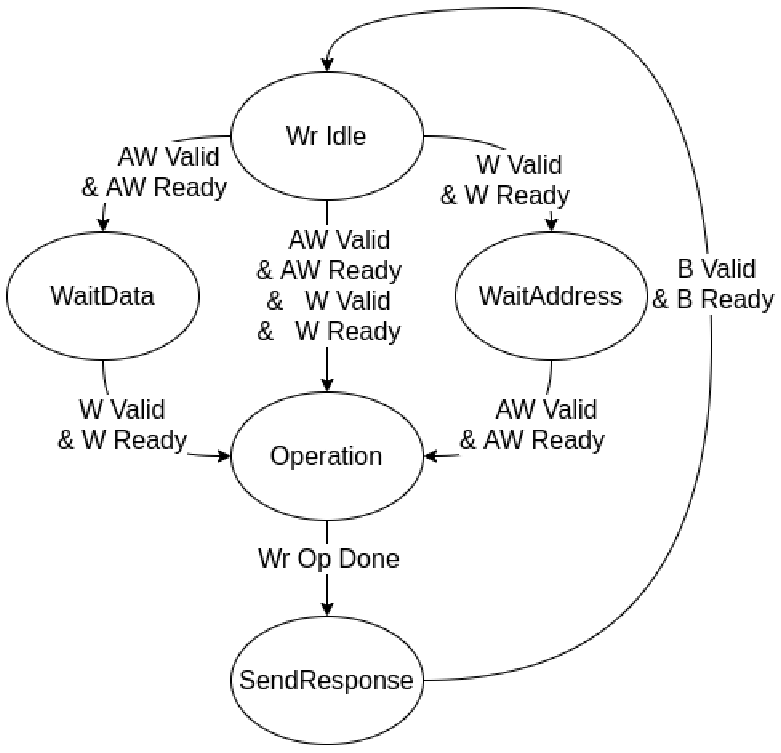


Figure 3.16: Controller Write FSM

The controller starts from the *Wr Idle* state and returns to that state when it is ready to receive a new write request. The controller then moves to one of the following 3 states:

- It moves to the *WaitData* state when it receives the address from the AW Channel but still waits for the data through the W Channel.
- It moves to the *WaitAddress* state when it receives the data from the W Channel but still waits for the address through the AW Channel.
- It moves to the *Operation* state when it receives at the same time both the data from the W Channel and the address from the AW Channel.

When the controller is in either the *WaitData* or *WaitAddress* state, it will move to the *Operation* state after it receives the missing information it requires. On the *Operation* state, it executes an operation depending on the 18 LS bits of the write address. Table 3.1 summarizes the possible write operations. When an operation finishes, it asserts the *We_Op_Done* signal, and the FSM moves to the *SendResponse* state. In the *SendResponse* state, the controller sends a response back that the operation is successful and waits until the master accepts it. After the master accepts the write response through the B Channel, the controller returns to the *Wr Idle* state.

Address 18 LS bits	Write
0aaaaaaaaaaaaaaaaa	Load 2 Instructions in Instruction Memory
10xxxxxttttttttxxx	Initialize PC for thread (tttttttt).
110xxxxxxxxxxggxxx	Enables threads in group (gg).
111xxxxxxxxxxxxxxx	Starts execution for enabled threads.

Table 3.1: Controller Write Operation

The controller looks at the 3 MS bits from the 18 LS bits of the address to find which operation to execute. The controller doesn't utilize the WSTRB signal of the W channel to simplify its implementation. So depending on those MS bits we have the following:

- The MS bit is 0. If the MS bit is 0 then the controller must write the two instructions (which come from the data of the W channel) in the instruction memory of the processor. The controller writes two instructions because each instruction is 32-bit and the data bus is 64-bit. The controller creates the address where it will write the instructions in the instruction memory from the *a* bits Table 3.1 shows. The controller finishes the operation when the processor doesn't need to read from the instruction memory. When the operation finishes it asserts the Wr_Op_Done signal.

- The 2 MS bits are equal to 10. In that case, the controller initializes the PC value of a thread in the PC File using the data the W channel provides. The 8 *t* bits Table 3.1 shows, create the thread for which we initialize the PC value. Since there are 8 bits we can initialize any of the 256 threads. The operation completes and asserts the Wr_Op_Done signal when the processor pipeline doesn't want to write in the PC file.

- The 3 MS bits are equal to 110. For this combination of MS bits, we enable which threads we want to run next time a start operation initiates. Since the bus is 64-bit and we have 256 threads, we split the threads into 4 groups of 64 threads each. The data this operation uses is a mask of which threads should be active and inactive in a group. The 2 *g* bits, which Table 3.1 shows, are the group for which the data correspond. The controller asserts the Wr_Op_Done signal as there is no direct interference with the processor.

- The 3 MS bits are equal to 111. When this combination of MS bits happens, it signifies a thread start operation. In the thread start operation, the processor can start to enter all the enabled threads from the 4 groups into the pipeline and execute the code in the instruction memory. The controller uses the 8 LS bits of the data to set the MAX_THREAD variable and ignores the rest. The controller also resets the performance registers when a thread start operation occurs and asserts the Wr_Op_Done signal.

3.3.2 Read Operations

The read requests work the same way as the writes, as they happen 1 at a time and use an FSM. Figure 3.17 shows the FSM the controller uses for read operations.

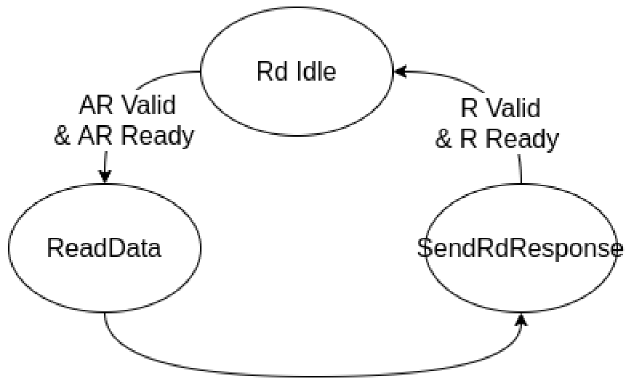


Figure 3.17: Controller Read FSM

The controller starts from the *RdIdle* state, where it awaits an address through the AR Channel. After it receives the address, it advances to the *ReadData* state, where it prepares the appropriate read data the operation wants depending on the 18 LS bits of the address. Table 3.2 summarizes the possible read operations.

Address 18 LS bits	Read
0xxxxxxxxxxxxggxxx	Threads running in a group.
10xxxxxxxxxxxxxxxx	Cycles since last thread start.
110xxxxxxxxxiiixxx	Issues since last thread start.

Table 3.2: Controller Read Operation

We use the same scheme that writes use of checking the 3 MS bits from the 18LS address bits to split the different read operations. The different read operations depending on the MS bits that Table 3.2 shows are the following:

- When the MS bit is equal to 0. Then we read how many threads still run in the group the 2 *g* bits (that Table 3.2shows) describe. The operation shows how many threads still run since the last thread start operation. The return data is a mask of 64-bit and has asserted bits for the threads that still run.
- When the 2 MS bits are equal to 10. Then we read the elapsed processor cycles since the last thread start operation. The counter stops counting when there aren't any running threads left.
- When the 3 MS bits are equal to 110. Then we read the number of instructions issued since the last thread start operation. The 3 *i* bits define one of the following cases:

- If iii is equal to 000. Then we want to read the total instructions the processor issued.
- If iii is equal to 001. Then we want to read how many times the processor scheduled 1 instruction.
- If iii is equal to 010. Then we want to read how many times the processor scheduled 2 instructions.
- If iii is equal to 011. Then we want to read how many times the processor scheduled 3 instructions.
- If iii is equal to 100. Then we want to read how many times the processor scheduled 4 instructions.

