EPFL

Thèse n° 9353

Exploring brain-inspired multi-core heterogeneous hardware templates for low-power biomedical embedded systems

Présentée le 31 mars 2023

Faculté des sciences et techniques de l'ingénieur
Laboratoire des systèmes embarqués
Programme doctoral en génie électrique

pour l'obtention du grade de Docteur ès Sciences

par

Benoît Walter DENKINGER

Acceptée sur proposition du jury

Prof. G. De Micheli, président du jury
Prof. D. Atienza Alonso, directeur de thèse
Prof. L. Pozzi, rapporteuse
Prof. K. Olcoz Herrero, rapporteuse
Prof. A. Burg, rapporteur

École
polytechnique
fédérale
de Lausanne

2023

The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity. — Albert Einstein

To my love Margot.
To my sons Gaspard and Marius.
To my parents Ivan and Sylvie.

-

A mon amour Margot.
A mes fils Gaspard et Marius.
A mes parents Ivan and Sylvie.

Acknowledgements

I am grateful to my P.hD. supervisor Professor David Atienza for all his support and guidance throughout the years. It all started in 2017 when I finished my master's thesis at EPFL. Thanks to a shared contact, Dr. Martino Ruggiero, I met David to discuss the possibility of starting as a Ph.D. student at the ESL. At that time, I knew very little about the topic, but I was very interested in David's proposal, and I would like to thank him for giving me this chance. Since then, I've constantly been learning new things thanks to David and the incredible team he created at ESL, making it a very stimulating environment, and I thank every person I met and worked with at ESL. I've always felt listened to, respected, guided, and trusted throughout the years, and I'll always be thankful to David for that.

This thesis research would not have been possible without the incredible support of the best post-doc: Dr. Miguel Peon-Quiros. Miguel joined ESL a few weeks before me, and I had the privilege of working with him until the end of my Ph.D. He is a fantastic source of knowledge on many subjects (not only scientific ones) and a very kind person. He has advised me and answered my doubts countless times. For all these reasons, I'll always be grateful to him.

I also had the incredible chance to collaborate during my thesis with the Interuniversity Microelectronics Centre (IMEC) in the Netherlands and Belgium. This collaboration has been possible thanks to Professor Francky Catthoor and Dr. Mario Konijnenburg, who actively followed me during my Ph.D. Francky is an endless well of knowledge with considerable expertise in many domains allowing him to guide research in the most promising directions. Although I did not always understand his global vision of my work and the reasons behind topics and directions he wanted to explore, he never forced it and let me try and understand the reasoning myself. It allowed me to learn many things during our collaboration. Mario did a fantastic job helping me set up the infrastructure and the simulation environment that allowed me to conduct my research. Mario is a passionate engineer with much expertise in research and industrial fields. He has always found time to help me whenever needed. I want to thank Francky and Mario (and IMEC through them) for their support during this four years collaboration, and I'll always be thankful for that and for everything I learned through them.

I also want to thank all the jury members: Prof. Andreas Burg, Prof. Katzalin Olcoz, Prof. Laura Pozzi, Prof. Andreas Burg, and Prof. Giovanni De Micheli. They dedicated

time to reading and evaluating my work. They raised interesting questions during my private defense and provided helpful feedback about my thesis. For these reasons, I'm very grateful to them.

Finally, I want to thank my family for being essential to succeeding in my thesis (and life in general). In particular, to the one person I have loved for more than half my life: Margot. We have already been through so many years of laughing, traveling, being patient, dreaming, crying, supporting, and much more. All these years together give us the envy and the courage to start our own family, with Gaspard first and Marius very recently. Kids and a P.hD. might not seem very compatible initially, but they help me escape and put in perspective the demanding work of a thesis. It is true we could have found a slightly better timing: finishing a thesis with an 18-month-old kid and another on the way is quite physically and mentally intensive. But I never doubted our strength to go through this, and once more, Margot supported me, encouraged me, and did everything she could to help me. Thanks, Margot, for that, for all these years and all the future ones to come. I also want to thank my parents, Sylvie and Ivan, for their unconditional love and support. Having children myself, I'm starting to realize everything they have done for me and all the essential values they transmitted to me. Thank you to my brothers and sister, Alexis, Camille, and Julien, which are also a source of inspiration. We have lived very different lives, but we managed to stay close, and they have always been very important to me.

Lausanne, February 6, 2023 Benoît W. Denkinger

Abstract

The miniaturization of integrated circuits (ICs) and their higher performance and energy efficiency, combined with new machine learning algorithms and applications, have paved the way to intelligent, interconnected edge devices. Many domains could benefit from their global higher efficiency —in costs, energy, performance, and environmental impact. In the medical domain, they could revolutionize how healthcare services are delivered to people. For example, continuous monitoring of a person's biosignals could help detect mental or physiological health conditions earlier and better prevent their potential degradation. It also enables more personalized treatments based on the data of a patient. Ultimately, these data could create a digital twin of a person used to simulate multiple treatments and find the optimal one.

Some systems already take advantage of today's ICs computation power and energy efficiency by providing long-term monitoring of some biosignals. For example, medically certified patches that can record throughout days and weeks exist, while this was only possible in medical centers before and for a short period. However, the stringent constraints due to the current certification regulations limit their optimizations. Therefore, their use is reserved for advanced monitoring of patients with a high suspicion of certain diseases, not for prevention purposes. On the other hand, publicly available devices that record various biosignals already exist, such as smartwatches and fitness trackers. These devices, connected together, could create wireless body area networks (WBANs) that could democratize personalized and preventive healthcare services. However, these devices usually require, in the best case, a daily charge which limits their monitoring capability. Additionally, these devices must approach the high quality of clinical devices used in hospitals. This is the key to providing proper monitoring and diagnosis in daily situations that reduce healthcare costs.

Therefore, higher efficiency is required, but tasks embedded systems with two opposite goals: low-power operation and high performance. The current trend to reach these goals is toward heterogeneous platforms, including multi-core architectures with heterogeneous cores and hardware accelerators. The latter can be divided into custom non-programmable accelerators and flexible (programmable) domain-specific cores. Fixed-function or custom accelerators, referred to as application-specific integrated circuits (ASICs) in this thesis, are very efficient at implementing a particular functionality for a given set of constraints

(e.g., fast Fourier transform (FFT) or finite impulse response (FIR) filter engines). However, they are inflexible when facing application-wide optimizations or functionality upgrades. Conversely, programmable cores, referred to as domain-specific instruction-set processors (DSIPs) in this thesis, offer higher flexibility but often with a penalty in area, performance, and, above all, energy consumption.

This thesis explores the performance versus flexibility tradeoff at the architecture level to advance the Pareto front of current solutions. In particular, the goal is to find innovative architectural features that improve the energy efficiency of embedded devices while maintaining (or increasing) their throughput. This exploration has led to VWR2A, an heterogeneous DSIP architecture template targeting the biomedical domain that integrates high computational density and low-power memory structures (i.e., very-wide registers and scratchpad memories). Compared to two state-of-the-art programmable architectures targeting the biomedical domain, an ARM Cortex-M4 based system-on-chip (SoC) and a baseline coarse-grained reconfigurable array (CGRA), the VWR2A instance displayed an energy-delay product (EDP) improvement of $104.8\times$ and $19.8\times$, respectively.

In addition, VWR2A enables the generation of architectures that narrow or close the energy and performance gap at the kernel level with respect to ASICs. In particular, one VWR2A instance has shown similar or better performance on FFT and FIR filter kernels compared to an FFT and a matrix processor ASICs, respectively. In terms of energy, at the kernel level, the VWR2A instance is still $4.9\times$ less efficient than the FFT ASIC, but consumes $22.7\,\%$ less energy than the matrix processor. However, VWR2A provides the flexibility to accelerate multiple kernels, resulting in significant energy savings at the application level compared to ASIC-based designs, with an EDP improvement of $4.4\times$. These results support the main hypothesis of this thesis: that programmable accelerators (i.e., DSIPs) can increase the energy efficiency of embedded devices, particularly in the biomedical domain.

Finally, as VWR2A remains fully programmable, it can execute most of the source code. In particular, control-intensive kernels, which are often present at the application level and usually left to be executed by the CPU (i.e., a general-purpose processor (GPP)), can be mapped. One VWR2A instance optimized for such code has demonstrated higher performance and energy efficiency compared to an ARM Cortex-M4 processor and a RISC-V Ibex processor. At the application level, the overhead of the programmability (compared to ASICs) is largely compensated with higher code coverage. This results in an EDP improvement as significant as $27.6\times$ when both control-intensive and data-intensive (e.g., FFT, FIR filter, median) kernels are executed by a VWR2A instance compared to an SoC using a GPP+ASICs combination.

Résumé

La miniaturisation des circuits intégrés (integrated circuits (ICs)) a permis une augmentation de leurs performances et de leur efficacité énergétique. Combinées à de nouveaux algorithmes et applications basées sur l'apprentissage automatique (machine learning), ces circuits ont ouvert la voie à des dispositifs de périphérie (edge devices) intelligents et interconnectés. De nombreux domaines pourraient bénéficier de leur meilleure efficacité globale en termes de coûts, d'énergie, de performances et d'impact environnemental. Dans le domaine médical, ils pourraient révolutionner la manière dont la population accède aux services de santé. Par exemple, la surveillance continue des signaux biologiques d'une personne pourrait aider à détecter plus tôt des problèmes de santé mentale ou physiologique et à mieux prévenir leurs dégradations potentielles. Ces appareils pourraient également permettre des traitements plus personnalisés basés sur les données des patients. L'ultime but étant de créer un jumeau numérique d'une personne, reposant sur ses données biologiques, sur lequel plusieurs traitements peuvent être simulés pour trouver celui qui est optimal.

Certains systèmes tirent déjà parti de la puissance de calcul et de l'efficacité énergétique des circuits intégrés existants et permettent une surveillance à long terme de certains signaux biologiques. Par exemple, des patchs médicaux existent et peuvent enregistrer sur plusieurs jours, voir semaines, alors que, précédemment, cela n'était possible que dans des centres médicaux et pour une courte période. Cependant, les réglementations en vigueur imposent des contraintes importantes qui limitent l'optimisation de ces appareils requérant une certification médicale. De ce fait, leur utilisation est réservée pour un suivi médical des patients avec une forte suspicion de certaines maladies, et non à des fins de prévention. D'un autre côté, il existe des appareils qui sont accessibles au grand public et qui enregistrent divers signaux biologiques, tels que les montres intelligentes (smartwatches) et les traqueurs de fitness. Ces appareils, connectés ensemble, pourraient créer des réseaux de capteurs corporels sans fil (wireless body area networks (WBAN)) qui pourraient démocratiser les services de santé personnalisés et préventifs. Cependant, ces appareils nécessitent généralement, dans le meilleur des cas, une recharge journalière ce qui limite leur capacité de surveillance. De plus, ces appareils doivent fournir une information d'une précision aussi proche que possible des appareils cliniques utilisés dans

les hôpitaux. C'est le point essentiel pour assurer une prévention efficace au quotidien et ainsi réduire les coûts des systèmes de santé.

Pour atteindre ces objectifs, il est nécessaire d'atteindre une plus grande efficacité globale, mais cela confronte les systèmes portatifs (embedded systems) avec deux objectifs opposés : un fonctionnement à faible consommation d'énergie et des performances élevées. La tendance actuelle pour atteindre ces objectifs est de développer des plates-formes hétérogènes, avec des architectures multicœurs et des cœurs hétérogènes ainsi que des accélérateurs matériels (hardware accelerators). Ces derniers peuvent être divisés en accélérateurs spécialisés non programmables et en accélérateurs flexibles (programmables) spécifiques à un domaine. Les accélérateurs avec une fonction dédiée ou spécialisée, appelés circuits intégrés spécifiques à l'application (application-specific integrated circuits (ASIC)) dans cette thèse, sont très efficaces pour exécuter une certaine fonctionnalité sous des contraintes données (par exemple, une transformée de Fourier rapide (FFT) ou des filtres à réponse impulsionnelle finie (FIR)). Cependant, ce type d'accélérateur est inadapté aux optimisations ou aux potentielles mises à niveau d'une application à cause de sa non-programmabilité. À l'inverse, les accélérateurs programmables, appelés processeurs à jeu d'instructions spécifiques au domaine (domain-specific instruction-set processors (DSIP)) dans cette thèse, offrent une plus grande flexibilité mais souvent avec une pénalité en termes de dimensions, de performances et, surtout, de consommation d'énergie.

Cette thèse explore le compromis entre performance et flexibilité au niveau de l'architecture matérielle dans le but d'améliorer les architectures programmables existantes. En particulier, l'objectif est de trouver des fonctionnalités architecturales innovantes qui améliorent l'efficacité énergétique des dispositifs portatifs tout en maintenant (ou en augmentant) leur puissance de calcul. Cette exploration a conduit à VWR2A, un modèle d'architecture DSIP hétérogène ciblant le domaine biomédical et qui intègre une densité de puissance de calcul élevée et des structures de mémoire à faible puissance (c'est-à-dire des registres très larges et des mémoires scratchpad). Par rapport à deux architectures programmables de pointe ciblant le domaine biomédical, un système-sur-puce (system-on-chip (SoC)) basé sur un processeur ARM Cortex-M4 et une matrice reconfigurable à gros grains (coarse-grained reconfigurable array (CGRA)), une instance de VWR2A a démontré une amélioration du produit énergie-délai (energy-delay product (EDP)) de $104.8\times$ et $19.8\times$, respectivement.

De plus, VWR2A permet la génération d'architectures qui réduisent ou comblent l'écart de performances et de consommation d'énergie au niveau d'un noyau (kernel) par rapport aux ASICs. En particulier, une instance de VWR2A a montré des performances similaires ou meilleures sur les noyaux de FFT et filtre FIR par rapport à un accélérateur spécialisé (c'est-à-dire un ASIC) pour exécuter des FFTs et un pour le calcul matriciel. En termes d'énergie, au niveau du noyau, l'instance basée sur VWR2A est toujours $4.9\times$ moins efficace que l'ASIC spécialisé pour les FFTs, en revanche il consomme $22.7\,\%$ moins

d'énergie que le processeur matriciel. Cependant, VWR2A offre la flexibilité d'accélérer plusieurs noyaux, ce qui entraîne des économies d'énergie significatives au niveau d'une application par rapport aux systèmes basés sur des ASICs, avec une amélioration de l'EDP de $4.4\times$. Ces résultats soutiennent l'hypothèse principale de cette thèse : que les accélérateurs programmables (c'est-à-dire les DSIPs) peuvent augmenter l'efficacité énergétique des dispositifs portatifs, en particulier dans le domaine biomédical.

Enfin, comme VWR2A est entièrement programmable, il peut exécuter la majeure partie du code source d'un domaine. En particulier, les noyaux à forte intensité de contrôle (c'est-à-dire avec beaucoup d'instructions de contrôle), qui sont souvent présents au niveau d'une application, peuvent aussi être exécuté par VWR2A, alors que généralement ils le sont par un processeur à usage général (general-purpose processor (GPP)). Une instance de VWR2A optimisée pour un tel code a démontré des performances et une efficacité énergétique supérieures à celles d'un processeur ARM Cortex-M4 et d'un processeur RISC-V Ibex. Au niveau d'une application, le coût supplémentaire de la programmabilité (par rapport aux ASICs) est largement compensé par une couverture plus élevée du code d'une application. Cela se traduit par une amélioration de l'EDP aussi élevée que $27.6\times$ lorsque les noyaux à forte intensité de contrôle (control-intensive kernels) et à forte intensité de données (data-intensive kernels) sont exécutés par une instance de VWR2A par rapport à un SoC utilisant une combinaison GPP+ASICs.

Contents

List of Figures

List of Tables

List of Acronyms

ALU arithmetic logic unit

ASIC application-specific integrated circuit

ASIP application-specific instruction-set processor

CGRA coarse-grained reconfigurable array

CISC complex instruction set computer

CMSIS-DSP common microcontroller software interface standard digital signal processing

CNN convolutional neural network

CPI clock per instruction

CPU central processing unit

DFG data flow graph

DMA direct memory access

DNN deep neural network

DSIP domain-specific instruction-set processor

DSP digital signal processor

ECG electrocardiogram

EDP energy-delay product

FFT fast Fourier transform

FIR finite impulse response

FPGA field-programmable gate array

FPU floating-point unit

FU functional unit

GPP general-purpose processor

GPU graphics processing unit

HPC high-performance computing

IC integrated circuit

ILP instruction level parallelism

IoT Internet of things

IPC instructions-per-cycle

ISA instruction set architecture

LCU loop-control unit

LSU load-and-store unit

MF morphological filtering

MXCU multiplexer-control unit

NCD noncommunicable disease

NRE nonrecurring engineering

PC program counter

PE processing element

PPG photoplethysmogram

RC reconfigurable cell

RF register file

RISC reduced instruction set computer

RMS root-mean-square

RTL register transfer level

SIMD single instruction multiple data

SoC system-on-chip

SPM scratchpad memory

SRAM static random access memory

SRF scalar register file

SVM support-vector machine

VLIW very-long instruction word

VWR very-wide register

WBAN wireless body area networks

1 Introduction

Intelligent and autonomous systems are becoming increasingly present in our daily lives: smart buildings [1, 2], autonomous vehicles [3, 4], smartphones, and wearable devices [5, 6] (e.g., for medical or security purposes). These domains benefit from the promise of better global efficiency and cost enabled by these smart devices [7]. In addition, I believe they are one inevitable answer (from the technology side) to the climate crisis by creating a more sustainable society in which human development and the prosperity of our planet are not opposed. Some domains have already embraced these devices and see their benefits, but many more could also take advantage of them. There are different reasons why some areas have not yet incorporated such devices: law restrictions, societal change resistance, or a lack of reasonable engineering solutions.

1.1 Intelligent devices in the biomedical domain

The biomedical domain is one example that could greatly benefit from intelligent devices. First, their use could help prevent some diseases by detecting early abnormal conditions thanks to continuous monitoring of biological signals (biosignals). This could trigger preventive treatment or even prevent death. According to the World Health Organization (WHO) 2019 statistics [8], persons aged between 30 and 69 years old have the most chance (i.e., $40.0\,\%$) of dying of cardiovascular diseases (e.g., ischaemic heart disease, stroke). They are part of what is called noncommunicable diseases (NCDs), which account for $77.0\,\%$ of the total deaths, as shown in Figure 1.1. For this reason, NCDs drive much research on the medical and engineering sides to understand their causes and effects, and provide technological solutions to prevent, detect, and monitor such diseases.

The early prevention of health conditions would also positively impact the cost of our health care system by preventing costly and heavy treatments that often occur at later stages of diseases. Figure 1.2 shows the total health expenditure as a percentage of a country's gross domestic product (GDP) from 1970 to 2021 as provided by the Organisation

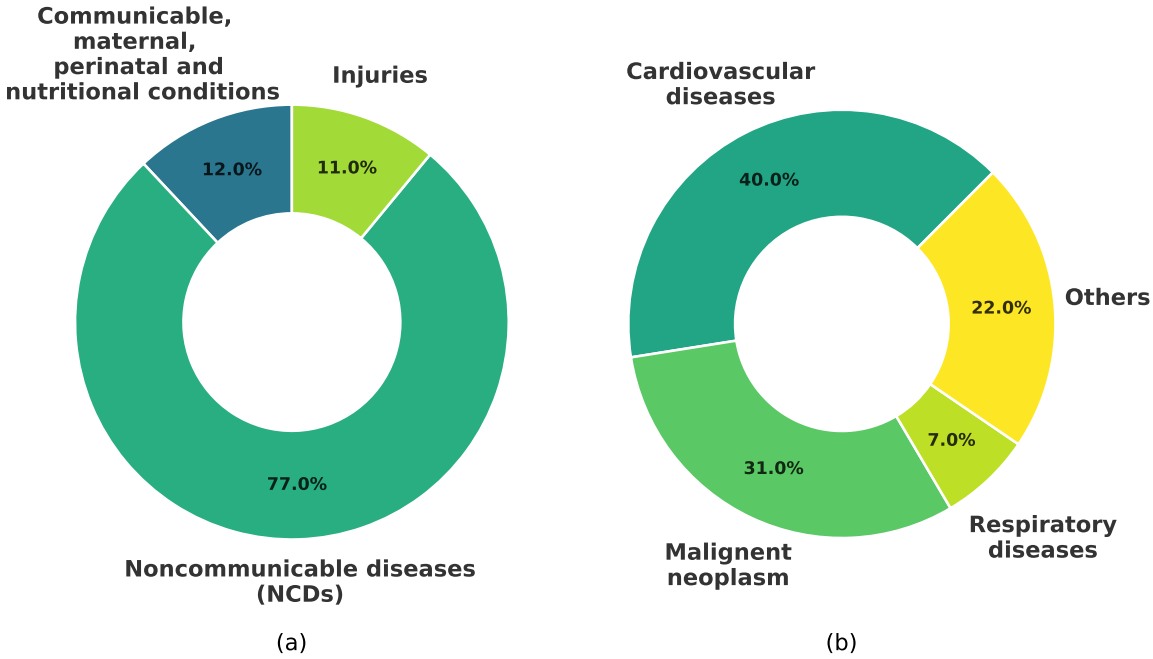


Figure 1.1: 2019 world death causes as reported by the WHO [8] for persons aged between 30 and 69 years old. (a) Three principal groups of death causes. (b) NCD death causes breakdown.

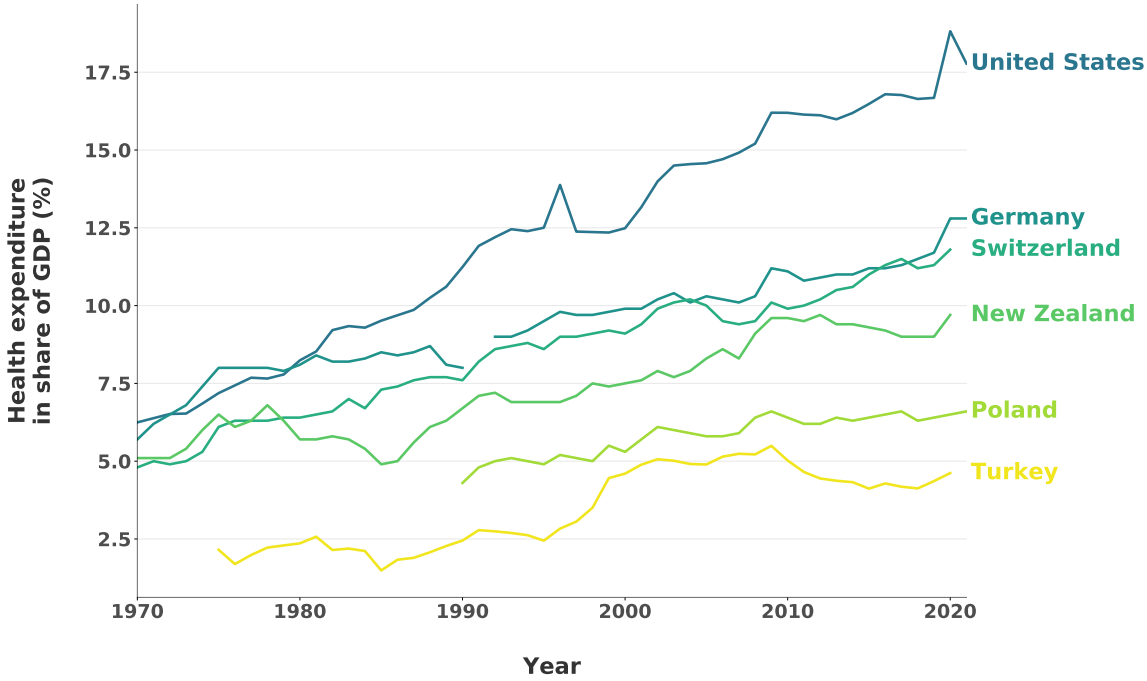


Figure 1.2: Health expenditure of countries in percentage of their GDP [9].

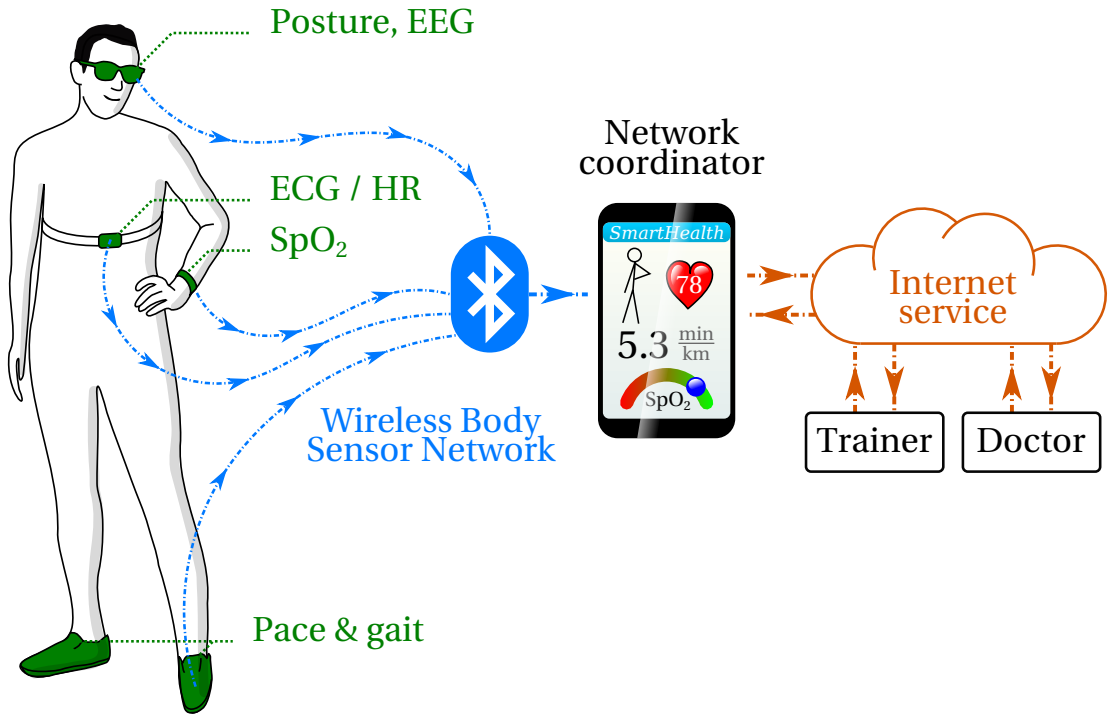


Figure 1.3: Wireless Body Area Network (WBAN) concept with multiple sensor nodes communicating through a wireless channel and to the internet through a network coordinator unit. Copied from [11], with the courtesy of Grégoire Surrel.

for Economic Co-operation and Development (OECD) [9]. This topic is critical, especially in countries with an aging population and a low birth rate. Additionally, the use of wearable devices could also lower the environmental impact of the healthcare system. For example, according to an analysis of the United States Energy Information Agency (EIA) [10], on the commercial energy consumption of buildings, hospitals are the third largest energy consumer (per square meter) after food service and food sales facilities in the United States. Avoiding extended hospitalization periods (or re-hospitalization) through better prevention would reduce the hospital's energy consumption.

