

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Designing an Instruction Set Based Coarse Grain Accelerator

Gonalo de Albuquerque Ferreira

Mestrado Integrado em Engenharia Eletrotcnica e de Computadores

Advisor: Dr. Nuno Paulino

Co-advisor: Dr. Joo Canas Ferreira

31st of March, 2024

Resumo

A sociedade moderna de hoje em dia cada vez mais utiliza dispositivos inteligentes, aumentando a sua dependência nos serviços digitais, onde dados são gerados a um ritmo cada vez maior. Por sua vez, o processamento dos mesmos também está a aumentar, o que fez com que a computação transitasse mais para os dispositivos no limite da rede (edge computing). No entanto, os CPUs genéricos modernos estão a alcançar os limites da lei de Moore, tornando-os menos adequados para atingir as metas de eficiência na computação de dados em larga escala, como machine learning ou processamento de vídeo e imagem. Por isso, a pesquisa em arquiteturas de hardware alternativas tem vindo a ressurgir, como a de Coarse-Grain-Reconfigurable-Arrays (**CGRAs**), uma arquitetura que se tem apresentado em muitas configurações e designs diferentes, com o objetivo de melhorar o escalonamento do desempenho. Porém, não há um método standard de como abstracionar um CGRA de um processador hóspede, dado o vasto espaço de implementações em várias configurações diferentes, cada uma com o seu tipo de camada de abstração e de conjunto de instruções, fragmentando o espaço, e aumentando a complexidade em adotar CGRAs nos processadores modernos. O presente trabalho tem como alvo a pesquisa de como um conjunto de instruções para um CGRA poderá ser desenvolvido, com o objetivo de abstracionar as especificidades do design para um desenvolvedor, permitindo o desenvolvimento de aplicações de uma forma agnóstica do hardware. Para isso, criou-se um simulador comportamental com precisão ao nível do ciclo de execução utilizando Java, onde se pode explorar de uma forma visual arquiteturas diferentes de CGRA e como interagir e configurar o mesmo utilizando instruções assembly.

Abstract

Nowadays modern society is increasingly utilizing smart devices, increasing its dependence on digital services, constantly producing data at a bigger rate. In turn, data processing is also increasing, which made computation shift more to devices on the edge of networks. However, modern general-purpose Central Processing Units (CPUs) are reaching the limits of Moore's law, becoming less adequate for fulfilling efficiency requirements on large-dimension data workloads, such as machine learning and image/video processing. Therefore, research on alternative hardware architectures has been re-surging, such as CGRAs, an architecture that has been presented in many different configurations and designs, aiming to further improve performance scaling. However, there is no single standardized method to abstract away a CGRA to a host processor, since there are many implementations in many different configurations, each with their abstraction layer and instruction set, fragmenting the space and increasing the complexity of adopting CGRAs in modern processors. The present work aims to research how can an instruction-set for CGRAs be developed, with the goal of abstracting away the design specificities to the developer, enabling application development in a hardware-agnostic way. For this, we have built a behavioral execution cycle-accurate simulator using Java, where one can explore in a visual way different CGRA architectures and how to interact and configure utilising assembly instructions.

Contents

Resumo	i
Abstract	ii
Abbreviations and Symbols	vii
1 Introduction	1
1.1 Motivation	2
1.2 Objectives	2
1.3 Report Structure	3
2 Background	4
2.1 ASICs	4
2.2 General-purpose CPUs	4
2.3 Reconfigurable Hardware	6
2.4 Heterogeneous Computing	7
2.5 Coarse Grain Accelerators	9
2.6 Microprocessor Architectures	10
2.7 RISC-V	11
2.8 Dataflow Philosophy	14
2.9 Managing Hardware Accelerators	15
2.10 Binary Acceleration	15
3 State of the Art	17
3.1 CGRA Design	17
3.2 Relevant CGRA Designs	18
3.3 RISC-V based Systems	21
3.4 Hardware Accelerator Management	23
3.5 Hardware Emulation	25
3.6 CGRA Design Frameworks	26
4 Proposed Approach	28
4.1 CGRA Design	28
4.2 CGRA-SIM	30
4.2.1 Software Architecture	31
4.2.2 Instruction Decoding	33
4.2.3 Simulator Environment	37

5	Results	40
5.1	Algorithm Implementation	40
6	Conclusion	44
6.1	Future Work	44
A	Verilog Code for ALU Processing Element	46
B	Set of instructions for loading the FIR algorithm onto the CGRA-SIM	48
C	C code for a FIR filter algorithm with set input data	50
	References	51

List of Figures

2.1	Typical CGRA design layout (adapted from [1])	10
3.1	Diagram of MATRIX Functional Unit (adapted from [2])	18
3.2	Diagram of MATRIX CGRA (adapted from [2])	19
3.3	Diagram of Morphosys functional unit (adapted from [3])	19
3.4	Diagram of Morphosys CGRA (adapted from [3])	19
3.5	Diagram of RSPA different architectures (adapted from [4])	20
3.6	Diagram of ADRES schematic (adapted from [5])	20
3.7	Diagram of DRAA CGRA (adapted from [6])	21
4.1	Diagram for CGRA Sim architecture	29
4.2	Flow chart for proposed workflow	30
4.3	Initial CGRA configuration window	37
4.4	Main CGRA-SIM window	38
4.5	CGRA windows for viewing registers, changing PE type, and setting a label	39
5.1	CGRA main window with FIR filter preset	40
5.2	CDFG Diagram for a 4-tap FIR filter	41
5.3	FIR filter results from C algorithm	42
5.4	FIR filter results from CGRA implementation	42

List of Tables

2.1	RISC-V Calling Convention Names	12
2.2	Main RV64G instructions.	13
2.3	RISC-V Instruction Format	14
4.1	CGRA-SIM Instruction opcodes.	34
4.2	ALU element control opcodes.	35
4.3	LSU element control opcodes.	36

Abbreviations and Symbols

FPGA Field-Programmable-Gate-Array

CGRA Coarse-Grain-Reconfigurable-Array

CPU Central Processing Unit

GPU Graphics Processing Unit

ISA Instruction Set Architecture

ASIC Application-Specific-Integrated-Circuit

SIMD Single-Instruction-Multiple-Data

SoC System-on-a-Chip

PE Processing Element

RISC Reduced Instruction Set Computer

ALU Arithmetic Logic Unit

IPC Instructions-per-clock

Chapter 1

Introduction

Technology over the last years has been becoming more and more accessible, thanks to the incessant advancements in manufacturing and cost reducing. This allows a bigger portion of the world human population access to digital devices such as computers and smartphones. On the other hand, society in more developed countries has also been increasing its dependence on digital services and devices. This, in conjugation with the constant increase of the world population, has led to an exponential growth in the generation of data[7], whether it be text messages, photos, audio, videos, and other data. However, modern digital services require processing this data, and it is not feasible to process such amounts of data in centralized data-centers or servers, due to ballooning costs. It is therefore becoming more viable to shift the data processing workload to devices on the edge, such as smartphones, TV boxes, video game consoles, etc., leading to the advent of edge computing. Even so, data processing is an intensive task, specially when edge devices are heavily power and energy-constrained. Traditional methods for scaling performance by increasing power, number of transistors, and clock frequency, are reaching a ceiling[8] as proven by the laws of Moore, Dennard and Amdahl, requiring new approaches to hardware design and development.

Although the concept of utilizing alternative hardware accelerators and co-processors as a means to increase performance is not new [9], only more recently exploration of new vectors for heterogeneous computing and research on alternative hardware architectures has become more prominent, such as Systolic Arrays or Coarse-Grain-Reconfigurable-Arrays (CGRAs), which is the focus of this paper. However, their mass implementation is still in a very early phase, due to several inefficiencies and problems to be solved, one of them being a lack of a standardized abstraction of the CGRA hardware to a host processor. This is due to the vast amount of different design choices, fragmenting the space and increasing the complexity of adopting CGRAs in modern processors. Also, it seems the industry is not leaning towards a closed-source approach. The present work therefore aims to research and develop ways of abstracting away a CGRA by means of a set of simple instructions, with various CGRAs designs taken into consideration, that aim to allow basic communication and interoperability of compute kernels.

The remainder of this section will summarize the current performance issues regarding conventional computing, and present alternative computing paradigms, with special focus given to

CGRAs. This will establish the motivation and objectives which conclude the section.

1.1 Motivation

Modern CPUs have for a long time reached the Dennard scaling limit (around the release of the Pentium 4 CPUs), which means that further improvements to the current technology will be focused in increasing Instructions-per-clock (**IPC**), in architectural changes and new technologies (for example, 3D cache stacking, HBM) instead of node shrinking and increasing clock speed. Modern general-purpose Central Processing Units (**CPUs**) are reaching the limits of Moore's law, becoming less adequate for fulfilling efficiency requirements on large-dimension data workloads, such as machine learning and image/video processing[10]. Therefore, research on newer hardware architectures has been re-surging, such as **CGRAs**, an architecture that has been presented in many different configurations and designs, aiming to further improve performance scaling. However, there is no single standardized method to abstract away a **CGRA** to a host processor, since there are many implementations in many different configurations, each with their abstraction layer and instruction set, fragmenting the space and increasing the complexity of adopting **CGRAs** in modern processors.

Although the tools of the present are much more mature and certain areas such as FPGAs and High-level Synthesis have evolved greatly, hardware design remains a lengthy and complex task. However, it seems that nowadays we are capable of designing and simulating custom hardware with much fewer resources than before, and with the inception of Instruction Set Architectures (**ISAs**) such as RISC-V and easily accessible open-source software, there has been a resurgence of interest this field, with promising results.

1.2 Objectives

As such, the present work aims to implement a custom set of instructions that should be transparent to both the host **CPU** and the coarse grain accelerator, creating an abstraction layer between the main processor and the **CGRA**. This will be initially explored using a custom developed behavioural simulator of a **CGRA**. The custom instructions should be able to configure a **CGRA** and instruct it to run computations. In the co-processor paradigm these instructions must also be able to manage the memory accesses, which might include directly from a processor's cache, or external memory. Instructions should be device and specifications-agnostic, allowing for transparent operation [11].

The following goals have been set:

- Designing a generic **CGRA** for basic proof-of-concept validation, exploring recurring ideas of academia;
- Creating a high-level simulator capable of producing functionally similar results to a physical **CGRA** design;

- Validating the functionality of custom microcode that is capable of abstracting away computation kernels to a CGRA;

The present work aims to implement, therefore, an agnostic abstraction layer through a custom instruction set where the CGRA can abstract away its hardware settings, reducing the fragmentation in the heterogeneous computing space, and open doors for a plethora of different hardware.

Currently, each implementation of a CGRA has its own specific instructions that the compiler must be aware of, in order to offload jobs to the accelerator. These custom instruction sets only work on a specific CGRA. This is especially troublesome when considering that often, optimizing a compiler to make use of a custom instruction set can be hard and could consume more time than designing the CGRA itself. Considering how sparse the literature on this topic is, one of the main goals of this project is to mitigate the fragmentation issue. This means that it should transparently pass information of the workload without prior knowledge of the hardware specifications of the CGRA, which can be extremely varied.

1.3 Report Structure

This chapter has introduced the motivation and objectives of this dissertation. The following sections are organized as follows: Chapter 2 provides a background of the technical aspects. On Chapter 3, it is described the state-of-the-art on CGRA architectures and simulation environments, and related work presented. On Chapter 4 a methodology for work flow and a general idea for a CGRA architecture will be provided. Chapter 5 presents the obtained results and future work. Chapter 6 presents the conclusion to the present work.

Chapter 2

Background

In this chapter, a background will be given for the technical aspects that underpin the current paradigm of heterogeneous computation and data-flow computation offloading to specialized hardware, which were touched upon in the previous section. This regards microprocessor architectures, data-flow computation aspects and reconfigurable hardware. An analysis is made for the topology of a CGRA.

2.1 ASICs

Application-Specific-Integrated-Circuits ([ASICs](#)) are an umbrella term for any purpose-driven processing unit, or in other words, an Application Specific Integrated Circuit, where a digital circuit is designed to compute and solve a specific predefined with the best performance possible, which could be, for example, mining cryptocurrencies [[12](#)], implementing Deep Neural Networks [[13](#)] (Google TPU), processing signals (Digital Signal Processors), and more. The vast majority of these devices belong to the realm of digital circuits, where transistors built with silicon and metal oxides perform logic operations, while some are both analog and digital. However, while being efficient at the workload they were designed for, [ASICs](#) are not typically functionally flexible, providing very low to no capacity for programmability, which means they are very inefficient or even incapable of performing other tasks.

2.2 General-purpose CPUs

It could be said that modern Central Processing Units ([CPUs](#)) are a special kind of [ASIC](#), since their main purpose is running generic workloads, usually provided by a person interfacing through an operative system such as Windows or Linux. [CPUs](#) usually are designed for the latest node size of semiconductor fabrication in integrated circuits, requiring cooperative research and development. During the past decades, companies such as Intel or IBM have presented [CPUs](#) implementing their own Instruction Set Architectures ([ISAs](#)), where nowadays it could be said that x86 is the hegemony of [ISAs](#) in modern [CPUs](#). [CPUs](#) were designed for performing tasks according to the

instructions that they receive and decode. The CPU dynamically decodes the instructions during runtime, responding to inputs from a program. A major distinction can be made in regards to the instruction-set, where there are two big categories - Reduced Instruction Set (RISC) and Complex Instruction Set (CISC).

x86 is a closed-source CISC-based instruction-set architecture developed by Intel, with participation from other corporations and labs. It is the dominant architecture in the CPUs present on most consumer-oriented computers, servers, and also many embedded devices and smaller computers. Although x86 is vastly different now in comparison to when it was created, it remains backwards compatible, which has advantages but also disadvantages. Backwards compatibility allows modern CPUs to run code designed for old CPUs from decades ago, maintaining compatibility with older OSes and software, which is critical for certain environments.

In contrast to x86 are competing Reduced Instruction Set Computer (RISC) ISAs like ARM and RISC-V. ARM is an instruction set architecture originally developed by Acorn for the BBC Micro. It builds upon a completely different philosophy as it is a RISC based ISA that catered to low power, smaller, embedded, portable devices, although nowadays ARM designs are present even in supercomputers. It is very popular in small, low-cost, embedded devices, as noted by their majority presence in smartphones and portable computers, and it is more developer-friendly than a CISC ISA like x86, thanks to a reduced amount of instructions. However, like x86 currently it has a lot of new iterations and improvements which have considerably raised the amount of bloat, creating a somewhat big entry barrier. Nowadays there are many categories of ARM ISAs with many different instructions and designs. Besides, it is also closed-source, and one must obtain licensing from the ARM company before being able to utilise their ISA for hardware design, which can be prohibitive for smaller developers, restraining overall democratization of the sector.