After the controller creates the data the operation wants, it moves to the *SendRdResponse* state. In that state, it sends back the data and waits until the one that requested them receives them. After the communicator receives the data the controller returns to the *RdIdle* state and waits for new read requests.

On both Table 3.1 and Table 3.2, when an address bit has an x value, it means the controller ignores it and doesn't considerate it for the operation it will execute. That way we simplify the implementation of the controller.

Chapter 4

Evaluation

In this chapter, we present the evaluation of Matzic . More specifically, the FPGA resources it requires and the performance of the core.

4.1 Evaluation Environment and Tools

The evaluation setup for Matzic is to use it as an accelerator to a host processor in both the simulation and real hardware environment. This section describes the tools we use for implementing and evaluating the core of this thesis. This section also describes the environments under which we use the tools and evaluate the processor.

4.1.1 Tools

This thesis uses the following tools for its implementation and evaluation:

- Software
  - Xilinx Vivado 2020.2
  - Linux
  - RISC-V GNU Toolchain
- Hardware
  - Ariane RISC-V CPU [12]
  - Xilinx Kintex Evaluation Board

4.1.1.1 Software: Xilinx Vivado 2020.2

The Xilinx Vivado 2020.2 design suite is the primary software we use for the design, simulation, and implementation of the core. Vivado also downloads and programs the design to the FPGA. Moreover, Xilinx also provides different modules we use to debug the real hardware implementation of the core, such as the Integrated Logic Analyzer (ILA) core.

4.1.1.2 Software: Linux

Linux is the OS that runs on the Host machine (Ariane CPU) and provides the base environment for running the applications. It allows the evaluation system to be as close to a real one as possible. It also provides ssh support, an NFS system that improves the collection of accurate metrics for the processor.

4.1.1.3 Software: RISC-V GNU Toolchain

The RISC-V GNU Toolchain provides the GCC C compiler. The GCC compiler is the basic compiler we use to compile all the code that runs on Matzic . Since it is an open-source compiler, it allows someone to modify it and add extra functionality to serve particular needs. To do this, they have to create a machine description file, which describes the pipeline of the machine so that the compiler does a better job when it optimizes the machine code. The compiler also provides the ability to add options in the compilation that can suit the needs of a RISC-V machine, but we don't use this feature. We don't implement it because it is beyond the scope of this work. However, we modify the compiler's settings so that the compiled code only uses half the registers the RISC-V architecture provides.

4.1.1.4 Hardware: Ariane RISC-V CPU

The Ariane RISC-V CPU [12] is a 64-bit, single issue, in-order RISC-V core with some limited out-of-order execution capabilities. It supports hardware multiply/divide, atomic memory operations, and contains an IEEE-compliant floating-point unit (FPU). It has 3 privilege levels with which it is allowed to run a Unix-like operating system such as Linux. The modes are the M, S, and U modes. Figure 4.1 shows an overview of the core. The figure also shows that Ariane has branch prediction by having a branch target buffer (BTB) and a branch history table (BHT). It also has a page-table walker (PTW) and translation look-aside buffer (TLB) to support fast memory translations for memory operations.

4.1.1.5 Hardware: Xilinx Kintex Evaluation Board

The FPGA we use to test our design is the xcku040-ffva1156-2-e, part of the Xilinx Kintex UltraScale KCU105 Evaluation Kit. The FPGA comes with $484,800$ LUTs, $242,400$ Flip-Flops, and 21.1 Mb Block RAM, as well as other resources that Figure 4.2 shows. The evaluation board, in addition to the FPGA, also contains other resources that we use in this thesis, such as:

- A $2GB$ DDR4 64-bit DRAM memory. From the 2 GB, we use the 1st as the memory for Matzic , and the Debian Linux kernel that runs on the Ariane CPU uses the 2nd.

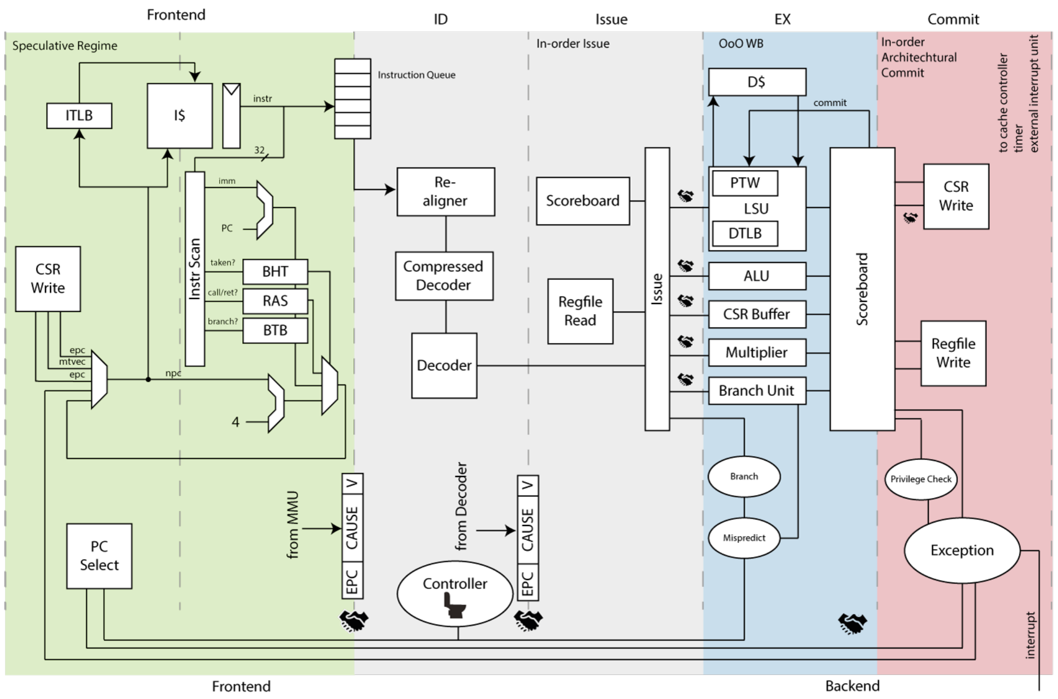
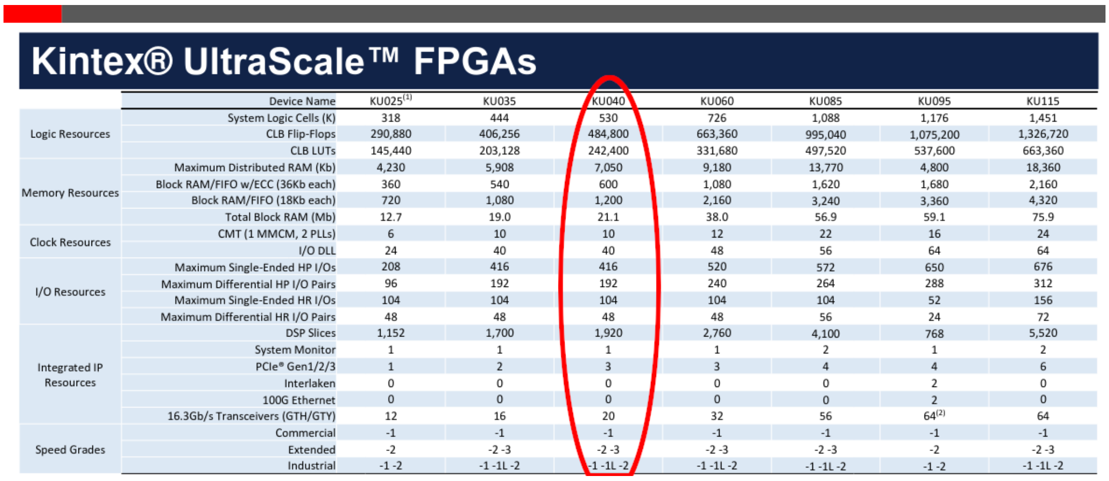