In this context, smart wearables and wireless body area networks (WBANs) are particularly promising by enabling continuous monitoring and personalized healthcare for everyone. Figure 1.3 shows the concept of WBAN, where multiple sensors record various biosignals, such as electrocardiogram (ECG), blood oxygen saturation, or body motion, and communicate through a wireless channel to create a WBAN. These devices should be worn in daily life situations to allow long monitoring periods and provide efficient healthcare services at *home* (i.e., not in a hospital). Therefore, these nodes should be as small as possible to reduce their discomfort. This limits their battery capacity, thus, their

processing power, as such devices should ideally be able to operate for multiple days or weeks without having to be recharged.

Ultimately, the collected data could create a digital twin: a virtual biological representation of a person. Combined with a human body's behavioral and prediction models, this could offer a virtual solution to evaluate various treatments and estimate their impact on a person to find the optimal solution [12, 13]. Without going this far, this thesis proposes solutions to the less disruptive home diagnosis situation where information is gathered and locally processed to provide continuous monitoring, preventive diagnosis, and alerts.

1.1.1 Embedded biosignal processing platforms

There are two main classes of devices with different requirements and goals: medically certified and recreational (available to the general public). The formers provide very accurate measurements that comply with medical standards and are often legally required to have a recognized diagnosis and access to the corresponding treatments. Recent devices, such as the Phillips ePatch [14] or the Smartcardia 7-Lead ECG patch [15], are examples of the potential change WBANs could produce in the biomedical field. These devices simplify the recording of ECG signals and extend the monitoring time compared to previous solutions, where such monitoring was only possible in medical centers (with bulky devices) and for a limited recording time. For example, the Smartcardia patch allows up to 14 days of monitoring and storage on the device, and seven days of real-time connectivity with the cloud [16].

However, medically certified systems have to respect many constraints that limit their optimization. For example, the ePatch and Smartcardia only record and transmit data with minimal processing on the platform, and the analysis is performed on the cloud. One reason is that full disclosure of data is still required for medical-grade devices. In this case, processing the data in the cloud is the best option, as the data must be transmitted. Nevertheless, such a scenario is reserved when a particular disease is suspected, and medical certification is required.

On the contrary, more preventive situations have different requirements. For example, recreational wearable devices that measure different biosignals exist. Smartwatches or fitness trackers are the most common example of such devices. They typically measure the heart rate (HR), the blood oxygen saturation (SpO2), and the body's motion. Coupled with a cardiac monitoring belt, they can also monitor a person's ECG. These devices offer continuous monitoring of various biosignals and enable the early detection or preventive monitoring of various health conditions. For example, many NCDs, including some types of diabetes, hypertension, and cardiovascular diseases, are linked to unhealthy lifestyles,

such as a lack of physical activities or an improperly balanced food regime. These devices provide daily monitoring that could help each individual be aware of unhealthy behaviors and try to minimize them to lower the risks of certain diseases. The research mainly focuses on ECG-based detection methods targeting cardiovascular diseases [17–19] as they represent the highest death risk, as shown in Figure 1.1. These devices have different constraints than the medically certified ones and can be further optimized by moving the computational load toward the edge and reducing the communication energy overhead. However, to provide helpful monitoring and accurate diagnosis, they must approach the high-quality information delivered by clinical devices and professionals. This scenario confronts embedded systems with two opposite goals: low-power operation and high performance.

1.1.2 Low-power architectures and limitations

Low-power architecture exploration is an active area of research, as we still need to obtain higher energy efficiencies to enable long-lasting operation between battery recharges. An avenue to reach these goals is toward heterogeneous platforms, including multi-core architectures with heterogeneous cores and hardware accelerators. The latter can be divided into custom accelerators, or application-specific integrated circuits (ASICs), and flexible (programmable) cores, or domain-specific instruction-set processors (DSIPs).

In the literature, ASICs are often presented as the most efficient architecture and as the solution to overcome existing embedded systems' computational power and energy efficiency bottleneck. The justification is shown in Figure 1.4, where the performance versus programmability tradeoff is illustrated for various architectures. This trend is quite intuitive as a circuit customized for a task and a given set of constraints is usually more efficient than a more flexible architecture. In this context, implementing ASICs seems to be the ideal solution. Unfortunately, this conclusion is based on simplified scenarios, and other constraints arise when complete applications and real-life scenarios are considered making ASICs suboptimal designs.

Three main factors limit the design and integration of ASICs in today's low-power commercial designs, the first being their lack of flexibility. While Figure 1.4 shows that ASICs are the optimal solution in terms of global performance (i.e., area, latency, and energy), this conclusion is mostly true at the kernel level. When complete applications are considered, many different kernels are executed, and an ASIC can accelerate only a few in the best case and none in the worst case. This limits the impact of ASICs at the application level, as described by Amdahl's Law [20] and formulated in Equation 1.1, to the weight of the accelerated kernels in the total execution time of the complete application. This gives an upper bound limit of $\frac{1}{(1-p)}$ for the maximum possible speed-up the acceleration of a

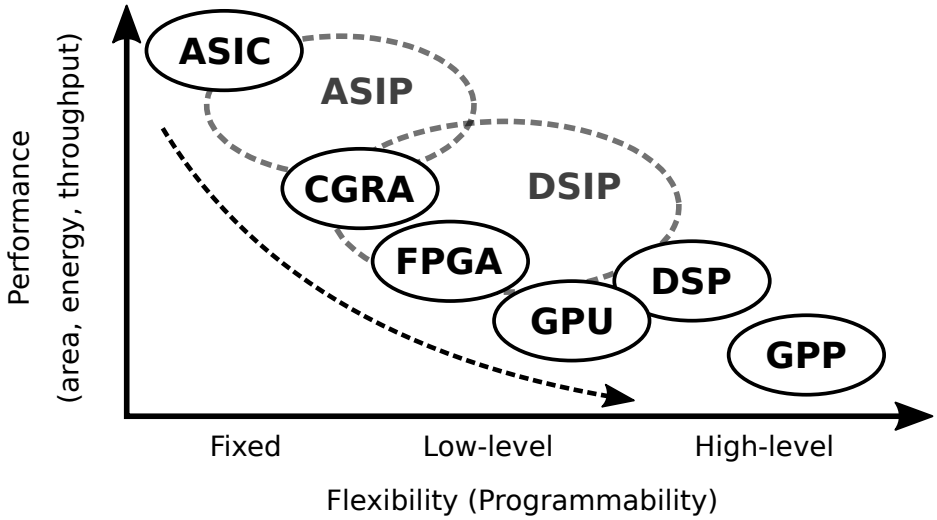


Figure 1.4: Common architectures' flexibility vs performance trade-off.

kernel can produce at the application level. For example, accelerating a kernel representing $10\,\%$ of the total execution time of an application cannot speed-up the application more than a factor of $1.1\,\times$.

$$\begin{cases} & app_{speedup}(s,p) = \frac{1}{(1-p)+\frac{p}{s}} \\ \lim_{s\to\infty} & app_{speedup}(s,p) = \frac{1}{(1-p)} \end{cases} \tag{1.1}$$

where:

- $app_{speedup}$ is the speed-up at the application level.
- s is the speed-up at the kernel level.
- p is the proportion of execution time the kernel represents originally at the application level.

In addition, applications are susceptible to functionality changes and the evolution of their algorithms, while ASIC designs are fixed. This could make them quickly obsolete. However, some commercial platforms integrate ASICs, particularly when low-power execution is required for computationally expensive kernels such as in deep neural network (DNN) applications. For example, the MAX78000 system-on-chip (SoC) [21] integrates a convolutional neural network (CNN) accelerator, the Greenwave's GAP8 chip a HardWare Convolution Engine (HWCE) [22, 23], and the Kendryte K210 a layer-based CNN accelerator [24]. While these accelerators increase the overall performance, they might become less efficient if the CNN generic model changes or even ineffective if the model becomes obsolete (e.g., replaced by a new and more accurate type of model).

Finally, the nonrecurring engineering (NRE) costs are growing with scaled technologies [25]. Therefore, massive production volume is necessary to reach profitability, while the inflexibility of ASICs restricts their range of applicability, hence their use cases. For all these reasons, ASICs are an economical show-stopper for their broad deployment inside integrated circuits (ICs) using scaled nanometer technologies.

On the other hand, programmable architectures offer higher flexibility but at the cost of much lower global performance. General-purpose processors (GPPs) are the most flexible architecture and cover any possible use case. A large variety of GPP architectures exist, ranging from ultra-low power to high-performance computing (HPC). Because of their limited-resource environment, embedded systems favor low-power designs at the cost of reduced performance. Two major trends have been proposed to increase their computational power and retain flexibility: tightly coupled co-processors (in-processor extensions) and multi-core systems. One example of the former is the DSP extension of the ARM Cortex-M4 processor [26]. It enables packed-single instruction multiple data (SIMD) execution and improves the performance of the GPP, especially for kernels using lower-bit representation (e.g., 16- or 8-bit). For a workload that can be parallelized, multi-core architectures offer a simple solution to further increase the computational power. Combining different types of processors and implementing a power-gating strategy offers a significant performance improvement while limiting the power and energy overhead by switching off part of the system when it is not needed [27]. However, the performance and energy efficiency of GPP-based designs are still orders of magnitude lower than ASICs [28–31].

In this context, it seems the performance gap between commercial platforms that favor programmable architectures and custom circuits that offer optimal performance (at the kernel level) is to remain. This work proposes and supports a way out of this fundamental dilemma.

1.2 Thesis contribution

In this thesis, I advocate another direction by designing DSIPs: the development of heterogeneous processors that do not contain any customized components and remain fully programmable for any source code that could be targeted. The difference with a GPP is that the compiled assembly code runs more efficiently for the target domain than for code outside this domain.

A DSIP enables optimizing the architecture for a target domain (i.e., a large set of possible use cases), which translates to high performance and energy efficiency while remaining programmable. This flexibility allows most of the kernels of the application

domain to be executed more efficiently than with GPP-based designs, which translates to higher application-level performance than ASIC-based designs.

To support and prove this idea, I conducted a DSIP architectural exploration for the biomedical domain. Then, I performed comparisons with other types of architectures (e.g., ASICs, GPPs) both at the kernel and application levels to have a fair and complete overview of the design space. The results of my experiments back up the main contributions of this thesis that are:

- A **DSIP template** called VWR2A: a very-wide-register reconfigurable-array that combines high computational density and a low-energy memory hierarchy. This template is meant to be optimized for a given set of requirements in area, power, energy, and throughput. It results from a design space exploration focused on innovative architectural features to advance the Pareto front of existing programmable architectures. VWR2A targets the biomedical domain's ultra-low power and energy applications while maintaining the required high throughput/latency performance.

  The main architectural features of the template are based on the state-of-the-art in different domains: the memory hierarchy (from many studies in embedded systems), computational density (from coarse-grained reconfigurable arrays (CGRAs)), instruction decoding efficiency (from CGRAs), and increased instruction parallelism (from very-long instruction word (VLIW) processors). In particular, these features are:

  1. A reconfigurable array based on the CGRA template that uses reconfigurable cells (RCs) with a high computational density.
  2. Specialized slots, which are loosely inspired by the brain regions and their focus on specific tasks (e.g., motor control, vision) and similar to the functional units (FUs) of a VLIW processor to increase the instruction level parallelism (ILP) by freeing the RCs from executing simple instructions unrelated to the actual data processing (e.g., loop control, branches). These specialized slots can perform the simple operations for which they are designed using less energy than the RCs.
  3. An energy-efficient instruction decoding scheme based on the CGRA template with a context memory containing pre-decoded instructions loaded to local program memories close to the processing elements (i.e., the RCs and the specialized slots).
  4. A low-energy wide memory hierarchy including:
     (a) Wide memories with reduced decoding energy, high bandwidth, and low switching in large output transistors.

(b) Single-ported memories to reduce their energy consumption to a minimum.
(c) A scratchpad memory (SPM) to reduce energy consumption by targeting most data accesses to a small memory closer to the processing elements (i.e., the RCs).

The complete template is presented and discussed in detail with examples based on two instances of VWR2A. The important optimizable parameters are examined, and typical values based on experiments are provided to guide the design of an instance based on the template. The mapping of kernels is illustrated with an example to better explain the specificities of the template.

I evaluate quantitatively the innovative architectural features introduced in VWR2A to assess their role in the template's high performance and energy efficiency. In particular, I compare the proposed architecture with two state-of-the-art programmable architectures targeting the biomedical domain: a low-power SoC [32] featuring a single-core general-purpose ARM Cortex-M4 processor and a CGRA based on two existing implementations [33, 34]. First, I analyze the assembly code of a typical kernel used in the biomedical domain: a fast Fourier transform (FFT). Three architectures are evaluated to provide a qualitative comparison of their efficiency in executing instructions, particularly their degree of ILP. Then, the three implementations were simulated in register transfer level (RTL) to get the cycle-accurate execution times of different FFT kernels. In addition, all the architectures have been synthesized, and their cell-switching activity (post-synthesis) was measured to estimate their respective energy consumption.

- A **DSIP versus ASICs** comparison. First, the design of a DSIP instance based on the VWR2A template is implemented, and design parameter choices are justified. This instance tries to minimize the performance and energy gap compared to ASICs and is optimized for data-intensive kernels.

  Two ASICs are used for comparison: an FFT accelerator and a matrix processor. They are two characteristic accelerators for the biomedical domain as they can execute typical biomedical kernels such as FFTs and finite impulse response (FIR) filters. The three implementations (the VWR2A instance and the two ASICs) have been simulated in RTL to get cycle-accurate execution times and synthesized to estimate their energy consumption from their post-synthesis netlist cell switching activity.

  The comparison is first conducted at the kernel level, the optimal scenario for ASICs. Then, a complete application is evaluated with an SoC integrating the ASICs and the VWR2A instance. The kernel and application level evaluations provide a complete and fair view of the design space and its tradeoffs. Based on these experiments, the higher energy efficiency of the specific features introduced into VWR2A is assessed and discussed in detail.

Finally, the template's parameters and impact on performance and energy consumption are evaluated and discussed to show the design tradeoffs when designing an instance based on VWR2A.

- The evaluation of a **DSIP instance for control-intensive code**. This type of code is usually not evaluated as research focuses on data-intensive kernels. However, such code is often present (and sometimes dominant) at the application level. This work proposes optimizing a DSIP instance based on VWR2A to improve the execution of control-intensive kernels and increase the coverage of efficiently executed code. First, the VWR2A instance is presented, and the specific features introduced to improve the execution of control-dominated kernels are presented and justified. Various control-intensive kernels and applications are evaluated on the VWR2A instance. In particular, the mapping of queues is evaluated. Such structures are an excellent example of control-intensive code used in many kernels and usually not executed by accelerators.

  A comparison with two state-of-the-art GPPs is made to assess the efficiency of these features. The first processor is an ARM Cortex-M4 optimized for executing data-intensive kernels at low power. The second processor is a RISC-V Ibex ultra-low power core explicitly optimized for control-dominated code. These two GPPs, optimized for different scenarios, are representative of typical architectures used for control-intensive code.

  The VWR2A instance and the two GPPs have been simulated in RTL to get cycle-accurate timing and synthesized to extract their cell switching activity post-synthesis and estimate their power consumption. The experiments have been conducted at the kernel and application levels. The results obtained demonstrate quantitatively the VWR2A instance's higher efficiency. In addition, the main parameters of the template that influence the execution of control-intensive code are examined, and their optimization is discussed qualitatively to guide the design of an instance focusing on such code.

1.3 Thesis outline

The rest of this thesis is organized as illustrated in Figure 1.5 and as follows. Each chapter of this thesis contains an introduction to provide the necessary background information and a review of the related works based on the context of the chapter.

Chapter 2 presents VWR2A: a very-wide-register and reconfigurable-array template for DSIPs targeting the biomedical domain. This template is proposed as a better alternative compared to existing programmable architectures, such as GPPs and CGRAs. VWR2A

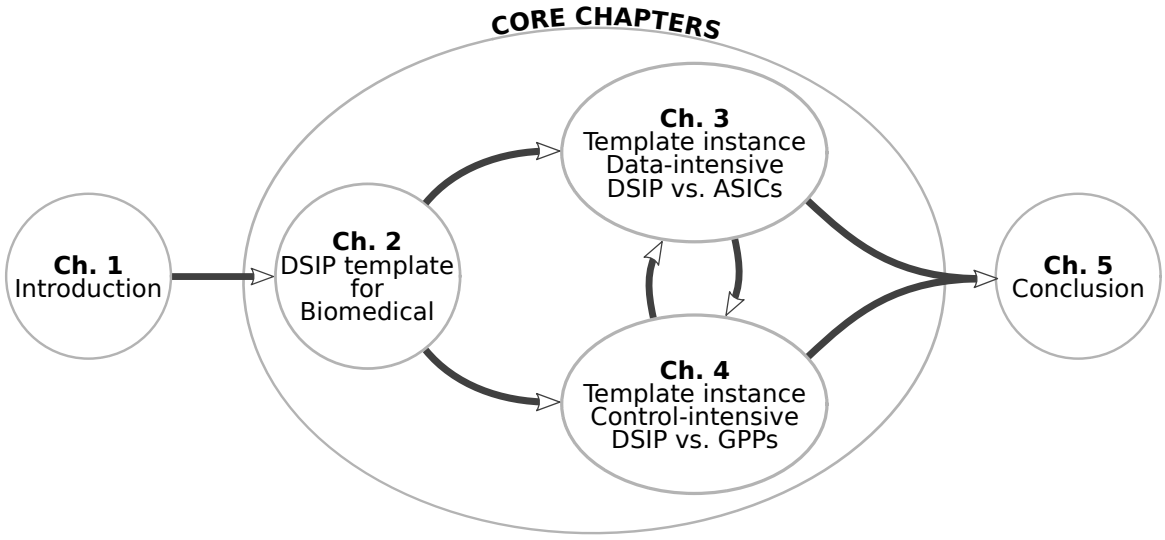


Figure 1.5: Chapters of the thesis and their summarized content to help readers navigate through this work.

combines a CGRA compute core fabric and a low-energy wide memory hierarchy made of an SPM and very-wide registers (VWRs). It is augmented with specialized slots, namely a load-and-store unit (LSU), a multiplexer-control unit (MXCU), and a loop-control unit (LCU). The human brain and its regions specialized for distinct tasks (e.g., motor control, vision, coordination) loosely inspire these specialized slots that are optimized for specific tasks (e.g., loop control) and increase the ILP of the architecture. The optimizable parameters of the VWR2A template are discussed and illustrated with examples from actual instances of the template. The specificities of VWR2A are shown through a simple kernel example mapped on one instance of the template. Finally, to showcase the innovative architecture features introduced in VWR2A (e.g., the specialized slots) and assess their impact on performance, one instance is compared to two state-of-the-art architectures: an ARM Cortex-M4 low-power GPP and a CGRA design, both optimized for the biomedical domain. The main differences and better performance of the VWR2A instance compared to these two architectures are discussed in detail to show the advantage of VWR2A's features from a high-level perspective. These features are at the heart of the architecture template, and their advantages will be visible in any instance based on it.

Chapter 3 presents one instance of VWR2A focusing on data-intensive code. The specific values of the template's parameters used in this instance are explained to provide insights into the design choices required when creating a new instance. To assess the performance of this VWR2A instance, an SoC targeting the biomedical domain [32] is considered. It features an ARM Cortex-M4 low-power processor and hardware accelerators or ASICs, such as an FFT accelerator and a matrix processor engine. The VWR2A instance is integrated inside the SoC and compared at the kernel level with the ASICs. Then, the

performance at the application level is evaluated considering the best possible mapping on the original SoC (i.e., using its hardware accelerators) and on the SoC augmented with the VWR2A instance (i.e., replacing the original SoC ASICs with the VWR2A instance).

Chapter 4 presents one instance of VWR2A focusing on control-intensive code. While ASICs focus on data-intensive kernels because they often represent most of the workload, other types of code are usually present at the application level. Compared to data-intensive kernels, control-dominated code has a significantly higher number of branch instructions and irregular memory accesses. Traditionally, this type of code is usually left to be executed by the processor (i.e., a GPP) as no accelerator is primarily designed for it, which translates to poor performance when this code is dominant. This can happen because an application is control-dominated by nature or because all the data-intensive kernels have been accelerated. This chapter shows the advantage of having a programmable architecture in this situation, allowing both data-intensive and control-intensive code to be accelerated. The VWR2A instance is compared to two state-of-the-art low-power processors: an ARM Cortex-M4 [26] and a RISC-V Ibex core [35], which is optimized explicitly for control-intensive code.

Chapter 5 concludes this thesis and resumes the main results of the previous chapters. Finally, the most promising research directions to continue and improve on this work are discussed.

2 Low-Power Domain-Specific Instruction-set Processor architecture

2.1 Introduction

In 2030, 24 billion Internet of things (IoT) devices are expected to be used worldwide [36]. Edge devices —being IoT, wearable, or embedded devices—, could benefit many domains with their promise of better efficiency and lower costs [7]. Smart autonomous vehicles (SAV) [3, 4], such as cars and drones, are revolutionizing the transport infrastructure [2]. Smart buildings and cities improve the energy efficiency of our habitats, working places, and manufacturing plants, paving the way to a sustainable society [1]. In the healthcare domain, wireless body area networks (WBANs) could help monitor and detect health problems sooner and lower the high economic and environmental costs of today's healthcare systems [5, 6]. However, existing platforms have not yet achieved long-lasting battery operation and often require a daily charge. In order to improve the overall energy efficiency of IoT systems —from the edge to the cloud— the computational load is pushed towards the edge. This reduces communication energy but tasks embedded systems with two opposite goals: low-power operation and high performance. Moreover, the current trend of growing machine learning application complexity is putting more pressure on embedded devices. Optimizations from the application to the circuit level are now mandatory to cope with this computation complexity and achieve higher energy efficiencies.

In particular, hardware architecture exploration for such devices is an active area of research. One avenue to reach these goals is toward heterogeneous platforms, including multi-core architectures with heterogeneous cores and co-processors (e.g., hardware accelerators). The inclusion of the latter has become a standard for computing platforms in recent years. These accelerators, or co-processors, can execute repetitive operations that account for a significant amount of the processing time (i.e., computational kernels) in a more efficient way than a general-purpose processor (GPP) or a generic graphics processing unit (GPU). The development of computing platforms and accelerators follows

two main trends: application-specific integrated circuits (ASICs)[1] or custom accelerators, and domain-specific instruction-set processors (DSIPs)[2] or flexible (programmable) cores.

Fixed-function accelerators (i.e., ASICs) are often the most efficient way of implementing a particular functionality for a given set of constraints. They are generally not programmable and are thus focused on a single task or a small family of related tasks. One example of a custom accelerator integrated inside a system-on-chip (SoC) targeting the biomedical domain is the fast Fourier transform (FFT) accelerator included in [32], which can execute FFTs of different sizes much more efficiently than the platform ARM Cortex-M4 core. On the other hand, it is possible to build fully programmable cores tailored specifically for algorithms from a given vast application domain (i.e., DSIP) without becoming general-purpose processors.

Figure 1.4 shows the programmability versus performance tradeoff of the main classes of architectures. Although ASICs offer the best performance at the kernel level, their main disadvantage is their very targeted specific context. This limits their impact on performance at the application level and, from an economic point-of-view, inevitably reduces their target market domain, reducing the yearly production volumes of the integrated circuits (ICs) that contain these accelerators. These platforms include the so-called application-specific instruction-set processors (ASIPs) [37, 38]. Given the huge nonrecurring engineering (NRE) costs of modern scaled technology nodes [25], this is an economical show-stopper for the broad deployment of such ICs. Hence, this chapter advocates another direction by designing DSIPs: the development of fully programmable heterogeneous processors that do not contain any customized components and remain fully programmable for any source code that could be targeted. The crucial difference with a GPP is that the compiled assembly code runs more efficiently for the target domain than for code outside this domain.