RISC-V is a relatively new ISA (although it is based on concepts that came much earlier, in the 1980s) based on a RISC architecture. It was developed by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman, who started the project in 2010 as part of the Parallel Computing Laboratory at UC Berkeley, together with a new high-level language for hardware design, Chisel[14]. It was a five-year project funded mainly by Intel, Microsoft, and also by the government of California and other companies. The project was released using the Berkeley Software Distribution license, which meant that unlike other ISAs such as x86, ARM, PowerPC, which are commercially protected by patents, it is open source, and no patents were filed, as *"(...) the RISC-V ISA itself does not represent any new technology. The RISC-V ISA is based on computer architecture ideas that date back at least 40 years" [15]*.

What brings the attention of many is that RISC-V's initial adoption and further development is being driven by the RISC-V Foundation, while also having the collaboration of the Linux Foundation. The foundation's directors are trusted by the community, and in exchange collaborate on researching and adopting the new technology. This brings clarity and confidence, which in turn would help bringing in the main proponents in the industry.

Here is a summary of the goals established by the RISC-V designers:

- A completely open ISA that is freely available to academia and industry.

- Avoid “over-architecting” for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology.
- Support specialization of the design for more specific purposes, through adding custom extensions;
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- Support for highly parallel multicore or many core implementations, including heterogeneous multiprocessors.
- Optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.

For the first time, an [ISA](#) is built from the ground up with a mindset of open access, research, collaboration, not only thanks to the foundation’s motives and ethic values, but also in an engineering standpoint, as it was built to be simple, powerful, and scalable. The RISC-V foundation includes members such as Google, Huawei, Seagate, Alibaba, with others such as Intel having announced their support of the RISC-V foundation. The [ISA](#) is actively developed and secured by the RISC-V’s international members, such as Arduino, Imagination, Infineon, IBM, Marvell, Mediatek, Microchip, Nvidia, NXP, Qualcomm, Samsung, Silicon Labs, ST, and many more.

2.3 Reconfigurable Hardware

To curb the problem of ASICs’ lack of programmability, while retaining the benefits of dedicated hardware, designs can be implemented on reconfigurable hardware, with Field-Programmable-Gate-Arrays ([FPGAs](#)) being the most prominent example. They consist of logical units in a grid that can be configured with code to emulate different digital architectures. These logical units usually consist of Lookup Tables, Adders, and D Flip-flops (registers). To program these devices, one can write a description of the digital design to implement in a hardware description language such as Verilog, then use other tools for verification, placement and routing. They are widely used for testing hardware and for educational purposes, and in recent years have been increasingly appearing as an edge device, usually increasing the efficiency of the process when compared to normal CPUs. [FPGAs](#) in a way explore the same concept of Coarse-Grain-Reconfigurable-Arrays ([CGRAs](#)), utilizing the most fine grains possible, where customization is possible to the bit level. This means that there is a lot of overhead in terms of area and routing, which is undesired, and reconfiguration is lengthy and complex. There are systems which combine a general-purpose or an application-specific processor with a reconfigurable logic array. They aim to:

- Improve application re-usability, where the hardware can easily adapt to changes in software.

- Allow for updates to the specification, permitting improvements on performance or other areas.
- Have similar performance to ASICs while not risking becoming obsolete as time passes and technology changes.

The design and manufacture of ASICs can take a very long time (around 18 months), have prohibitive costs (around 300M« million US Dollars for a tape-out of the current leading-edge node), require immense expertise, and no margin for errors, which further reiterates the importance of FPGAs when it comes to product testing, reconfiguration, and debugging. Nowadays, there are FPGAs present in the cloud, such as the AWS F1 or Nimbix, which enables on-the-fly design testing, facilitating the development of new digital architectures.

But, in order to use an FPGA as a coprocessor, or part of a heterogeneous computing design, there has to be an efficient compilation flow from the software to the FPGA. Although High-level synthesis allows a developer to directly port an algorithm to a digital design, if one wants good performance in line with current implementations, he must have expert knowledge in digital architectures, regarding spatial and time-scheduling, optimisation of memory accesses, understanding of parallel computing, etc. An unoptimised merge-sort algorithm can be several orders slower than the norm using high-level synthesis, while with proper optimisation of the points mentioned above, it can run with much improved performance. On these cases, the usual bottleneck is the memory bandwidth, which is not an easy to solve problem.

Recent developments have shown (Microsoft Brainwave)[16] that FPGAs can accelerate Deep Neural Networks, while also showcasing novel research in reduced precision floating point numbers, among other things.

2.4 Heterogeneous Computing

Heterogeneous computing is the term used to refer to a system which employs different types of interconnected processors or architectures with the goal of improving overall efficiency, by exploiting different hardware architectures that cater to different kinds of workloads. These kinds of devices are often regarded as System-on-a-Chip (SoC). As mentioned before, it has been increasingly becoming a hot topic of research, expanding on other vectors of hardware design, in order to improve efficiency and performance while dodging the limitations of the Moore's Law and Dennard scaling [17].

Modern general-purpose CPUs were designed to be able to execute any kind of workload, but this does not mean that they are efficient at such tasks. Since on any given day all kinds of workloads can be given to a CPU, it might benefit from possessing different kinds of more specialized hardware. For example, most modern CPUs include an integrated Graphics Processing Unit (GPU). Smartphones are typically implemented as a single SoC, which features a CPU, integrated memory, graphics, modem, ADC/DAC, and more.

Heterogeneous computing can be considered a separate field in digital electronics, since it encompasses both software and hardware, and their respective tools and compilation frameworks, requiring combined specialized know-how in hardware/software co-design[18].

Currently, one can see some sort of heterogeneous computing in almost all devices. Heterogeneous devices can be seen at the level of the system, where different ICs and specialized boards perform computations and communicate with each other, or inside the die of an IC, where different silicon chips can be seen on the same die, or even different kinds of hardware tightly coupled with each other. Modern SoCs feature chiplets (discrete individual chips, which might be different in their physical specifications), and are interconnected to each other on the same die, thanks to new advancements made in the technology of the interconnection networks. GPUs are ASICs that accelerate processing of algorithms for displaying pictures on a screen. They are hardware accelerators that, for example in a videogame, work in tandem with the CPU to draw each frame to a frame buffer and nowadays many SoCs can be seen where a GPU is present in tandem with the CPU. Some SoCs also feature specialized microprocessors for data processing and Machine Learning/AI purposes, which are heavily utilized in, for example, the camera app of a smartphone, while taking pictures or videos. The key points behind the performance improvements of hardware accelerators are:

- They are tailored for specific specialized data types and operations, and designed to work in the most efficient way for these types;
- They can be designed in a way that enables massive parallelism of computations;
- They possess a specific architecture of memory communication, also tailor made to the necessities of driving data to the processing elements;
- They can have much reduced control-path overhead, since they sacrifice general-purposeness for specificity and simplicity of design;
- They are co-designed with the algorithm they were designed to compute, sometimes enabling great uplifts in performance (some significant speedups in several orders of magnitude can be seen specially regarding computation of big amounts of data, in comparison to general-purpose CPUs)

On the software side, there must be integration present in the programming language, usually in the form of APIs, and compiler tools. Otherwise, for instance, the aforementioned camera app would not be able to harness the AI hardware capabilities for enhancing the photos, or the CPU would be unable to communicate with the GPU component(s) in the system. standard APIs such as OpenCL [19] exist to enable this communication layer, offloading computation to those external devices. There are also compilers for these languages, that assemble the code using a specific instruction-set defined into the external device's circuit, usually by the proprietary. In recent years AMD were the main force pushing for a heterogeneous computing philosophy, where CPU scalar and GPU's massive parallel computation are combined for a benefit in performance, while Nvidia

developed the CUDA API for utilizing the resources of the graphics card as a massively parallel computing card.

As mentioned before, nowadays there are consumer devices where an SoC is present with new co-processors that enable, for example, improved performance in machine learning, such as the Google TPU. This device consists of a systolic array, a very specific type of CGRA that utilizes multiply-accumulate elements while directly transposing the directionality of the neural network's graph to the interconnect of the processing elements. However, since this is an architecture with specific design constraints, its real-world performance will heavily depend of the programmer's knowledge of both the target application and the hardware. He must be able to identify systolic-array-like parts of the algorithm and transpose them to the TPU, and continually tune both, while also utilising a proprietary API that will have some overhead, and this, according to research, could take months of work[20].

2.5 Coarse Grain Accelerators

The new paradigm of massive data processing has led to the rethinking of how should processors be designed, and, while it is not new, it led consequently to more and more research and development on CGRAs.

CGRAs, or Coarse Grain Reconfigurable Arrays, are processors that consist of a 2D array or array-like structure of Processing Elements (PEs), which are the basic unit of computation, and they are controlled through a specialized set of instructions. These instructions, sometimes called control-words, are heavily dependent on the architectural details of the CGRA. CGRAs are usually co-processors, to whom a main CPU is responsible for assigning a certain workload, much like to a GPU if there is one present. Unlike FPGAs, which are very finely-grained in processing elements, CGRAs, as the name implies, are more coarse grained, where the basic block usually features a unit capable of doing certain arithmetic and logic operations. CGRAs, due to their coarse grained blocks, present many benefits in terms of runtime operation, in comparison to FPGAs. This is due to much better area efficiency of the logic blocks, and much more simplified routing scheme. Besides, since their reconfiguration time is much lower, they can efficiently and quickly switch context, further increasing diversity of workloads and applicability in real life scenarios. The main goal of CGRAs is therefore to enable a high degree of parallelism [21], efficient throughput specially for data-driven workloads, and wide range of possible use-cases, since the interconnection between the processing blocks can enable a direct exploitation of data-flow philosophies. This special interleaved organization of the logical units has been touted as a promising technology while being a good compromise between flexibility, energy efficiency, and performance. CGRAs promise improvements either in terms of power, area, or time, and more simplicity in architecture and control signalling, specially in accelerating audio/video/signal processing, telecommunications, neural networks, and more. Ideally, a CGRA would be in a configuration that reflects the flow of data between operations in a certain program. This allows for a potentially massive improvement in Instruction-Level Parallelism, which nowadays still presents a

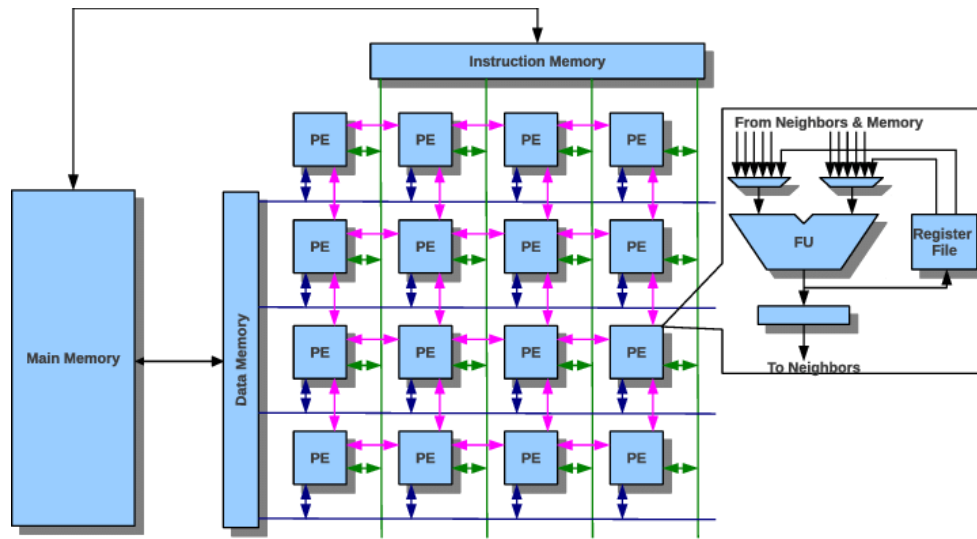


Figure 2.1: Typical CGRA design layout (adapted from [1])

huge opportunity for improvement, according to studies[22]. However, there are many different architectures, with many different array sizes, with different designs for processing elements, not to mention the inclusion of specialized units, memory, load-store units, etc., which also leads to having different kinds of abstraction layers and ways to communicate (via specialized instructions), therefore heavily fragmenting the design-space and making adoption of CGRAs very opaque and difficult.

The space of CGRA implementations is separated into loosely-coupled, and tightly-coupled, where CGRAs can be either separate or integrated into the main die of a CPU.

2.6 Microprocessor Architectures

In the beginning, processors took a similar amount of time accessing memory as performing instructions, as most of them did not even include a cache on the main die[23], so ISAs and CPUs were designed and built as scalar machines, where instructions are decoded and executed one-by-one, sequentially. However, as technology progressed, transistor count increased in inverse proportion to their size, and base clock speed increased until reaching a ceiling, while memory did not quite keep up with the advances, creating a big bottleneck. So nowadays, since instruction execution is several times faster than memory read/write operations, modern CPU performance heavily depends on how many instructions can be ran in between memory accesses. Therefore, newer techniques were brought forward to try to mitigate this issue, such as superscalar architectures, who aim to exploit Single-Instruction-Multiple-Data (SIMD) instructions by utilizing single instructions for multiple operations on multiple data points; Multi-threading, which aims to temporally schedule a program into multiple threads of execution, increasing concurrency and decreasing the amount of dark silicon; VLIW architectures where instruction-level parallelism is explicitly done by statically scheduling operations, and many other examples.

Current general-purpose CPUs are built with a superscalar architecture, where an instruction triggers a multi-staged control path with careful time scheduling, where techniques such as out-of-order execution, speculative execution, branch prediction are utilized so as to better utilize the available resources, with the aim of improving overall efficiency. However, these approaches are not ideal and may present inefficiencies, complexity of programming, performance issues, stalling risks, and security hazards. This is due to the often long instruction pipelines and time multiplexing, which can lead to non-negligible overhead for the aforementioned cases [24].