Figure 4.1: Ariane CPU pipeline overview. source:[12]



Kintex® UltraScale™ FPGAs

	Device Name	KU025[1]	KU035	KU040	KU060	KU085	KU095	KU115
Logic Resources	System Logic Cells (K)	318	444	530	726	1,088	1,176	1,451
	CLB Flip-Flops	290,880	406,256	484,800	663,360	995,040	1,075,200	1,326,720
	CLB LUTs	145,440	203,128	242,400	331,680	497,520	537,600	663,360
Memory Resources	Maximum Distributed RAM (Kb)	4,230	5,908	7,050	9,180	13,770	4,800	18,360
	Block RAM/FIFO w/ECC (36Kb each)	360	540	600	1,080	1,620	1,680	2,160
	Block RAM/FIFO (18Kb each)	720	1,080	1,200	2,160	3,240	3,360	4,320
	Total Block RAM (Mb)	12.7	19.0	21.1	38.0	56.9	59.1	75.9
Clock Resources	CMT (1 MMCM, 2 PLLs)	6	10	10	12	22	16	24
	I/O DLL	24	40	40	48	56	64	64
I/O Resources	Maximum Single-Ended HP I/Os	208	416	416	520	572	650	676
	Maximum Differential HP I/O Pairs	96	192	192	240	264	288	312
	Maximum Single-Ended HR I/Os	104	104	104	104	104	52	156
	Maximum Differential HR I/O Pairs	48	48	48	48	56	24	72
Integrated IP Resources	DSP Slices	1,152	1,700	1,920	2,760	4,100	768	5,520
	System Monitor	1	1	1	1	2	1	2
	PCIe® Gen1/2/3	1	2	3	3	4	4	6
	Interlaken	0	0	0	0	0	2	0
	100G Ethernet	0	0	0	0	0	2	0
	16.3Gb/s Transceivers (GTH/GTY)	12	16	20	32	56	64[2]	64
Speed Grades	Commercial	-1	-1	-1	-1	-1	-1	-1
	Extended	-2	-2 -3	-2 -3	-2 -3	-2 -3	-2	-2 -3
	Industrial	-1 -2	-1 -1L -2	-1 -1L -2	-1 -1L -2	-1 -1L -2	-1 -2	-1 -1L -2

Figure 4.2: Xilinx Kintex FPGAs infos. source:[4]

- An Ethernet port that allows the Ariane CPU to have network access for communication.
- An USB-UART Connector, which we use to provide direct access to the Linux kernel on the Ariane CPU. The UART connector provides information about the state of the kernel and allows for communication with the kernel when

there is no network.

- An USB-JTAG connector that we use to download our design in the FPGA.
- A PCIe connector that we use to boot and start the Linux Kernel in the Ariane CPU.

For more information about the board evaluation board you can refer to source [3]

4.1.2 Simulation Environment

In the simulation environment, in which we simulate and fine-tune the core, we use a simple RISC-V 64IM simulation processor to act as the host processor. In that setup, both processors have access to the same memory and use physical addresses when entering that memory. The host (simulation) core completes 1 instruction per cycle, in most cases, except when it communicates with the controller of Matzic
.

We use 2 matrix multiply programs to test and tune the core in the simulation. The first program (mm32-fp32) multiplies 2 32 $\times$ 32 32-bit float matrices, while the second (mm32-int64) multiplies 2 32 $\times$ 32 64-bit integer matrices. Each of the programs respectively executes 477,428 and 574,528 instructions.

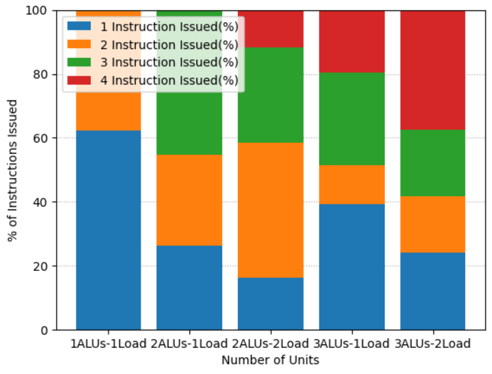
During the simulation phase, we try to increase the average number of instructions for the core issues each time a valid thread is in the Decode stage of the pipeline. For that reason, we measure the hazards that block the issue of all 4 instructions. We define the hazards that can happen as one of the following:

1. Control Hazards: The problem this hazard presents is the change in the normal flow of execution. It occurs if one of the 3 first instructions fetched is a branch, jump, barrier, or ecall. When that happens we don't schedule all the instructions after the first such instruction, as it can lead to an erroneous result.
2. Memory Hazards: Memory hazards happen when there are more than one instructions that want to access the memory, and at least one wants to write in the memory.
3. Read-After-Write Hazards: These happen because, in the batch of instruction currently present at the Decode stage for issue, an instruction produces the result that another one uses as an operand.
4. Structural Hazards: These hazards occur when we can't schedule other instructions because there are no available units that can serve them.

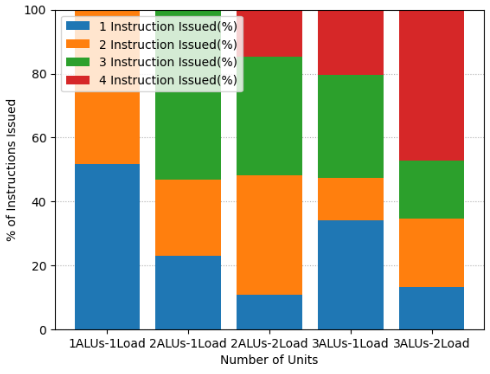
From the different kinds of hazards, we identify that structural hazards are the most common. After identifying that the structural hazards are the most common, we then measure which of the execution units are the ones that the instructions (that we fail to schedule) most request. Finally, we find that the most common

Program	1ALUs 1Load	2ALUS 1Load	2ALUs 2Load	3ALUs 1Load	3ALUs 2Load
mm32-fp32	346,688	217,856	201,440	208,512	175,712
mm32-int64	387,616	249,632	224,992	240,320	192,128

Table 4.1: Total issues for each simulation program for different combination of available units



(a) mm32-fp32



(b) mm32-int64

Figure 4.3: Percent of how many instructions where issued during each Issue for different combination of available units

requested units are the ALU and the Load unit. After that, we increase the number of those units and see how it affects our metrics. The different setups we try are with 1 ALU and 1 Load (the starting setup), 2 ALUs and 1 Load, 2 ALUs and 2 Loads, 3 ALUs and 1 Load, and 3 ALUs and 2 Loads.

In Table 4.1, we can see how many issues each setup needs to execute each program. In Figure 4.3, we show the percent where an issue scheduled 1 instruction, 2 instructions, 3 instructions, or 4 instructions from the total issues each evaluation setup makes.