2.1.1 Low-power architectures and limitations

Traditional knowledge from the literature says that DSIPs and programmable cores are less efficient than fixed-function ones and that the latter should be preferred to improve the energy efficiency of the systems. Figure 1.4 shows the well-known and intuitive flexibility-performance tradeoff of different generic architectures at the kernel level [29, 31]. Based on this figure, designing ASICs or ASIPs as accelerator components seems to be the obvious solution to solve the aforementioned challenges of embedded systems. However, three main reasons limit the design and integration of ASICs in today's low-power real chips:

[1]The term ASICs refers here to any custom circuits and not necessarily to full chips.

[2]The ASIP term has been used in the past to refer to a similar concept, but its name implies some application-specific optimizations. Therefore, the term DSIP, which refers to optimizations for a complete domain, is more precise and appropriate to the ideas articulated by this thesis.

1. Their lack of flexibility:

   (a) Limits their range of applications, therefore, restricting their use cases.

   (b) Limits their source code coverage, thus, their impact at the application level, as described by Amdahl's Law [20] and formulated in Equation 1.1, to the weight of the accelerated kernels in the total execution time of the complete application.

   (c) Prevents potential application updates making them unadapted to rapidly evolving fields such as machine learning applications.

2. Although the literature has demonstrated the optimal performance of ASIC designs on very specialized computation kernels (e.g., FFTs, matrix multiplications, convolutional neural networks (CNNs)) [39, 40], their impact at the complete application level (i.e., in a real-life scenario encompassing the entire context) is often not evaluated.

3. Custom design development costs are growing with scaled technologies, and massive production volume is necessary to reach profitability, which might not represent an actual market demand. Moreover, custom designs might have limited reusability, further increasing their costs.

Considering these drawbacks, low-power commercial platforms usually prefer the flexibility of programmable general-purpose processors at the cost of lower overall performance, although most of the literature advocates for ASIC designs [39, 41, 42]. To reduce the performance gap between GPPs and ASICs, ARM, for example, proposes a range of low-power general-purpose processor architectures —Cortex-M0 to Cortex-M4— with various extensions (e.g., digital signal processing, floating-point unit (FPU)). However, compared to ASICs, general-purpose processors, even with their extensions, are still multiple orders of magnitude less efficient in all performance metrics. The literature proposed many different architectures —some of which are used in commercial chips— in the mid-range of the flexibility-performance tradeoff (see Figure 1.4): field-programmable gate arrays (FPGAs) [43, 44], coarse-grained reconfigurable arrays (CGRAs) [31, 45], or ASIPs [46, 47], for example. Although better in performance than GPPs and digital signal processors (DSPs), they are usually at least one order of magnitude less performant than ASICs [28–31]. The limitations of these architectures are of two natures: their lack of advanced architectural optimizations to maintain high flexibility and their high cost, either in energy or latency, for data accesses and movements. In this context, it would seem we are fated to choose between the limited applicability of high-performance and energy-efficient ASIC designs proposed by the literature and the existing commercial platforms that trade flexibility for lower performance and efficiency. This thesis proposes and supports a way out of this fundamental dilemma.

2.1.2 Contributions and outline of the chapter

In this chapter, I advocate for the development of DSIP architectures instead of ASICs in order to tackle the aforementioned challenges and overcome the current limitations of low-power and low-energy embedded systems. To show the benefit of such an approach, I propose the Very-Wide-Register and Reconfigurable Array (VWR2A) template, a combination of two concepts: a very-wide register (VWR) as foreground memory and a CGRA as processor core fabric. CGRAs have demonstrated good performance and energy efficiency, particularly for the biomedical domain [33, 34, 45]. Additionally, the CGRA template is a programmable architecture making it an excellent building block for creating a DSIP. A VWR is a low-energy memory designed for data with high spatial locality (e.g., arrays of data), which makes it a perfect candidate for biosignal processing.

Both these elements have been proposed in earlier literature from a conceptual point of view [33, 48], but they have not been combined and instantiated for the biomedical target domain. VWR2A enables designs with significantly better key metrics —in both latency/throughput and energy— compared to state-of-the-art low-power instruction-set processor architectures (e.g., GPPs) and programmable accelerators (e.g., CGRAs). Designing an instruction set processor that supports most of the source code of a complete application domain to be compilable forces it to be flexible by nature (i.e., amenable to a potential application or functionality upgrades) while allowing advanced architectural optimizations based on the specificities of this domain (i.e., workload complexity, type of code, timing constraints). When carefully designed, a DSIP can efficiently cover a large portion of possible source code, translating to high performance at the application level compared to an ASIC that only accelerates specific kernels, as demonstrated in Chapters 3 and 4. Moreover, such DSIP still covers all the other code sections with a reasonable, though not optimal, efficiency. In particular, the key contributions of this chapter are:

1. VWR2A: a DSIP architecture template targeting the biomedical domain's ultra-low power and energy applications while maintaining a required high throughput/latency performance.
2. An analysis of the VWR2A template through a kernel mapping and execution example.
3. A performance and energy comparison against two representative state-of-the-art architectures targeting the biomedical domain: a low-power SoC [32] featuring a single-core general-purpose ARM Cortex-M4 processor and a CGRA [33].

The rest of this chapter is organized as follows. First, I discuss related architectures for low-power computing in Section 2.2. Then, the proposed VWR2A template is presented

and illustrated with examples in Section 2.3. The mapping of kernels on the template is illustrated and discussed with one example in Section 2.4. The experimental setup used to evaluate an instance of the VWR2A template is presented in Section 2.5. The detailed evaluation and comparison against two state-of-the-art low-power architectures are completed in Section 2.7. Finally, Section 2.8 finishes the chapter by drawing the main conclusions.

2.2 Related work

Many low-power programmable designs have been proposed in the literature. From the architecture point of view, two central elements can be optimized: the computing and the memory parts. Existing solutions from the academic community are reviewed first: low-power computing architectures are discussed in Chapter 2.2.1, and low-power and low-energy memory organizations are reviewed in Chapter 2.2.2. Finally, commercial platforms are presented in Chapter 2.2.3.

2.2.1 Low-power and low-energy CPU architectures

General-Purpose Processors

The complex instruction set computer (CISC) vs. reduced instruction set computer (RISC) instruction set architectures (ISAs) debate has been going on for decades, and recent studies [49–51] tend to show they do not have much impact on performance (energy, latency, area). Nevertheless, RISC ISAs are usually preferred for low-power applications and CISC ISAs for high-performance ones. More importantly, architectural design choices, such as simpler architectures with in-order shallow pipelines, have been demonstrated to be a better compromise for low-power embedded devices [52]. The last decade has seen the rise of the open RISC-V ISA widely adopted in academia and many processors based on it. In the low-power domain, the PULP project significantly impacted the research community with the proposal of the lowRISC Ibex processor (formerly Zero-riscy [53]) and the OpenHW Group CORE-V CV32E40P (formerly RI5CY [54]), two state-of-the-art low-power processors (which are even used in some commercial platforms today). For data-intensive workload edge applications, ultra-low power cores, such as Ibex or Bellevue [55], are inappropriate as they lack fast computing resources. To tackle this issue, low-power and high-performance processors usually possess extensions. For example, the RI5CY core supports packet-single instruction multiple data (SIMD) and has a zero-overhead loop feature (hardware loop support) [56]. It also takes advantage of the extensible RISC-V ISA option and provides custom instructions with hardware support, such as fixed-point arith-

metic or bit manipulation operations. These extensions increase power consumption, but when they are correctly designed (and adapted to the workload), the gain in performance is even higher, and the total energy consumption is reduced [54]. These extensions can be integrated inside a GPP, as for the two examples presented above, or as a co-processor (e.g., a hardware accelerator) to improve their efficiency. The latter option is discussed later in this section.

However, GPPs, even with extensions, are still orders of magnitude less efficient than ASICs. The authors of [57] have evaluated the overhead of programmable architectures compared to ASICs and proposed solutions to reduce the energy gap, while maintaining programmability, within a factor $3\times$ compared to an ASIC. Starting from a GPP, they optimized it for one application (similar to an ASIP), but they acknowledged the need to add flexibility to the custom features they introduced in order to enable the coverage of other similar applications and increase the use case of the architecture. This idea is at the center of a DSIP design (discussed in the next section). Finally, their main conclusion is that a memory organization tailored to the application need is the key to reducing to a minimum the energy gap between programmable and ASIC designs. The dedicated wide memory hierarchy of the architecture template proposed in this chapter (see Section 2.3.2) has been proposed for the same reason, and the experimental results show, similar to [57], the large improvement such an optimized memory organization enables.

Domain-specific Instruction-set processors

The DSIP acronym is not widely spread in the literature, and very few architectures have been proposed. DSIPs originate from ASIPs. Although relatively close, the original idea of an ASIP was to build a platform —with a central processing unit (CPU), memory, hardware modules, and peripherals— for an application (or a small set of applications) using high-level synthesis and a compiler supporting this platform [37]. The hardware modules comprise CPU extensions and hardware accelerators or co-processors, for example. The work that followed usually focused on a few aspects of the original concept. Similar to an ASIC, an ASIP implies application-specific optimizations, while this chapter is interested in architectures with a broader range of applications, hence the use of the DSIP acronym, which is more precise. To my knowledge, the term was first proposed in [58], where the authors presented the Flexible Extremely ENergy Efficient Configurable System (FEENECS): an architectural template for DSIP. The VWR2A template is partially based on FEENECS, particularly regarding the wide memory hierarchy (see Chapter 2.2.2). The other parts of the template are similar from a high-level perspective. However, our proposal is more specific regarding the compute core fabric block of the template, while FEENECS is very generic. Moreover, no actual instantiation of the complete FEENECS template has ever been evaluated, only sub-blocks. Another DSIP (although not explicitly presented in such a

term) is the TamaRISC low-power core [59]. It is a custom design ISA optimized for the bio-signal processing application domain that uses a minimal subset of the microchip PIC24 ISA [60]. TamaRISC features a 3-stage pipeline, a 16-bit datapath, and a 24-bit instruction encoding width. A multi-core version based on this core has also been proposed to accommodate for higher workloads. As the DSIP template proposed in this chapter targets the same domain, its base ISA presented in Section 2.3 (and the ISAs of the instances in Chapters 3 and 4) is similar to that of TamaRISC regarding the supported instructions.

Homogeneous multi-core architectures

Low-power homogeneous multi-core architectures have been proposed in the last decade to improve the performance and efficiency of any processor (e.g., GPPs, DSIPs). Similarly to the concept of extension, it increases computing power by augmenting the number of processors. However, compared to GPP extensions (e.g., digital signal processing, packet-SIMD), adding one or more processors represents a higher power overhead. All recently proposed architectures use power domains to limit the power consumption overhead, in particular, the leakage power during idle periods, as in [61], for example. Such architectures have shown excellent performance and energy efficiency compared to single-core designs [59, 61]. Nonetheless, their efficiency is still limited due to their memory hierarchy (see Chapter 2.2.3) and the commonly unoptimized GPP template used to design their multi-core system. In VWR2A, the multi-processing paradigm is leveraged in the form of multiple processing elements (see Chapter 2.3).

Very Long Instruction Word Processors

Pipelining [62] is used inside a CPU to enable the concurrent *execution* of multiple instructions, increasing its throughput and maximum frequency due to the shortened logic paths in each of the stages of the pipeline. However, the pipelining concept does not define how many instructions can *start* at a time and their order of execution. The class of processors mentioned above (i.e., the Cortex-M0/M4, Ibex, and RI5CY) are known as in-order scalar or single-instruction, single-data (SISD) processors where the instructions are executed in the compiler-generated order and a single instruction starts every cycle, working on a single data.

Two concepts have been proposed to improve performance: out-of-order execution [62, 63] and superscalar or multiple issue processors [62, 64, 65]. Although both concepts are closely related and developed simultaneously, their nature is different. A superscalar processor enables multiple instructions to be issued simultaneously to take advantage of the instruction level parallelism (ILP) present in many algorithms. It has multiple functional

units (FUs) that can execute in parallel multiple instructions within the compiler-generated order (in-order execution). This allows a clock per instruction (CPI) less than one, but data-dependencies limit instructions parallelization on the FUs. An out-of-order processor verifies the dependencies between instructions at runtime and dynamically schedules them on the available FUs regardless of their compiler-generated order, allowing multiple out-of-order instructions to *execute* in parallel. However, the runtime dependency check and scheduling represent significant hardware, hence, power overhead. Both concepts are orthogonal and can be applied to build out-of-order superscalar processors, usually used for high-performance applications.

Very-long instruction word (VLIW) processors propose an alternative in which the dependencies are identified at compile time (instead of runtime), and the parallel execution of the instructions is scheduled explicitly by the compiler. It removes the hardware and power overheads of out-of-order superscalar processors while exploiting ILP. VLIW processors are often used for digital signal processing applications as most of these algorithms exhibit high ILP. In a VLIW architecture, or in a superscalar processor, all the functional units (FUs) share a common register file requiring a multi-port design which consumes much energy [48]. This limitation is further discussed in Chapter 2.2.2, where memory hierarchies are reviewed.

Low-power co-processor architectures

Incorporating co-processors and hardware accelerators in the computing platform has become standard practice in recent years. They can be integrated into a platform as tightly-coupled or loosely-coupled accelerators. GPP extensions, such as the digital signal processing unit in the ARM Cortex-M4 [26], are an example of tightly-coupled accelerators. They improve performance, but their integration within the processor limits their maximum gains, as shown in [66]. Loosely-coupled co-processors, or hardware accelerators, are circuits external to the CPU, enabling further performance improvement (in latency and energy) by taking advantage of private memory blocks tailored to their needs [66]. These accelerators rely on a direct memory access (DMA) to transfer data from the system's main memory to the accelerators' private memory.

The development of accelerators follows two main trends: custom accelerators, or ASICs, and more flexible (partly programmable) cores. Although providing the best performance at the kernel level, ASICs are less desirable in real-life applications (see Chapter 2.1.1), and therefore not considered here. Conversely, existing partly programmable accelerators are more interesting in the context of this chapter. Although flexible, these accelerators still have a limited code coverage compared to the DSIP solution proposed in this thesis.

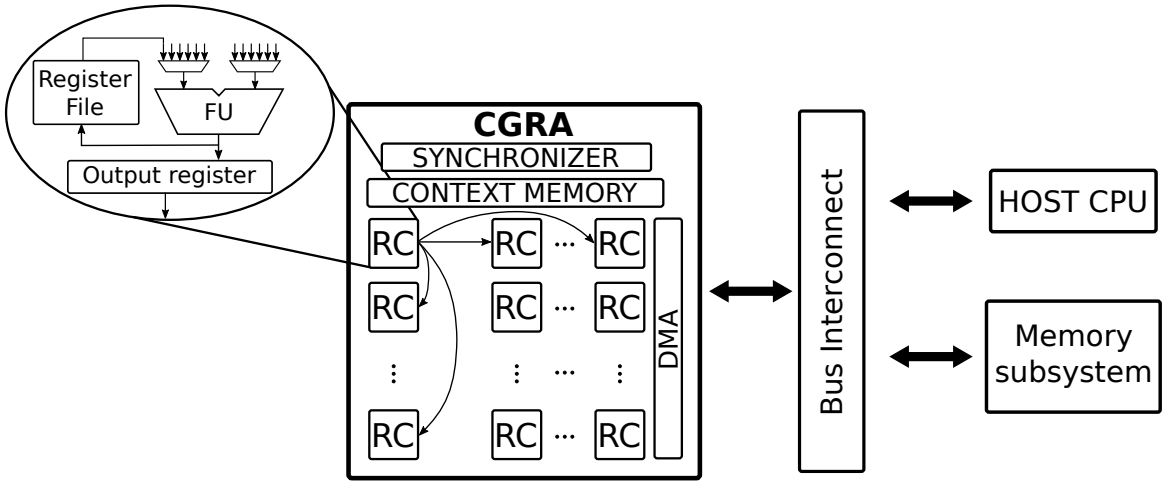


Figure 2.1: CGRA architecture template.

CGRAs have often been used as they provide good flexibility with reasonable energy overhead compared to ASICs [31, 67]. Additionally, their efficiency for biomedical applications has been demonstrated several times [33, 34, 45]. CGRAs offer flexibility thanks to their datapath that is reconfigurable at runtime. The concept is somewhat similar to an FPGA, but the configuration is done at a coarser granularity, namely the datapath level, limiting the required hardware overhead to control the reconfiguration, hence improving the overall energy efficiency compared to an FPGA. Figure 2.1 shows the CGRA template and its typical integration inside a platform. Its main components are:

- A context memory
- A reconfigurable array made of multiple reconfigurable cells (RCs) (or processing elements (PEs))
- An interconnection between the RCs
- An interconnection with the system data bus (e.g., DMA)
- A synchronization mechanism with the host processor

A context switch can reconfigure the array of RCs at runtime. This is done by loading new configuration words from the context memory to the local program memory of the RCs. One major advantage of the CGRA architecture template, which is kept in the proposed DSIP template, is the high computation density because the bits of the configuration words correspond directly to the control signals in the cell datapaths, without an actual decoding process. Figure 2.1 shows one possible interconnection called a torus closest neighbors interconnect, but other schemes are possible.

Many platforms have integrated a CGRA alongside the CPU to improve performance and energy efficiency. Morphosys [68] is made of a single-core RISC GPP, augmented with a CGRA that executes computation-intensive kernels. Although this architecture has shown better performance than ASICs, in some cases, it focuses on high performance and not low-power execution. Its high number of RCs (up to 256) and memory hierarchy using caches is not optimized for embedded systems. Healwear [45] is a multi-core DSIP platform implementing eight TamaRISC cores [59] and a CGRA optimized for biosignal processing on embedded devices. However, because of the limited code coverage of the CGRA, the authors' experimentation shows minimal energy savings at the application level. In [34], the authors proposed a similar platform using the PULP platform [27] (a single-core with a multi-core cluster, both using RISC-V cores) and a CGRA. Compared to Healwear, the savings from the CGRA are even lower due to the better performance of the multi-core system. The compute fabric core of the DSIP template proposed in this chapter uses the CGRA architecture template, but it is augmented with specialized slots (see Chapter 2.3) in order to improve its performance and energy efficiency, and increase its code coverage.

2.2.2 Low-power and low-energy memory organization

The memory organization can be divided into two parts: the background or L2 memory and the foreground or L0 memory. The L2 background memory is usually the biggest and slowest on-chip memory block which contains the entire set of data (e.g., variables, input signals, constants, instructions), and all the processing elements share it. The background memory can also contain an L1 memory that acts as a buffer between the L2 and the foreground memory. This optional L1 background memory is smaller and can be shared by all the processing elements or be private. Finally, the foreground memory is the smallest and closest memory to the processing element(s) and is private. Ultra-low power and low-performance systems usually have a two-level memory hierarchy (i.e., L2+L0) as each level is relatively small (i.e., a few hundred kilobytes maximum) and the processing system is made of a single-core (i.e., no data bus access contentions due to many masters). The power and energy consumption of the background and foreground memories account for a large amount of the total consumption of any embedded device. The reason is that reading data from memory, particularly background memory, requires significantly more energy than the energy used to process these data (e.g., addition or multiplication). Therefore, optimizing the memory organization and hierarchy is essential to design low-power and energy systems. To this end, in my work, I proposed using a dedicated low-energy memory hierarchy in the DSIP template.

Background memory: L1 and L2

A system's main memory (i.e., L2) is the lowest on-chip memory block in the hierarchy. This memory block is usually implemented with static random access memory (SRAM) for low-power devices and goes from a few kilobytes to a few megabytes. It is usually made of smaller, concatenated single-port memory banks (for a memory larger than 32 KiB to 64 KiB) that can be clock gated, put in retention mode, or even power-gated to reduce power consumption.

The VWR2A template is meant to be integrated inside a platform as a loosely-coupled co-processor. Therefore, the definition of the main memory is done at the platform level and not embedded inside the template (i.e., not elaborated in this thesis). Nevertheless, the template requires access to this memory to store and share data with the other components of the platform.

Low-power architectures, especially single-core GPPs, do not consistently implement an L1 memory hierarchy level, as there is usually a single master accessing a relatively small main memory (<512 KiB). Multi-core systems (of any nature) could also access the main memory directly through the system bus. However, in this case, the performance of algorithms with many data accesses depends on the system bus latency and bandwidth, which can negatively impact the overall performance, as shown in [66] for the specific case of loosely-coupled accelerators. Moreover, data access through the system bus is costly in energy and can be reduced by including a private background (i.e., L1) memory, particularly for data.

This L1 memory can be private or shared depending on the requirements. A typical approach is to use caches, but they incur a significant energy penalty due to their inherent control overhead, therefore, privileged for high-end multi-core edge devices. Scratchpad memories (SPMs) are an alternative that offers similar performance at lower energy as the control is moved to the software side [69], at the cost of making the software responsible for the explicit placement of data. For these reasons, an L1 shared SPM is proposed in VWR2A to achieve high performance while limiting the power overhead (compared to a cache).

Foreground memory: L0

The foreground memory is the smallest and closest memory to the processing elements, hence the fastest and most energy-efficient memory regarding data access. Register files (RFs) are a common solution to implement this memory. Small RFs of a few tens to hundreds of bytes are usually built with standard cells (i.e., flip-flops). Depending on the

computing architecture, the RFs can be single- or multi-ported to provide parallel access to the same memory block and enable an easy way to share data between the multiple processing elements. RFs, in particular multi-ported ones, represent a significant part of the total power consumption according to [48, 70, 71].

An asymmetrical RFs, based on very wide registers (VWRs), was introduced in the FEENECS template [58] as a better alternative, in terms of energy, to standard multi-ported register files. The first reason is that the cells of the VWRs are single-ported, while those of the register files are often multi-ported. Second, their wide interface still allows multiple words to be loaded at once, leading to a lower overall energy per word access than traditional register files. To fully benefit from the VWRs, the background memory needs to match the VWR width, allowing it to fill a VWR in one cycle. It has been proposed to use a SPM as the background memory [48]. This effectively makes the VWR interface asymmetric with respect to the SPM side and the datapath side. The authors of [48, 72] also show that such a design is better for place and route, as both memories (i.e., the SPM and the VWR) can be aligned, and the wire length of the most active connection is reduced to a minimum.[3] Indeed, only the outputs of the multiplexers switch in every cycle, not the outputs of the VWRs themselves, reducing energy consumption.

2.2.3 Commercial low-power platforms

Many commercial low-power and high-performance platforms integrating numerous of the concepts presented above exist. For single-core systems, the ARM Cortex-M family of processors [26] dominates the market. Many platform providers, such as Texas Instruments [73], ST [74], or Silicon Labs [75], use this family of cores in their designs to provide low-power, high-performance solutions. The ARM Cortex-M processors range from ultra-low-power architecture, with the Cortex-M0 (ARMv7E-M) or Cortex-M23 (ARMv8-M), to low-power, high-performance architecture with the Cortex-M4 (ARMv7E-M) or Cortex-M33 (ARMv8-M). The Cortex-M0 is similar to the Ibex core [53] presented before and is not adapted to computationally intensive applications for the same reasons (low computation resources and no extensions such as SIMD or DSP). The Cortex-M4 core has a digital signal processing module enabling 32-bit wide packet-SIMD operation on four 8-bit values or two 16-bit values and a single-cycle multiplier and MAC unit. While better in performance and energy, it is still far from competing with ASICs. The next processor in the Cortex M family is the M7, a 6-stage pipeline superscalar processor with branch prediction. While providing higher performance, the energy efficiency depends on multiple parameters such as the targeted workload. Finally, none of these platforms explicitly target the biomedical domain, but the Cortex-M4 processor is the most common GPP used when low-power and high performance are required.

[3] This point is discussed in 5.2.1 as a future work direction.

To improve performance (and possibly energy efficiency), multi-core platforms, such as the Greenwave GAP-8 board [22] (originating from the PULP project [27]) or the TI AM57x SoCs exist. Although targeting high energy efficiency execution, they use the GPP template for their cores and traditional memory hierarchies based on caches or multi-ported memories, limiting their overall efficiency. Moreover, they target domains with higher performance requirements (e.g., edge AI applications), making them unadapted to ultra-low power biomedical embedded systems.

2.3 VWR2A: a template for low power and energy Domain Specific Instruction-set Processors

Very-Wide-Register Reconfigurable-Array (VWR2A) is the low power and energy DSIP architecture template proposed to advance the current state-of-the-art embedded devices targeting the biomedical domain and fill the current gap in existing commercial platforms. It combines various previously presented concepts to further improve the current state-of-the-art low-power designs' performance and energy efficiency. Figure 2.2 shows the main features of the VWR2A template, which integrates high computational density (i.e., a CGRA-like architecture) and low power memory structures (i.e., VWRs and SPMs). The reconfigurable array is made of multiple independent columns that can be reconfigured at runtime by a context switch from the context memory. Each column has multiple RCs to compute on data stored in VWRs and three specialized slots: a load-and-store unit (LSU), a loop-control unit (LCU), and a multiplexer-control unit (MXCU). The LSU is responsible for the data movement within a VWR (i.e., shuffling unit) and between the VWRs or the scalar register file (SRF) (of one column) and the shared SPM. The LCU manages the kernels' execution by issuing, for example, branch or jump instructions and controls the program counter (PC) shared by all the units in a column (i.e., RCs and specialized slots). Finally, the MXCU generates the addresses to access the VWRs and the SRF from the RCs' side. The kernels are executed upon request from the host platform in which an instance of VWR2A is integrated. The synchronizer manages these requests and schedules their execution on the available columns. The DMA transfers data between the shared SPM of VWR2A and the platform's main memory.