The scalar nature of x86 processors is very useful for performing tasks of any kind on-demand. However, as it was mentioned before, modern society has imposed new paradigms, which brought forward new workloads that impose processing large amounts of data. In an attempt to accelerate data processing on some x86 general-purpose CPUs, Instruction Set Extensions were created, which extend the base x86 assembly language with extra SIMD instructions for vector processing. Intel, for example, has created many extensions (and even extensions of extensions) of the instruction set, where several new instructions were added.¹ Even worse, some extensions such as AVX-512 are fragmented into many parts, further increasing the difficulty of programming for x86 CPUs. Some CPUs even have to throttle clock frequency in order to meet TDP constraints when using vector extensions[25]. And further adding Instruction Set Extensions could potentially cause more problems:

1. x86 processors must be backwards compatible, therefore each new SIMD extension cannot replace or improve upon another older one.
2. The compiler must be updated to use the new instructions, adding bloat;
3. The compiler algorithm must be upgraded for increased awareness of which SIMD instructions to use and when, increasing code complexity;
4. Instruction Set Extensions further complicate x86 assembly programming, making it difficult for programmers to optimize their code and increasing the entry barrier for programming CISC assembly;
5. Instruction Set Extensions must also be decoded by the processor, increasing the amount of logic present on the decoder and other components and power draw;
6. Dedicated hardware for vector processing, such as data processors or AI accelerators is at worst similar, and at best orders of magnitude better in comparison to contemporary x86 CPUs.

2.7 RISC-V

Like many other RISC architectures, RISC-V is a load-store ISA, where:

¹For example, the AMD Ryzen 4000, a common CPU in 2021, features the following extensions: MMX+, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, AVX, AVX2, FMA3, SHA, AMD64, AMD-V, AES, where only the last four are not SIMD extensions.

1. The only way to access main memory is by load/store instructions;
2. All arithmetic and logic operations occur between registers.

The RISC-V [ISA](#) exists in both 32-bit and 64-bit address space variants. It features a base instruction-set that must be present in any RISC-V design, and optional extensions, organized into Privileged specifications and User-level specifications. The chosen [ISA](#) can be classified as "RV" + "address space" + "Extension letter". The User level [ISA](#) consists of a base Instruction set, which is the Integer set, so it is written as RV32I, and its opcode is 11 at [1:0], very similar to earlier [RISC](#) processors. User-level spec also features the following extensions:

- M - Integer Multiplication and Division
- A - Atomic Instructions
- D - Single-Precision Floating-Point
- D - Double-Precision Floating-Point
- C - Compressed Instructions (16 bit)
- G - amalgamation of IMAFD, which is the general purpose ISA.

These extensions allow easy separation and expansion of the [ISA](#) according to one's needs, allowing for simplicity and ease of adoption/education. Also, RISC-V allows anyone to freely develop and test their instruction sets themselves, utilizing the instruction space in any kind of way.

Much like many other [RISC](#) designs, it contains 32 integer registers (32 or 64 bit), from x0 to x31, where x0 contains the constant '0'.

Table 2.1: RISC-V Calling Convention Names

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved/frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/return	Caller
x12-x17	a2-a7	Function args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

It also contains 32 floating-point registers, which may contain a single or double precision value, and an FP status register, for rounding modes and exception reporting. The processor state also depends on a register holding the program counter.

Table 2.2: Main RV64G instructions.

Instruction Opcode	Explanation
<i>Data transfers</i>	<i>Move data between either registers and memory, or between integer and FP registers</i>
lb, lbu, sb (or lh, lw, and other variants)	Load/store byte/half word/double (unsigned).
<i>Arithmetic/logical</i>	<i>Arithmetic or logical operations on data contained in general registers.</i>
add/sub(i, w, iw)	Addition and subtraction, with immediate and word versions.
slt(u, i, iu)	set-less-than, signed, unsigned and/or immediate.
and/or/xor(i)	logical operations, can be between registers or register and immediate.
lui	Load upper immediate, which are the lower 12 to 31 bits (upper 32 set to 0).
auipc	Addition between immediate and upper 20 bits of the pc (for branching to any 32 bit address)
sll/srl/sra(i, w, iw)	Shifts, logical left or right, and arithmetic, with immediate and word versions.
mul(w, h, hsu, hu), div(w, u), rem(u, w, uw)	Integer multiplication, division, remainder, with signed, unsigned word versions and support for 64-bit.
<i>Control</i>	<i>Conditional branches, jumps, either relative to pc or through a register</i>
beq, bne, blt, bge, bltu, bgeu	Branch according to condition (if (not) equal, if less/greater than, signed or unsigned.
jal, jalr	Jump-and-link absolute or relative to pc.

The base RISC-V [ISA](#) has fixed-length 32-bit instructions, with specific boundaries for bits with different purposes², and supports bi-endian³ operation. The RV64G instructions can be summed on table 2.2.

Instruction-set Extensions for RISC-V were separated by the authors into two groups: Greenfield, which occupy a new specific region of the address space, and Brownfield, where specific parts of another extension, "gaps" in a certain address space, are borrowed for other purposes. All standard extensions are of the Greenfield type, since they occupy exclusive regions in the address space, for standardization goals.

²The standard encoding scheme supports [ISA](#) extensions with variable-length instructions, where each instruction can be any multiple of 16-bit instruction fragments. The Compressed instruction set reduces code size by also utilizing 16-bit instructions.

³Bi-endian means both little or big-endian.

On the RISC-V User Spec [ISA](#) manual, it is stated that one can "recycle" the Compressed instructions extension's opcodes, which are the 2 least-significant bits 00, 01, and 10, freeing up $3 \cdot 2^{30}$ different instructions.

Table 2.3: RISC-V Instruction Format

31 ————— 25	24 — 20	19 — 15	14 — 12	11 ——— 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][10:5]	rs2	rs1	funct3	imm[4:1][11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20][10:1][11][19:12]				rd	opcode	J-type

Assuming this, for configuring a [CGRA](#), there are 30 bits available to specify configurations for the operations and interconnections, which is 3,221,225,472 different instructions.

The official RISC-V Instruction Set manuals can be freely viewed online, where one can read more in detail about the designers' motives behind the decisions made, and other technicalities that are not the scope of the present work.

2.8 Dataflow Philosophy

A data-centric philosophy of digital architectures differentiates from the normal control-flow in the regard that the spatiality and directionality of the data propagation is what dictates the flow of the computation kernel. In a sequential way, a grid of interconnected elements will execute mathematical operations on their operands, which could be of the type integer, float, vector, complex number, etc., and the spatiality of said grid will reflect the graph representing the whole mathematical function. These graphs are DFGs, or dataflow graphs, where the nodes represent mathematical operations, and the edges the placement of the operands. This means that data propagates sequentially in a potentially massively parallel way. This is a complete mentality shift from normal [CPUs](#) in the sense that there is no execution phase consisting of a single mathematical operation in an ALU, in each cycle of execution of the entire pipeline; instead, it allows data to suffer many transformations in various ways, before reaching the phase of writing the result to memory.

Currently, as mentioned before, there can be seen devices like this such as Google's Tensor Processing Unit (TPU), implementing a very specific type of [CGRA](#) called a systolic array. At the time of release, for machine learning it could outperform the competing product from Intel, which was the Haswell generation of Intel Core CPUs, by up to 196x.

There have been other approaches in trying to adapt other high-performance devices such as [GPUs](#) to this different computing mentality; however, their static hardware-managed caches makes them unoptimised for big flows of data, besides other problems. For data-centric devices, the prime focus should be eliminating all possible memory bottlenecks, for they impact directly the

performance of those devices. For maximum efficiency, data must be available on all processing blocks at each cycle.

2.9 Managing Hardware Accelerators

Hardware accelerators can be implemented as either tightly-integrated, as a coprocessor, or as a discrete peripheral device. Tightly integrated devices are part of the pipeline of the CPUs (usually at the later stages), where each execution cycle must traverse all of said device. Coprocessors are usually part of an SoC, not tightly integrated into the processor, but managed through a network-on-chip. A peripheral device can be installed to a computer through extension ports such as the PCIe standard.

SoCs are of heterogeneous nature, which means different kinds of hardware communicate through a common bus, for example the ARM Advanced Microcontroller Bus Architecture[26] (an open standard). This means that in order to control each of these coprocessors, the CPUs must be able to coordinate memory accesses in order to avoid memory hazards. Data can be accessed from shared caches or DMA units so as to not further burden the CPU. Coprocessors, which are not part of the CPUs pipeline, require coordination of user-space and kernel-space drivers, which provide the functions that are to be exploited by the developer. However, drivers could also rely on the Operating System system calls to manage the scheduling of hardware resources' utilization and other things, which could be a big source of inefficiencies[27]. On the other hand, tightly-coupled devices are part of the CPU's pipeline, and due to the scalar philosophy of CPUs architectures, could incur on heavy latency penalties and memory bottlenecks, while falling prey to the shortcomings of out-of-order execution and memory hazards. This would ultimately nullify the performance benefits of such a data-oriented device. The inherent parallel nature of Data-flow centric devices would necessitate utilizing decoupled instructions, which require special care for keeping synchronization of the devices. This implies careful and complex control-flow for avoiding "Read After Write" or "Write After Write" hazards. Ideally a co-processor would work in an "offloading computation" mode where, once it is properly configured and has the data available for consumption, it can work in parallel to the rest of the system. For this, special instructions would be used to manage the hardware accelerators in a state-machine mode, with exclusivity rules for facilitating reservation, work offloading, synchronization, and fine-grained control of local memory (important for the spatial aspect of certain co-processors such as CGRAs). There is a huge potential in exploring safe and efficient ways to manage co-processors with as little overhead as possible, while also respecting the data necessities of said devices.

2.10 Binary Acceleration

Binary acceleration is the term used to describe a process of binary translation (recompiling a binary for a different target ISA, or the same but extended with different instructions) that enables utilization of dedicated hardware [28] in order to accelerate certain workloads. Without this step,

a processor will never acknowledge that certain resources can be utilized to improve performance. Therefore the compiler must also be configured to target a different ISA. When an instruction set extension is implemented, the compiler must also be aware that there are different instructions made to exploit different resources.

Binary translation can occur either at the compilation stage, or at runtime [29], resorting to analysis of the stream of instructions being directed to the main processor [30]. Examples of the latter techniques are JIT compilation, virtualization, and emulators. Therefore, it can theoretically support both dynamically (Java) and statically compiled (C, C++) languages. Although, being able to adapt to the instruction stream (online paradigm) could be better since a statically compiled binary does not include information that changes dynamically throughout runtime (hence, they must simulate execution first in order to heuristically learn about cycles, dynamic memory accesses, and more).

Succinctly, offline and online binary acceleration approaches work by analysing an application, building a data-flow graph, and automatically optimising subgraphs utilising techniques such as modulo scheduling [31] (unwrapping and optimising loops). These subgraphs are called binary segments, and represent blocks of code where instructions occur sequentially, manipulating specific and constrained blocks of data. The resulting CDFGs - Cyclic Directed Flow Graphs - can be mapped onto CGRAs [32], providing that area, power, and timing constraints are met. Contemporary work has shown that the process of mapping graphs onto CGRAs is in itself a vector for optimization [33] [34], where graph isomorphisms and repeated subsequences can be exploited for better area and power efficiency.

There are also drawbacks in utilising such a technique. For example, when analysing binary code, in contrast to high-level code, it is difficult to associate the manipulation of data to its physical presence in the memory, and detecting memory access patterns or trip counts is complex[35]. This puts great emphasis on the quality of the compiler.

In the present work it is not intended to optimize a compiler for binary acceleration, nor translating instructions in real time.

Chapter 3

State of the Art

In this chapter, the state of the art on workflows for Coarse-Grain-Reconfigurable-Array (CGRA) design, implementation, simulation, and inter-operation in heterogeneous systems is provided, together with methodologies for instruction set extensions and their use in simulations (where possible).

3.1 CGRA Design

The architecture of a CGRA consists of several different aspects: the dimensions of the Processing Element (PE) matrix, the type of interconnect, the architecture of said PEs, the homogeneity or heterogeneity of the PE matrix, presence of local/external/hybrid memory, through accessing a shared cache, or communication through Direct-Memory Access, and possibly many other ways. One must note that while there are some things that can be reconfigured, where some of these configurations can occur at runtime, others cannot be reconfigured, only redesigned instead, which involves a change of the architecture. The design space exploration for the CGRA's memory is in itself immense, since it not only depends on working as either part of the pipeline, as a coprocessor, or peripheral device, but also on the availability of the external memory, which there are several different types of, each with different bandwidths and latencies. The CGRA's internal registers themselves may also present different topologies. The vast amount of possibilities [36] equates to an enormous potential for exploring, for example, hybrid architectures, where some PEs could have larger input/output sizes for vector processing, registers for constant numbers, could include look-up tables, and more. Even more complex would be allowing PEs to support subgraphs. However, exploiting this at the fullest level requires an equally complex compiler. The custom instructions must also be able to express this specialization.

CGRAs feature a simple control-path that serves the purpose of separating the configuration, running, and memory managing steps. As will be later seen, it can be controlled with a small set of instructions that operate the state changes. The main focus however is the data path, which is characterized by the compute kernel applied to the CGRA, and defined by the compiler/programmer.

CGRAs are commonly devised in either a row or a mesh architecture. In the latter, the PEs are organized in a 2d array, enabling communications between an unit and its neighbouring ones, with specific zones for memory-managing elements. Since there is such a big spectrum of design aspects that can be changed, the original designer may choose different aspects according to constraints or goals he wants to accomplish. For example, with constrained area, the designer must be careful with the area for internal cache and registers, while there must be also efficient ways to communicate with PEs of a Manhattan distance of two or bigger.

Different architectures are chosen by the designer to cater the CGRA to different needs and constraints - embedded, low-power, tightly-coupled accelerator, HPC, etc. - with different philosophies for power consumption, performance, and area. A CGRA designed for HPC could include several specialized functions and more complex PEs designed to execute them. Several authors have proposed different methods for efficiently routing [37] data in certain memory architectures, with register files, register row/column, rotating registers, shared cache pools, temporally-aware and spatially-aware schemes [38], and more. There are different ways to map computations onto CGRAs that can also be explored, where authors suggest directly mapping CDFGs to CGRAs. Some authors have also proposed methods for multi-threading, together with context switching [39]. There are many different designs with proven functionality [40], implementing some of the different features mentioned above.

3.2 Relevant CGRA Designs

In this section are displayed some of the most prominent CGRA designs available [41].