From the plots in Figure 4.3, we see that we can schedule more instructions with the setup this thesis uses, which is with 3 ALUs and 2 Load Units. Following, we show the reason why that happens by studying the hazards that occur. In Table 4.2 and Figure 4.4, we see the total number of hazards that occurred based on the setup of available ALU and Load units and continue the analysis with the plots in Figure 4.5.

Figure 4.5 shows the percent of each of the 4 types of hazards as a percent based on the unit setups available. Because we classify every hazard as one of the 4 previously described types above, all those percentages sum up to 100% of the total hazards Table 4.2 shows.

In the Figure, we see that in the basic setup (1 ALU and 1 Load unit), the

Program	1ALUs 1Load	2ALUS 1Load	2ALUs 2Load	3ALUs 1Load	3ALUs 2Load
mm32-fp32	346,688	217,760	177,600	167,424	109,856
mm32-int64	387,616	249,536	191,936	191,072	101,728

Table 4.2: Total hazards for each simulation program for different combination of available units

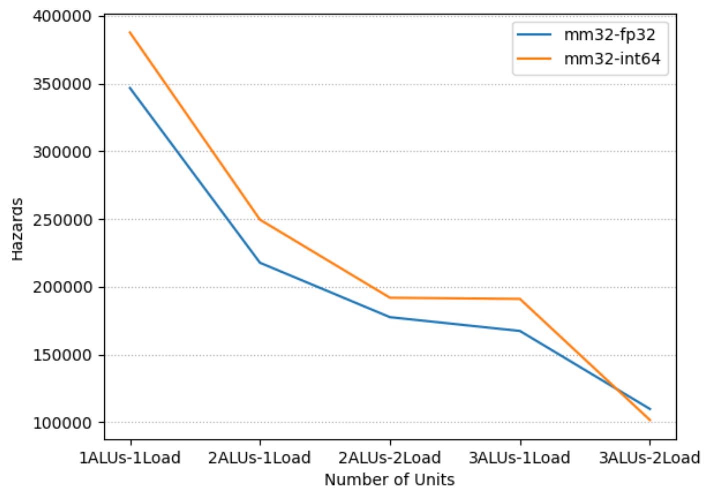
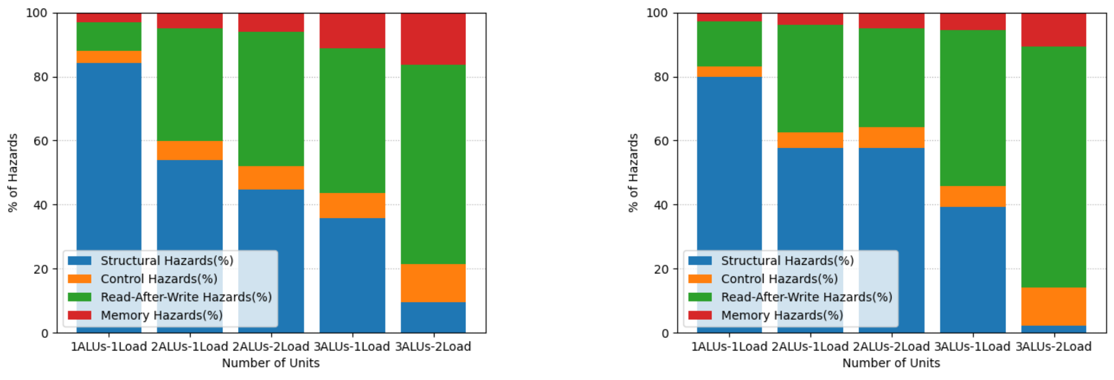


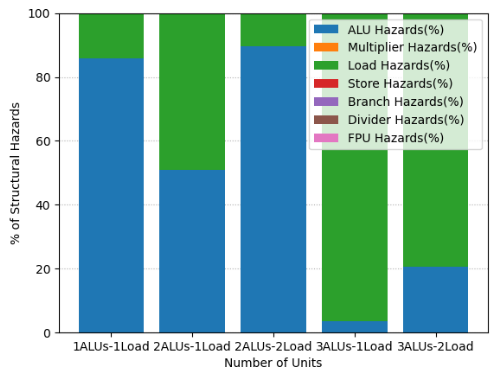
Figure 4.4: Plot showing the total hazards that happened for different combination of available units.



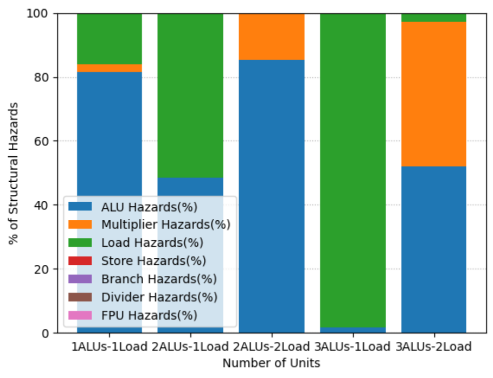
(a) mm32-fp32

(b) mm32-int64

Figure 4.5: Percent of each type of hazard from total hazards, for different combination of available units.



(a) mm32-fp32



(b) mm32-int64

Figure 4.6: Percent of each unit responsible for the structural hazards, for different combination of available units.

largest percentage of the hazards that occur are of the structural type and is more than 80%. As we add units and observe the results of the other setups, we see that the structural hazards reduce to a low percentage, and the majority of hazards that remain are of the Read-after-Write type. We expect the compiler to try to minimize and address that type of hazard during compilation and optimization. We don't address those hazards another way, as this type of hazard defines a true dependency between instructions, and we can't execute them in parallel.

The following plot, in Figure 4.6, depicts which units are the ones that cause the structural hazards. The plot shows the values as a percent of all the structural hazards.

As per the Figure, most structural hazards happen due to unavailable ALUs and Load units. For that reason, the evaluation setups increase the available number of ALU and Load Units. After testing the different setups, we conclude that 3 ALUs and 2 Load Units are enough to reduce the number of hazards to a satisfying degree and increase the average number of instructions the Decode stage schedules each time an issue happens. In Figure 4.7, we show how the average instructions issued per issue (IPI) change per the different setups we evaluate.

In the Figure, we see the IPI double from the original setup of 1 ALU and 1 Load Unit.

We try arrays of different sizes and setups during the simulation to reach as close as possible to use all the 256 threads. Due to the limitations of the simulator and the time a run takes to complete once we use 128 threads, we stopped trying to reach the full 256 threads in the simulation. But from the different setups, there is no change in the ratios we use for the evaluation after the 32 threads. Also, we focused on improving the IPI because when all the threads are active and there are no stalls in the pipeline, IPI will be close to the instructions per cycle (IPC) metric, which measures a processor's performance.

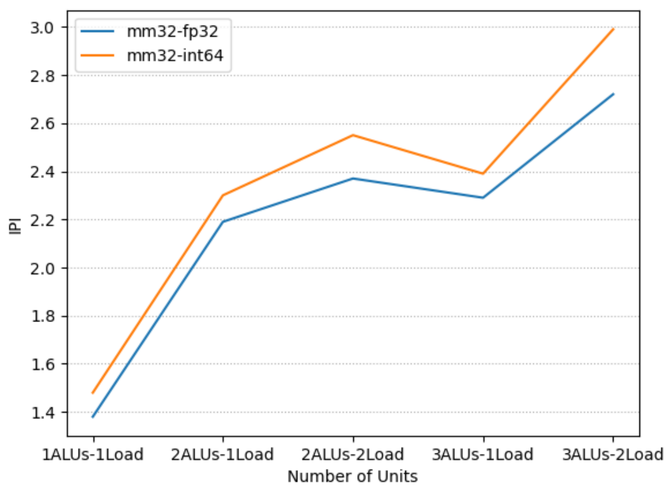


Figure 4.7: Average Instructions Issued per Issue (IPI).