The high performance and energy efficiency of VWR2A architectural features are inspired by our brain structure and its different regions responsible for specific functions (e.g., vision and motor control). All these regions are optimized (e.g., with different neurons) and work in parallel, enabling us to achieve complex tasks very efficiently. The VWR2A units (i.e., the specialized slots and the RCs) are very similar, from a high-level perspective, by enabling the parallel execution of multiple instructions. Additionally, they are heterogeneous units that are each optimized (e.g., datapath width, supported

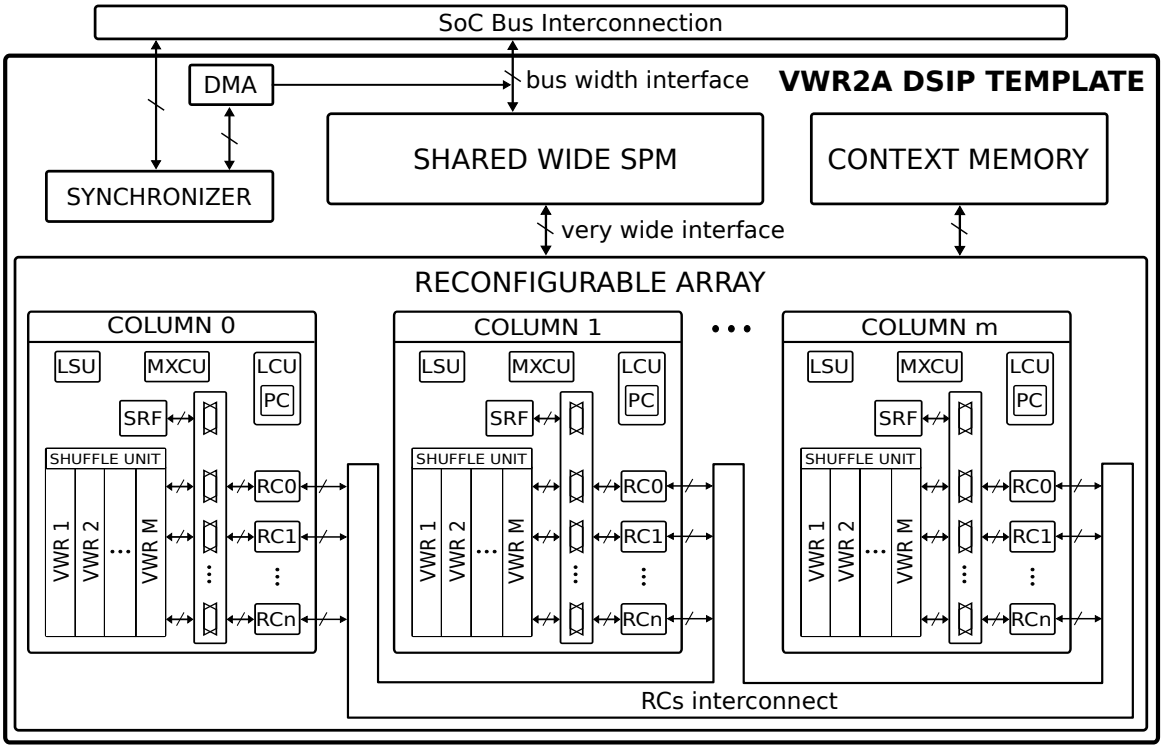


Figure 2.2: VWR2A architecture template block diagram for low power and energy DSIP design. The RCs and the context memory are detailed in Section 2.3.1. The wide memory hierarchy (SPM and VWRs) is discussed in Section 2.3.2, and the shuffle unit in Section 2.3.3. The specialized slots (LCU, MXCU, and LSU) are presented in Section 2.3.4.

instruction set) for the specific set of tasks they are designed for. This makes VWR2A a high-performance and energy-efficient architecture.

VWR2A provides high-level features (e.g., specialized slots, a specific memory hierarchy) meant to be optimized in an actual instance targeting a particular set of applications. The DSIP concept proposes to design architectures (e.g., ISAs, memory hierarchies) tailored to a specific domain to improve energy efficiency, but it does not define what makes a domain of application. The definition of a domain relies chiefly on the area or cost, latency, and energy performance a designer wants to achieve. For example, from an economic perspective, a domain is defined by the minimal production volume required to make a product viable. Based on this information, the design of the architecture can be explored to reach the optimal power, performance, and area (PPA) tradeoff for a given set of requirements.

In particular, the VWR2A template is proposed as a starting point for the architecture exploration and final implementation of DSIP designs targeting biomedical applications. In this chapter, the VWR2A template is presented in detail and illustrated with some

design choice examples to better demonstrate the template's use. For a complete and detailed review and analysis of two instances of the VWR2A template —one focused on data-intensive code and the other on control-intensive code— refer to Chapters 3 and 4, respectively.

2.3.1 Reconfigurable Array

The compute core fabric of VWR2A is based on the CGRA architecture template. It is made of RCs that are specialized for data processing. Each RC has an arithmetic logic unit (ALU), a local program memory, a register file, and an output register used for the RCs' interconnection, as shown in Figure 2.3. To reduce control overhead, the RCs are grouped in columns (see Figure 2.2), where all the RCs of a column are synchronized through a shared PC. This defines the minimum number of RCs that work in parallel (i.e., the number of RCs in a column). The columns are independent, allowing as many kernels as the number of columns to run in parallel. The columns can also execute kernels in synchronization (e.g., one kernel on two columns). In this case, the program counters of the columns are synchronized.

As in VWR2A the specialized slots execute all the extra computations not directly related to data (see Section 2.3.4), the RCs' design can be optimized compared to that of a standard CGRA. First, the operations supported by the ALU will significantly impact the performance of a specific instantiation. For example, the VWR2A instance proposed in Chapter 3, which is optimized for biomedical data-intensive applications, implements a single-cycle multiplier with two working modes: a standard mode, where the lowest 32 bits are kept, and a fixed-point mode, where the lower 16 bits are discarded and the next 32 bits are kept. This enables single-cycle fixed-point multiplication in q17.15 format and good performance for algorithms that require decimal representation, such as the FFT.

The datapath width is another optimization parameter of the template. In this thesis, all the template instances have a 32-bit datapath width for the RCs, which simplifies the data exchange with the system CPU (which is most of the time a 32-bit processor). The input of the ALU operand multiplexers can also be customized; however, they usually depend on other parameters of an instantiation (e.g., RCs' interconnection, number of VWRs). The last important parameter is the depth of the local RFs of the RCs. Although the architecture has VWRs (see next section), having a small local RFs is more efficient for storing locally partial results, scalar variables, or a variable accessed many times, for example.

The number of RCs in a column and the number of columns are additional parameters of the template and a tradeoff between performance and power consumption. Increasing

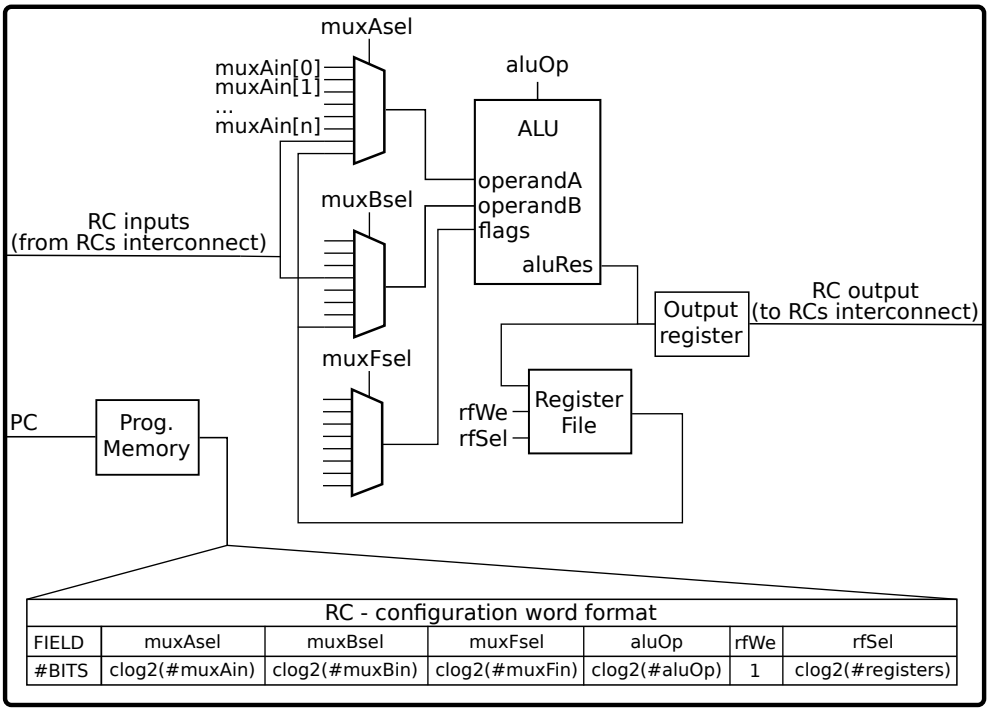


Figure 2.3: Block diagram of the RC in the architectural template.

their number increases performance but also power consumption. Regarding the number of RCs in a column, as long as the workload can be parallelized on the additional RCs, energy consumption will be lower, especially if the reconfigurable array can be power gated after its work is finished. The minimal throughput of the targeted domain is also important and might force the number of required processing elements. The number of columns defines how many different kernels can execute concurrently and depends on the potential parallelization of the workload at the application level. For the biomedical domain, many parallelization opportunities exist [34], especially when multiple biosignals or/and multiple leads are monitored.

The RCs have direct connections with one another through their interconnect network. Although the template does not enforce any specific scheme, the closest neighbors' interconnection is a good tradeoff between low-power, high-performance, and flexibility [33]. This network is usually synchronous (i.e., passing through a register) and enables fast and energy-efficient data exchange between the RCs (without going through the memory subsystem) and allows the creation of a data computation flow between the RCs. Similar to the RCs' interconnection for data, the RCs can select their internal flags (i.e., zero and sign flags) or those from their neighboring RCs. In both cases, the flag refers to the result of the operation executed in the previous cycle. These flags can be used, for example, to

Table 2.1: Main optimizable parameters of the VWR2A template related to the RCs and a non-exhaustive range of plausible values.

Parameter	**Typical values**
Number of RCs in a column	2, 4, 8
Number of columns	2, 4
Program memory depth (words)	16, 32, 64, 128
Scalar register file depth (words)	2, 4, 8
Datapath width (bits)	16, 32
RCs' interconnection	closest neighbors (with or w/o diagonal)
ISA	example: single-cycle multiplier

select one of the two ALU input operands as output in order to implement locally (i.e., in the RCs) simple *if* conditions (see Chapters 3 and 4).

CGRAs are reconfigurable thanks to their context switching mechanism. The configuration words (i.e., instructions to configure the RCs) are stored in a context memory. The CGRA architecture template offers a high computation density because the bits of the configuration words correspond directly to the control signals in the cell datapaths, saving the cost of the decoding logic of GPPs. When a kernel execution starts, its configuration words are loaded to the RCs' local program memory. The size of the context memory and local program memory are domain-specific parameters that can be optimized. Because of the shared program counter of a column, this architecture has evident parallelism with a VLIW processor template in which all the execution slots are equivalent. Indeed, the instructions of the different RCs can be seen as a wide (predecoded) instruction word.

The configuration word length is different depending on the specific value of the parameters of the template. The bottom of Figure 2.3 shows a generic configuration word for the RCs and the most common fields required by any application domain. Table 2.1 summarizes the main optimizable parameters of the template related to the RCs and provides a non-exhaustive range of plausible values. The scalability of the VWR2A template, especially regarding the number of columns, depends on the limitations of the physical layout implementation (see Chapter 5.2.1). Although not explored in this thesis, multiple instances of the VWR2A template could exist within a single chip to scale further than the value proposed in Table 2.1 if a higher throughput is required, for example. These instances could be identical or optimized for a subset of the algorithms used in the application domain.

2.3.2 Ultra-low energy memory organization

VWR2A is currently meant to be integrated as a co-processor or a programmable hardware accelerator. As discussed in Section 2.2.2, a proper memory hierarchy design is needed to avoid the limitations of the platform bus in which a VWR2A instance is integrated. Therefore, to accommodate for workloads with high data access needs, VWR2A integrates a low-energy wide memory hierarchy made of a dedicated SPM and VWRs [72], as shown in Figure 2.4. Such a memory hierarchy is perfectly suited for the biomedical domain as most of the applications gather multiple biosignals over a defined sampling period and then process these data in blocks. During the processing, many data reads and writes are performed, and the wide memory hierarchy offers a fast and energy-efficient solution. The SPM serves as an L1 memory for data, and the VWRs as L0 or foreground memories. The SPM is shared between the columns and has a double interface: on the system side, it has the system bus width. On the accelerator side, it has the same width as the VWRs. A DMA performs the data transfers between the SPM and the system's main memory (i.e., L2 memory), while the LSU (see Chapter 2.3.4) moves the data between the SPM and the VWRs. The VWRs act as a buffer between the SPM and the RCs, which can access the elements in the wide registers word by word. The SPM size depends on the amount of data used by the applications of the targeted domain. If its size is restricted (e.g., due to power or area limitations), a double buffering scheme can be implemented between the SPM and the system's main memory (see Chapter 4.6).

The RCs and the VWRs are connected through a network of multiplexers, allowing multiple RCs to work in parallel on different sections of a VWR (i.e., each RC can only access a fixed range of each VWR to maintain their single-port nature). Theoretically, wider VWRs reduce the access energy per word (see Section 2.2.2). However, the size of the VWRs has to consider the application target domain and the datapath width consuming the data words. On the one hand, the VWRs need to be large enough to minimize the energy access per word and the frequency at which new data must be loaded. On the other hand, wider VWRs have higher leakage, and their usage might be suboptimal depending on the amount of data to process and the timing constraints of the targeted domain. As shown in Chapter 3, an analysis of the targeted domain gives an approximate range, and design exploration is required to fine-tune this value.

The number of VWRs is also an important design choice, although the possible range of values is smaller than their possible bitwidth. The need for multiple VWRs originates from their single-port nature. Two VWRs are required at a minimum to access two operands. A third VWR avoids overwriting one of the input VWRs when writing back the result. Therefore, this design choice is often limited to two or three VWRs. For the two instances of Chapters 3 and 4, three VWRs are used. A fourth VWR would significantly increase power consumption without necessarily improving performance, as a VWR can be refilled with

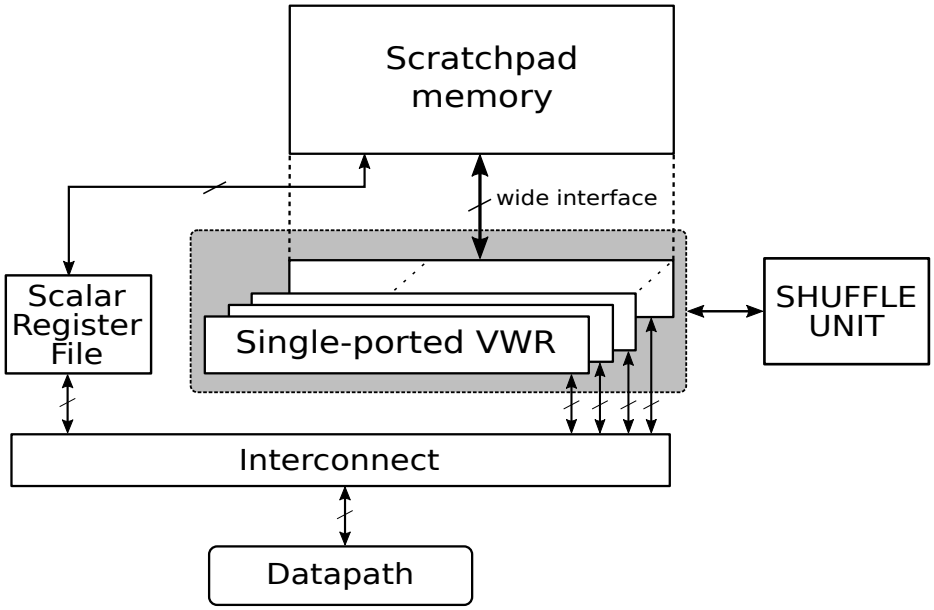


Figure 2.4: Wide memory hierarchy organization as proposed by the FEENECS template of [58].

new data from the wide SPM in one cycle, but could potentially be required and beneficial for some domains.

VWRs are meant for data with strong spatial locality, such as data arrays of biosignals. For this reason, it has been proposed to use a dedicated SRF for scalar variables that do not exhibit enough access regularity to combine with the vectorized access to data arrays [48]. The SRF depth depends on the applications. It can be single- or multi-ported as it is usually small compared to the SPM and VWRs, therefore almost not contributing to the total area and power consumption (see Chapters 3 and 4). The SRF contains scalar values, such as constants, mask values (e.g., to compute addresses), or loop sizes. These values are initially stored in the SPM and loaded inside the SRF at runtime. This further limits the required size of the SRF as new values can be loaded during execution.

2.3.3 Shuffle unit

The architecture is completed with the integration of a shuffle unit, as proposed in [72]. As each RC only accesses a fixed slice of a VWR, data reordering is needed to move the data inside the full VWR. Such reordering is possible through the RCs interconnection matrix, but it is only efficient if it can be integrated into the computation flow without any cycle penalty. Otherwise, that would be highly inefficient in terms of performance and energy, whereas the shuffle unit enables fast and energy-efficient data reordering [48].

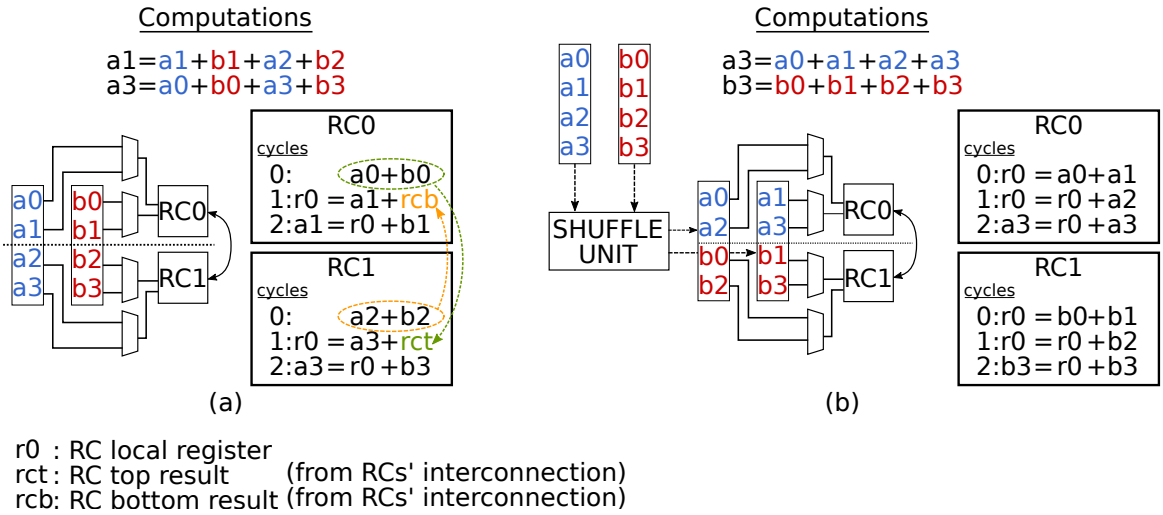


Figure 2.5: Data re-ordering inside VWRs: (a) each RC computes on its own slice of the VWRs and passes the result over the interconnection to the other RC (efficient only if no cycle penalty) and (b) the shuffle unit is used to re-order the data inside the VWRs before each RC computes on its own slice. Using the RCs' interconnection in (b) would add a lot of latency and using the shuffle is more efficient.

Figure 2.5 shows two examples, one where the data reordering can be efficiently implemented with the RCs' interconnection (Figure 2.5.a) and one where the shuffle unit is more efficient (Figure 2.5.b). In the first case, Figure 2.5.a, RC0 can only access a0, a1, b0, and b1, while RC1 can only access a2, a3, b2, and b3 (because of the VWRs single-port nature). However, RC0 needs to store the sum of a1, b1, a2, and b2 in a1, and RC1 the sum of a0, b0, a3, and b3 in a3. In this specific example, RC0 and RC1 can take advantage of the RCs' interconnection to share the data of their slice without any cycle penalty. On the contrary, Figure 2.5.b, cannot do it without penalty cycles in which RC0 and RC1 would only load one data to share it with the other RC. In this case, using the shuffle unit to reorder the data is more efficient. These are simplified examples with only two RCs, random operations, and two small VWRs, but they illustrate scenarios encountered multiple times while mapping kernels on instances of the template. This becomes more problematic with more realistic designs (i.e., more RCs and larger VWRs) such as the ones in Chapters 3 and 4.

The shuffle unit takes as input the data contained in one or more VWRs, applies a hardcoded shuffle operation on the data, and stores the result in a VWR. Various specific implementations are possible. The template instances of Chapters 3 and 4 use the same design based on three VWRs: the input data inside the first two VWRs are shuffled and stored in the third VWR. The shuffling operations are limited to a few hardcoded schemes to limit power consumption. Some examples of data shuffling are words interleaving, even or odd index pruning, and circular shifting (see Chapters 3 and 4 for more details).

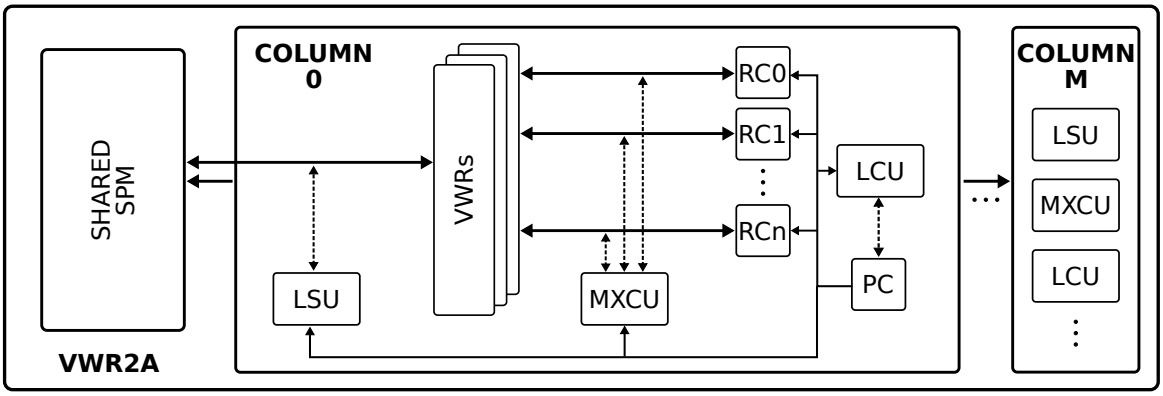


Figure 2.6: Illustration of the LCU, MXCU, and LSU specialized slots corresponding function.

2.3.4 Specialized slots

The concept of specialized slots, borrowed from the VLIW architecture, is introduced with an LSU, an LCU, and an MXCU on top of the RCs of each column, as shown in Figure 2.6. They all have their own instruction stream synchronized with the RCs in the column via the shared PC. Similar to the RCs, all the specialized slots have a local program memory that can be reconfigured at runtime by loading configuration words from the context memory. The specialized slots further enable leveraging the ILP inherent to any code. Inspired by the brain organization and its region's distinct roles (e.g., speech, vision, motor control), the specialized slots and the RCs of a column work in parallel on a common task but with their own dedicated functions.

Each specialized slot is optimized for its specific set of tasks (see Figure 2.6) with a dedicated ISA, datapath, and register file. This allows performant and energy-efficient execution of different types of kernels. However, while the slots are designed for certain tasks, they can also execute certain general instructions. For example, the LSU usually takes care of incrementing an address pointer. However, if the LSU is already executing another instruction at a certain cycle, the address pointer incrementation can be offloaded to another slot, as long as the ISA of that slot supports it. Such a scenario uses the SRF to share data between the slots and improves performance by increasing ILP. Here I describe each specialized slot, its tasks, and the related optimizations.