The MATRIX [2] architecture features a 8-bit datapath and mainly uses Arithmetic Logic Units (ALUs) in the PEs. For its time, it was considered to have an advanced network topology, where basic computing elements could directly communicate with neighbours on a two-square Manhattan distance.

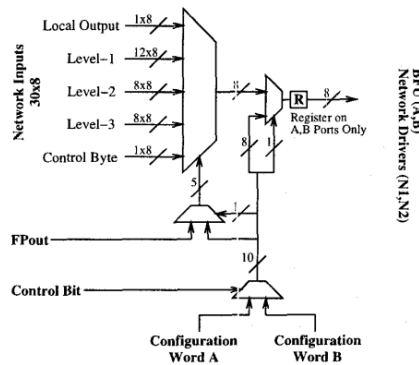


Figure 3.1: Diagram of MATRIX Functional Unit (adapted from [2])

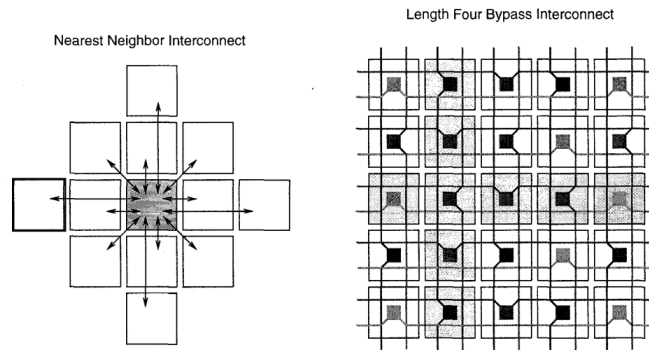


Figure 3.2: Diagram of MATRIX CGRA (adapted from [2])

MorphoSys [42] consists of an 8 by 8 array of PEs serving as a coprocessor to a TinyRISC core host. The array performs 16-bit computations. It performs well in regular code segments, specially in intensive computation paradigms, at the cost of total area.

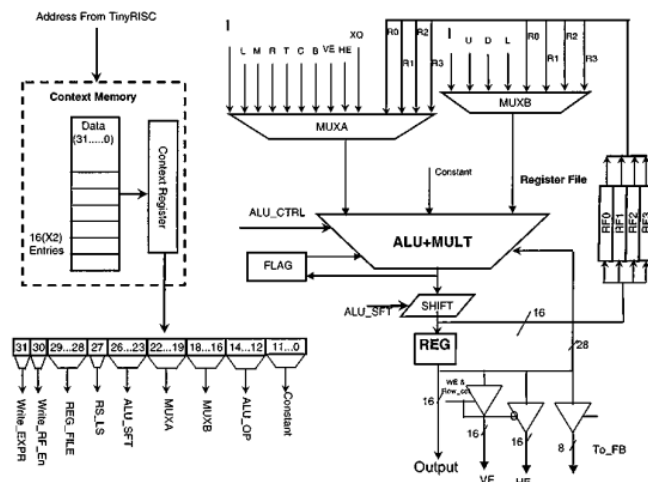


Figure 3.3: Diagram of Morphosys functional unit (adapted from [3])

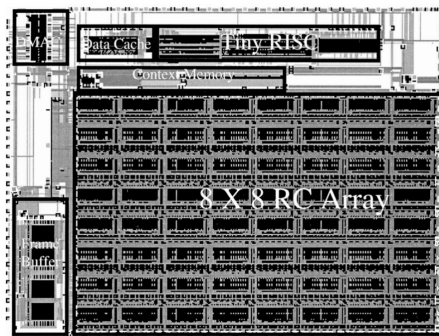


Figure 3.4: Diagram of Morphosys CGRA (adapted from [3])

Similarly to MorphoSys, in RSPA[4] each PE contains one ALU and one array multiplier. On the other hand, the base architecture is different since memory have multiple read/write data buses and a configuration cache is allocated to each PE. This is to favour loop scheduling over the Single-Instruction-Multiple-Data (SIMD) approach of Morphosys.

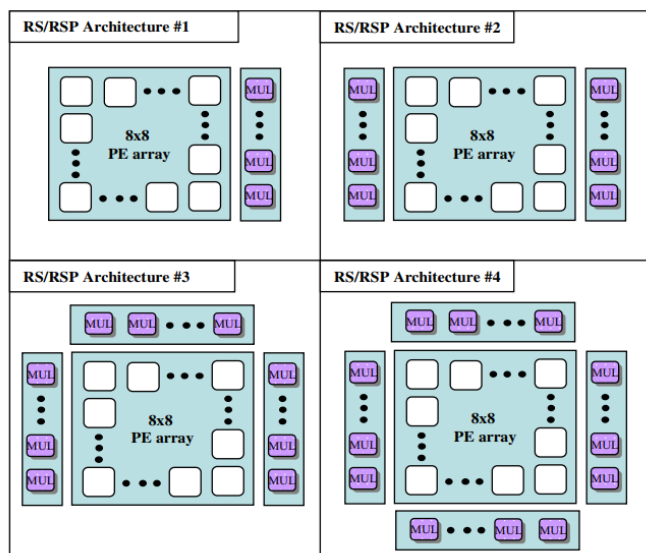


Figure 3.5: Diagram of RSPA different architectures (adapted from [4])

ADRES [5] is a tightly-coupled CGRA which consists of a virtual template in which a VLIW processor and a CGRA share resources. Thanks to careful modulo scheduling optimization by the compiler and shared memory access between both host processor and accelerator, it is presented as an efficient design for performance optimization. Each PE comprises of an ALU and a register for storing configuration, and have communication between the 4 neighbouring PEs, and all of its row. It has seen commercial implementation and tapeout.

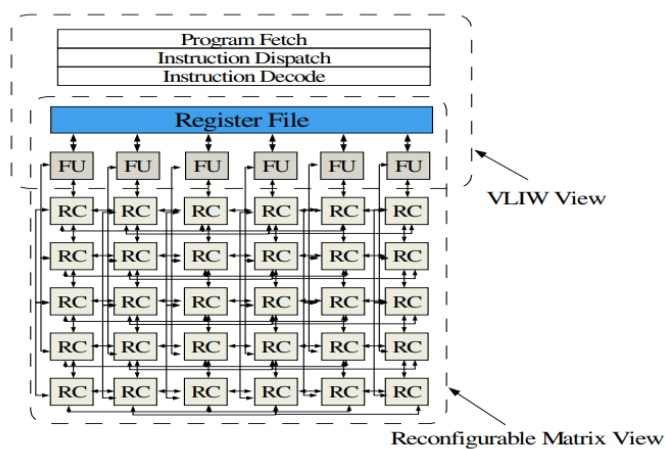


Figure 3.6: Diagram of ADRES schematic (adapted from [5])

The Versat [43] topology consists of separate Control and Data Engines, where the Control Engine is tasked with running the programs, while the Data Engine is responsible for the data-heavy computations. Its functional units consist of ALUs, embedded memories, multipliers, or barrel shifts. It features a complex databus that spans across all the basic elements, with different topologies.

Other known architectures in the literature are, for example, DRAA, TRIPS, DySER, and more. Hartenstein [44] presents an extensive survey on existing CGRA architectures.

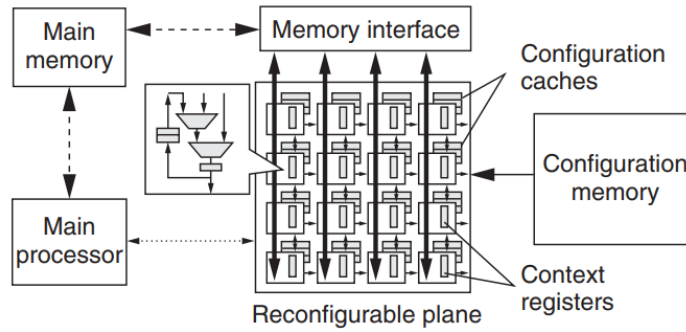


Figure 3.7: Diagram of DRAA CGRA (adapted from [6])

3.3 RISC-V based Systems

RISC-V is an accessible, powerful, extensive architecture, with a growing full-stack framework for enabling research and development [45], naturally supporting Linux and gcc compiler and debugger.

The RISC-V specification allows and facilitates the creating of a custom instruction set extension [46] [47], where researchers can exploit different functionalities and expand the range of operations that can be performed. This is very useful for designing an instruction-set based CGRA, specially since there are many tools available with great interoperability between them and compatibility with Hardware Description Languages, such as Chisel3.

As stated in the RISC-V User Spec Instruction Set Architecture (ISA) Manual v2.2 [48], subchapter 21.2: "RISC-V Extension Design Philosophy", "One use-case [for custom instruction-set extensions] is developing highly specialized custom accelerators, designed to run kernels from important application domains. These might want to drop all but the base integer ISA and add in only the extensions that are required for the task in hand. The base ISA has been designed to place minimal requirements on a hardware implementation, and has been encoded to use only a small fraction of a 32-bit instruction encoding space.", which, as will be seen later, coincides perfectly with the goals in mind for the present work.

Here are some relevant RISC-V cores:

- CVA6 [49] - 6-stage, single issue, in-order CPU, supporting RV64IMAC, as well as other features for support of Unix-like operating systems. It is one of the oldest and most reputable RISC-V cores.
- SHAKTI - A family of RISC-V processors developed with funding from the Indian government. The aim of the project is to build an open-source production-grade processor, together with full SoC, development boards, and High-Level Synthesis language. According to its website, it is competitive with other offerings in the area.
- CV32E41P is a 32-bit, in-order RISC-V core with a 4-stage pipeline, adopting the RV32IMC architecture (with F, Zfinx, Zce), and most notably the Xpulp custom extension set. It is part of the PULP [50] framework, a collection of high-performance RISC-V cores commonly used by academics. Unfortunately, it was not possible to find more information on the custom extension set, other than a reference for the new instructions. Its team is now also part of the EUPilot project, an initiative for advancing research and independence on chips in Europe.
- The BOOM Core [51] (Berkeley Out-of-Order Machine) is an open-source RV64GC core written in Chisel. It is also one of the earliest cores, with a focus on high performance for architecture research.
- Rocket [52] is one of the most complete projects built with RISC-V. Made in UC Berkeley too, it is a RISC-V in-order Central Processing Unit (CPU) core, which features an extension port for implementation of co-processors, called RoCC. The Rocket Core is a part of Chipyard, an open-source software framework for the generation of SoCs. With it, one is able to generate an System-on-a-Chip (SoC) consisting of a rocket core, an optional coprocessor, memory system, peripheral ports, etc.

Chipyard also provides software for RTL simulation, Field-Programmable-Gate-Array (FPGA) simulation acceleration, circuit synthesis (PnR), and more. Using a rocket core allows us to integrate a co-processor, like a CGRA, capable of hardware acceleration. Chipyard already permits the implementation of other co-processors, such as Gemmini, a DNN accelerator (whose main component is a systolic array, similar in structure to a CGRA), and Hwacha, a vector computing unit.

Chisel, as mentioned before, is a hardware description language made in tandem with RISC-V, to facilitate circuit generation and re-utilization. And together with libraries, it promotes sharing and co-developing code much like it is done today in Software Engineering. It is based in Scala, an Object-Oriented Language based in Java, with similar syntax to C, combining other characteristics such as functional programming. Thanks to the Scala back-end, it allows hardware blocks to support templates, polymorphism, and inheritance.

Chisel code, containing the hardware description, is later compiled into an intermediate representation in a language called FIRRTL, which contains the product of all the Chisel code transformations into a single "circuit" description. Then, it is finally compiled into Verilog.

There is great interest in researching RISC-V heterogeneous systems which implement a [CGRA](#) as a coprocessor, as the full chain of computation can be viewed and debugged, including the main CPU's orchestration of the computation offloading. There is unfortunately still sparse research on systems where both the RISC-V architecture for a host processor, and a [CGRA](#) as a co-processor, are present.

Fiolhais et al. [53] explore existing RISC-V cores and [CGRA](#) solutions, and create a system based on their own RISC-V design named Adept, and the published Versat [CGRA](#), mentioned above. Work was also published on a compiler for Versat [54]. In here, one can better understand the ideas behind some design choices for the mentioned [CGRA](#) architecture. The authors have opted for a top-down approach, where high-level C/C++ code is compiled, and then the assembler on the [CGRA](#) in charge of building the instructions for configuring the [PEs](#) and other elements. Those instructions are different between the control path and the data path, which are decoded by different parts of the [CGRA](#).

3.4 Hardware Accelerator Management

As mentioned before, managing coprocessors and peripheral devices has its own space for exploration and improvement. There are several aspects which need to be considered for working with network based, or tightly coupled CGRAs, since they can share caches or access external memory.

The ARM architecture presents a standard coprocessor interface, defining instructions that allow interaction with up to 16 tightly-coupled hardware accelerators. However, since they share the control path with the cores, being part of the pipeline, they are not fit for implementing dataflow-centric coprocessors such as [CGRAs](#). This is due to the inherent parallelism of a data-centric coprocessor, where the main [CPU](#) cannot know in advance in how many clock cycles a compute kernel will be computed, which blocks out-of-order execution. This is a similar phenomenon as what occurs with the RoCC interface of the RISC-V Rocket chip, as will be seen later.

Management of coprocessors is usually done through drivers, in kernel space, with access through user-space, enabling the possibility of developers invoking functions that are part of a high-level API. This abstracts away the instruction set that directly operates on the coprocessor. Besides, drivers rely on system calls to the operating system. Some authors have noticed that this high-level access translates to inefficiencies, which are always present for each operation/-communication to a coprocessor. Which means that for a lot of small operations, the burden from these inefficiencies can escalate to a great percentage of the total work done. Therefore was proposed the IXIAM[55] framework, for implementing an instruction-based coprocessor management system, through a custom-made Instruction-set Extension for the RISC-V spec. Such [ISA](#) was designed in such a way that the safety features that come with OS (software) management are not ignored, while also acknowledging different formats of instructions for control and memory/cache management - "The supported hardware infrastructure implements communication and some tasks normally delegated to the OS, such as reservation. As a result, the entire proposal can be, in principle, implemented in user space while still guaranteeing security and isolation."