4.1.3 Hardware Environment

During the evaluation of the hardware environment, we use the Ariane CPU as the host processor. Furthermore, the host also runs a *Debian* Linux operating system kernel (OS). The OS in the host makes the evaluation as close to real-world conditions as possible because the application couldn't access the memory directly. Each memory access the application makes uses a virtual address, while the accelerator can only use physical addresses. To solve those issues, we develop a user-level library in C language, that we call "Muda", that we use to compile and run the applications using both cores. Table 4.3 shows the functions that the user-level library has.

Of the functions the table describes, there are three that also have arguments. Tables 4.4, 4.5 and 4.6 show the functions we implement and the specific arguments that launch_kernel, mudaMalloc and mudaMemcpy receive respectively. The user-level library also provides some directives we use for the functions that will run on Matzic and not the host processor. Those directives are the *__global__* and *__device__* directives. If a function that will run on Matzic doesn't have one of the directives, then the program behavior is undefined, and anything can happen.

In the hardware implementation, we also use a delayer for the requests that go to the DRAM memory. The delayer introduces latency to the memory requests and allows the hardware evaluation to be even closer to real-world scenarios. Figure 4.8 presents a diagram of the hardware setup together with the used components.

Function Name	# Args	Description
mudaInit	0	Initializes the environment to allow the use of Matzic .
launch_kernel	8	This function is responsible for setting up Matzic to run a code for the device.
mudaMalloc	2	It allocates memory that is accesable by Matzic .
mudaMemcpy	4	It transfers data between Matzic and the host processor.
mudaDestroy	0	Unsets the environment that was initialized by mudaInit.
get_thread	0	Returns the ID of the hardware thread that executed it.(Only used in code for Matzic)
get_deployed_num	0	Returns the number of threads that were launched on Matzic .(Only used in code for Matzic)
barrier	0	Used to set a rendezvous point for all threads running on Matzic to meet before code execution continues. (Only used in code for Matzic)

Table 4.3: Functions provided by user level library in order to run.

Argument	Type	Description
threads_no	unsigned int	The number of threads that should run on Matzic . They map to the ids belonging in range [0,threads_no).
func	pointer	It is the pointer for the function to run on Matzic . The function must have the *__global__* attribute in order to run on Matzic .
arg0, arg1, ... arg5	unsigned int / pointer	Those are the arguments the function that will run on Matzic will use. We limited to only six arguments and them either being pointers or integers, but it is possible for more arguments to be used if encapsulated and transferred as a struct in the C language. Also pointers passed as arguments have to be gotten through the mudaMalloc function.

Table 4.4: Arguments of launch_kernel function.

4.2 Resource Utilization

This section analyzes the hardware resources we use during the implementation of Matzic . The resources we use are:

Argument	Type	Description
devPtr	pointer	It is the location in memory where malloc will write the address of the allocated space that can be used by Matzic .
nbytes	unsigned int	The amount of space that should be alocated.

Table 4.5: Arguments of mudaMalloc function.

Argument	Type	Description
dst	pointer	The address to which the data will be copied to.
src	pointer	The address from where the data will be copied from.
nbytes	unsigned int	The amount bytes to copy from src.
kind	mudaMemcpyKind	One of the two following value 1. HostToDevice: To copy from host memory to memory compatible for Matzic . 2. DeviceToHost. To copy from memory compatible for Matzic to host memory. .

Table 4.6: Arguments of mudaMemcpy function.

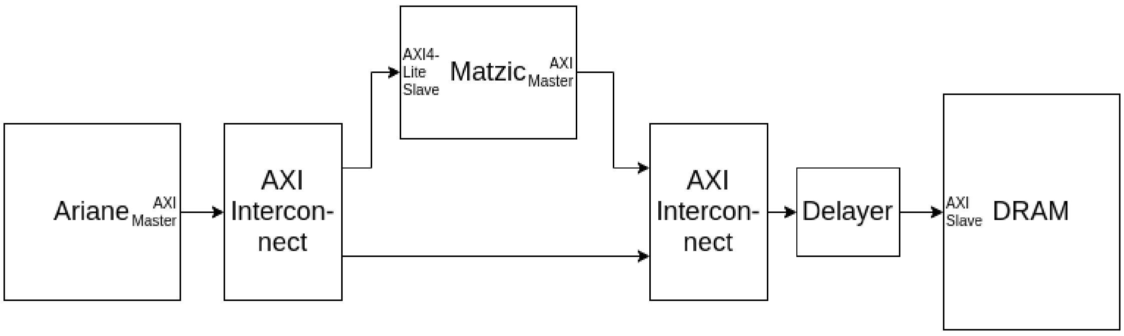


Figure 4.8: Hardware setup where evaluation took place.

- LookUp Tables (LUT): Basic component of a FPGA that simulates logic functions of up to 6 inputs.
- Registers (Regs)
- Carry Lookahead (CARRY8): fast lookahead carry logic allows to perform fast arithmetic addition and subtraction.
- F7 Mux: Multiplexer combining LUTSs to form logic functions of up to 13

Stage	LUTs	Regs	CARRY8	F7	F8	BRAM's	DSP
Controller	907	1,067	83	0	0	0	0
Thread Scheduler	1,247	523	23	102	50	0	0
PC Fetch	200	0	0	0	0	1	0
Instruction Fetch	196	76	0	0	0	32	0
Decode	831	0	24	1	0	0	0
Operand Fetch	8,541	120	0	2,560	1,280	31	0
Execute	13,455	7,505	284	90	1	6	27
Commit	2,472	1,466	0	314	17	8	0
Matzic (Total)	29,941	13,570	493	3,067	1,348	81	27

Table 4.7: FPGA Resources used.

inputs,

- F8 Mux: Multiplexer combining F7 Muxes to form logic functions of up to 27 inputs.
- Block RAM tile(BRAM)
- Digital Signal processors (DSP)

In Table 4.7, we show the resources each pipeline stage utilizes and the resources that the Controller and the Thread Scheduler utilize.

At the end of Table 4.7 is the total number of resources that Matzic utilizes. These numbers are not the sum of all the values above because of the pipeline registers, logic, and other sources that exist between pipeline stages. The reason we do not include this info in a separate line is that we do not consider those values to have a noticeable impact on the main resource usage.

In Figure 4.9 we show the percent of utilization that each of the entries in Table 4.7 use. The plot also contains the percent of resources that exist between the stages of the pipeline. From the plot, we also see that the Execute stage of the pipeline has the most dominant percent for LUTs, Regs, CARRY8, and DSPs, which the instructions use for execution. In the plot, we also see that a huge percentage of resources go to the Operand Fetch stage, but most of those resources implement the multiplexors at the outputs of the register files. Moreover, we see that the Operand Fetch stage also uses a large amount of BRAMs because of the volume of registers we have that we implement as memory.

To evaluate the utilization of Matzic we compare it to the utilization of the Ariane CPU. In Table 4.8 we show the resources that Ariane utilizes.

When we compare the results, we conclude Matzic has 22.65% fewer LUTs, 56.90% fewer Regs, and 12.17% less CARRY8s, but it has 40.30% more F7 muxes, 3.17× more F7 muxes and 2.18× more BRAMs.