LSU: Load-and-Store Unit

The LSU controls the data transfers between the SPM and the VWRs or the SRF. The data arrays are allocated to VWRs while scalar values (e.g., loop size) are stored in the SRF. The

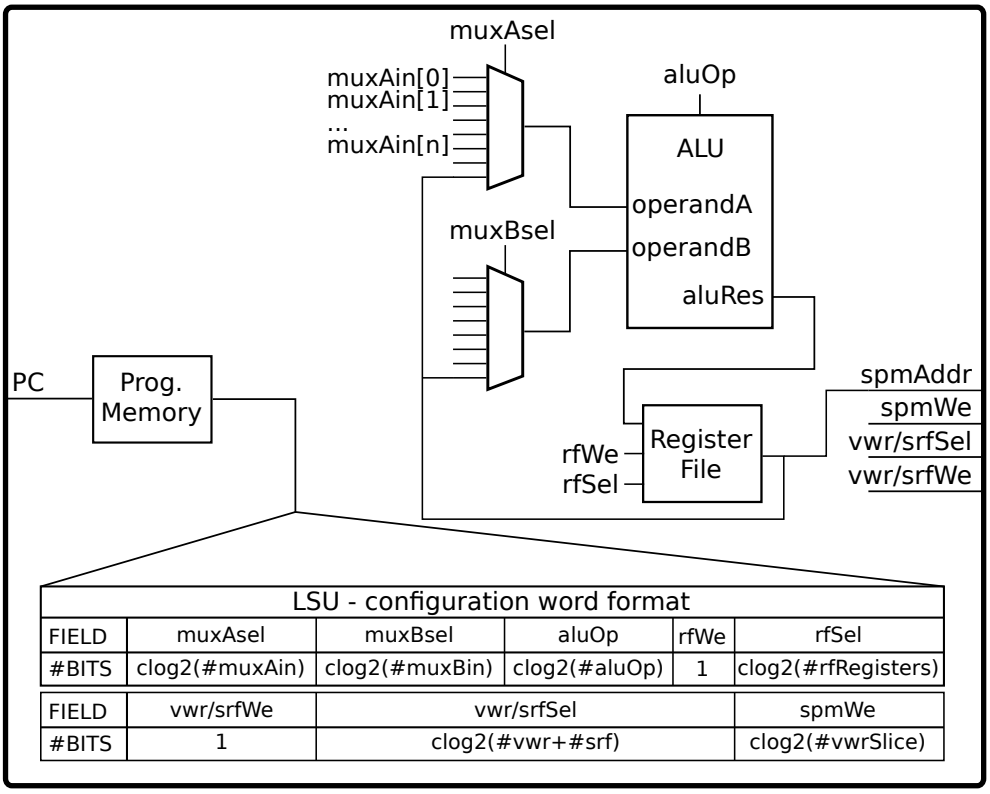


Figure 2.7: Block diagram of the LSU in the architectural template.

main task of the LSU is to generate addresses, and its datapath is optimized for such a task. Advanced arithmetic operations (e.g., multiplications, divisions) are rarely required to generate addresses. Therefore an ALU implementing logical bit operations (and, or, xor), addition/subtraction, and left/right logical bit shift is often sufficient. Depending on the targeted domain, it can be extended with other specific instructions. For example, the instantiation of Chapter 4 has a bit reversal operation to improve the performance of FFT computation. The template does not recommend integrating an immediate field inside the configuration words as it uses many bits and would increase the local program memory size and the context memory size. More efficient solutions exist, such as fixed input to zero or any other constant value. The SPM depth fixes the datapath width of the LSU because it has to generate these SPM addresses. The register file depth is also adjustable and its optimal value is domain-dependent. Nevertheless, based on experimentation, an eight entries RFs seems to be a good design starting point in many cases. Figure 2.7 shows the template of the LSU specialized slots and some of the optimizable parameters.

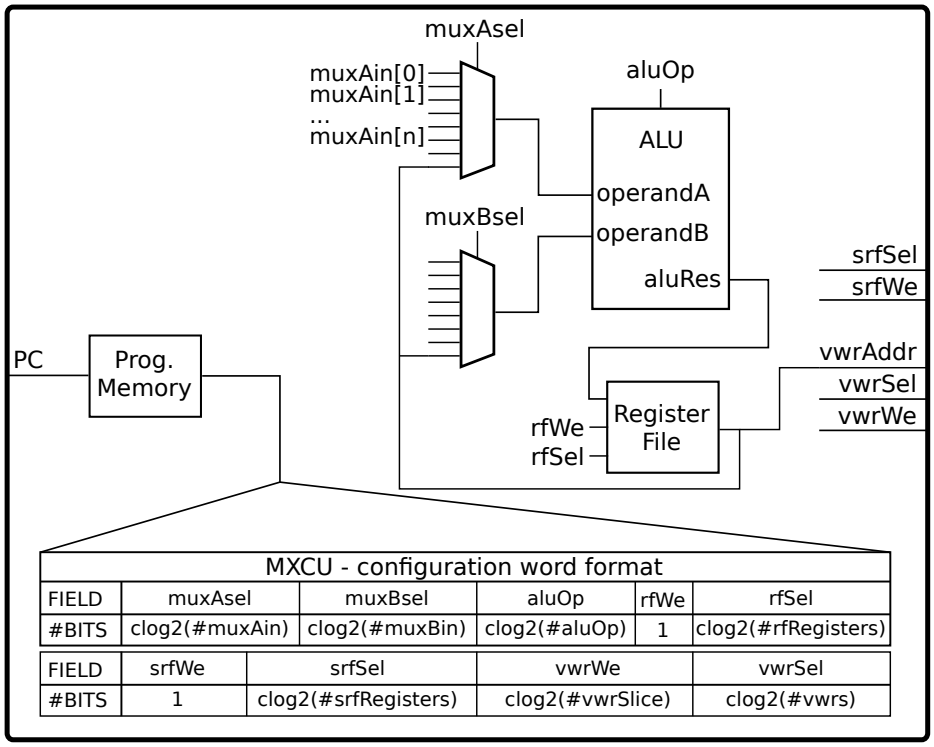


Figure 2.8: Block diagram of the MXCU in the architectural template.

MXCU: MultipleXer-Control Unit

The MXCU computes the addresses of the VWRs' words passed to the RCs. To achieve high performance, the number of instructions of the innermost loop mapped on the architecture is critical, and the MXCU usually plays an essential role. As all RCs access their own VWR slice, they could access a different word of that slice. However, such a design would require one address to be generated per RC, and the MXCU should have multiple FUs in order to generate these addresses fast enough. Consequently, it would require wider instructions (to control all the FUs), and larger local program memory and context memory. To avoid this important power and complexity overhead, the VWR2A template proposes a unique address generation shared by the RCs to access their VWR slice. This address is also used to write the data back to any of the VWRs. Although this structure adds some constraints to the kernel mapping, they can be solved with careful data placement and proper use of the shuffling unit. Like the LSU, the MXCU's main task is to generate addresses, and its ALU is optimized for it by executing addition/subtraction, logical bit operations (and, or, xor), and left/right logical bit shift. Depending on the targeted domain, more supported operations might be relevant. The implementation of an immediate field is also discouraged, and the use of a few constant entries for the ALU's input multiplexers is preferred. For example, the instance in Chapters 3 has entries fixed to 0, 1, and 2. These

values are enough to create most of the access patterns. The address range of the VWRs fixes the datapath width of the MXCU (taking into account that the RCs access only a slice of the VWRs). For example, a column with 4 RCs and a VWR bitwidth of 4096 bits (i.e., 128 32-bit words) gives a datapath width of 5 bits. The reason is that the width of the VWRs in 32-bit words (i.e., 128) divided by four slices (one per RC) gives each RC access to 32 words, and 5 bits ($2^5 = 32$) are required to access them. Figure 2.8 shows the template of the MXCU specialized slots and recapitulates the optimizable parameters.

LCU: Loop-Control Unit

Compared to a common CGRA using modulo scheduling and hardware kernel execution control —with a prologue, steady-state, and epilogue scheme [4]— the LCU is programmable. Its primary task is to handle loop control (e.g., counter increments, branches) and conditions (e.g., if-else blocks). Therefore, its generic ISA includes jump, conditional jump, signed addition/subtraction, logical bit operations (and, or, xor), and left/right logical shifts. The jump and branch instructions update the PC register shared by all the units in a column (i.e., the RCs and the specialized slots). When multiple columns execute one kernel conjointly, their respective PCs are synchronized. The LCU is the only unit for which the template recommends an immediate field to store the destination address of the conditional branch instructions. Therefore, the minimum bitwidth of this field depends on the depth of the program memory.

In theory, the LCU can implement any nested loop depth; however, an instantiation is limited by its internal data register file and local program memory size. Therefore, these two parameters must be optimized based on the application domain. However, thanks to the intrinsic flexible nature of a DSIP, an instantiation can often adapt to unexpected or rare scenarios. For example, the potential loop depth limitation of a specific LCU design can be overcome by using more columns or the SRF. The LCUs of the VWR2A instances in Chapters 3 and 4 have a four entries local register file and execute kernels that have up to three level nested for loops, all executing on two columns which means two register files (one per LCU) are available. However, the two columns are mainly used to increase the number of RCs. For example, data-intensive kernels with simple control structures (e.g., *for* loops, as in the finite impulse response (FIR) Filter evaluated in chapter 3), could be executed in one column even with a three-nested loop implementation. In this case, each loop size is stored in the SRF and loaded to the LCU local register file whenever the loop is initialized. Then, the LCU decrements each register every iteration accordingly and compares them with its constant zero input to decide whether to jump.

The datapath width of the LCU is not fixed by other parameters compared to the LSU or the MXCU. However, a lower bound value can be defined based on the VWR's

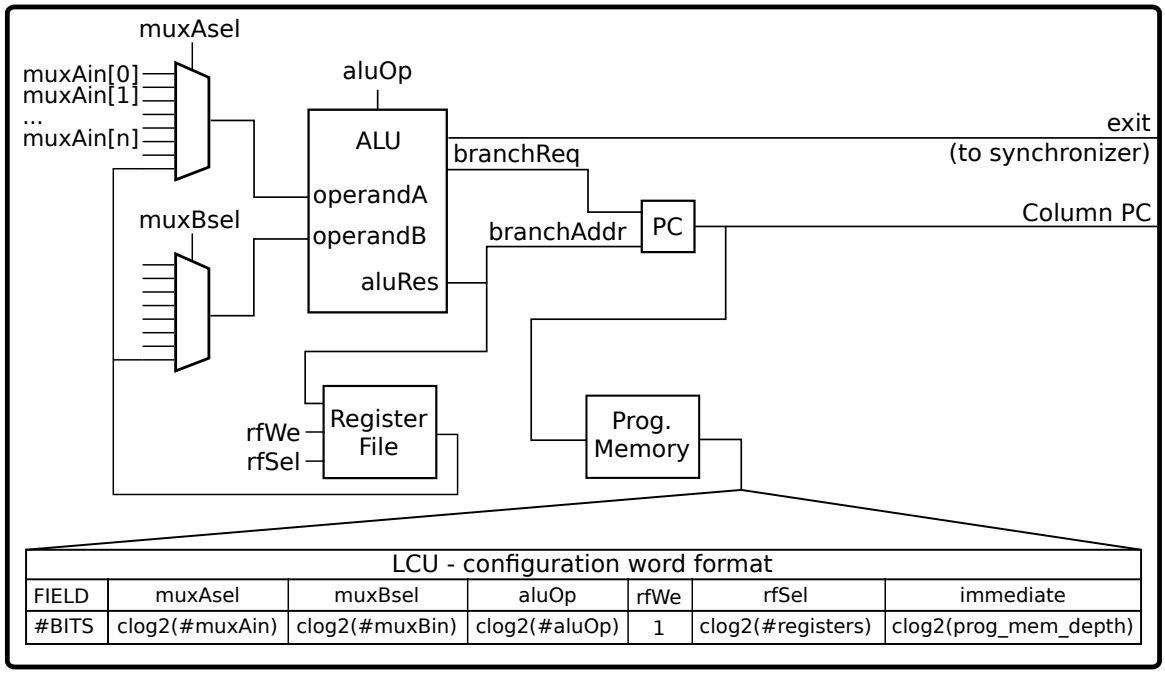


Figure 2.9: Block diagram of the LCU in the architectural template.

size. In the previously used example with a column containing four RCs and VWRs with 128 words (i.e., 32 words per RC), the loop size is usually limited to a maximum of 32 iterations. This is due to the size of a VWR slice (i.e., 32 words per RC) and the fact that in each iteration usually each RC consumes at least one data element. Therefore, after 32 iterations, the loop is exited to load the next batch of data using the LSU. Additionally, the LCU is able to generate PC values to access all the configuration words stored in the local program memory of all the RCs and specialized slots. Therefore, its minimum datapath width is defined as: $dp\_width_{min} = max(clog2(slice_{size}), clog2(prog\_mem\_depth))$. A loose upper bound value can be defined similarly: considering the worst-case scenario where only one RC consumes one data element at a time, the loop would take 128 iterations corresponding to the VWRs size. While the lower bound value is precise, the latter gives more of an approximation to help the designer. Figure 2.9 shows the template of the LCU specialized slots and recapitulates the optimizable parameters.

2.4 Illustrative kernel mapping analysis

The high-level mapping of a C code example is illustrated here to explain better the use of an instance generated from the template. For conciseness and clarity, a simple instance of one column with four RCs is used, and a simple kernel adding two vectors is considered.

Original C code

```
void vec_add (
      int32_t *v0,
      int32_t *v1,
      int16_t size) {
 int16_t i;
 for(i=0; i<size; i++) {
   v0[i] = v0[i]+v1[i];
 }}
```

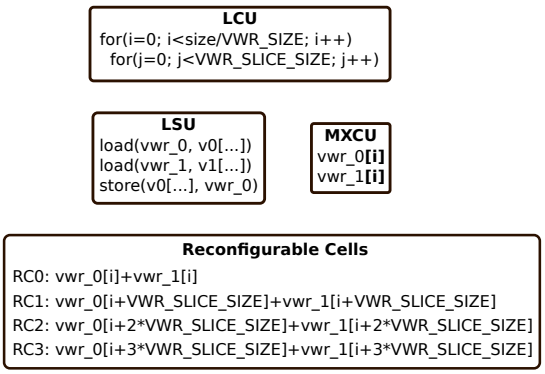
(a)

VWR2A pseudocode

```
void vec_add (
      int32_t *v1,
      int32_t *v2,
      int16_t size) {
 int16_t i, j;
 for(i=0; i<size/VWR_SIZE; i++) {
   load(vwr_0, v0[i*VWR_SIZE:(i+1)*VWR_SIZE])
   load(vwr_1, v1[i*VWR_SIZE:(i+1)*VWR_SIZE])
   for(j=0; j<VWR_SLICE_SIZE; j++) {
      vwr_0[i] = vwr_0[i]+vwr_1[i];
   }
   store(v0[i*VWR_SIZE:(i+1)*VWR_SIZE], vwr_0)
 }}
```

(b)

VWR2A mapping



(c)

Figure 2.10: Illustrative kernel mapping analysis with the original C code for two vectors element-wise addition (a), its translation to pseudocode for the VWR2A instance (b), and the distribution of the tasks to the specialized slots and RCs (c).

Figure 2.10 shows the C code kernel, its translation to pseudocode for the VWR2A instance, and finally, the high-level distribution of the various tasks to the different units (from left to right). It shows how the specialized slots increase the ILP of the design. In this case, the innermost loop (i.e., adding two elements from $v0$ and $v1$) would take only one cycle:

- The LCU updates the counter (i) and checks the loop condition (considering a single cycle counter update and conditional branch instruction is supported).
- The MXCU generates the addresses to access the two VWRs (i.e., vwr_0 and vwr_1).
- The RCs add the two elements of their respective VWR slice. In this example, four additions happen in parallel as there are four RCs.

A similar high-level decomposition can be found in almost any code, which is why the template proposes the implementation of these three specialized slots in particular. Nevertheless, the actual computations are domain-dependent and have to be optimized accordingly. For example, in the C code example of Figure 2.10, the access pattern of the VWRs is simple (continuous addresses), which simplifies the addresses generated by the MXCU. Depending on the complexity of the access patterns, the MXCU can support different operations, or the shuffle unit can support a re-ordering scheme that simplifies the access pattern.

Figure 2.10 shows the effects of the wide memory hierarchy on the original C code. The first one is the abstraction of the load and store operations outside the original loop.

Instead of loading data every iteration, the LSU loads a batch of data to the VWRs, and then the RCs compute on these data. The results are written back to the background memory (i.e., the SPM) only when all the data inside the VWRs have been consumed. The second effect, not occurring in the simple example of Figure 2.10, is the data placement constraints imposed by the single-port nature of the VWRs. From the datapath side (i.e., the RCs), the VWRs are decomposed in private memory slices (one per RC). It becomes a problem when one RC requires data stored in another slice. The FFT kernel (see Section 2.6.2) was the most challenging in that regard, although it has a very regular access pattern. The problem is that this pattern changes every stage, and at every new stage, the RCs require data stored in another slice. Such a problem requires intelligent use of the RCs interconnection to share data between the RCs efficiently, but more is needed. As discussed in Section 2.3.3, the shuffle unit is there to help with this problem. However, to limit its power and area overhead, the supported shuffling operations must be carefully evaluated to cover multiple scenarios and kernels instead of a single use case.

2.5 Comparative evaluation of the VWR2A template with respect to other programmable architectures

To illustrate the flexibility and performance advantages of the proposed VWR2A template, this section proposes a quantitative comparison with two representative state-of-the-art programmable low-power architectures: a low-power SoC and a baseline CGRA architecture, both targeting the biomedical domain. In this case, the template uses the instance proposed in Chapter 3 targeting data-intensive kernels of the biomedical domain. Although a VWR2A instance can hypothetically execute any C code, a particular instance has some practical limitations. The first one is the size of the units' local program memory. Although it is theoretically possible to increase its size, there exist limitations, such as power consumption. The second is the currently missing compiler support that makes it challenging to map complete applications. Therefore, VWR2A is integrated inside platforms as a hardware accelerator or a co-processor.

2.5.1 Evaluated architectures

Low-power SoC

To demonstrate the advantages of the proposed DSIP architecture, I compare it with a low-power SoC intended for biomedical signal acquisition and processing [32]. As shown in Figure 2.11, this platform features an ARM Cortex-M4F (CM4F) processor, 192 KiB of

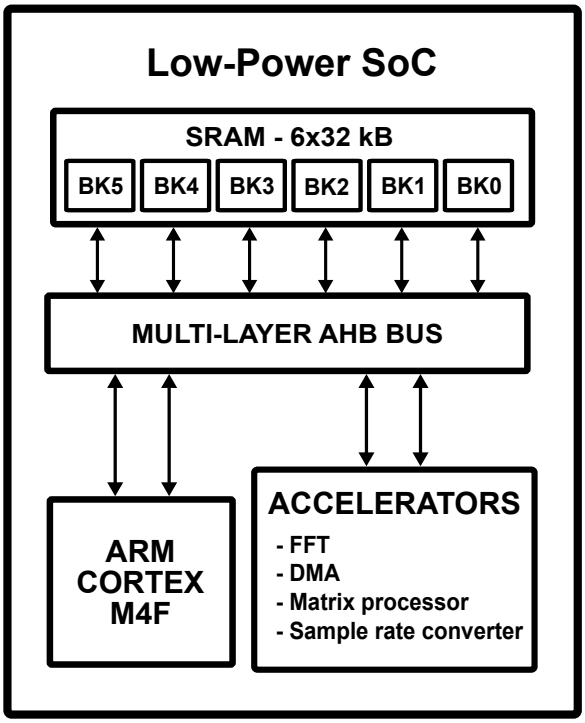


Figure 2.11: Low-power SoC for biomedical signal acquisition and processing architecture proposed in [32].

SRAM (divided into six banks that can be individually power gated), an analog front end for the acquisition of biosignals (e.g., electrocardiogram (ECG), photoplethysmogram (PPG)), a DMA, and several fixed-function accelerators. The ARM CM4F processor is a middle-class, low-power 3-stage pipeline core with VFPv4-SP (single-precision FPU) and digital signal processing extensions. The FPU extension is not relevant in the context of this thesis as floating-point arithmetic operations have not been considered, although the VWR2A template does not exclude them. The digital signal processing extension is more important in our context as it optimizes the execution of fixed-point operations. It enables 32-bit wide SIMD operation on four 8-bit values or two 16-bit values. The ARM CM4 also supports 32-bit integer multiplication and MAC operations in one cycle and 32-bit integer division in 2-12 cycles.

Baseline CGRA

Figure 2.12 shows the baseline CGRA used for comparison. Its design is based on two very similar architectures optimized for the biomedical domain introduced in [33] and [34]. It features a 4-by-4 reconfigurable array organized in four columns of four RCs. Similarly to VWR2A, all the RCs of a column have a shared PC, and the columns can work independently

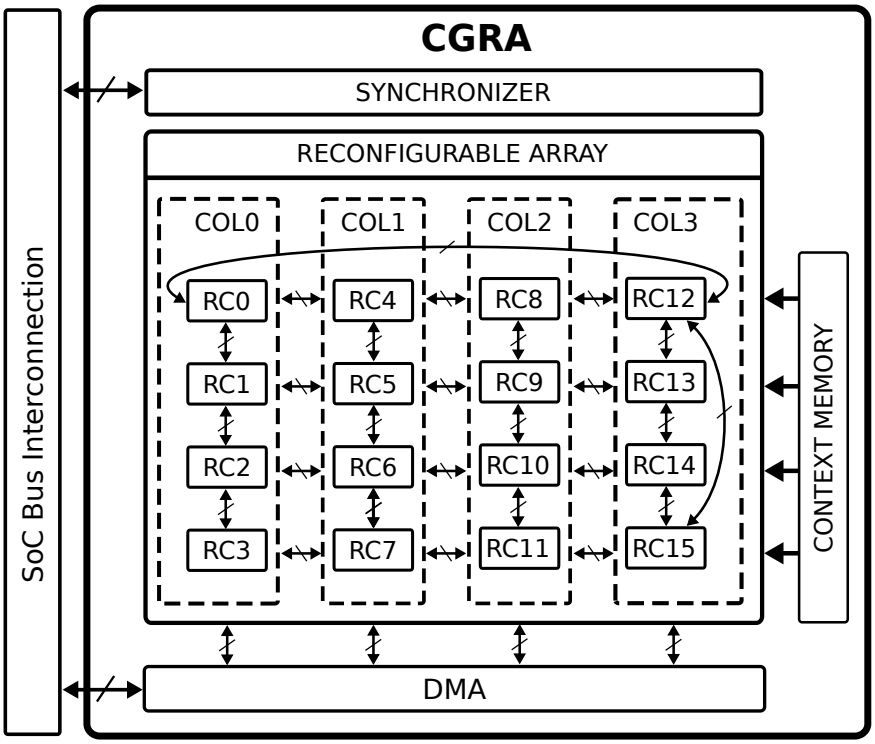


Figure 2.12: CGRA architecture baseline based on [33] and [34]. The RCs nearest neighbors interconnection torus is only shown for a few cells at the edges (RC0-RC12 and RC12-RC15) to simplify the figure, but all the other RCs at the edges have similar connections.

on different kernels or in synchronization on one kernel. This architecture does not have any dedicated data memory and relies on the platform memory in which it is integrated. The instruction flow (e.g., branch, data computations, addresses generation) is entirely handled by the RCs since there are no specialized slots. All the RCs have the same design: a 32-bit datapath, a program memory of 32 words of 32 bits, and a four-entry RFs. The RCs are interconnected with their closest neighboring cells in a torus fashion (see Figure 2.12).

This architecture has been integrated inside the low-power SoC described above. Each column possesses a master port connected to the AMBA-AHB bus of the platform, limiting the data access request to one RC per column (all the RCs can issue a load or store request, but only one at a time per column).

VWR2A

The VWR2A instance used for the experiments is taken from Chapter 3 and shown in Figure 3.3 of that chapter. This instantiation is described in detail in its corresponding chapter and it is used here to illustrate the proposed VWR2A template. Its main features are

a 4-by-2 reconfigurable array (i.e., two columns of four RCs), three VWRs and one SRF per column, and a $32\,\mathrm{KiB}$ shared SPM. The goal is to show the template's high-level features, their impact on the mapping, and how they can significantly improve performance. VWR2A is integrated inside the SoC similarly to the baseline CGRA.

2.6 Experimental setup

2.6.1 Performance and energy characterization

The complete SoC, either including the baseline CGRA or the VWR2A instance, is synthesized with the Cadence genus tool [76] using the TSMC 40 nm LP CMOS technology [77] at $80\,\mathrm{MHz}$ (the original SoC frequency) and post-synthesis simulations are run with Cadence Incisive Verification Platform [76] to compare the performance and the energy of all the implementations. This allows cycle-accurate simulations from which the switching activity of the cells is extracted and used for power estimation with Synopsys PrimePower [78].

2.6.2 Benchmark kernel

The Fourier transform is a good example of a kernel used in many biomedical applications [79–81] as it enables analysis and feature extraction in the frequency domain. This kernel is used throughout this chapter as a representative example of code used in the biomedical domain (complete applications are evaluated in Chapters 3 and 4). It represents a significant workload —in terms of data computation and data access— for low-power architectures and therefore is an excellent kernel to compare the three considered architectures and illustrate the use of VWR2A. The Fourier transform is usually computed with an optimized algorithm called FFT, which reduces the computation complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, where n is the number of points. A common implementation is the in-place radix-2 FFT algorithm [82]. The radix-2 algorithm divides the computation into k stages, where $2^k = N$ and N is the number of input points, as shown in Figure 2.13 (with $N = 8$). Therefore, a radix-2 FFT expects a power-of-two input size. Other variants of radix-M FFT, where M is a power-of-two index (e.g., radix-4), are also possible, but they expect a power-of-M input size, limiting their use. It is also possible to mix different radix implementations, but we focus only on the radix-2 version for simplicity and clarity.

All the stages of a radix-2 FFT execute the same flow of operations. The only changes are the coefficients and the data ordering. The basic block of operation is called a butterfly (see Figure 2.13). The output of the radix-2 algorithm is in bit-reversed order (i.e., the output signal indexes are in bit-reversed order), as shown in Figure 2.13, and an extra reordering

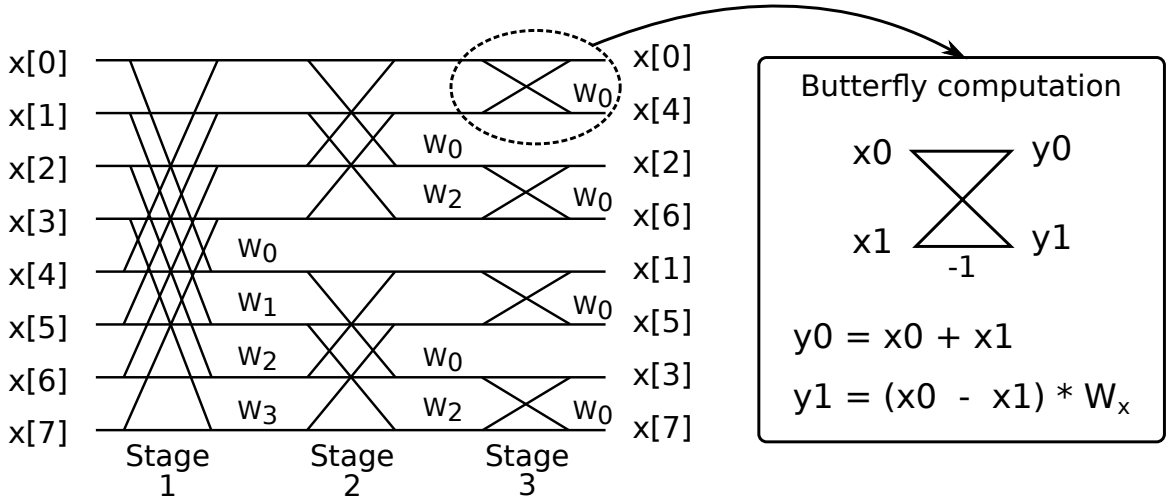


Figure 2.13: 8 points radix-2 FFT computation graph and butterfly computation.

step is required to obtain the correct output order according to the definition of the Fourier transform. This step can also be performed before the radix-2 execution without any impact on the algorithm complexity as it only changes the order of the computations.