The authors therefore propose six instructions, The proposed solution is encompassed into the following points:

- A general [ISA](#) extension for a wide variety of integrated accelerators, which is independent of the specific features of each of them while allowing their effective and low-latency usage;
- A limited architectural support for our [ISA](#) extension that allows reducing OS involvement in accelerator management through a distributed hardware reservation mechanism of accelerators;
- Avoiding [ISA](#) bloating, which is particularly dangerous in the case of fixed-size instruction sets, and promoting easy scalability, as newly-introduced accelerators can be seamlessly supported;
- Efficient use of die-area and power, since the same [ISA](#) extension and micro-architecture can effectively interact with many accelerators;
- Design independence, as the proposed instructions do not dictate the design choices of the accelerator manufacturer, as many accelerators can be made compatible with our proposal using a thin hardware interfacing layer.

Their research shows a performance benefit inversely proportional to the number of computations, which shows that this solution effectively greatly reduces the overhead introduced by API calls and OS management - "Our proposal does not need the traversing of all these layers and can work with no driver in the middle. Since we map accelerator-oriented commands on user-space instructions, these can be directly invoked in user code with inline assembly."

Unfortunately little to no explicit work on custom flexible and portable instruction-set extension interfaces for RISC-V cores was found freely available. As reported by some authors [56], only a small subset of RISC-V cores provide an open interface for integrating [ISA](#) extensions, allowing expansion of the processor capabilities, and the degree of possibility of such expansion can vary greatly between implementations, stressing the need for standardization. Most of those implementations are designed for specific goals in mind, not allowing flexibility and expansion, and only work for those specific cores. For example, the EAI interface [56] is designed for the E203 core; the eXtension Interface [56] for the CV32E40X, and the RoCC for the Rocket Chip. The extension framework for the PicoRV32 does not allow control flow and memory manipulation. Damian et al. explore the creation of a feature rich ISE interface called SCAIE-V, in order to solve the problems mentioned above. It is feature rich as it allows "custom control flow, decoupled execution, multi-cycle instructions, and memory transactions", and very importantly, allows instructions to be decoupled from the main CPU, running in parallel to the main pipeline. It works as a framework for generating a least-cost interface based on the needs of the instruction set extension. While supporting all the features one would need for controlling a CGRA, while also abstracting away the details of the core's microarchitecture and the coprocessors', it still restricts a

specific core to a specific coprocessor management system, where in a hardware level would provide limited flexibility unless the architecture is redesigned, leading to incompatibilities between different versions of the core. In any case, the authors show a great deal of care taken regarding supporting different architectures, avoiding memory hazards, and coordinating communication with as little overhead possible.

These approaches fall in line with the overall philosophy of the present work. It shows that it is possible to create an abstraction layer between main CPU and other hardware accelerators present in the same network, such as a CGRA, in a hardware-level instruction decoding approach. It also allows the possibility for either directly controlling a CGRA, from the developer point of view, or abstracting away the control to the compiler.

3.5 Hardware Emulation

Hardware emulation is an important aspect of the design flow since, for work based on extending the instruction set, it allows validation of custom microcode implementations, while also being an useful learning tool, specially when combined with a GUI and a graphical representation of the digital logic. If accurate, although most of the times very slowly, they can also allow one to run in real time and debug a program written for a target architecture being simulated, if the speed penalty can be tolerated.

MARS is very likely one of the most simple and well known examples of such software. It is an embedded IDE and simulator for a MIPS architecture, which is especially useful for educational purposes, written in Java. It allows the user to write assembly code, debug the code, and view the state of the MIPS simulated processor being altered in real time, during the execution of a piece of code. One can also add features through plugins, that allow the user to, for example, view the schematic of the processor, with the datapath being updated in real time.

QEMU is a very well known machine emulator/virtualizer, which allows one to run a simulation of a processor architecture in real time. As per the user-made wiki page:

"QEMU is a generic and open source machine emulator and virtualizer.

When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.

When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, 64-bit POWER, S390, 32-bit and 64-bit ARM, and MIPS guests."

Unfortunately while allowing for simulating RISC-V architectures, it cannot as of yet simulate a standalone CGRA, or one associated as a coprocessor in another main CPU architecture. For this, we have built a simulator, based on all the principles learned and inspired by tools such as CGRA-ME.

Another similar simulator where inspiration was taken from is the Microblaze Simulator built by Dr. João Bispo. This program allows one to simulate a Microblaze architecture with full control and debugging features of the device. It allows the user to dynamically input assembly instructions, and thanks to its GUI, view the registers' content (with dynamic updates during runtime), view the trace of the instructions, watch the result of the operations, and more. The GUI is simple and efficient while allowing one to have several ways of controlling the emulated device.

3.6 CGRA Design Frameworks

While many authors have released already several different [CGRA](#) designs, the research done is often too closed on scope, without showing other vectors for exploration, and without much information regarding performance comparisons and competitiveness regarding other designs. There is a need for [CGRA](#) frameworks and simulation tools that can provide methods for designing, exploring different architectures, allow simulations and execution of benchmarks, and physical realization through HDLs, and their respective compilation frameworks.

One such framework, and possibly the single one so far, is CGRA-ME, a tool that allows producing a generic description of the [CGRA](#) architecture, mapping of algorithms, and physical implementation. For describing a [CGRA](#) it uses the CGRA-ADL description language, based on XML. The tool allows for execution of applications written in C code through a LLVM based compiler. This compiler converts the C syntax into an intermediate assembly language, with instructions representing logical and arithmetical operations. The resulting DFGs (most common data and algorithm structure fed to CGRAs) are mapped to the CGRA, as the [CGRA](#) itself is also presented as a graph, where each different [CGRA](#) context is a different graph.

Although there are several tools and software developed for compiling code into graphs, mapping algorithms, or turning [CGRA](#) designs into HDL code, there is no other modelling and evaluation framework for [CGRAs](#) (on the other hand, there is a pretty solid tool set for FPGAs). There are also no CAD algorithms that help with calculating area, frequency or power consumption metrics.

This tool allows exploring the immense design space of [CGRA](#) on several fronts; One can choose different topologies for:

- Types of basic processing elements, and which operations they support;
- Homo- or heterogeneity of the Processing Element ([PEs](#)) matrix;
- Interconnect type (number and direction of possible connections between each PE);
- Local/external/hybrid memory system, with respective methods for memory access;
- Possibility of context-switching.

However, when it comes to the instructions used, as was mentioned before, the instruction set is an intermediate language between the compiler and the mapper, only consisting of logical and

arithmetical operations, which means it is not complete, on the standpoint of control mechanisms and memory access. The present work formulates a simulator where manipulation of a CGRA is done through a previously-defined instruction set, designed for the optic of the developer, in which full control, assigning and mapping of operations and data propagation is possible. Therefore these instructions can abstract away the CGRA hardware from the runtime environment, allowing for CGRA interoperability with reduced overhead.

Ling et al [57]. explore how CGRA-ME is used as both a simulator and generator of a program for the CGRA, through analysis of a dataflow graph (CDFG), although the CGRA mapping process is not very clear, and there is no transparency on the actual instructions being used between the host processor and the coprocessor.

Chapter 4

Proposed Approach

In this chapter a methodology for work flow and a general idea for a Coarse-Grain-Reconfigurable-Array (CGRA) architectures will be provided. Then, the CGRA-SIM simulator tool is shown, together with an example of a simple instruction set for configuration and execution.

The proposed workplan is the following:

- Prototyping a base generic CGRA, with an array of Arithmetic Logic Unit (ALU) Processing Elements, which allows different preset combinations and types of said PEs, interconnect types, and other design vectors.
- Developing a high-level simulator for the CGRA (CGRASim), using an object-oriented programming language and an freely-available IDE (in this case, Java and Eclipse);
- Designing a possible micro-code and enabling CGRA interpretation of the instructions (inline assembly) through a software parser.
- Testing different CGRA architectures to evaluate different instructions.

4.1 CGRA Design

The design of the CGRA is an amalgamation of the most common characteristics found in other CGRA designs, although it is purposefully similar to the ADRES CGRA architecture, since this configuration has been utilized and tested extensively by academia. However, it must also follow the instruction-set based CGRA specification, which means it must allow full configuration and utilisation through specific well-defined instructions, as mentioned in previous chapters. So each of its components must be designed in such a way that allows manipulation through instructions, which have their own size constraints.

For the interconnect configuration, a mesh topology was chosen, as opposed to a single-dimensional topology, or other kinds. Mesh topologies allow for bi-dimensional interconnections, where Processing Elements (PEs) can route data to each-other in the x and y axis. This topology allows for different types of interconnections, such as Nearest Neighbour, Nearest Neighbour with

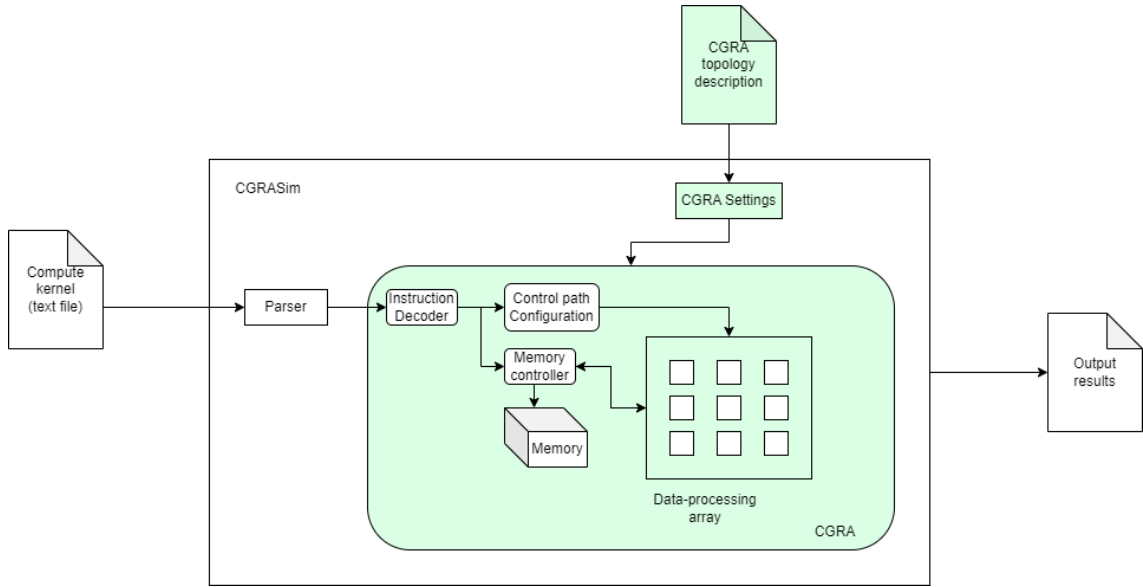


Figure 4.1: Diagram for CGRA Sim architecture

diagonals, toroidal (the axis limits are connected and loop back to each other), and more. A mesh topology is the most common, and is preferred in this case, since by enabling two-dimensional connections between blocks, it permits high levels of data throughput capabilities, although other hierarchical organizations can also be contemplated in the future. The basic processing element grid can also be configured homogeneous (all PEs of the same type) or heterogeneous, although in the end the CGRA is always heterogeneous since it must have PEs for accessing memory, and for doing calculations.

As for the processing elements, they can be separated in two groups, according to the number of inputs: Unary Processing Elements, and Binary Processing Elements. The unary PEs are for unary operations where only one input is required. These operations can be, for example, absolute value ($|x|$), negation ($-x$), one's complement, square root, squaring, cubing, and others. Binary Processing Elements are for implementing the more common calculus, such as sum, subtraction, multiplication; bit-wise logic operations, such as AND, OR, XOR; bit manipulation, such as shifts; memory operations, and more. These require two inputs and produce one output. In a CGRA this means that these PEs will always connect to two PEs in upstream and accept data from them, and create an output value in downstream. There is a vast amount of possibilities for the mathematical operations and types that can be integrated on a PE. However, we opted for simplicity and similarity to common hardware constructs found in digital circuits and processors. This includes the functionality from Central Processing Unit (CPU) ALUs, multiplication units, and similar elements. On Appendix A, one can see code for an ALU-based PE in Verilog, a common Hardware-Description Language.

To summarize, the CGRA consists of a matrix of processing elements, operating in parallel with each other, and propagating data to each other according to a set of configurations that fits the rules of a predefined type of grid. The processing elements generate an output according

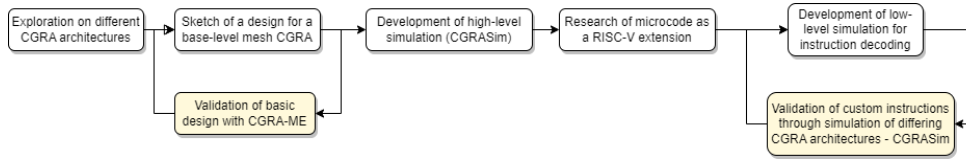


Figure 4.2: Flow chart for proposed workflow

to their inputs and the configured operation, and store it on their internal register. This is the execution step. Then the data from the registers is propagated to the next element according to a interconnection configuration. This is the propagation step. Both steps together form one single execution cycle.

To generate initial payloads for performing the operations, and to store the final results, the **CGRA** must be able to communicate with the memory. This is done through special binary **PEs** called Load-Store Units (LSUs for short). These accept an address input and a data input, and perform a memory access according to their configured operation.

4.2 CGRA-SIM

The tool henceforth developed is CGRA-SIM, a user-interface-guided **CGRA** simulator, written in the Java programming language. It was started in 2021 by Dr. Nuno Paulino, which featured already some of the foundational functions for enabling a **CGRA** simulator environment, but was incomplete and did not feature a GUI. This program aims to fulfill several important steps of the workflow for exploring Instruction-set based CGRAs. It allows the user to easily configure a preset bi-dimensional CGRA, with different levels of design space exploration (that will be expanded further), where the GUI provides easy ways to configure, change its memory contents, read direct instructions, and operate the control-flow of the CGRA. It is intended as a way to visualize the effect of the instructions being sent, helping to creating **CGRA** compute kernels and directly view their operation. The simulator is execution-cycle accurate, and produces true results. It is not however intended to be used as a runtime simulator, as it is not performance oriented. It is intended to be use as a research and academic tool, and as a means to validate the concept of instruction-set based CGRAs. The list of features is as follows:

- GUI oriented utilisation;
- Configuration window for the initial **CGRA** design setup;
- Visual **CGRA** main window with abstract representation of the **PEs** and interconnection;
- Visual **PE** representation allowing for visualizing type of operation, register content, and more;
- Visual representation of interconnect types and connections between **PEs**;

- GUI items for easily viewing the **CGRA** configurations and data values, with different vectors for interaction and design space exploration;
- Console-like text field for manually inputting **CGRA** instructions;
- Menus for changing visual aspects, changing **PE** configurations, manipulation of **CGRA** memory;
- Manipulating the simulation by user interactions;
- Register content window;
- Preset compute kernels;
- Button to load **CGRA** programs utilizing the custom instruction-set;

And more smaller features not contemplated in this list.