Finally Matzic runs at 50 MHz on the FPGA, which is the same speed as Ariane. The critical path of the core is in the FPU implementation.

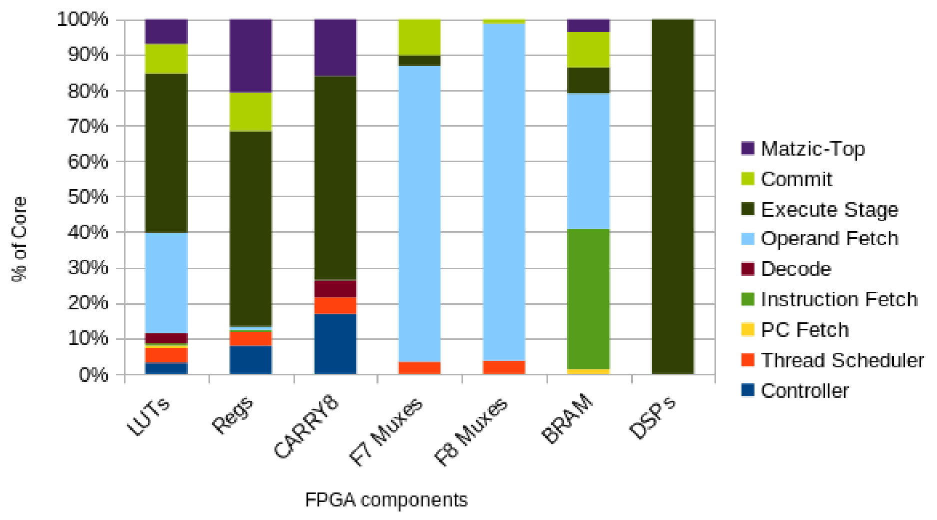


Figure 4.9: Stage Core Utilization

Stage	LUTS	Regs	CARRY8	F7	F8	BRAM's	DSP
Ariane (Total)	36,724	21,291	553	2,186	425	37	27

Table 4.8: Ariane CPU used resources.

4.3 Performance Evaluation

This section contains the performance evaluation for the hardware implementation of Matzic . We evaluate the performance of Matzic by comparing it to the execution of a program run on Ariane. The programs that we use for the evaluation are the following:

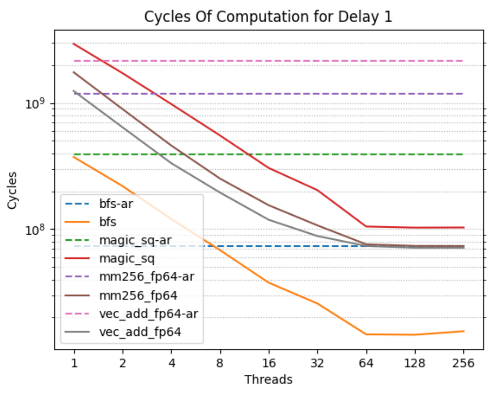
1. bfs: This is a breadth-first search program that is a port from the Rodinia Benchmark suite[6]. We execute the program with a graph with $65,536$ nodes.

2. magic_sq: Program that counts how many solutions a magic square with missing elements has. The test executes on a 3×3 magic square with 9 missing elements. It has to check $362,880$ permutations of the missing elements.

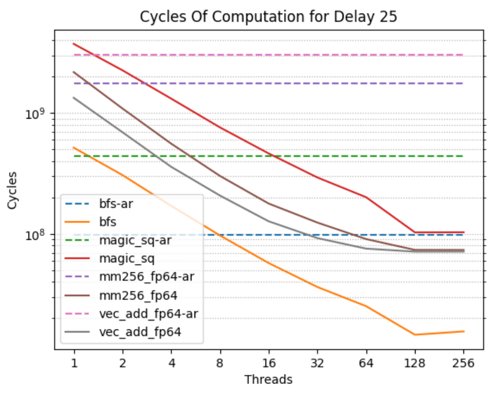
3. mm256_fp64: A matrix multiply program between 2 256×256 matrices of 64-bit float numbers.

4. vec_add_fp64: A program that initializes and adds 2 vectors with $4,194,304$ each.

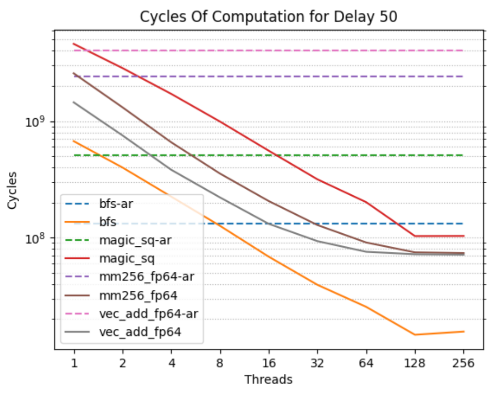
We also implement the programs on Ariane, as well as Matzic . We run the programs on Matzic with 1, 2, 4, 8, 16, 32, 64, 128, and 256. Also, we benchmark



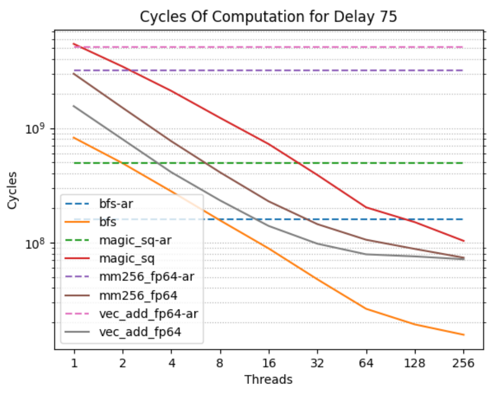
(a) delay 1



(b) delay 25



(c) delay 50



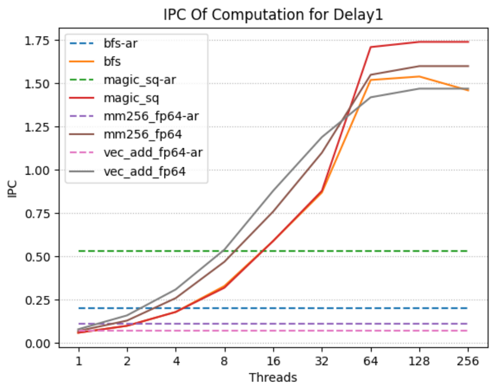
(d) delay 75

Figure 4.10: Cycles of computation of each program

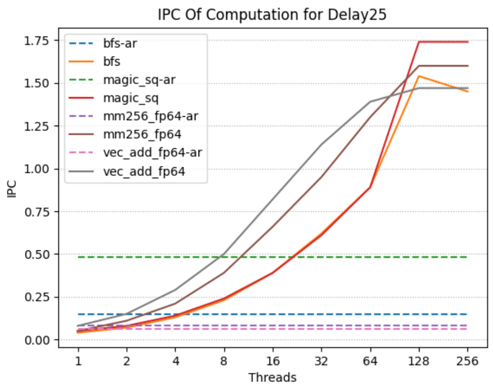
them with different memory latencies that we artificially create with the memory delayer. The delay setups we use are 1, 25, 50, and 75 additional clock cycles delay. The delay starts from 1 clock cycle, which is the performance without introducing latency to the system, and we raise it to 75 clock cycles which is the delay that we expect in a real-world system scenario.

In the plots we show in Figures 4.10 and 4.11, we present the number of clock cycles (cc) and instructions per cycle (IPC) each program takes respectively. The Figures show how the clock cycles and IPC develop as the number of threads increases and with the different delay setups.