An optimized version is used for real-valued FFTs (i.e., the imaginary part is null). The N real values sequence is transformed into an $N/2$ complex sequence. Then, the regular radix-2 implementation is used. This technique reduces the computations of the FFT kernel but requires some additional operations to recover the correct output. The overall gain is approximately a factor of 2 compared to a complex FFT of size N where the imaginary part is zero. The three evaluated architectures use the radix-2 FFT algorithm (sometimes a radix-4 implementation for the common microcontroller software interface standard digital signal processing (CMSIS-DSP) library), but the mapping on each architecture is different (see Chapter 2.7.1).

The recursive C implementation of the radix-2 FFT algorithm is shown in Figure 2.14. This specific implementation expects the input signal to be in bit-reversed order. This figure shows that for each butterfly step, 10 accesses to the memory are required (6 reads and 4 writes), 6 addresses have to be generated, and 10 arithmetic operations are performed (considering the *fxp_mult()* function as one operation). For a single-core scalar processor, it represents a theoretical minimum of 26 instructions (much more in reality as each of these high-level instructions is typically made of multiple real machine instructions).

```
00: int n_group = FFT_SIZE/2;
01: int offset = 1;
02: for(int i=1; i<=LOG2(FFT_SIZE); i++){
03:   for(int k=0; k<n_group; k++){
04:     int base_idx = k << i;
05:     for (int j=0; j<offset; j++){
06:       fxp even_real = Real_Out[base_idx+j];
07:       fxp even_imag = Imag_Out[base_idx+j];
08:       fxp odd_real = Real_Out[base_idx+j+offset];
09:       fxp odd_imag = Imag_Out[base_idx+j+offset];
10:       fxp w_r = factors_real_fxp[j*n_group];
11:       fxp w_i = factors_imag_fxp[j*n_group];                    6 data reads
12:       fxp p_sum_r = fxp_mult( w_r, odd_real ) - fxp_mult( w_i, odd_imag );
13:       fxp p_sum_i = fxp_mult( w_i, odd_real ) + fxp_mult( w_r, odd_imag );
14:       Real_Out[base_idx+j] = even_real + p_sum_r;      4 data writes
15:       Real_Out[base_idx+j+offset] = even_real - p_sum_r;
16:       Imag_Out[base_idx+j] = even_imag + p_sum_i;
17:       Imag_Out[base_idx+j+offset] = even_imag - p_sum_i;  10 arithmetic ops
18:     }
19:   }
20:   n_group >>= 1;
21:   offset <<= 1;
22: }
```

Figure 2.14: C code of the FFT radix-2 algorithm and inner loop analysis.

2.7 Experimental results

2.7.1 Software mapping comparison

The radix-2 FFT is used as an example kernel used in the biomedical domain to show the different mapping of kernels on the three evaluated architectures. The ARM Cortex-M4 low-power SoC, referred to as the **CM4** architecture, has two implementations: one using the plain C code of Figure 2.14 and one using the CMSIS-DSP library of ARM[83]. The disassembly of the former is shown in Figure 2.15. Only the innermost loop is shown as it is the most relevant to understand the performance differences. Each call to the *fxp_mult* function takes four instructions. Because of these calls, the compiler enforces the use of r0 and r1 to pass the two functions' arguments (i.e., the two numbers to multiply). If the function was inlined, a few instructions could be saved (at the cost of extended code), but it would not significantly impact performance. The most important fact to notice is the large number of load (ldr) and store (str) instructions. This significantly impacts performance and energy consumption. Each of these instructions accesses the memory and makes the long wires of the bus switch every time. They also consume cycles of the processor.

The disassembly of the version using the CMSIS-DSP library is not shown here because its complexity does not allow a simple analysis. Although this version is much faster than the plain C version of Figure 2.14, it is not due to the actual use of the CM4 digital signal processing extensions. From the disassembly analysis, there is no use of any packed-SIMD instructions, because 32-bit input data are used. Therefore, the performance improvement is only due to the manually optimized code of the CMSIS-DSP by using, for example, loop unrolling. This is at the cost of a much larger code ($6.7\times$ more bytes of code compared to the plain C for a 512-point real FFT). Based on the size of the FFT, the library uses either the radix-2 or the radix-4 algorithm.

The mapping on the baseline CGRA, referred to as the **CGRA** architecture, can exploit the radix-2 algorithm ILP and data-level parallelism thanks to its multiple RCs. For simplicity, the mapping of the FFT has been limited to two columns of the reconfigurable array. Figure 2.16 shows the disassembly code of the innermost loop of the code mapped on the CGRA. It clearly shows the benefit of having multiple processing elements (i.e., the RCs), as the innermost loop size is reduced to 7 instructions per RC. A second advantage of the CGRA is that it supports fixed-point multiplication in hardware, reducing the four instructions of the CM4 to only one. The first column (i.e., RC0-3) works on the real part of the complex input data, while the second column (i.e., RC4-7) works on the imaginary part.

```
void fft_cplx_radix2_iter (fxp *Real_Out, fxp *Imag_Out, int fft_s, int nbits){
...
00:  ldr      r3, [sp, #0]                    Addresses generation
01:  str      r2, [sp, #60]                            +
02:  ldr.w    r3, [r3, r4, lsl #2]   butterfly input read from memory
03:  str      r3, [sp, #12]
04:  ldr.w    r3, [fp, r4, lsl #2]
05:  str      r3, [sp, #16]
06:  ldr.w    r3, [r2, r4, lsl #2]
07:  str      r3, [sp, #20]
08:  ldr.w    r3, [sl, r4, lsl #2]
09:  str      r3, [sp, #24]
10:  ldr      r3, [sp, #44]
11:  ldr      r2, [sp, #48]
12:  ldr      r3, [r3, r7]
13:  ldr      r2, [r2, r7]
14:  ldr      r1, [sp, #20]
15:  str      r2, [sp, #28]
16:  mov      r0, r3
17:  str      r3, [sp, #56]                   Butterfly element
18:  bl       <fxp_mult>
19:  ldr      r1, [sp, #24]
20:  mov      r9, r0
21:  ldr      r0, [sp, #28]
22:  bl       <fxp_mult>
23:  ldr      r1, [sp, #20]
24:  sub.w    r9, r9, r0
25:  ldr      r0, [sp, #28]
26:  bl       <fxp_mult>
27:  ldr      r3, [sp, #56]
28:  str      r0, [sp, #20]
29:  ldr      r1, [sp, #24]
30:  mov      r0, r3
31:  bl       <fxp_mult>
32:  ldr      r3, [sp, #20]
33:  ldr      r2, [sp, #0]
34:  add      r3, r0
35:  mov      r0, r3
36:  ldr      r3, [sp, #12]
37:  add      r3, r9
38:  str.w    r3, [r2, r4, lsl #2]
39:  ldr      r3, [sp, #12]     Butterfly output written to memory
40:  ldr      r2, [sp, #60]
41:  sub.w    r3, r3, r9
42:  str.w    r3, [r2, r4, lsl #2]
43:  ldr      r3, [sp, #16]
44:  add      r3, r0
45:  str.w    r3, [fp, r4, lsl #2]
46:  ldr      r3, [sp, #16]
47:  subs     r0, r3, r0
48:  str.w    r0, [sl, r4, lsl #2]
49:  adds     r4, #1
50:  ldr      r3, [sp, #40]                   Inner loop condition check
51:  cmp      r8, r4
52:  add      r7, r3
53:  bgt.n    00
...
}
```

Figure 2.15: Cortex-M4 ARMv7E-M ISA disassembly code for the radix-2 FFT algorithm (plain C code of Figure 2.14) innermost loop.

```
void fft_cplx_radix2_iter (fxp *Real_Out, fxp *Imag_Out, int fft_s, int nbits)

RC0:                    RC1:                    RC2:                    RC3:
  ...                     ...                     ...                     ...
L0:mvs    r1, data      L0:nop                  L0:ld     self, data    L0:adds   r3, #1
   nop                     nop                     mvs    self, rcr        ld     self, [r0]
   fxpmul self, rct, r1    nop                     fxpmul self, rcb, self  nop
   ld     self, [r0]       nop                     adds   self, rcr, self  mvs    self, r1
   adds   r1, self, rcr    nop                     subs   self, rcr, self  adds   r0, r0, self
   str    r1, [r0]         mvs    self, rcb        mvs    self, r0         mvs    self, r3
   adds   r0, #4           str    rcr, [rcb]       adds   r0, #4           nop
   ...                     ...                     ...                     ...

RC4:                    RC5:                    RC6:                    RC7:
  ...                     ...                     ...                     ...
L0:ld     r1, [r0]      L0:nop                  L0:mvs    self, data    L0:nop
   nop                     nop                     mvs    self, rcl        ld     self, [r0]
   fxpmul self, rct, r1    nop                     fxpmul self, rcb, self  nop
   subs   self, rcl, self  nop                     ld     self, [r0]       nop
   subs   self, rcl, self  nop                     adds   r1, rcl, self    adds   r0, r0, rcl
   mvs    self, r0         mvs    self, rct        str    r1, [r0]         nop
   adds   r0, #4           str    rcl, [rct]       adds   r0, #4           blt    rcl, r3, L0
   ...                     ...                     ...                     ...
```

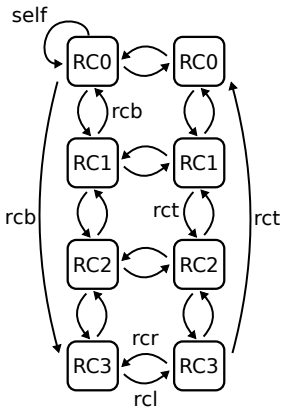


Figure 2.16: CGRA disassembly code for the radix-2 FFT algorithm innermost loop.

The disassembly clearly shows the use of the RCs' interconnection (i.e., rc[t/b/l/r] and self), which enable fast data sharing between the RCs. This reduces the number of accesses to the memory. However, Figure 2.14 shows that six load and four store operations are required at least per iteration for the data transfer only. Although the mapping on the CGRA has two ports to the memory bus (i.e., one per column), as they both access the same memory bank, they cannot happen in parallel. Using more columns would not improve performance as the bus throughput —and not the processing power— is the bottleneck. The SoC has multiple banks and a multilayer AHB bus, which enables parallel access to all the banks. However, to take advantage of it, the data have to be placed knowing the access pattern of the algorithm, the FFT in this case, which changes every iteration (every outer loop iteration for the FFT). Without hardware support for such a re-ordering, implementing it in software would take more time than the bus contention penalty of the current mapping.

Figure 2.17 shows the disassembly of the VWR2A instance, referred to as the **VWR2A** architecture for simplicity. The VWR2A implementation executes the bit-reversal reordering after the FFT computation. The mapping also uses two columns, as for the CGRA, but it is very different due to the use of the VWRs. While the CGRA and the CM4 implementations treat the data one by one, the VWR2A first loads a batch of data inside the VWRs (lines 03, 04, 09, and 11 in Figure 2.17) and processes it (lines 06, 07, 10, 13, and 14 in Figure 2.17) before loading the next one. As each RC has access to its own VWR slice, all the RCs of a column can access data in parallel, removing the bus throughput bottleneck. The previously mentioned data re-ordering problem is also present with the memory hierarchy of VWR2A. It is solved using the shuffle unit after each outer loop iteration to re-order the data and the FFT weights inside the VWRs. Finally, as the specialized slots take care of all the necessary extra computations not directly linked to the input data (e.g.,

void fft_cplx_radix2_iter (fxp *Real_Out, fxp *Imag_Out, int fft_s, int nbits)

COLUMN 0

```
LCU:                    LSU:                                        MXCU:                   RC0-3:
00:      ...            00:      ...                                00:      ...            00:      ...
01: L0: nop             01: L0: li      r1, #0                      01: L0: mv      r4, r6  01: L0: nop
02:      nop            02:      sadd r0, r1, r2                    02:      li      r6, #31 02:      nop
03:      nop            03:      sadd r7, srf[3], r0 / ld.vwr #1, [r7] 03:      li      r0, #0  03:      nop
04:      nop            04:      sadd r7, srf[3], r1 / ld.vwr #0, [r7] 04:      nop            04:      nop
05:      nop            05:      sadd r1, r0, #1                    05:      nop            05:      nop
06:      nop            06:      nop                                06:      nop            06:      sadd vwrc, vwra, vwrb
07: L4: nop             07: L4: nop                                 07: L4: sadd r0, r0, #1 07: L4: ssub vwra, vwra, vwrb
08:      beq     r1, #0, .L1    08:      sadd r7, srf[4], r3 / str.vwr #2, [r7] 08:      li      r0, #0  08:      nop
09:      beq     r1, #1, .L3    09:      sadd r7, srf[5], r3 / ld.vwr  #1, [r7] 09:      mv      r6, r4  09:      nop
10: L5: nop             10: L5: nop                                 10: L5: sadd r0, r0, #1 10: L5: smul vwrc, vwra, vwrb
11: L1: beq     r1, #0, .L2    11: L1: sadd r7, srf[3], r0 / ld.vwr  #1, [r7] 11: L1: li      r0, #0  11: L1: nop
12:      beq     r1, #1, .L3    12:      land  r1, r1, r4           12:      nop            12:      nop
13: L6: nop             13: L6: nop                                 13: L6: nop            13: L6: smul self, vwra, vwrb
14:      nop            14:      nop                                14:      sadd r0, r0, #1 14:      ssub vwra, vwrc-rcr
15: L2: nop             15: L2: sadd r3, r3, r6                     15: L2: nop            15: L2: nop
16:      bgepd r2, #0, .L0      16:      land  r3, r3, srf[1]       16:      nop            16:      nop
17:      ...            17:      ...                                17:      ...            17:      ...
```

COLUMN 1

```
LCU:                    LSU:                                        MXCU:                   RC0-3:
00:      ...            00:      ...                                00:      ...            00:      ...
01: L0: nop             01: L0: li      r1, #0                      01: L0: mv      r4, r6  01: L0: nop
02:      nop            02:      sadd r0, r1, r2                    02:      li      r6, #31 02:      nop
03:      nop            03:      sadd r7, srf[3], r0 / ld.vwr #1, [r7] 03:      li      r0, #0  03:      nop
04:      nop            04:      sadd r7, srf[3], r1 / ld.vwr #0, [r7] 04:      nop            04:      nop
05:      li      r3, #31        05:      sadd r1, r0, #1            05:      nop            05:      nop
06:      nop            06:      nop                                06:      nop            06:      sadd vwrc, vwra, vwrb
07: L4: bgepd r3, #0, .L4       07: L4: nop                         07: L4: sadd r0, r0, #1 07: L4: ssub vwra, vwra, vwrb
08:      li      r3, #31        08:      sadd r7, srf[4], r3 / str.vwr #2, [r7] 08:      li      r0, #0  08:      nop
09:      li      r1, #0         09:      sadd r7, srf[5], r3 / ld.vwr  #1, [r7] 09:      mv      r6, r4  09:      nop
10: L5: bgepd r3, #0, .L5       10: L5: nop                         10: L5: sadd r0, r0, #1 10: L5: smul vwrc, vwra, vwrb
11: L1: li      r3, #31        11: L1: sadd r7, srf[3], r0 / ld.vwr  #1, [r7] 11: L1: li      r0, #0  11:      nop
12:      li      r1, #1         12:      land  r1, r1, r4           12:      nop            12:      nop
13: L6: nop             13: L6: nop                                 13: L6: nop            13: L6: smul self, vwra, vwrb
14:      bgepd r3, #0, .L6      14:      nop                        14:      sadd r0, r0, #1 14:      ssub vwra, vwrc-rcl
15: L2: nop             15: L2: sadd r3, r3, r6                     15: L2: nop            15: L2: nop
16:      nop            16:      land  r3, r3, srf[1]               16:      nop            16:      nop
17:      ...            17:      ...                                17:      ...            17:      ...
```

Figure 2.17: VWR2A instance disassembly code for the radix-2 FFT algorithm innermost loop. *rc[t,l]* refers to the RC[right,left] connection from the RCs' interconnection, and *self* to the output register of an RC that is connected to this interconnection network.

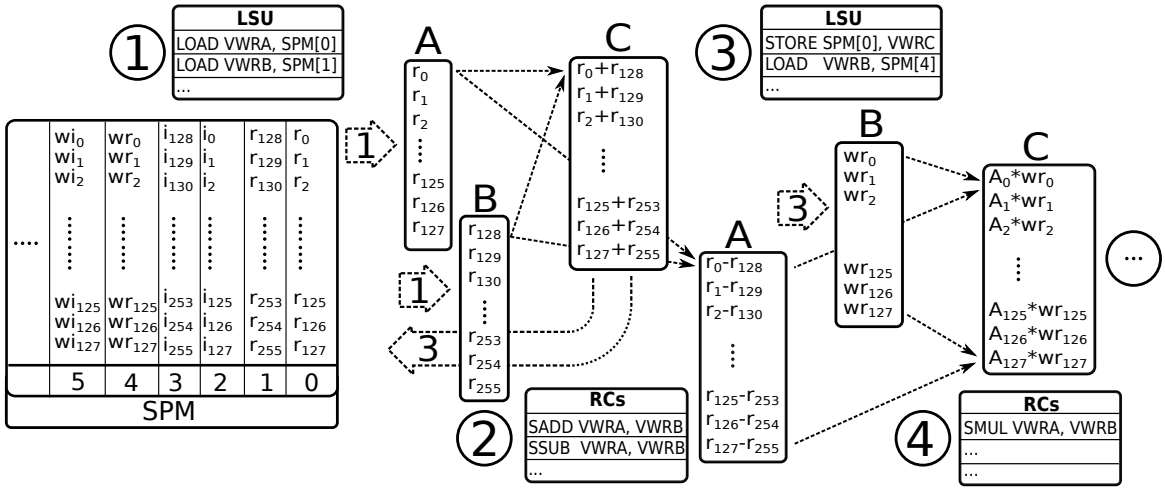


Figure 2.18: Graphical illustration of a few steps of the FFT radix-2 kernel mapping on the VWR2A.

loop counters, address generation), the eight RCs can entirely focus on the computation of the FFT butterfly elements. These are the reasons for the performance improvement of the VWR2A architecture compared to the CGRA architecture, in particular the use of specialized slots.

Figure 2.18 illustrates in a more readable way the first few instructions of Figure 2.17 for a 256 point complex FFT. The different steps are the following:

1. The LSU loads the real part (r_i) of the data from the SPM in VWRs A and B.

2. The RCs add the entries of VWRs A and B into the VWR C and subtract them into VWR A.

3. The LSU stores VWR C in the SPM and loads the real weights (wr_i) in VWR B.

4. The RCs multiply the entries of VWRs A and B into VWR C.

5. ...(kernel execution continues)

The shuffle unit is used between the stages to reorder the data. For example, in Figure 2.18, r_0 is added to r_{128}, but in the next stage r_0 needs to be added to r_{64}. At the end of the current stage, r_0 and r_{64} will be in the same VWR, which is not possible if we reuse the same code for all the stages. Therefore, the shuffle unit applies the "*VWR words interleaving*" shuffling to create the correct data layout for the next stage. Fig. 2.18 shows the execution for one column of RCs. The second column executes the same instructions but on the imaginary part (i_k) of the data. The output of the kernel is in bit-reversed order, and the shuffle unit is again used to reorder the data.

Table 2.2: Comparison of the radix-2 FFT algorithm mapping on the Cortex-M4 SoC (**CM4**) plain C version, the baseline CGRA (**CGRA**), and the VWR2A instance (**VWR2A**).

	CM4	**CGRA**	**VWR2A**
inner loop #instructions	73	7	16
inner loop #cycles (est.)	91	15	181
middle+inner loop #iterations	$\frac{FFT\_SIZE}{2}$	$\frac{FFT\_SIZE}{2}$	$\frac{FFT\_SIZE}{VWR\_SIZE*2}$
Total cycles[6] 1 outer loop	23 296	3840	362
Speed-up	1.0 ×	6.1 ×	64.4 ×

Table 2.2 shows the number of instructions and estimated cycles[4] per inner loop iteration for the three implementations. Because of the code structure on the VWR2A architecture, a fair comparison is only possible at the outer loop level. Table 2.2 shows how many times the middle and inner loops are repeated. For the CGRA and CM4 implementations, this value is fixed to the FFT size divided by two.[5] For the VWR2A architecture, this value is fixed to $FFT\_SIZE/VWR\_SIZE * 2$ because it processes a batch of data equal to the VWRs' size. The total number of cycles of one outer loop iteration of a 512 point complex value FFT is given for the three implementations. The absolute values are only estimations, but their ratios are aligned with the results of the following experiments section and perfectly explain the performance improvement of the CGRA and the VWR2A implementations compared to the CM4 one. The CGRA enables ILP over the RCs, and the VWR2A further improves it by integrating specialized slots. In addition, the VWR2A removes the bus bottleneck with its wide memory hierarchy based on VWRs, further improving performance. This analysis is based on the radix-2 FFT algorithm example.

2.7.2 Performance and energy consumption comparison

The performance of the three evaluated architectures is shown in Table 2.3 for three different FFT sizes and for complex and real sequences. For the CGRA and VWR2A implementations, the time to configure the platform is included (i.e., loading the configuration words from the context memory to the local program memories of all the units). For VWR2A, the data copy from the SoC SRAM to the VWR2A SPM (and back) is also taken into account. The index bit-reversal step (see Chapter 2.6.2) and the split operations (recovery operations after a real-valued FFT) are also executed by the CGRA and VWR2A.

[4]For simplicity, the number of cycles are estimated using 1.5 cycles for a load or store operation (as they usually take more than 1 cycle, but the request can be pipelined) and 1 cycle for any other operation.

[5]The implementation uses two loops, but their total cumulated size is fixed to $FFT\_SIZE/2$ as shown in Figure 2.14

[6]For a 512 points complex FFT.

Table 2.3: FFT kernel performance comparison for various sizes.

Cplx-valued	**CM4 (plain C)** cycles	**CM4 (CMSIS)** cycles	speed-up	**CGRA** cycles	speed-up	**VWR2A** cycles	speed-up
512	351 450	142 497	2.5 ×	48 556	7.2 ×	7211	48.7 ×
1024	742 872	279 576	2.7 ×	106 424	7.0 ×	14 610	50.8 ×
2048	1 624 455	670 594	2.4 ×	213 446	7.6 ×	29 908	54.3 ×
Real-valued							
512	196 024	102 148	1.9 ×	31 780	6.2 ×	3795	51.7 ×
1024	403 821	233 705	1.7 ×	66 759	6.0 ×	7259	55.6 ×
2048	878 053	459 784	1.9 ×	141 354	6.2 ×	14 528	60.4 ×

Compared to the CM4 plain C code, the CMSIS-DSP library gives an average speed-up of $2.2\times$, the CGRA an average speed-up of $6.7\times$, and the VWR2A an average speed-up of $53.6\times$. The CGRA and VWR2A use both 8 RCs, but the latter is $8.1\times$ faster on average. The wide memory hierarchy explains part of this performance improvement, and the rest is due to the specialized slots, which demonstrates their relevance and crucial role in the performance of VWR2A. These speed-ups also translate into energy savings as shown in Table 2.4. As the CM4 power consumption is almost constant, using the CMSIS-DSP library reduces the energy by $46.6\,\%$ on average. The energy savings for the CGRA and the VWR2A instance are limited by their higher power consumption compared to the CM4, but they are still significant: $67.3\,\%$ and $86.5\,\%$, respectively. The energy-delay product (EDP) combines both metrics (latency and energy) and provides a better overall comparison factor between the different implementations. Compared to the baseline CM4 plain C code implementation, the CM4+CMSIS, the CGRA, and the VWR2A instance have and EDP improvement of $4.4\times$, $20.8\times$, and $397.6\times$, respectively. Compared to the optimal implementation using the CMSIS-DSP library for the CM4, the CGRA, and the VWR2A instance have and EDP improvement of $5.1\times$, and $104.8\times$, respectively.

Mapping. The first observation is that all three implementations (CM4+CMSIS, CGRA, VWR2A) have optimized code, while the CM4 plain C is a straightforward implementation of the radix-2 algorithm, showing the importance of mapping to reach optimal execution. The flexibility-performance tradeoff discussed in Chapter 1 and 2.1 and shown in Figure 1.4 is validated for the CGRA and VWR2A architectures compared to the CM4: they are faster and more energy efficient, but they are not as flexible or programmable as the CM4 (e.g., they support a smaller ISA and only 32-bit integer data). The comparison of the CGRA and VWR2A instances is also interesting to investigate. They have a similar architecture, as the VWR2A uses a CGRA design for its compute core fabric, and can, in theory, execute the same kernels, thus can be considered equivalently programmable, but the VWR2A is

Table 2.4: FFT kernel energy consumption comparison for various sizes.