4.2.1 Software Architecture

As it was previously said, the simulator is written in Java, an object-oriented language. This allows the use of more advanced coding techniques that were useful for the purposes of the simulator. Besides, it also allows a tight integration with GUI elements through Java Swing libraries. This was all done with the Eclipse IDE which also features a Java Swing designer. Therefore, the code is divided into simulator code and GUI code, with each having its own hierarchies and integrations with each other. For the simulator, there are several different sections of code representing a distinct aspect. These sections are separated in classes, following a hierarchy. The main class is the `GenericSpecsCGRA`, which has the parametric builder for the **CGRA**, together with the objects for its other components. The builder, as the name says, builds the **CGRA** according to different parameters, which will affect the characteristics mentioned before, such as interconnect type, size of memory, and so on. Then the methods are mostly getters and setters, with also methods for executing the simulation cycles, printing information, manipulating the simulation runtime, communicating with the GUI, and others. The **CGRA** main object holds objects for other things, such as the interconnect, mesh, instruction decoder, and more. The `AbstractInterconnect` class and its `Interconnect` interface are responsible for the map holding all **PE** port connections, manipulation of said connections, also defining which connections are possible, according to the interconnect types mentioned above, and the data propagation part of the simulation cycle. It also processes context switching. The `Mesh` class holds the two-dimensional array of PEs, and is responsible for its manipulation, also according to the rules defined by the user through the GUI configuration window. It is also responsible for the execution part of the simulation cycle. The `GenericMemory` class and its `Memory` interface hold the data structure that represents any memory in the **CGRA**, and methods for its access and manipulation. It is used for the **CGRA** random access memory, the live memory (liveins and liveouts), and context memory. The Instruction Decoder

class holds the mapper for instructions to functions in the CGRA. This will be better explained in a further section.

These are the main structural components of the CGRA. The following classes are for other specific functions. The datatypes classes, which are composed of PEInteger, PEBoolean, the interface PEData, and others, represent the data that is exchanged between the PEs. They also contain the methods for manipulating the data according to certain functions, which for the PEIntegers, are the arithmetical operations conducted by the ALU elements, for example. So this is where the PE operations are actually implemented, in the code architecture point of view. The PEs' abstract parent class is Abstract Processing Element. This class implements the ProcessingElement interface and holds most of the fields and constants that represent characteristics of the PEs such as its control data, constants memory, register file, simulation settings, ports, and methods for configuring the PE in relation to the CGRA. This class is extended by Unary and Binary PEs, which as mentioned before define the amount of input ports, and in turn are extended by the specific types of PEs integrated into the CGRA. For the Unary PEs, there is the UnaryPE type, and for binary PEs, there are ALU Elements, Load-store elements, and Adder and Multiplication elements. These PEs execute commands according to the control data they are given, which is part of another class, PEControlSetting, a helper for the control classes for ALU (ALUControlSetting) and LSU (LSUControlSetting). These define the allowed operations for these specific types of PEs.

To summarize, the simulator is separated into the following classes:

- Control - for control-flow datatypes, stored into different config registers throughout the CGRA
- Datatypes - for the data quanta that is to be processed by the CGRA matrix
- Instruction Decoder - a class holding the instruction decoder process code;
- Context - a data collection for a CGRA context - consisting of a representation of the connections set between PEs;
- Interconnect - classes for the different possible interconnects for the CGRA (Nearest-neighbour, Toroidal, etc.)
- Memory - classes for the local CGRA memory;
- Mesh - helper class for holding the 2D array of PEs and associated code;
- PEs - classes for each different PE type, with a hierarchical organization depending on main function and architecture/number of inputs
- GenericSpecsCGRA - parent class for holding the CGRA
- GUI-related classes - hold the code and information for the different windows and other objects such as enums for configuring the CGRA;

In essence, the simulator code consists of the inter-operation between two main classes - the mesh and the interconnect - where each will execute different tasks: the execution of the computing algorithms, and the propagation of the data to the neighboring PEs. The main **CGRA** class holds an instruction decoder where controlling the **CGRA** can be done through the parsing of in-line instructions. These instructions are then translated into the execution of different methods.

The **CGRA** starts as a mesh of configurable x and y dimensions. The user can configure which **PEs** are of which type, and their connections with neighbouring classes. Each **PE** holds the control settings in its configuration memory, and each connection is stored in the interconnect class.

Once the configuration step is done, after the user having loaded data onto the local memory, and the respective addresses being set in the load-store units, execution of a computing kernel can start.

The control phases of configuring, managing memory, and execution, are controlled via state machine code. During a step phase of execution, the **CGRA** performs an execute command on each **PE** in order (in real life it would happen all simultaneously), and once that process is finished, it executes the propagation phase, where all the payloads are driven to the inputs of the next **PE**. For load-store units it is different, where this process would not necessarily happen for a store command, for example.

4.2.2 Instruction Decoding

The goal of this simulator is not only to prove the **CGRA** as a massively parallel processing machine, but also to validate the concept of it being an instruction-set based accelerator. Therefore, the simulator must integrate this capability as a means to fully execute operations with pseudo-assembly instructions. The instruction parser class is `InstructionDecoder`. This class is instantiated once in a **CGRA**, and it is responsible for mapping all the instructions to different operations. These operations can be for communicating with the **CGRA**, manipulating its settings, setting the operations on processing elements, or manipulating the simulator environment. The instructions are in table 4.1.

These instructions fit into three main categories:

- Configuration - "set", "set-io", "set-constant" for example, whose parameters are numbers that define coordinates, operation codes, or provide input arguments.
- **CGRA** information - "gen-info", "pe-info" - used for retrieving information about the **CGRA** and its elements. These can be thought as if an external device was performing a query of the **CGRA** through an API, requesting info as a data structure in an organized manner.
- Simulation - "run", "step", "config-setPE", which serve for manipulating the simulation environment. "Config-setPE" is not realizable outside of a simulator as it implies a change in the **CGRA** architecture.

The Instruction decoder is responsible for verifying the syntax of the commands, which means verifying if the input arguments are correct, and in the correct order. Then, it executes methods

Table 4.1: CGRA-SIM Instruction opcodes.

Instruction Opcode	Explanation
<i>Configuration</i>	<i>Configure PE settings, PE inputs and outputs, constant values</i>
set (op) (x, y)	Set an operation for an ALU or LSU Processing Element. op is a 4-bit opcode.
set-io (x1, y1) (x2, y2) (x3, y3)	Set the inputs for the PE in coordinates (x1, y1). CGRA interconnect then checks which connections are possible.
set-constant (x, y) (const)	Sets a constant value to be stored in a PE's constant register.
set-input-constant (x, y) (input) (boolean)	Sets which input will be fed the constant value instead of the payload at the input port. Boolean turns this on or off.
<i>CGRA Information</i>	<i>Retrieves information and design parameters about the CGRA</i>
gen-info	Retrieves information about the CGRA characteristics: Mesh dimensions and topology, interconnect type, memory size, and more.
pe-info (x, y)	Gets information about a PE characteristics - type of PE, current operation, register file size, constant, and more.
mem-info	Retrieves information of the characteristics of the CGRA's generic memory and contents, if possible.
store-ctx	Stores current topology as a context in the CGRA's context memory.
switch-ctx (ctx-id)	Switches context to a different one in the CGRA's context memory.
<i>Simulation Control</i>	<i>Controls the simulation environment</i>
run (x)	Runs/stops execution of the CGRA. Equates to running "step" x times.
step	Executes a simulation cycle (execution and propagation steps).
pause	Pauses/resumes execution of the CGRA simulation. Currently not implemented in practice.
reset	resets the CGRA to an original state, clearing the connection map, registers, PE control objects, memory.
config-setPE (x, y) (type-id)	Changes a PE to a different type, set by 'type-id'. Simulator-only configuration type instruction.

exposed in the main [CGRA](#) class. The instructions were purposely defined as to be able to fit the RISC-V custom instructions specifications, namely regarding the size of the instruction. For example, for setting the inputs/outputs of a binary [PE](#) on a 4 x 4 array:

1. 4/5 bits for specifying the instruction type;
2. 4 bits for specifying the coordinates of the destiny [PE](#) (from 0 to 15th PE, 2 bits for x axis, 2 for y axis);
3. 4 bits for the coordinates of each of two input PEs, totalling 8 bits.

Where one would add extra bits for future-proofing, for example for bigger arrays or more complex ALUs. In total leads to a minimum of 17 bits for the address space, which is fine regarding instruction word size.

The specification for the arithmetical operations ran by an [ALU](#) Element are defined in table 4.2, and the load-store operations in table 4.3.

Table 4.2: ALU element control opcodes.

Operation Opcode	Explanation
0000 - PASSNULL	Outputs a null value (equivalent to 0's in hardware)
0001 - ADD	Addition
0010 - SUB	Subtraction
0011 - MUL	Multiplication
0100 - DIV	Divison (rest)
0101 - LSHIFT	Left shift logical
0110 - RSHIFT	Right shift logical
0111 - AND	logical AND
1000 - OR	logical OR
1001 - XOR	logical XOR
1010 - PASSL	outputs "left" input value (first input)
1011 - PASSR	outputs "right" input value (second input)
1100 - SLT	outputs the smaller value between inputs
1101 - SEQ	outputs an input value if inputs are equal, null if not.

Table 4.3: LSU element control opcodes.

Operation Opcode	Explanation
0000 - PASSNULL	Outputs a null value (equivalent to 0's in hardware)
0001 - LOAD	Loads a value stored in local CGRA memory with address provided by the first input
0010 - LOAD-STREAM	Loads a value stored in local CGRA memory with address provided by the first input + internal counter (increments by 1 in each simulation cycle)
0011 - STORE	Stores a value from the second input, in address provided by first input
0100 - STORE-STREAM	Stores a value from the second input, in address provided by first input + internal counter (increments by 1 in each simulation cycle)

4.2.3 Simulator Environment

Upon opening the simulator, the user is presented with a **CGRA** configuring window. This window allows setting different aspects of the accelerator, such as:

- X and Y dimensions of the **PE** matrix;
- Homo- or heterogeneity of the **PE** Mesh (could also be empty for a blank **CGRA** preset)
- Type of **PEs** for homogeneous mesh;
- Type of interconnect, which defines the possible connections with different distances
- Local Memory size, in number of PEData-type objects it can hold;
- First row of LSUs - these **PEs** are necessary for loading data onto the CGRA, where this option saves the trouble of defining them one by one.

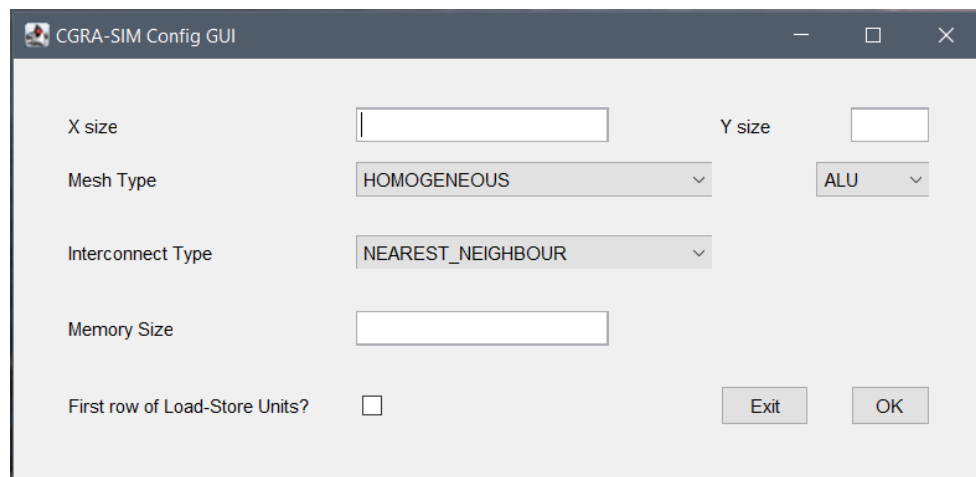


Figure 4.3: Initial CGRA configuration window

Once these settings are set, by pressing the "OK" button the simulator then proceeds to loading the main simulation window, presenting a user-interface representation of a CGRA, together with a text box for inputting instructions, a text window for showing **PE** information, buttons for execution and opening files, and several other options. On this window, the user can perform the following operations:

- Start/stopping an execution runtime;
- Executing only a single execution step;
- Opening and parsing instructions of a text file;
- Clicking a **PE** for displaying its configuration and register data;
- Sending instructions to the **CGRA** directly through a text box;

- Resetting the CGRA;
- Setting the type of a PE (this cannot happen during runtime, as it is a design choice)
- Loading contents into memory from a file;
- Storing memory contents to a file;
- Setting an interconnect context;
- Loading a context from a file to the context memory;
- Saving a context from context memory to a file;
- Setting a preset compute kernel;
- Obtain help;
- View about page;

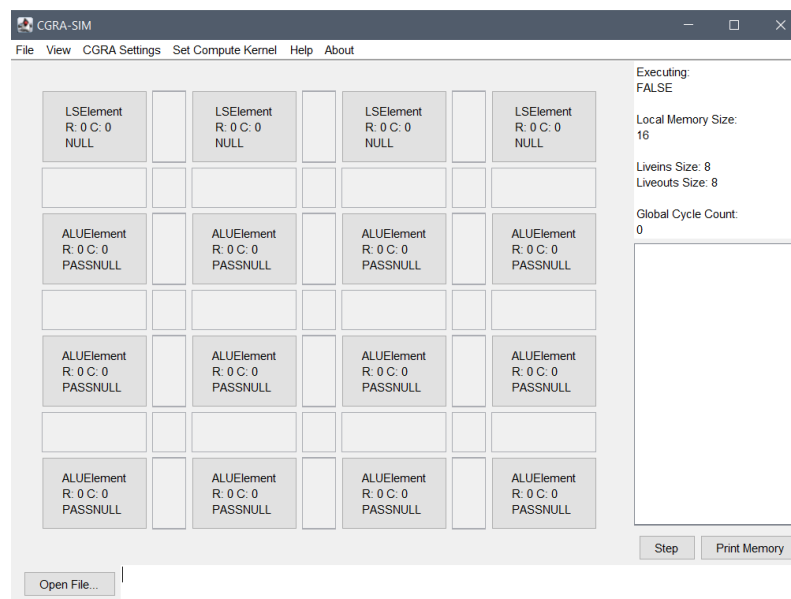


Figure 4.4: Main CGRA-SIM window

There are more helper dialog boxes/windows; these are for setting a specific PE to a type, and choosing a preset compute kernel of several types, to run on the CGRA. For now, the simulator comes with the choice for the following preset computing kernels:

- Dot Product ($A[0] * B[0] + A[1] * B[1]$);
- Discrete convolution/FIR Filter (4 taps)
- Affine function ($A * x + b$)

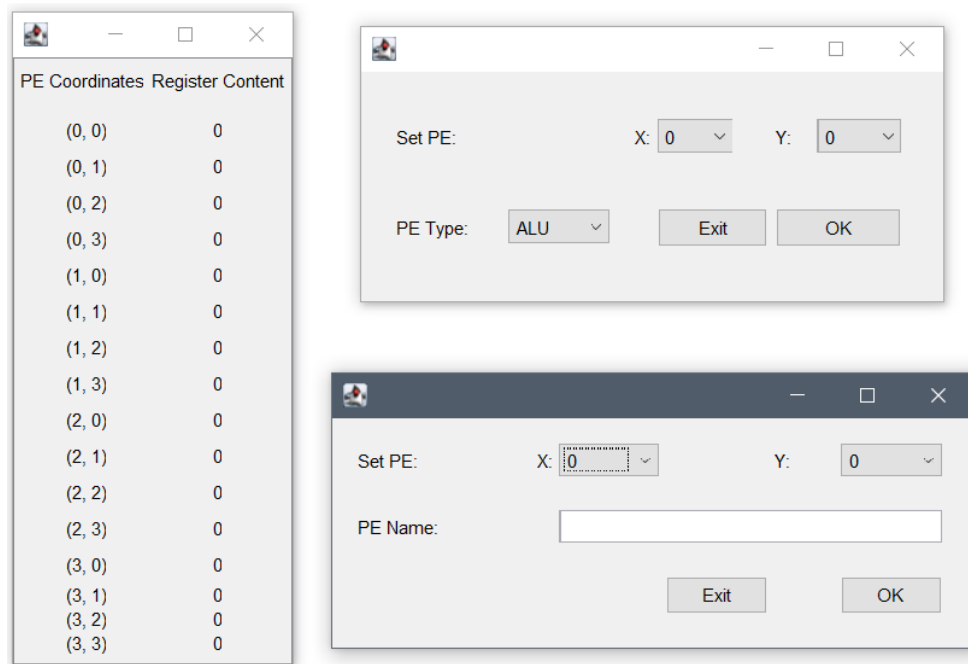


Figure 4.5: CGRA windows for viewing registers, changing PE type, and setting a label

- Complex number Division

There is plenty of room for adding other kernels. Many different authors explore different ways to use a CGRA, mostly for data-intensive applications. They can be, for example, Search algorithms, Sorting algorithms, Image Processing/filtering, Fast-Fourier Transform, Machine Learning kernels (MLP, Gaussian, etc), Video encoding, Cryptography, Complex Maths, Floating-point maths, etc.

Chapter 5

Results

In this chapter the results of an algorithm implementation will be shown, for validation of the initial hypothesis, and remarks for future work to be done.

5.1 Algorithm Implementation

We can validate the results of the development of the Coarse-Grain-Reconfigurable-Array ([CGRA](#)) Simulator and configuration through a custom Instruction Set, by implementing a known algorithm. Then, we feed test data to the algorithm and compare the output values to the expected theoretical values. They are expected to be the same.

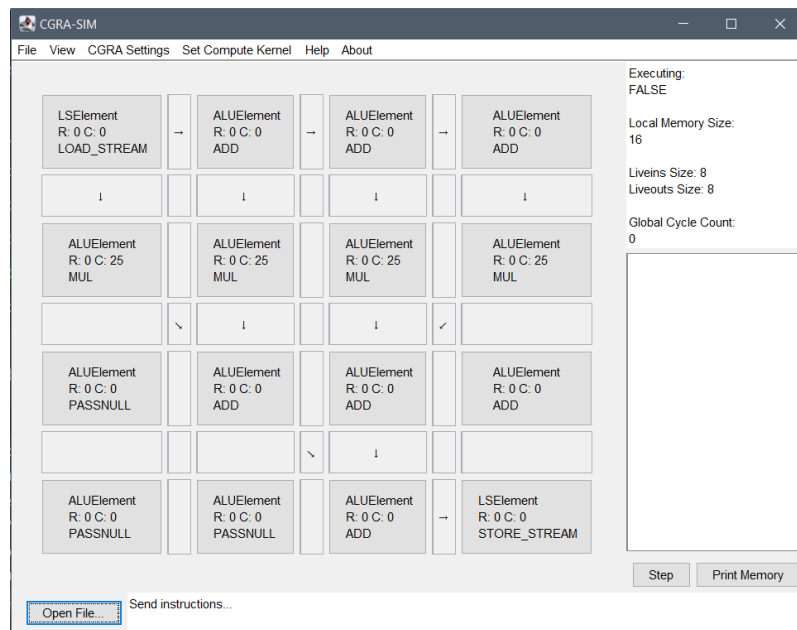


Figure 5.1: CGRA main window with FIR filter preset

For this, the algorithm chosen was the FIR Filter kernel. More specifically, an implementation of a 4th order simple averaging filter, which outputs the average of the last 4 samples of a data

stream. This can also be seen as a low pass filter. It only requires a 4x4 CGRA, with one input and one output LSUs. Implementing this algorithm was straightforward by exploiting the inher-

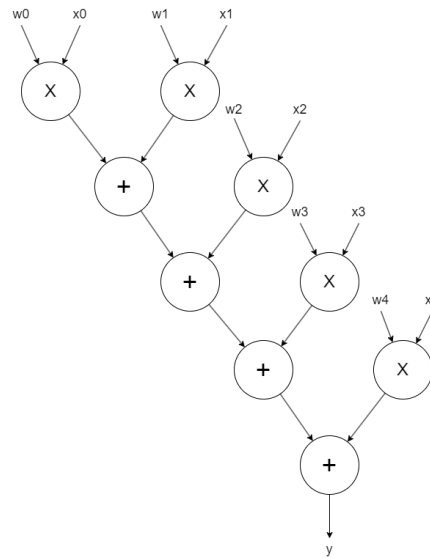


Figure 5.2: CDFG Diagram for a 4-tap FIR filter

ent delay nature of the propagation of data. In other words, propagating data to another adjacent Processing Element (PE) performs the exact same as a Z^{-1} block, which is a delay in Z notation. This allows for easy adaptation of discrete frequency-domain algorithms, such as the FFT, convolutions, 2D convolutions (such as a gaussian blur filter), and more. However, this inherent delay with each propagation movement can be a hindrance to other algorithms, or produce undesired effects. Besides, one can see a small limitation of a CGRA already, which is the necessary hopping between many Arithmetic Logic Unit (ALU) elements when executing an operation with many input operands. The required instructions for this algorithm can be seen on Appendix B. In Figure 5.1 it is shown the FIR filter implementation on the main window. Note that since it only operates with integers, the weight "0.25" is scaled up 100 times, to "25", which means the end result would need to be scaled down by a factor of 100 again. On a hardware perspective, it would make more sense to scale in powers of 2, leveraging the simple bit-shift operation instead of the much costlier multiplications and divisions.

For comparison, a similar FIR filter was implemented in C code, which can be seen on Appendix C. It also produces results scaled by 100x, for ease of comparison. The C algorithm yielded the results in figure 5.3, which we can compare to the results obtained by saving the local CGRA memory to a file (as shown in 5.4

```
0 - 2050
1 - 2175
2 - 7025
3 - 7975
4 - 5950
5 - 5950
6 - 2050
7 - 1800
8 - 2500
9 - 4500
10 - 25900
11 - 33900
12 - 33225
13 - 31300
14 - 10625
15 - 2000

...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 5.3: FIR filter results from C algorithm

```
1 - 0
2 - 0
3 - 0
4 - 0
5 - 2050
6 - 2175
7 - 7025
8 - 7975
9 - 5950
10 - 5950
11 - 2050
12 - 1800
13 - 2500
14 - 4500
15 - 25900
16 - 33900
17 - 33225
18 - 31300
19 - 10625
20 - 2000
```

Figure 5.4: FIR filter results from CGRA implementation

From the [CGRA](#) results we can notice one of the aforementioned undesired effects: the delay of the summing tree also delays the storage of the results, hence the 0's being stored in memory before the results appear. But the results are otherwise correct and the same as those produced by the C algorithm.

In terms of performance, it is notoriously difficult to compare benchmarks on modern processors. Out-of-order executions means some processing units are capable of reaching an Instructions-per-clock ([IPC](#)) value greater than 1, but these do not take into account possible penalties incurred by branch prediction misses and pipeline stalls. There are several papers showing how, due to this, synthetic benchmarks do not equate to real life performance, and increasing the pipeline does

not always equate to performance improvements[58]. However, some authors of RISC-V design topologies have produced benchmarks for several computing algorithms, such as FIR filters [59] and 2D convolutions [60] [61]. Processors such as PULP are in-order and as such, we can compare with the theoretical performance of the CGRA. These works show that those processors usually reach an IPC number close to 1. However, a direct comparison still cannot be made, as CGRAs are a different kind of hardware accelerator. In this case it was shown that the only instructions used are mostly for configuring the CGRA topology, and with context switching, they can execute computing kernels with little instructions. Also, simulation cycles are not equivalent to real hardware clock cycles.

Chapter 6

Conclusion

Nowadays there are many different CGRA topologies released by researchers all over the world, although most of them have not reached the tapeout phase, having remained only as theoretical concepts. This is due to, among other things, the lack of standardization for the interconnection between the CGRA and a host processor, which creates overhead and increases complexity. There is also much research done on methods of binary acceleration and dataflow integration on processors. Nowadays there are also many open-source tools, or at least easily accessible, for system simulation and testing, utilizing varied architectures. What remains to be done is mostly integration and adaptation work, where one combines the different tools, architectures, and topologies, in order to explore new ideas and systems, or improving upon old ones. Therefore, unfortunately not much work is present yet on full CPU + CGRA integration, specially regarding RISC-V architectures, which is what we are more focused on. Hopefully this dissertation will allow and facilitate others to perform more integrations by utilizing an architecture-agnostic protocol of communication via a custom ISA.

6.1 Future Work

This work leaves a lot of space for exploring new ideas and implementations of Coarse-Grain-Reconfigurable-Arrays (CGRAs). The simulator itself permits a lot of design space exploration and algorithm testing. However, its current CGRA design implementation cannot directly translate to hardware. Some Arithmetic Logic Unit (ALU) operations have very high latency, and the memory access is totally abstracted. Still, the scalability and efficiency of CGRAs is a proven concept. For the future, one could explore the following points:

- Optionally creating the physical representation of the high-level CGRA simulation in a hardware description language, such as Verilog (this could be done automatically, exploring other tools such as CrispyHDL, developed on the SPeCS Lab)
- Test the HDL code through a low-level simulator, present in software such as ModelSim or similar - for RTL and gate-level simulation, validating the CGRA model.

- Optionally, research RISC-V development tools mentioned in previous chapters, and utilize them to fit the microcode as a custom instruction set extension, then later validate a possible design of a [CGRA](#) loosely coupled to a RISC-V host Central Processing Unit ([CPU](#)) (Chipyard, Rocket core, FireSim/gem5, etc.)
- Explore other preset algorithms, utilising context switching to its full potential;

Appendix A

Verilog Code for ALU Processing Element

Listing A.1: Example of a Processing Element (PE) implementing an ALU unit

```
1  module alu(  
2      input [7:0] A,B, // ALU 8-bit Inputs  
3      input [3:0] Sel, // ALU Selection  
4      output [7:0] Out, // ALU 8-bit Output  
5  );  
6  
7      reg [15:0] result;  
8      assign Out = result[7:0]; //wire out  
9  
10     always @(posedge)  
11     begin  
12         case(Sel)  
13             4'b0000: // Pass Null  
14                 result <= 8'b0;  
15             4'b0001: // Sum  
16                 result <= A + B  
17             4'b0010: // Sub  
18                 result <= A - B;  
19             4'b0011: // Mul  
20                 result <= A * B;  
21             4'b0100: // Div (combinational)  
22                 result <= A/B;  
23             4'b0101: // Logical shift left  
24                 result <= A<<1;  
25             4'b0110: // Logical shift right  
26                 result <= A>>1;  
27             4'b0111: // And  
28                 result <= A & B;  
29             4'b1000: // Or  
30                 result <= A | B;  
31             4'b1001: // Xor  
32                 result <= A ^ B;  
33             4'b1010: // output A  
34                 result <= A;  
35             4'b1011 // output B
```

```
36     result <= B;
37     4'b1100: // the smaller
38     result <= (A<B) ? A : B;
39     4'b1101 //pass if same
40     result <= (A==B) ? A : 8'b0;
41
42     default: result <= 8'b0 ;
43     endcase
44 end
45 endmodule
```

Appendix B

Set of instructions for loading the FIR algorithm onto the CGRA-SIM

Listing B.1: FIR algorithm in CGRA-SIM

```
1 config_setpe 0 0 2
2 config_setpe 3 3 2
3 set 0 0 2
4 set 1 0 1
5 set 2 0 1
6 set 3 0 1
7 set 0 1 3
8 set 1 1 3
9 set 2 1 3
10 set 3 1 3
11 set 1 2 1
12 set 2 2 1
13 set 3 2 1
14 set 2 3 1
15 set 3 3 1
16 set 3 3 4
17 set_io 1 2 0 1 1 1
18 set_io 2 2 2 1 3 1
19 set_io 0 1 0 0 3 3
20 set_io 1 0 0 0 3 3
21 set_io 2 0 1 0 3 3
22 set_io 3 0 2 0 3 3
23 set_io 1 1 1 0 3 3
24 set_io 2 1 2 0 3 3
25 set_io 3 1 3 0 3 3
26 set_io 2 3 1 2 2 2
27 set_io 3 3 1 1 2 3
28 set_input_constant 1 0 1 1
29 set_input_constant 2 0 1 1
30 set_input_constant 3 0 1 1
31 set_input_constant 0 1 1 1
32 set_input_constant 1 1 1 1
33 set_input_constant 2 1 1 1
34 set_input_constant 3 1 1 1
35 set_constant 0 1 25
```