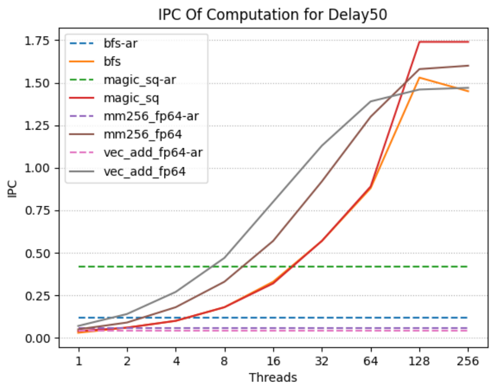
Except for the plots of clock cycles a run needs for the execution and the IPC of the programs, we also present in Figures 4.12 and 4.13 the speedup, or slowdown, the execution has with the different setups with the Ariane execution as the baseline. In Figure 4.12, we show the speedup to the cycles the program needs for the computation as we use more threads for the computation, and we show how the speedup also changes between the different delays. From these results, we like



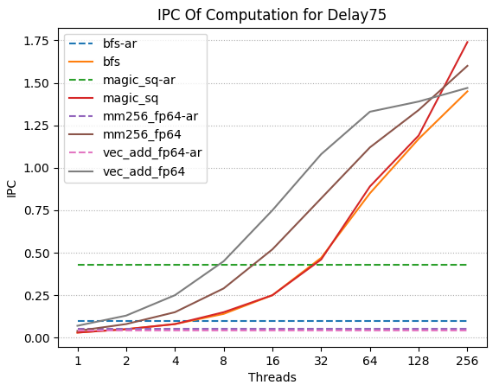
(a) delay 1



(b) delay 25



(c) delay 50

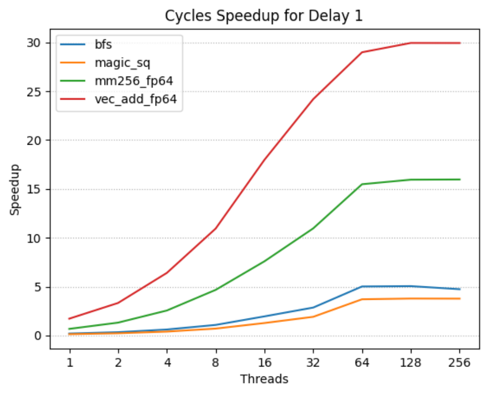


(d) delay 75

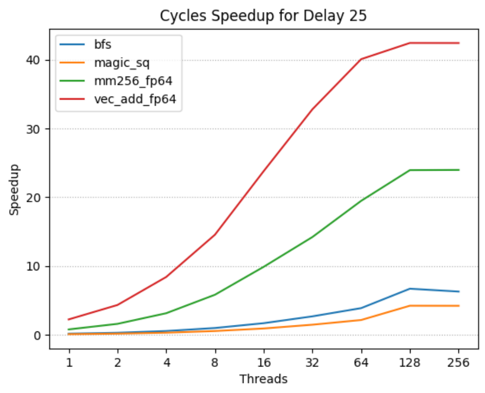
Figure 4.11: IPC of programs executed

to highlight the greater than 70× speedup we get for the vec_add_fp64 program for 256 threads and a delay of 75. Figure 4.13 shows the same things that Figure 4.12 shows but for the IPC. The highlight of the IPC graphs is the 37× speedup to IPC with the setup of 256 threads and 75 cycles delay.

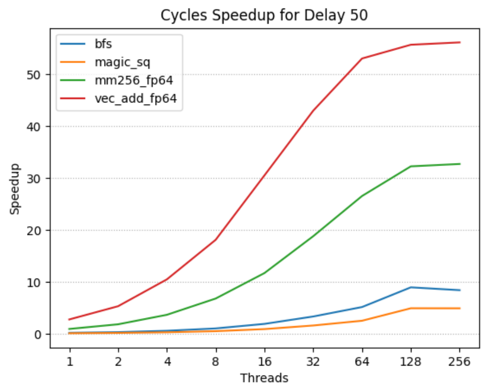
Figures 4.14 and 4.15 show how the cycles and IPC develop as the memory delay increases for the different execution setups of using different numbers of threads on Matzic and running only on Ariane. The highlight of the plots is that the number of cycles and the IPC of the programs that run on all 256 threads remains the same, despite the access latency to the memory increasing.



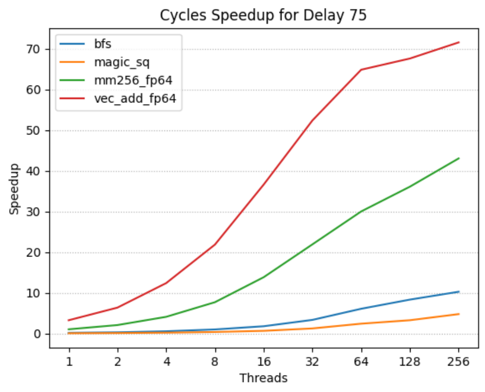
(a) delay 1



(b) delay 25

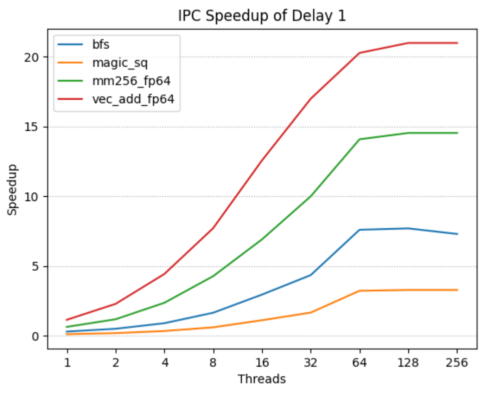


(c) delay 50

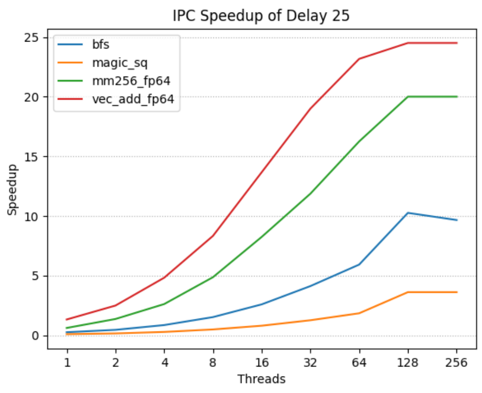


(d) delay 75

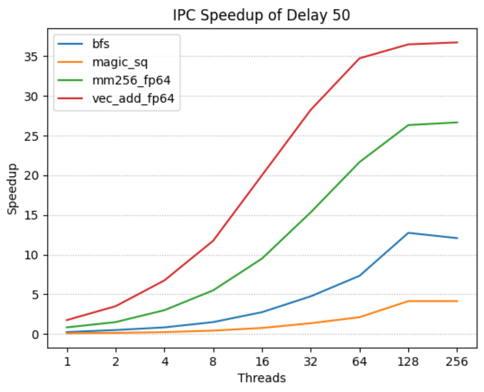
Figure 4.12: Cycles of computation of each program