	CM4 (plain C)	**CM4 (CMSIS)**		**CGRA**		**VWR2A**	
Cplx-valued	Energy (µJ)	Energy (µJ)	Savings	Energy (µJ)	Savings	Energy (µJ)	Savings
512	2.93	1.31	−55.3 %	0.88	−69.9 %	0.41	−86.1 %
1024	6.43	2.55	−60.4 %	1.93	−70.0 %	0.88	−86.3 %
2048	13.95	6.00	−57.0 %	4.29	−69.3 %	1.91	−86.3 %
Real-valued							
512	1.69	1.13	−33.2 %	0.61	−64.0 %	0.23	−86.4 %
1024	3.61	2.44	−32.3 %	1.29	−64.3 %	0.49	−86.4 %
2048	8.27	4.83	−41.6 %	2.77	−66.5 %	1.04	−87.4 %

faster and more energy efficient. Therefore, one crucial contribution of VWR2A is that it advances the Pareto front of the flexibility-performance tradeoff.

VWR2A performance. The main features of VWR2A are evaluated and compared to the baseline CGRA architecture to explain the higher efficiency and to provide insights that could be transposed to other architectures. One important reason for the improved performance of VWR2A is the use of a dedicated memory tailored to the need of the accelerator, as recommended in [57, 66]. For VWR2A, the SPM plays this role by providing a dedicated data memory close to the RCs (i.e., short wires) and a wide interface to enable multiple words to be accessed in parallel. In comparison, the CGRA uses the SoC limited bus bandwidth (32-bit) and main memory that is far (i.e., long wires), which increases the latency and energy consumption. Figure 2.19 shows the power consumption reduction in the SoC bus and main memory when VWR2A is used compared to the CGRA. As expected, this figure also shows that the VWR2A consumes more power because it contains a SPM. In total, the wide memory hierarchy —the SPM and the VWRs (with the decoding multiplexers)— is responsible for more than $90\,\%$ of the power consumption increase compared to the CGRA. Although it seems part of the power consumption has only moved from the SoC main memory and bus to the VWR2A instance, this organization is more energy efficient for two reasons: 1) the wide memory hierarchy is $1.7\times$ more energy efficient on average compared to using the SoC main memory and bus,[7] and 2) VWR2A is much faster thanks to this memory hierarchy. The VWR2A instance speed improvement compared to the CGRA is due to the wide memory hierarchy and the specialized slots that increase the ILP. The impact of the specialized slots can be estimated by considering the RCs would have to execute their instructions. For the FFT kernels evaluated in this

[7]This is evaluated in post-synthesis simulation. In a real chip or post-layout simulation, the long bus wires should penalize the CGRA implementation even more.

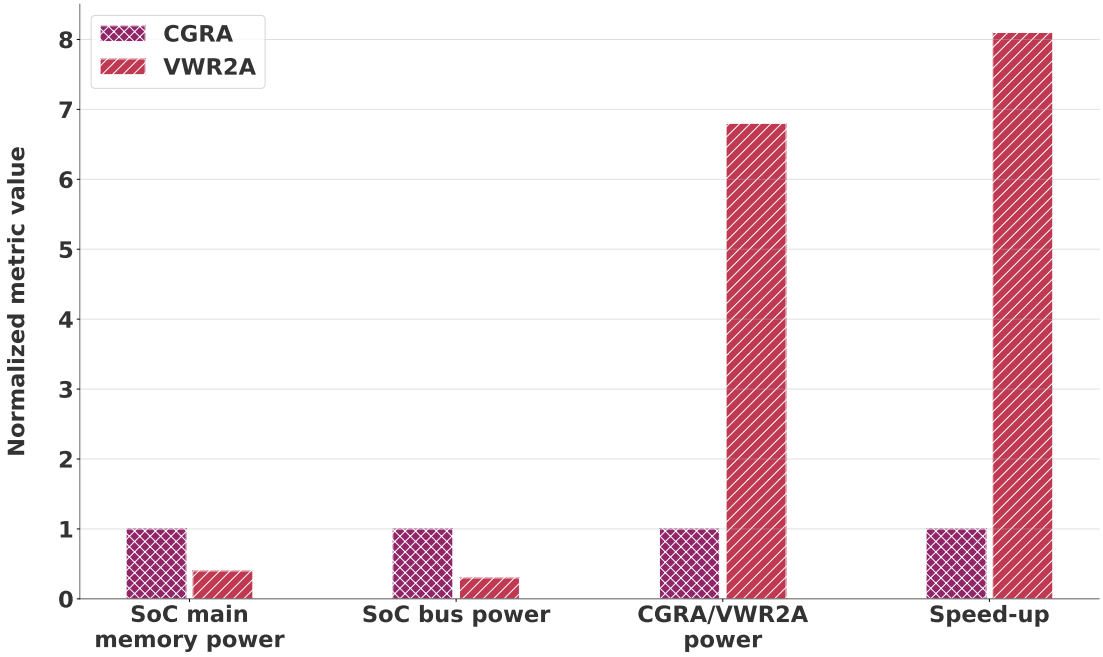


Figure 2.19: Normalized comparison of VWR2A main features on power and performance compared to the CGRA.

chapter, this would increase the execution time by approximately a factor of $2.1\times$. This shows the massive benefit of the specialized slots as they only account for $5.1\,\%$ of the VWR2A instance power consumption on average. Without them, it would be less than $20\,\%$ more energy efficient than the CGRA, compared to the actual $58.3\,\%$. Considering the ILP and the wide memory hierarchy are the main contributors to the speedup compared to the CGRA (as both architectures are using the same number of RCs), it means the wide memory hierarchy contributes to a factor $3.9\times$ improvement in speed. When energy and execution time are combined, the VWR2A instance has an EDP improvement of $19.8\times$ compared to the CGRA. If the area is considered in an energy-delay-area[8] product, the VWR2A is still $3.5\times$ better compared to the CGRA.

Comparing performance with state-of-the-art architectures from the literature is often not trivial because the experimental setup must be standardized for a fair comparison, especially for energy numbers. The technology and the frequency used to synthesize a design, the considered benchmarks, or the maturity of the architecture (e.g., complete SoC or a simplified platform) are, for example, parameters that considerably modify the performance and energy consumption of a design. The Ultra-Low Power Samsung Reconfigurable Processor (ULP-SRP) [84], a recent instantiation of the ADRES template [85]

[8]Post-synthesis area estimation.

that uses the same technology node (i.e., TSMC $40\,\mathrm{nm}$ LP) is a good comparison point. Compared to it, VWR2A exhibits significant performance and energy gains. The authors reported an execution time of $839.1\,\mu\mathrm{s}$ at $100\,\mathrm{MHz}$ and an energy consumption of $19.9\,\mu\mathrm{J}$ for a 256-Point FFT,[9] while VWR2A executes that same kernel in $35.6\,\mu\mathrm{s}$ at $80\,\mathrm{MHz}$ and consumes $0.3\,\mu\mathrm{J}$. These numbers correspond to a factor $23\times$ improvement in performance (without considering the frequency difference) and a factor of $66\times$ in terms of energy. It is important to note that post-layout simulation has been done for the ULP-SRP, while I ran post-synthesis simulation, which can explain part of the significant difference in energy.

2.8 Summary and conclusions

This chapter presented VWR2A: a low-power and high-performance template to design DSIPs for the biomedical domain. It is proposed as a better alternative, at the application level, in terms of energy and costs than low-power ASICs or GPPs. VWR2A is an architecture template combining the high computational density of the CGRA template and the low energy of a wide memory hierarchy based on VWRs and a wide SPM. To further increase the ILP, specialized slots, namely an LSU, MXCU, and LCU, are introduced for several basic blocks common to many algorithms. Each specialized slot is assigned to a set of tasks and can be optimized for it. The LSU handles the data movement between the background memory (i.e. the SPM) and the foreground memories (i.e., the VWRs). The MXCU generates the addresses for the RCs to access the VWRs, and the LCU controls the execution flow of the kernel (e.g., conditions, jumps) through the program counter. The RCs are optimized for the computations directly related to data as all the other tasks are executed by the specialized slots.

VWR2A is a template that can be used as a basis to implement a DSIP for the biomedical domain. The tunable parameters of the template, such as the number of RCs, their configuration (i.e., number of columns), or the specialized slots' datapath width, have been presented and illustrated with concrete examples based on the two instances of Chapters 3 and 4. The restriction imposed by the VWRs and the wide memory hierarchy has been discussed and illustrated with some examples to help the design of new instances.

One instance of the template (based on the one of Chapter 3) has been compared against two state-of-the-art low-power, high-performance architectures: an ARM Cortex-M4 GPP and a baseline CGRA. The experiment results show the better latency and energy efficiency of an instance based on VWR2A and the central role of the specialized slots. Compared to the ARM Cortex-M4 and the baseline CGRA, the VWR2A instance showed

[9]The authors do not specify if they implement a complex-valued or a real-valued FFT. A 256-Point complex-valued FFT is considered, corresponding to the worst case for VWR2A.

an EDP improvement of $104.8\times$ and $19.8\times$, respectively. The results also demonstrate that judicious architectural optimization can advance the Pareto front of the flexibility-performance tradeoff and further improve the energy efficiency of embedded systems. This brings us closer to the long-lasting operation of low-power edge devices and to the full potential of edge computing and IoT systems, particularly for the biomedical domain.

3 Acceleration of Data-intensive Applications for Embedded Systems

3.1 Introduction

The general idea of edge computing (e.g., Internet of things (IoT), wearables, embedded systems) could revolutionize our world in many domains (e.g., smart cities, self-driving vehicles, industry). To fully embrace this concept, we still need to obtain higher energy efficiency for today's embedded devices. The workload paradigm shift from the cloud to the edge reduces the communication energy and the reaction latency and increases data privacy. On the other hand, this increases the computational load on edge devices, making it challenging to design high-performance and energy-efficient embedded systems.

The biomedical domain is one example that could benefit from this revolution. Wearable biosignal-processing embedded systems are poised to become an essential tool toward the goals of personalized medicine and preventive healthcare. However, to achieve these goals, we still need to obtain higher energy efficiencies that enable long-lasting operation between battery recharges. The inclusion of co-processors or hardware accelerators in the computing platform has become standard in recent years. These accelerators can execute repetitive operations that account for a significant amount of the processing time (i.e., computational kernels) more efficiently than a general-purpose processor (GPP).

The development of co-processors follows two main trends: custom accelerators, or application-specific integrated circuits (ASICs), and flexible (programmable) cores without any fully custom components, usually considered as domain-specific instruction-set processor (DSIP) [86]. Fixed-function accelerators are often the most efficient way of implementing a particular functionality for a given set of constraints. They are generally not programmable and are thus focused on a single task or a small family of related tasks. One example of a custom accelerator is an fast Fourier transform (FFT) accelerator. The FFT is a crucial transformation used in the biomedical domain, and platforms targeting this

domain have proposed to use an ASIC for its computation [32, 87] to improve its execution compared to GPPs. However, as it has been proposed in the literature, it is possible to build programmable cores which have instruction sets that are tailored specifically for algorithms from a given application domain (i.e., DSIPs) without becoming GPPs [86]. Traditional knowledge says that these flexible cores are less efficient than fixed-function accelerators and that the latter should be preferred to improve the energy efficiency of the systems. This chapter demonstrates that the opposite is valid at the application level by using the VWR2A template proposed in Chapter 2 to build a DSIP instance that is more performant and energy efficient than ASICs. The reason is that a programmable accelerator covers larger portions of the source code which translates to higher savings at the application level (although less efficient than an ASIC at the kernel level).

3.1.1 Low-power architectures for biomedical applications and their limitations

Existing low-power commercial platforms for biomedical applications are limited by their processing capability and their power/energy consumption. The reason is that they mostly rely on GPP(s) as the principal processing element(s), translating to poor performance, both in latency and energy, as demonstrated in Chapter 2. The adoption of more customized circuits (i.e., ASICs) inside commercial platforms is limited because of three main reasons:

1. Their inflexibility limits their code coverage and adaptation to potential application updates.
2. Their optimized circuit design translates into high development costs, further increasing their limited reusability.
3. Having a set of ASICs to cover various applications (i.e., multiple kernels) is even more expensive, and defining this set is usually challenging.

Therefore, to keep high flexibility, commercial platforms only integrate accelerators or co-processors that cover almost any application, such as a graphics processing unit (GPU), digital signal processing extensions, or a direct memory access (DMA), limiting the overall performance gain.

The considerable computation requirement of the latest machine learning algorithms, in particular Deep Neural Networks (deep neural networks (DNNs)), has forced platforms to integrate custom accelerators (i.e., ASICs). The reason is that they are currently the only solution to provide good performance at reasonable power consumption. One example is

the MAX78000 system-on-chip (SoC) [21] that integrates a convolutional neural network (CNN) accelerator, the Greenwave's GAP8 chip and its HardWare Convolution Engine (HWCE) [22, 23] or the layer-based CNN accelerator of the Kendryte K210 [24]. While these accelerators increase the overall performance, they might become less efficient if the CNN generic model changes or even ineffective if the model becomes obsolete (e.g., replaced by a new and more accurate type of model).

In the biomedical domain, no such optimized platforms (i.e., including ASICs) have been commercialized, although several solutions have been proposed in the literature. One example is the SoC presented by the authors of [32]. It includes many hardware accelerators, particularly an FFT accelerator and a matrix processor. These two ASICs are significantly more energy efficient than the Cortex-M4 GPP at the kernel level (i.e., executing an FFT and matrix computations, respectively). However, their impact is limited at the application level —as described by Amdahl's law (see Equation 1.1)— or even inexistent in the worst case (e.g., if there is no FFT or matrix computation in the application). This analysis at the application level is usually not done, and many papers limit their experiments to kernels, concluding ASICs are the optimal solution.

3.1.2 Contributions and outline of the chapter

In this chapter, I propose to use the Very-Wide-Register Reconfigurable-Array (VWR2A) template presented in Chapter 2 to build a DSIP targeting the data-intensive applications in the biomedical domain. This instance of the VWR2A template shows how to move in the flexibility-performance tradeoff and build flexible domain-specific cores that close or narrow the efficiency gap in terms of energy and performance with respect to custom ones for individual computation kernels. Moreover, as these cores still have an instruction set covering a broad domain, they can be used in more parts of the application. As a consequence, they achieve larger improvements when the complete application —rather than individual kernels— is taken into account. In particular, the key contributions of this chapter are:

1. The design of a DSIP instance based on the VWR2A template proposed in Chapter 2 targeting the data-intensive biomedical applications domain.

2. An analysis of the target domain and its translation into design parameter choices for the instance of the VWR2A template.

3. An energy and performance comparison at the kernel level of VWR2A with two ASICs: an FFT accelerator and a matrix processor.

4. An energy and performance comparison at the application level with a low-power GPP and an SoC optimized for biomedical applications.

The rest of this paper is organized as follows. First, the typical data-intensive biomedical application and the existing computing architecture targeting this domain are discussed in Chapter 3.2. Then, VWR2A instance is presented and analyzed in Chapter 3.3. In Chapter 3.4, I describe the experimental setup used to evaluate our proposals, whereas in Chapter 3.5, the obtained results are analyzed. Finally, in Section 3.6, I draw the main conclusions of my study.

3.2 Related work

As this chapter focuses on the data-intensive biomedical applications domain, the typical processing steps of such applications are discussed here. Figure 3.1 shows the different steps of a biomedical application: signal filtering and enhancement, delineation, feature extraction, and inference. This thesis does not cover the signal acquisition phase as it implies other kinds of optimizations (e.g., power-gating, low-power analog front-end) that are not covered by the proposed DSIP template of Chapter 2. However, these optimizations can be combined to the ones discussed here and can be applied on top of the DSIP template.

3.2.1 Biomedical applications

Filtering and enhancement

Digital filtering removes or enhances specific frequencies from the acquired biosignals. Depending on the signal nature, there exist different filtering techniques [88]. For example, a raw electrocardiogram (ECG) signal contains an undesirable low-frequency harmonic (called baseline wander) and high-frequency noises. A typical technique is to apply a band-pass filter using a finite impulse response (FIR) filter. A FIR filter order corresponds to the number of weights and previous samples used to compute the filter's output. This algorithm is very data intensive as it loads many input data and weights (corresponding to the filter's order), multiplies them (one by another), and takes their mean value. The access pattern is usually very regular, as all the data are contiguous. Other methods exist, such as morphological filtering (MF), but this code is more control-intensive and discussed in Chapter 4. Signal enhancement techniques can be applied on top of filtering to improve the input signals further. For example, the root-mean-square (RMS) combines multiple leads, or the signal derivative can be extracted.

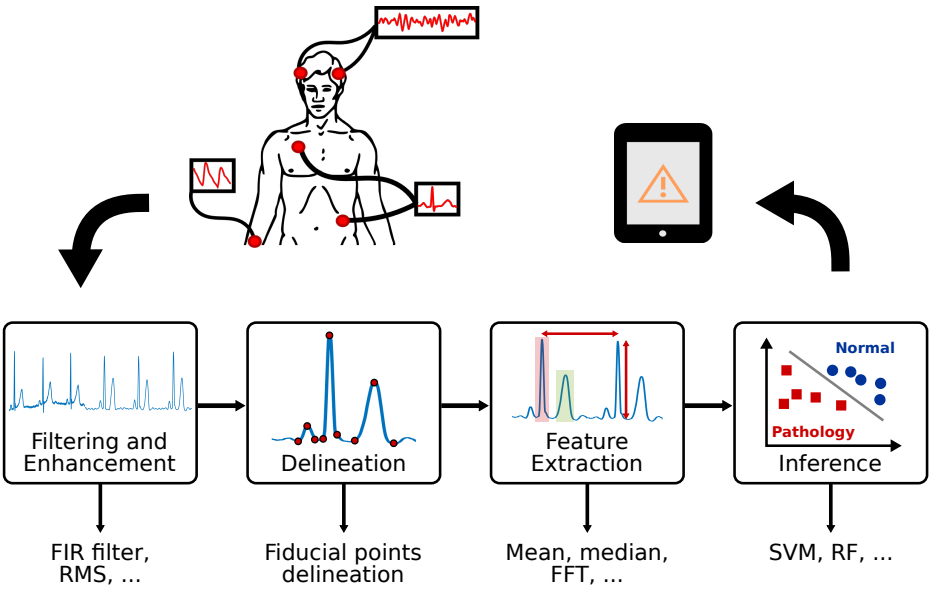


Figure 3.1: Typical steps of a biomedical application and examples of algorithms used for each of these steps.

Delineation

Biosignal delineation is a crucial step in many biomedical applications as it extracts the fiducial points of a signal (i.e., the signal's main characteristics). For example, Figure 3.2 shows an ECG and a respiration signal with their respective fiducial points. This step is highly control-intensive as the points are detected through multiple nested if-else conditions. Therefore, this phase is not discussed here but is evaluated in Chapter 4.

Feature extraction

Based on the fiducial points, the waveform or its frequency, or any other characteristics, the relevant features of a biosignal can be computed [88]. Figure 3.2 shows some features that can be extracted from the delineated points of an ECG and a respiration signal. Other examples are: mean or median point-to-point value, mean frequency, bandpower (e.g., 0.32 to 0.4 Hz), and average power. These features use data-intensive kernels, but their workload varies significantly. For example, computing a mean value is an addition of values (usually in contiguous memory locations) followed by a division and can be considered a data-intensive kernel. However, the computational workload is almost negligible compared to calculating an FFT, for example.

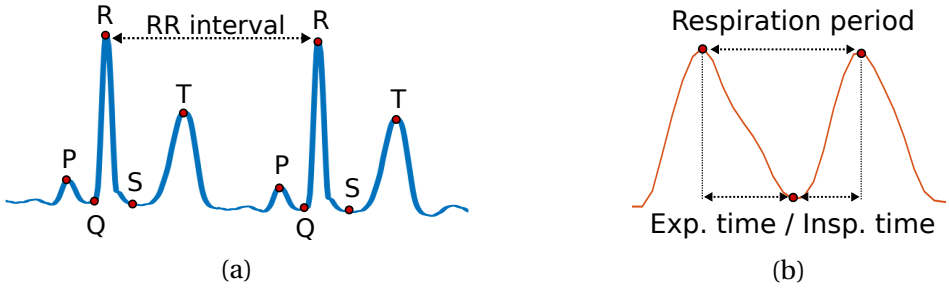


Figure 3.2: (a) Electrocardiogram (ECG) biosignal fiducial points (PQRST) and feature example (RR interval). (b) Respiration (RSP) biosignal fiducial points and feature examples (respiration period, expiration time, inspiration time).

Inference

The inference step allows an automatic diagnosis of a medical condition and can trigger an appropriate response (e.g., medication, alert, or deeper analysis). In the biomedical domain, the selection of features can often be based on practitioners' knowledge or existing studies, thanks to the tremendous past and ongoing research on the human body. This results in high-quality features used for the inference phase (i.e., classification or regression) and allows low- to middle-complexity machine learning algorithms (e.g., support-vector machine (SVM), Random Forest, or Fuzzy classifier) to obtain high classification precision [81, 89–91]. DNNs have been proposed in some research [92, 93], mainly for ECG applications, but they do not always provide higher accuracy than the standard approaches mentioned above [94]. In addition, the high computational workload of current models is usually unsuitable for low-power embedded systems. More importantly, their current nonexistent or low interpretability nature discards their use for real-life medical application certification. Therefore, DNNs are currently not considered a typical workload in the biomedical domain for embedded systems, but the aforementioned problems are being investigated, and future work could change the current situation. This would require the re-evaluation of the proposed template and probably a redefinition of it to consider such workloads.

3.2.2 Low-power architecture for biomedical embedded systems

Only a few low-power processing architectures have been proposed for biomedical applications in the literature. In [59], the authors proposed TamaRISC, a custom-designed RISC architecture supporting a subset of the PIC24H/F instruction set architecture (ISA) with a Harvard memory model. This ISA subset, the datapath width (16 bits), and the instruction width (24 bits) are all optimized for the biomedical domain, hence making it a

DSIP architecture. As this ISA has proven its efficiency for its domain, the VWR2A instance proposed in this chapter supports a very similar set of instructions (XOR, AND, OR, shift operations, ...) as TamaRISC. One main difference is related to the memory hierarchy: while TamaRISC implements an x-bar interconnect for accessing data in its multi-core version, VWR2A uses a low-energy wide memory hierarchy (see Chapter 2).

Duch et al. [33] proposed a multi-core reconfigurable architecture: a multi-core system (based on TamaRISC) augmented with a coarse-grained reconfigurable array (CGRA) hardware accelerator to improve further energy efficiency. The CGRA accelerator executes computationally intensive kernels. At the kernel level, it enables an average energy saving of 73.3% compared to the multi-core architecture. The reason is that the CGRA is faster thanks to its 16 reconfigurable cells (RCs), while the multi-core system has eight processors. Moreover, the design of the RCs is simpler than the processors translating to higher energy efficiency. However, because of its simple architecture, this CGRA can only execute basic kernels (e.g., no nested loops or complex if-else conditions). This limits its code coverage at the application level, with a maximum of $18.6\,\%$ energy savings reported for one application and around $10\,\%$ for two other applications. The DSIP template proposed in Chapter 2 and the instance of this chapter try to cover a larger fraction of the code of a domain to have similar energy savings and speed-up both at the kernel and application level.

To address this issue of limited code coverage, Mei et al. proposed the ADRES framework [85]: an architectural template composed of a very-long instruction word (VLIW) processor and a CGRA. The VLIW allows efficient execution of control-intensive code, and the CGRA still accelerates the computation-intensive kernels. Although this chapter focuses on data-intensive kernels and applications, the goal of the proposed DSIP template is to cover a domain. Therefore, compared to the ADRES template, the VWR2A template regroups all the processing elements in a single block. This allows a large portion of an application to be executed on an instance of the VWR2A template, removing the need for custom hardware accelerators or advanced general-purpose processors like the VLIW of the ADRES platform, thus reducing the overall system energy consumption. In [84], the authors proposed ULP-SRP, an instantiation of the ADRES template for the biomedical domain. However, as demonstrated in Chapter 2.7, VWR2A is significantly faster and more energy efficient thanks to its low-energy wide memory hierarchy and single flexible processing system.

More recently, Song et al. proposed a single-chip SoC optimized for biomedical applications known as MUSEIC [32]. Compared to all the architectures presented above, MUSEIC has a single-core ARM Cortex-M4F GPP and a set of ASICs (e.g., FFT accelerator, matrix processor, DMA) to improve its performance in latency and energy. While the goals —high-performance and low-power/energy— are similar to the DSIP template advocated

in this thesis, the methods are diametrically opposed: MUSEIC tries to cover the application domain with multiple ASICs, while I propose to build DSIP architectures that cover most of the code related to a domain. The disadvantages of ASICs over DSIPs have been discussed in Chapter 2.1.1, and this chapter provides a quantitive comparison to fully demonstrate the advantages of DSIPs.

3.3 VWR2A instance for data-intensive biomedical applications

The detailed implementation of the VWR2A instance used in this chapter is described here and its main features are shown in Figure 3.3. It illustrates how the template is used and optimized within the biomedical domain. The VWR2A template is meant to cover the complete biomedical domain, but an instance is optimized for particular constraints. These can be the specific biosignal being processed, the required throughput, the power envelope, or the area (i.e., cost). Any instance based on the VWR2A template, which is optimized for the biomedical domain (e.g., CGRA-like processing system, wide memory hierarchy), can potentially execute any code of this domain. However, there exist many applications and use cases. Therefore, depending on the computational demand, the VWR2A template flexibility allows the design of instances targeting higher or lower workloads.