```
36 set_constant 1 1 25  
37 set_constant 2 1 25  
38 set_constant 3 1 25
```

Appendix C

C code for a FIR filter algorithm with set input data

Listing C.1: FIR C algorithm with fixed input data

```
1 #include <stdio.h>
2 #include <math.h>
3
4 void low_pass_fir_filter(int *input_signal, int *output_signal, int signal_length) {
5
6     int coefficients[4] = {25, 25, 25, 25};
7
8     for (int i = 0; i < signal_length; i++) {
9         int filtered_value = 0;
10        for (int j = 0; j < 4; j++) {
11            if (i - j >= 0) {
12                filtered_value += coefficients[j] * input_signal[i - j];
13            }
14        }
15        output_signal[i] = filtered_value; // Divide by 100 to scale back to original range
16    }
17 }
18
19 int main() {
20     int signal_length = 16;
21     int input_signal[16] = {82, 5, 194, 38, 1, 5, 38, 28, 29, 85, 894, 348, 2, 8, 67, 3};
22
23     int output_signal[signal_length];
24
25     low_pass_fir_filter(input_signal, output_signal, signal_length);
26
27     for (int i = 0; i < signal_length; i++) {
28         printf("%d - %d\n", i, output_signal[i]);
29     }
30
31     return 0;
32 }
```

References

- [1] Aviral Shrivastava, Jared Payer, Reiley Jeyapaul, Mahdi Hamzeh, and Sarma Vrudhula. Enabling multithreading on cgras. In *2011 International Conference on Parallel Processing*, pages 255–264, 2011. doi:[10.1109/ICPP.2011.77](https://doi.org/10.1109/ICPP.2011.77).
- [2] Mirsky and DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, April 1996. doi:[10.1109/FPGA.1996.564808](https://doi.org/10.1109/FPGA.1996.564808).
- [3] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M.C. Filho, and Vladimir Castro Alves. Design and implementation of the morphosys reconfigurable computing processor. *The Journal of VLSI Signal Processing*, 24(2/3):147–164, 2000. URL: <http://link.springer.com/10.1023/A:1008189221436>, doi:[10.1023/A:1008189221436](https://doi.org/10.1023/A:1008189221436).
- [4] Yoonjin Kim, M. Kiemb, C. Park, Jinyong Jung, and Kiyoun Choi. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Design, Automation and Test in Europe*, pages 12–17 Vol. 1, 2005. doi:[10.1109/DATE.2005.260](https://doi.org/10.1109/DATE.2005.260).
- [5] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In Peter Y. K. Cheung and George A. Constantinides, editors, *Field Programmable Logic and Application*, Lecture Notes in Computer Science, pages 61–70, Berlin, Heidelberg, 2003. Springer. doi:[10.1007/978-3-540-45234-8_7](https://doi.org/10.1007/978-3-540-45234-8_7).
- [6] Jong eun Lee, Kiyoun Choi, and N.D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design & Test of Computers*, 20(1):26–33, 2003. doi:[10.1109/MDT.2003.1173050](https://doi.org/10.1109/MDT.2003.1173050).
- [7] Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper. URL: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>.
- [8] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey. *ACM Computing Surveys*, 53(1):6:1–6:36, February 2020. doi:[10.1145/3369764](https://doi.org/10.1145/3369764).
- [9] Russell Tessier, Kenneth Pocek, and André DeHon. Reconfigurable Computing Architectures. *Proceedings of the IEEE*, 103(3):332–354, March 2015. doi:[10.1109/JPROC.2014.2386883](https://doi.org/10.1109/JPROC.2014.2386883).

- [10] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3):202–210, 2005. URL: <http://www.gotw.ca/publications/concurrency-ddj.htm>.
- [11] Lee Moore, Simon Davidmann, Larry Lapides, and Carl Shaw. EW19 Paper on Methodology for Implementation of Custom Instructions in the RISC-V Architecture | Imperas - Embedded Software Development. URL: <https://www.imperas.com/articles/ew19-paper-methodology-implementation-custom-instructions-risc-v-architecture>.
- [12] Michael Bedford Taylor. The evolution of bitcoin hardware. *Computer*, 50(9):58–66, 2017. doi:10.1109/MC.2017.3571056.
- [13] Kaz Sato and Cliff Young. An in-depth look at google's first tensor processing unit (tpu) | google cloud blog. URL: <https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu>.
- [14] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi:10.1145/2228360.2228584.
- [15] History - risc-v international, Jan 2023. URL: <https://riscv.org/about/history/>.
- [16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018. ISSN: 2575-713X. URL: <https://ieeexplore.ieee.org/abstract/document/8416814>, doi:10.1109/ISCA.2018.00012.
- [17] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), jul 2015. doi:10.1145/2788396.
- [18] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, San Francisco California, June 2015. ACM. URL: <https://dl.acm.org/doi/10.1145/2744769.2744794>, doi:10.1145/2744769.2744794.
- [19] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.
- [20] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning, October 2019. arXiv:1907.10701 [cs, stat]. URL: <http://arxiv.org/abs/1907.10701>, doi:10.48550/arXiv.1907.10701.
- [21] Tom Vander Aa, Praveen Raghavan, Scott Mahlke, Bjorn De Sutter, Aviral Shrivastava, and Frank Hannig. Compilation techniques for CGRAs: exploring all parallelization approaches. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 185–186, New York, NY,

- USA, October 2010. Association for Computing Machinery. doi:10.1145/1878961.1878995.
- [22] Gabe Loh, Murali Thottethodi, and Gurindar Singh. Instruction-level parallelism in the era of power-constrained multi-core processors. *Proceedings of the IEEE*, 103(8):1391–1418, 2015.
- [23] Carlos Carvalho. The gap between processor and memory speeds. 01 2002.
- [24] Sarah L. Harris and David Money Harris. 7 - microarchitecture. In Sarah L. Harris and David Money Harris, editors, *Digital Design and Computer Architecture*, pages 384–484. Morgan Kaufmann, Boston, 2016. URL: <https://www.sciencedirect.com/science/article/pii/B9780128000564000078>, doi:<https://doi.org/10.1016/B978-0-12-800056-4.00007-8>.
- [25] Xeon Gold 6148 - Intel - WikiChip. URL: https://en.wikichip.org/wiki/intel/xeon_gold/6148#Frequencies.
- [26] Roopak Sinha, Parthasarathi Roop, and Samik Basu. *The AMBA SOC Platform*, pages 11–23. Springer New York, New York, NY, 2014. URL: https://doi.org/10.1007/978-1-4614-7864-5_2, doi:10.1007/978-1-4614-7864-5_2.
- [27] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association. URL: <https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls>.
- [28] N.T. Clark, H. Zhong, and S.A. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, 54(10):1258–1270, October 2005. doi:10.1109/TC.2005.156.
- [29] Ricardo Ferreira, Waldir Denver, Monica Pereira, Jorge Quadros, Luigi Carro, and Stephan Wong. A run-time modulo scheduling by using a binary translation mechanism. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 75–82, July 2014. doi:10.1109/SAMOS.2014.6893197.
- [30] Matthew A. Watkins, Tony Nowatzki, and Anthony Carno. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 138–150, March 2016. ISSN: 2378-203X. doi:10.1109/HPCA.2016.7446060.
- [31] David Padua, Amol Ghoting, John A. Gunnels, Mark S. Squillante, José Meseguer, James H. Cownie, Duncan Roweth, Sarita V. Adve, Hans J. Boehm, Sally A. McKee, Robert W. Wisniewski, George Karypis, Allen D. Malony, Steven Gottlieb, Rolf Riesen, Arthur B. Maccabe, Gianfranco Bilardi, Andrea Pietracaprina, Arun Kejariwal, Alexandru Nicolau, Christian Lengauer, John L. Gustafson, William Gropp, Jean-Pierre Prost, David Padua, Geoff Lowney, Patrick Amestoy, Alfredo Buttari, Iain Duff, Abdou Guermouche, Jean-Yves L’Excellent, Bora Uçar, Robert H. Halstead, Mario Nemirovsky, Patrick Amestoy, Alfredo Buttari, Iain Duff, Abdou Guermouche, Jean-Yves L’Excellent, Bora Uçar, and Scott Pakin. Modulo Scheduling and Loop Pipelining. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1158–1173. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_65.

- [32] Jong Kyung Paek, Kiyoungh Choi, and Jongeun Lee. Binary acceleration using coarse-grained reconfigurable architecture. *ACM SIGARCH Computer Architecture News*, 38(4):33–39, January 2011. doi:[10.1145/1926367.1926374](https://doi.org/10.1145/1926367.1926374).
- [33] CCCP: Coarse-Grained Reconfigurable Architecture.
- [34] Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava, and Reiley Jeyapaul. LASER: A hardware/software approach to accelerate complicated loops on CGRAs. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1069–1074, March 2018. ISSN: 1558-1101. doi:[10.23919/DATE.2018.8342170](https://doi.org/10.23919/DATE.2018.8342170).
- [35] Aaron Wood and Scott Hauck. Offset pipelined scheduling: Conditional branching for cgras. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 227–230, 2015. doi:[10.1109/FCCM.2015.51](https://doi.org/10.1109/FCCM.2015.51).
- [36] Jackson Melchert, Kathleen Feng, Caleb Donovanick, Ross Daly, Clark Barrett, Mark Horowitz, Pat Hanrahan, and Priyanka Raina. Automated Design Space Exploration of CGRA Processing Element Architectures using Frequent Subgraph Analysis. *arXiv:2104.14155 [cs]*, April 2021. arXiv: 2104.14155.
- [37] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. RAMP: Resource-Aware Mapping for CGRAs. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018. doi:[10.1109/DAC.2018.8465892](https://doi.org/10.1109/DAC.2018.8465892).
- [38] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee W. Yoon, Doosan Cho, and Yunheung Paek. High Throughput Data Mapping for Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1599–1609, November 2011. doi:[10.1109/TCAD.2011.2161217](https://doi.org/10.1109/TCAD.2011.2161217).
- [39] Jared Pager. Improving CGRA utilization by enabling multi-threading for power-efficient embedded systems. Technical report, Arizona State University, 2011. URL: <https://keep.lib.asu.edu/items/150460>.
- [40] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective. *IEEE Access*, 8:146719–146743, 2020. doi:[10.1109/ACCESS.2020.3012084](https://doi.org/10.1109/ACCESS.2020.3012084).
- [41] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.*, 52(6), oct 2019. doi:[10.1145/3357375](https://doi.org/10.1145/3357375).
- [42] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Tomás Lang. Morphosys: An integrated re-configurable architecture. 2000. URL: <https://api.semanticscholar.org/CorpusID:59778117>.
- [43] Joao D Lopes and José T de Sousa. Versat, a minimal coarse-grain reconfigurable array. In *International Conference on Vector and Parallel Processing*, pages 174–187. Springer, 2016.
- [44] R. Hartenstein. Coarse grain reconfigurable architectures. In *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, pages 564–569, February 2001. doi:[10.1109/ASPDAC.2001.913368](https://doi.org/10.1109/ASPDAC.2001.913368).

- [45] Andrew Waterman, Krste Asanovic, and John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. Risc-v international edition, December 2021.
- [46] Adding Custom Instructions to the RISC-V GNU-GCC toolchain. URL: <https://hsandid.github.io/posts/risc-v-custom-instruction/>.
- [47] Duncan Graham and Simon Davidmann. Optimizing RISC-V Custom Instructions with Software Driven Analysis and Profiling. URL: <https://www.youtube.com/watch?v=GrHB66Zpmng>.
- [48] Five Embeddev. RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA. URL: <https://www.five-embeddev.com/riscv-isa-manual/latest/rvc-opcode-map.html>.
- [49] OpenHW Group CVA6 User Manual — CVA6 documentation. URL: <https://docs.openhwgroup.org/projects/cva6-user-manual/>.
- [50] PULP platform. URL: <https://www.pulp-platform.org/index.html>.
- [51] RISC-V. BOOM. RISC-V BOOM. URL: <https://boom-core.org/>.
- [52] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4, 2016.
- [53] Luís Fiolhais, Fernando Gonçalves, Rui P. Duarte, Mário Véstias, and Jose T. de Sousa. Low energy heterogeneous computing with multiple risc-v and cgra cores. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2019. doi:10.1109/ISCAS.2019.8702538.
- [54] Rui Santiago, João Lopes, and José Sousa. Compiler for the versat reconfigurable architecture. January 2017.
- [55] Biagio Peccerillo, Elham Cheshmikhani, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. Ixiam: Isa extension for integrated accelerator management. *IEEE Access*, 11:33768–33791, 2023. doi:10.1109/ACCESS.2023.3264265.
- [56] Mihaela Damian, Julian Oppermann, Christoph Spang, and Andreas Koch. Scaie-v: an open-source scalable interface for isa extensions for risc-v processors. In *Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22*, page 169–174, New York, NY, USA, 2022. Association for Computing Machinery. URL: <https://doi.org/10.1145/3489517.3530432>, doi:10.1145/3489517.3530432.
- [57] Xiaoyi Ling, Takahiro Notsu, and Jason Anderson. An open-source framework for the generation of risc-v processor + cgra accelerator systems. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 35–42, 2021. doi:10.1109/DSD53832.2021.00015.
- [58] Janmartin Jahn. Resource Allocation for Software Pipelines in Many-core Systems. 2014. URL: <https://publikationen.bibliothek.kit.edu/1000040472>, doi:10.5445/IR/1000040472.

- [59] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gurkaynak, and Luca Benini. A near-threshold RISC-V core with DSP extensions for scalable IoT Endpoint Devices. 2016. URL: <https://arxiv.org/abs/1608.08376>, doi:10.48550/ARXIV.1608.08376.
- [60] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision. *Journal of Signal Processing Systems*, 84(3):339–354, September 2016. URL: <http://link.springer.com/10.1007/s11265-015-1070-9>, doi:10.1007/s11265-015-1070-9.
- [61] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017. doi:10.1109/PATMOS.2017.8106976.