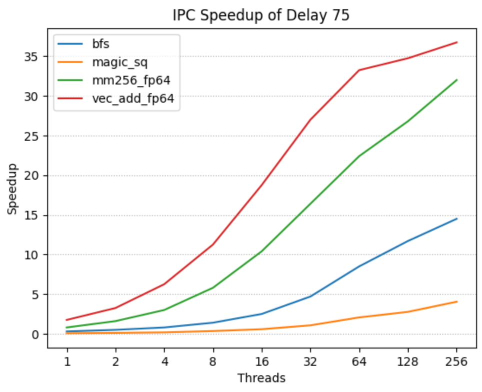
(a) delay 1



(b) delay 25

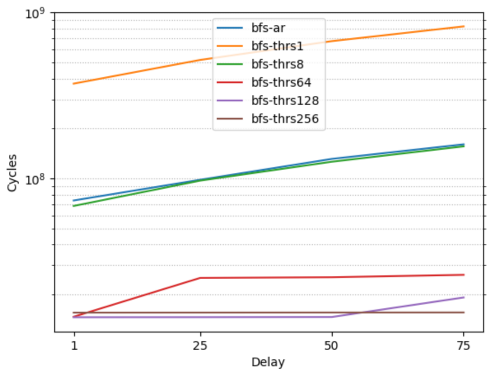


(c) delay 50

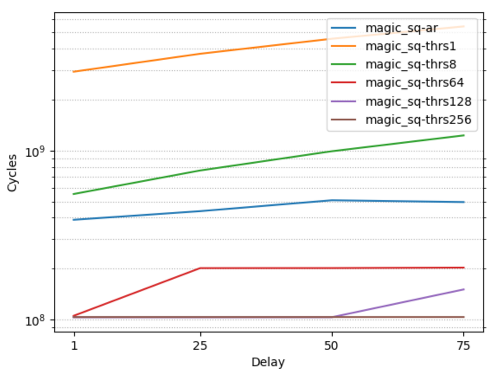


(d) delay 75

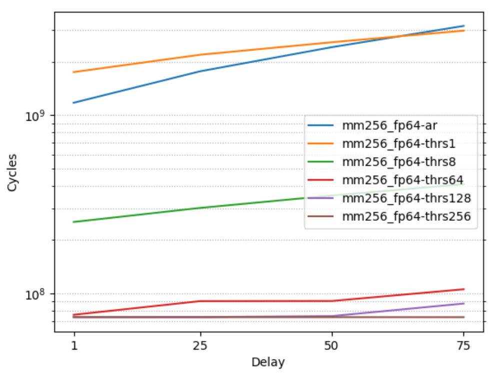
Figure 4.13: IPC of programs executed



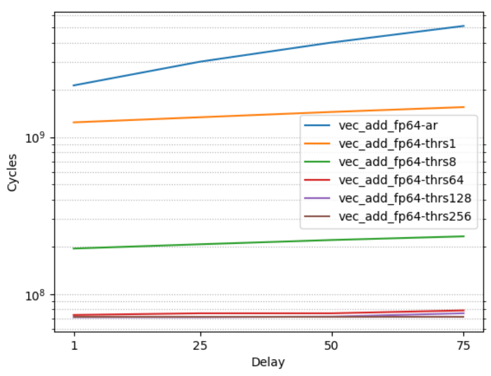
(a) bfs



(b) magic square

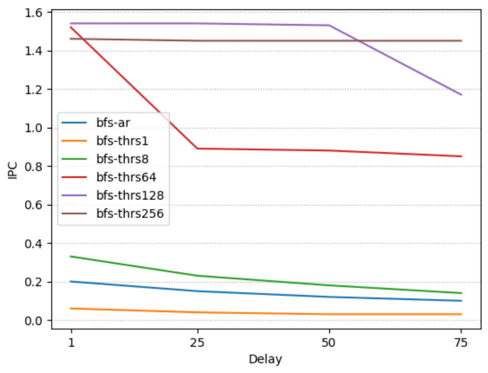


(c) matrix multiply

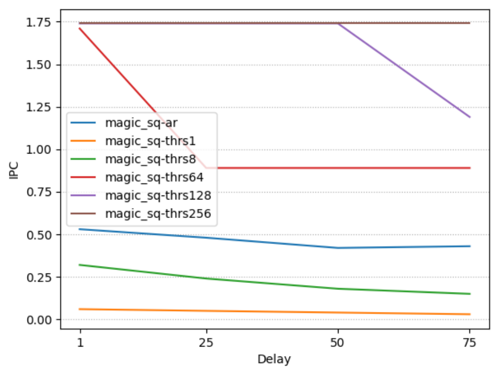


(d) vector add

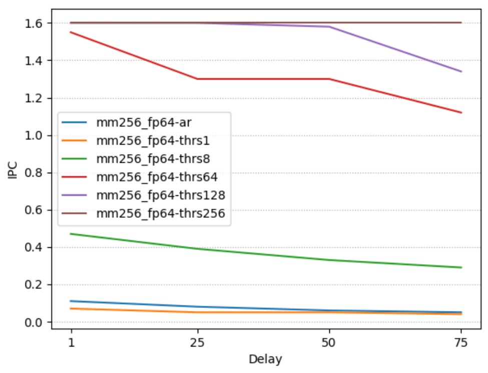
Figure 4.14: How delay affected the cycles of each program run



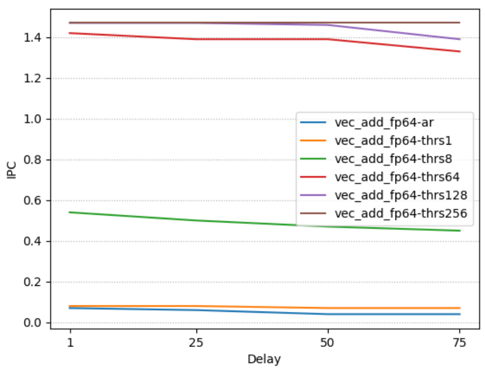
(a) bfs



(b) magic square



(c) matrix multiply



(d) vector add

Figure 4.15: How delay affected the IPC of each program run

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis and presents potential future work to advance this research topic.

5.1 Summary

In this thesis, we presented a superscalar massively multi-threaded RISC-V 64EMFD processor named Matzic , which can issue up to 4 instructions per cycle and is tolerant to high latency events. Matzic also runs along with another host processor in real hardware. The setup of host and Matzic can run in real-world conditions where the host runs a Linux operating system. Finally, the host successfully used Matzic to accelerate computing tasks with a sizeable speedup gain when it utilizes all the threads.

5.2 Future Work

We can develop, explore and improve a lot of different areas in this thesis. Specifically, the areas we would like to improve and explore are:

1. To reduce the hardware implementation resource usage to become more efficient.

2. Use a form of branch prediction to reduce the impact of control hazards and increase the IPI metric.

3. Issue load and store instructions together when the addresses don't conflict and increase the IPI.

4. Implement a better user library to use the resources more efficiently.

5. Increase the optimization of the compiler-generated code for Matzic to reduce the Read-After-Write hazards.

6. Use an IOMMU to virtualize the memory operations of Matzic and allow it to have a common address space with the host processor, thus using the same memory.

7. Develop a runtime environment that will allow OpenCL programs to use Matzic and also port it for the One-API environment, as the processor of this thesis is compatible to be used for programming models that take advantage of a large number of available threads.

Bibliography

[1] AMBA® AXI™ and ACE™ ProtocolSpecification.

[2] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.

[3] Xilinx Kintex UltraScale FPGA KCU105 Evaluation Kit.

[4] Xilinx UltraScale FPGA Product Tabless and Product Selection Guide.

[5] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[7] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[8] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(4):774–787, 2020.

[9] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[10] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[11] Theo Ungerer, Borut Robic, and Jurij Silc. Multithreaded Processors. *Comput. J.*, 45:320–348, 03 2002.

[12] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.