3.3.1 Reconfigurable array

The reconfigurable array is the core processing system of the VWR2A template. Defining the number of RCs and columns, their interconnection, and their ISA is a domain-dependent process and the key to overall high performance and energy efficiency. Some parameters have been defined based on the literature best practices for the template instance presented in this chapter. For example, [84] and [33], two architectures using the CGRA template and targeting the biomedical domain, have 9 and 16 RCs, respectively. Eight RCs are used in the proposed instance, a value close to these numbers. This choice is also related to the very-wide register (VWR) width, as discussed in Chapter 2.3, and further developed in the next section. The RCs are divided into two columns of 4 RCs each. This enables the execution of two independent kernels (one on each column). The RCs' interconnection scheme of the closest neighbors is also based on [33], as they have shown an excellent performance-energy tradeoff. ULP-SRP [84] also proposes to connect the closest diagonal neighbors, but this has not been implemented as the current design already offers high performance.

The RCs' ISA is limited to a few operations: logical bitwise operations (AND, OR, XOR), selection operations, signed addition, subtraction, multiplication, and logic and arithmetic

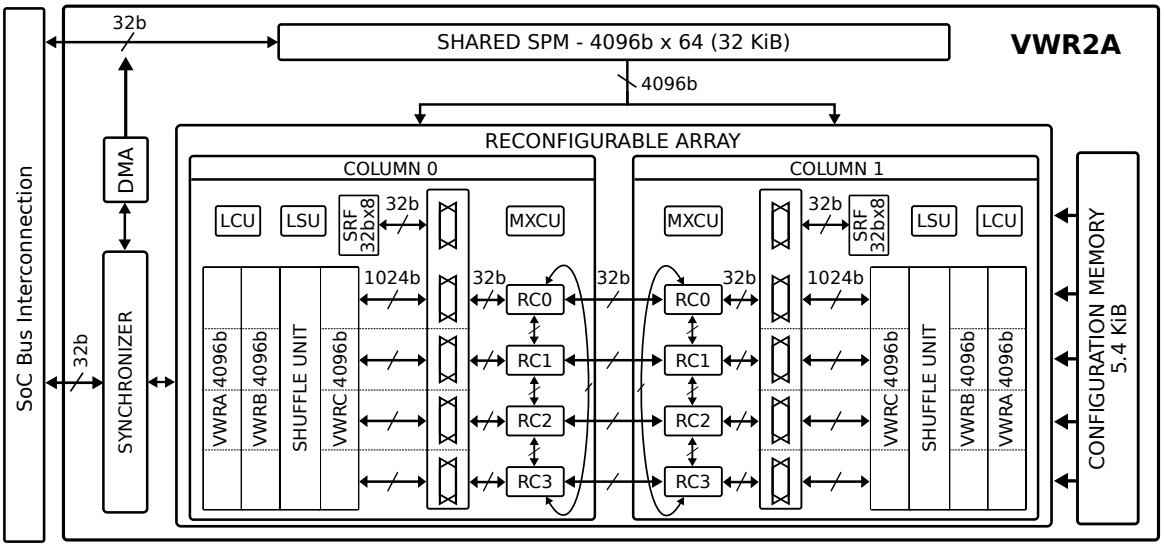


Figure 3.3: VWR2A instance for data-intensive biomedical applications.

bitshift operations. All operations happen in one clock cycle. The selection operations pick one of the two arithmetic logic unit (ALU) input operands as output based on the value of the sign flag or the zero flag. Similarly to the RCs' data interconnection, the sign and zero flags are also shared between the RCs. While these instructions are more critical for control-intensive code (see Chapter 4), they help the execution of division or square-root operations in software (as they are not supported in hardware), used in some data-intensive kernels. The multiplier has two working modes: a standard mode, where the lowest 32 bits are kept, and a fixed-point mode, where the lower 16 bits are discarded, and the next 32 bits are kept (i.e., q17.15 format). For example, this enables fast multiplication of decimal numbers used in an FFT. The datapath width is 32 bits to stay generic and compatible with standard processors, and each RC has a two-entry local register file (RF) for data.

Table 3.1 shows the instruction format of the RCs. To accommodate the execution of large kernels, such as an FFT, the local program memory of the RCs (and the specialized slots) has a depth of 64 instructions. The reconfiguration of the RCs (and the specialized slots) happens in parallel by loading their respective instructions from the context memory. Only one column can be configured at a time. In the case of two columns running (in synchronization or on different kernels), their configuration happens sequentially in case they both receive a request simultaneously. The number of cycles depends on the size of the kernel that is loaded (1 cycle per instruction) into the reconfigurable array with a maximum of 64 cycles (corresponding to the local program memory depth), or 128, when one, respectively two, columns are used, as all the RCs and specialized slots are loaded in parallel.

Table 3.1: RC configuration word format and size (see Figure 2.3 for the RC internal architecture).

Field	muxAsel	muxBsel	muxFsel	aluOp	rfWe	rfSel
Bits	16:13	12:9	8:5	4:2	1	0
Configuration word width						17 bits

Table 3.2: Typical biosignals of biomedical application with their number of channels (or leads) and sensor frequency range.

Biosignal	Description	#Channels	Sensor Sampling Frequency
ECG	Heart electrical activity	1–12	125-1000 Hz
EEG	Brain electrical activity	2–32	250-2000 Hz
EMG	Muscular electrical activity	1–4	100-300 Hz
GSR	Skin conductance	1–2	5-500 Hz
SpO2	Blood oxygen	1	100-1000 Hz

3.3.2 Low-energy memory organization

An important design parameter of the DSIP template is the scratchpad memory (SPM) and VWRs width. This number is estimated on an analysis of the target domain and fine-tuned experimentally. In the case of biosignals, the acquisition frequency range goes from a few Hz to approximately 1 kHz, as shown in Table 3.2, and data are processed in windows of a few seconds to tens of seconds. This creates a variety of scenarios ranging from tens of samples up to thousands being processed at once. The present instantiation, which targets this domain, has a 4096 bitwidth for the VWRs and the SPM (512 B), allowing 128 words of 32 bits to be transferred in a single cycle. This bitwidth (i.e., 4096) has been found experimentally to achieve a good tradeoff between performance and power consumption. It is highly related to the number of data that has to be processed. In this case, the kernels are extracted from an application that uses a 5 Hertz respiration sensor and a 1-minute and 40 seconds sampling window, which generates 512 samples. Consequently, four loads (from the SPM to the VWRs) are needed for all the data to be accessed once, which adds negligible delay as a load takes one cycle compared to the many cycles required to compute on these data. This parameter also depends on the number of RCs processing the data of the VWRs. For example, from the energy point of view, it is inefficient to have a very high bitwidth with a very small datapath (i.e., the number of RCs). It would result in data not being accessed very often and only increasing the leakage. In this instance of the VWR2A template, the four RCs can access 32 words (one-fourth of a VWR), and the data are consumed relatively fast, but loading new data from the SPM has a latency cost of one cycle.

One current limitation to the maximum bitwidth is the use of standard cell libraries and memory macros supplied by the technology provider to build the wide interface. A custom design for these memories will undoubtedly reduce power consumption and enable wider VWRs to be designed. This limitation and potential solutions are discussed in Chapter 5.2.1. The number of VWRs per column is fixed to 3, called VWR A, B, and C, as shown in Figure 3.3. This covers all the RC operations, which operate on a maximum of 3 variables (two input operands and one result).

The SPM is single-ported but shared between columns and the DMA. As this VWR2A instance has only two columns, a simple arbitration mechanism is used: the DMA has the highest priority, followed by column 0 and then column 1. Although such a design favors column 0, it has not created unbalanced access in any experiments. The reason is that both columns are synchronized (through their program counters (PCs)) when executing a kernel in coordination, forcing both columns to be granted access to the SPM if they request it simultaneously. In case both columns run two kernels in parallel, column 0 has a higher priority than column 1, but accesses to the SPM are rare, and column 1 will be granted access relatively quickly. This also avoids stalling the execution of the two columns when they require multiple subsequent load or store operations. For example, if column 0 requires three loads from the SPM before resuming its execution, it will be granted the accesses in a row while column 1 is stalled (if it requests access to the SPM). Then column 0 starts processing the data, and column 1 can access the SPM multiple times as column 0 does not have to access it for many cycles.

Each column of the reconfigurable array also has a shared scalar register file (SRF) of 8 32-bit entries. The SRF is used for scalar variables (e.g., loop size, address mask, kernel constant) and sometimes to share data between the different units (i.e., the RCs and the specialized slots). To limit its power consumption, the SRF is single ported, and its output is broadcasted to all the units. All the specialized slots can write to it (one at a time), and because the RCs are principally reading values from it, only the top RC of a column (i.e., RC0 in Figure 3.3) can write to it.

Finally, a shuffle unit is integrated into the VWRs. This unit is essential to overcome the single-port limitation of the VWRs by enabling fast and energy-efficient data reordering. The inputs of the shuffle unit are always VWR A and B, and the output is stored in VWR C to reduce the hardware control overhead. The load-and-store unit (LSU) controls the shuffle unit, and the supported shuffling operations are listed in the LSU description paragraph.

3.3.3 Specialized slots

Optimizing the specialized slots for the targeted domain is important, but their respective tasks are more generic and less dependent on the workload than for the RCs. Moreover, some parameters are fixed by other design choices (see below). For data-intensive kernels, the multiplexer-control unit (MXCU) and the LSU are particularly important to achieve good performance.

MXCU: MultipleXer-Control Unit

Data-intensive kernels are characterized by many data accesses, therefore requiring many addresses. While many data access patterns are regular for data-intensive kernels, generating these addresses is still challenging. A fast address generation is essential as the number of instructions of the innermost loop mapped on the architecture is critical for high performance.

The MXCU has an instruction width of 27 bits (see Table 3.3), and a RF of eight 5-bit entries. Its first register (r0) controls the VWRs slice entry passed to the RCs datapath (i.e., the word address of the VWRs slice). The configuration words do not have an immediate field, but the ALU's input multiplexers have entries hardcoded to 0, 1, and 2. It also has two values hardcoded to the middle address and last address of a VWR slice (i.e., 32 words), respectively, 15 and 31. These values are enough to create most of the access patterns. The ALU can execute signed addition and subtraction, logical bitwise operations (AND, OR, XOR), and left/right logical bitshift. The datapath width of 5 bits is fixed by the address range of the VWR slice: $2^5 = 32$ (i.e., the VWRs word width, 128, divided by four slices, one per RC). As shown in Figure 3.4, the MXCU registers 5 to 7 ($r5$ to $r7$) are used to mask the VWR slice address (held in $r0$) for the three VWRs (respectively, VWR A to C) to provide more flexibility in the data access pattern. For example, setting $r7$ to zero, as in Figure 3.4, forces the RCs to access only the first value of their slice for VWR C, regardless of the value of $r0$. This allows the RCs to access different addresses of the three VWRs while only one value is used as the base address. Without this masking feature, accessing different addresses would require three values to be generated (one per VWR), limiting the performance of many kernels.

As proposed in Chapter 2.3, the MXCU can control which RCs are writing their results and to which VWR (vwr_we and vwr_sel fields in Table 3.3) to further improve the flexibility, as illustrated in Figure 3.4. Finally, the MXCU controls the read and write single-port access to the SRF, through the srf_we and srf_sel fields (the same address is used for reading and writing).

Table 3.3: MXCU configuration word format and size (see Figure 2.8 for the MXCU internal architecture).

Field	muxAsel	muxBsel	aluOp	rfWe	rfSel
Bits	26:23	22:19	18:16	15	14:12
Field	srfWe	srfSel	srfWmux	vwrSel	vwrWe
Bits	11	10:9	8:6	5:4	3:0
Configuration word width					27 bits

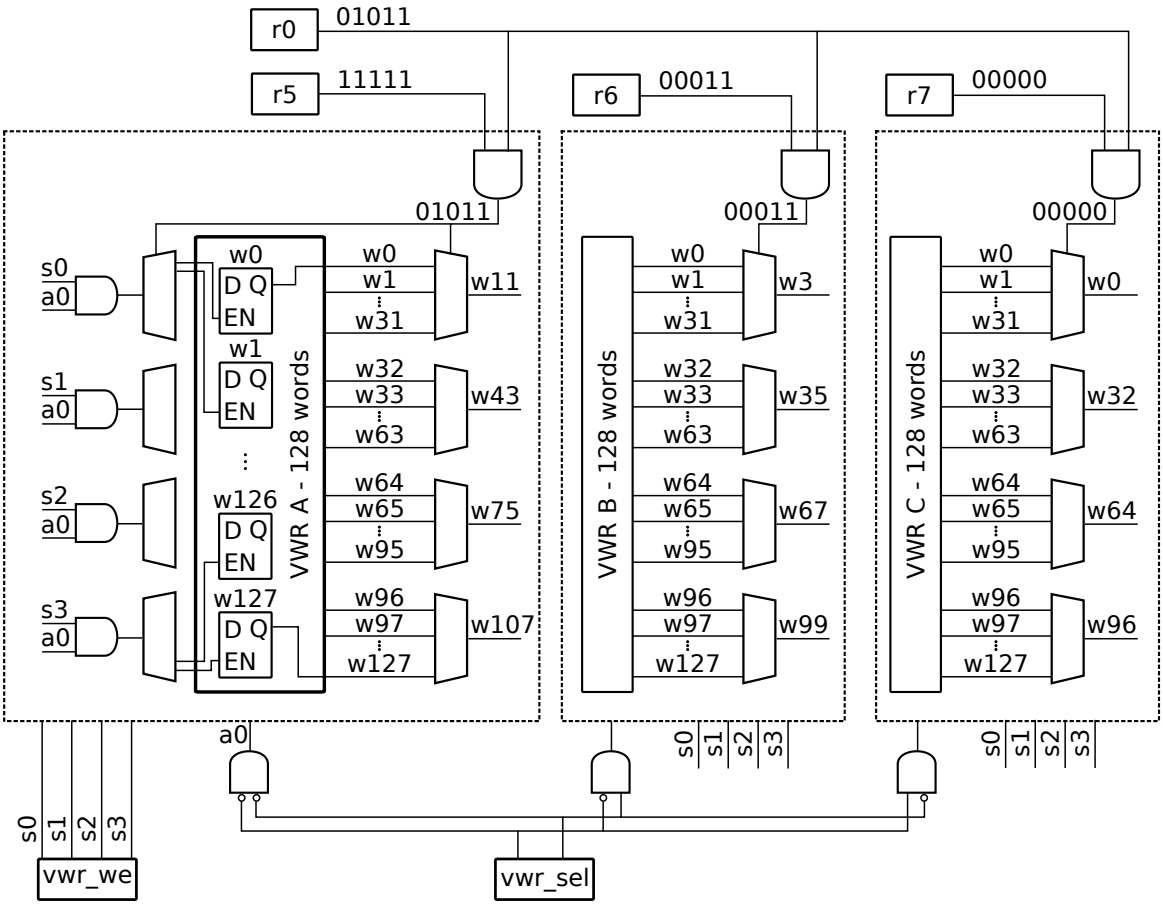


Figure 3.4: MXCU $r0$ (VWRs' slice address) and the mask registers ($r5$ to $r7$) to allow different words to be access in different VWRs. The write back to the VWRs is controlled by the vwr_sel (VWR A, B, or C) and vwr_we (one write enable per slice) fields.

Table 3.4: LSU configuration word format and size (see Figure 2.7 for the LSU internal architecture).

Field	muxAsel	muxBsel	aluOp	rfWe	rfSel	vwr/srfOp	vwr/srfSel
Bits	19:16	15:12	11:9	8	7:5	4:3	2:0
Configuration word width							20 bits

LSU: Load-and-Store Unit

The LSU is particularly relevant for data-intensive kernels as many loads and stores between the VWRs and the SPM are necessary. The LSU has an instruction width of 19 bits (see Table 3.4), and a RF of eight 6-bit entries. Its last register (r7) holds the SPM address for load and store operations. Similarly to the MXCU, the LSU has no immediate field, but the ALU's input multiplexers have entries hardcoded to 0, 1, and 2. The ALU supports logical bitwise operations (AND, OR, XOR), signed addition and subtraction, logical left and right bitshift, and a bit-reversal operation. The latter reverses the bit order of operand a and shifts it by the amount indicated in operand b. This operation is used, for example, for the FFT kernel.

To increase the instruction level parallelism (ILP), the LSU has two operation fields: vwr_op and alu_op, as shown in Table 3.4. The latter corresponds to the ALU operation listed above. The vwr_op specifies a load, store, or shuffling operation. This enables, for example, to issue a load request to the SPM while the address of the next request is computed, allowing subsequent loads, or stores, requests to different addresses. For a load or store, r7 is the read address, respectively the write address, of the SPM. The destination VWR, respectively the source VWR, is specified in the field vwr_sel (0=VWR A, 1=VWR B, 2=VWR C). For a shuffling operation, the vwr_op field holds the data shuffling operation to execute:

- *Words interleaving*: VWR A and B words are interleaved. The result is twice the size of a VWR, and the upper or lower half can be selected as the output.
- *Even or odd index pruning*: prunes the even/odd elements of VWRs A and B, and outputs the remaining elements (of both A and B).
- *Bit-reversal*: applies bit-reversal shuffling to the concatenation of VWRs A and B. The result is twice the size of a VWR, and the upper or lower half can be selected as the output.

Table 3.5: LCU configuration word format and size (see Figure 2.9 for the LCU internal architecture).

Field	muxAsel	muxBsel	aluOp	rfWe	rfSel	immediate
Bits	18:16	15:13	12:9	8	7:6	5:0
Configuration word width						19 bits

LCU: Loop-Control Unit

The loop-control unit (LCU) principally controls loop execution for data-intensive kernels, which does not require complex hardware support. It has an instruction width of 19 bits (see Table 3.5), and a RF of four 7-bit entries. The ALU can execute conditional branch operations, signed addition and subtraction, logical bitwise operations (AND, OR, XOR), and left/right logical bitshift. The supported conditional branch operations are branch-if-equal, branch-if-not-equal, branch-if-less-than, and branch-if-greater-or-equal. The latter implements a pre-decrement of the counter before executing the greater-or-equal comparison, enabling a single-cycle counter update and branch condition solving. The LCU is the only slot with an immediate field. It is essentially used to store the destination address for conditional branch operations. Because the immediate field holds the branch destination address, the ALU's input multiplexers have entries hardcoded to 0 and 1, mainly used for comparison purposes during branch operations. The LCU also controls the end of the kernel execution with its *exit* operation.

The PC shared by all the units of a column —the RCs and the specialized slots— is controlled by the LCU. By default, the PC is incremented by one, and the LCU provides a new value only when the flow of execution changes. The PC of both columns are synchronized when they execute together one kernel. The LCU of any columns can issue a branch instruction for both PCs to be updated.

3.4 Experimental setup

3.4.1 Biosignal processing ultra-low power embedded platform

In Chapter 2, we demonstrated the performance improvement of this VWR2A instance over an existing state-of-the-art CGRA architecture and a low-power, high-performance ARM Cortex-M4 GPP, all of them targeting the biomedical domain. In this chapter, we focus on showing the advantage of our DSIP template, particularly its VWR2A variation, over ASIC designs. Therefore, an instance of the VWR2A template is integrated into an ultra-low power SoC intended for biomedical signal acquisition and processing [32]. This

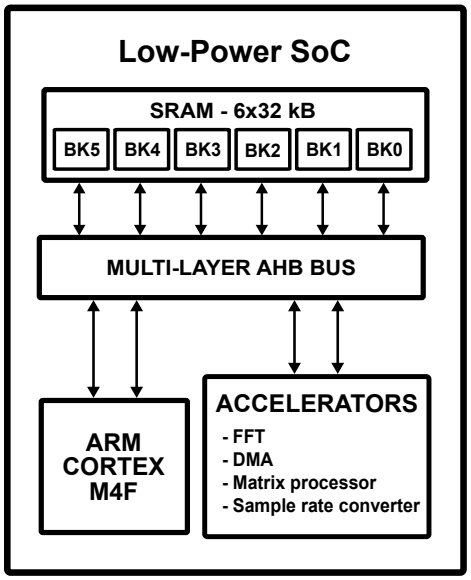


Figure 3.5: Low-power SoC for biomedical signal acquisition and processing architecture proposed in [32].

platform, shown in Figure 3.5, features an ARM Cortex-M4F processor, 192 KiB of static random access memory (SRAM) (divided into six banks that can be individually power gated), an analog front end for the acquisition of biosignals (e.g., ECG, photoplethysmogram (PPG)), a DMA, and several fixed-function accelerators. The SoC elements (e.g., accelerators, memories, processor) are connected through a multi-layer AMBA-AHB bus interface enabling parallel access of multiple masters to many slaves. The SoC has multiple power domains that can be turned on and off during execution to optimize energy consumption further. The FFT and matrix fixed-function (i.e., ASIC) accelerators are used for comparison.

FFT ASIC

It computes FFTs and inverse FFTs up to 4096 points, with an optimized flow for real-valued inputs (see Chapter 2). The FFT weights are stored in internal ROMs, whereas a dual-port local memory is used to store the data. At the beginning of an acceleration, the input signal is copied from the SoC SRAM to this dual-port internal memory (similarly to VWR2A, which first copies the signal to its internal SPM). Once the data are copied, the FFT computations start, using the radix-2 or radix-4 algorithm, depending on the FFT size. During the last stage, the data are directly copied back to the SoC SRAM once the final value is ready, so there is almost no write-back penalty time at the end of the execution. The hardware is highly optimized for FFTs. Its high-level representation is very similar to

the proposed DSIP template, with an address generation unit, a loop controller unit, and one radix-4 butterfly computing element (12 additions and three multiplications). Their main difference is that one is optimized for an application or a kernel (i.e., an FFT) with highly customized hardware, whereas the other (i.e., the DSIP) does not have any custom operation to maintain high flexibility and cover a domain of applications. One example of a custom optimization is the FFT accelerator datapath width of 18 bits with dynamic scaling to avoid overflow. The FFT execution flow is a 4-stage process decomposed into address generation, data load, butterfly computation, and results write back. When the execution is done, the FFT accelerator issues an interrupt to the processor. Such a design will normally be faster and more energy-efficient than any more flexible architecture but will only be able to execute FFTs.

Matrix processor ASIC

Compared to the FFT ASIC, the matrix processor implementation is rather simple. It has three master ports on the SoC AMBA-AHB bus: 2 for input data and one for output data, and it has no internal memories (it uses the SoC SRAM to read and write data). The execution flow is a 4-stage pipelined process: addresses generation, input data read, data computation, and output data write. It reaches maximum efficiency when multiple banks are used for the three different master ports because it removes bus conflicts and reduces latency, as evaluated in [95]. Although matrix-vector operations are pretty generic and used in many different applications, this accelerator is still considered an ASIC because it can only execute matrix-vector operations. The matrix processor supports signed addition, subtraction, and multiplication, logical bitwise operations (AND, OR, XOR) on matrix and/or vectors of 8-, 16-, or 32-bit. It also implements accumulation and bitshift operations that are applied at the output. It can execute 22 different operations in total, such as element-wise multiplication of 2 matrices/vectors, adding a constant to a matrix/vector, or the sum of differences between 2 matrices/vectors. The matrix processor is used in the experiments to compute a FIR filter, a typical kernel used in biomedical to filter the raw input biosignal data. When the execution is done, the matrix processor issues an interrupt to the processor.

3.4.2 Integration of our programmable core

The VWR2A instance is connected to the AMBA-AHB bus interface (see Figure 3.3), precisely like the other hardware accelerators, to have a fair comparison within the original SoC design. The VWR2A instance has one master port, controlled by its DMA, to transfer data between the SoC SRAM and its SPM. The kernel acceleration and DMA transfer requests from the central processing unit (CPU) are received through an additional slave

port. The VWR2A instance informs the processor when a kernel execution, or a DMA transfer, is finished through an interrupt line. It is included in the same power domain as the other accelerators and can therefore be power gated.

3.4.3 Performance and energy characterization

The complete SoC is synthesized, including the VWR2A instance, with the TSMC $40\,\mathrm{nm}$ LP CMOS technology at $80\,\mathrm{MHz}$ (the original SoC frequency), and post-synthesis simulations are run with Cadence Incisive Verification Platform [76] to compare the performance and the energy of all the implementations. (i.e., custom accelerators versus programmable core). This allows cycle-accurate simulations from which the cells' switching activity is extracted and used for power estimation with Synopsys PrimePower [78].

3.4.4 Representative set of software benchmarks

Standalone kernels

Although we are interested in the comparison at the application level, an evaluation at the kernel level is done to provide a complete analysis. The VWR2A instance is first compared with the SoC FFT accelerator and the matrix processor using a standalone FFT kernel and a FIR filter kernel, respectively, which are typical kernels used in biomedical applications [79–81, 88]. Different FFT sizes have been implemented for both complex and real-valued sequences. As discussed previously, the FFT accelerator uses a mixed radix-2 and radix-4 implementation. For the VWR2A instance, I used the radix-2 algorithm presented in Chapter 2.7.1. The second kernel is an 11-order FIR filter. Three different input sizes have been used to compare our VWR2A instance with the CPU and the matrix processor. Our mapping uses two columns of the reconfigurable array that work on different slices of the input array.

Biosignal application

To study the impact at the application level of the proposed programmable architecture, a cognitive workload estimation application [81] is considered. This application comprises four steps: preprocessing, delineation, feature extraction, and prediction. First, the preprocessing applies a FIR filter over the raw input data. Second, the delineation detects the maximums and minimums of the filtered signal to extract inspiration and expiration times. Third, these values are used for the extraction of time features (mean, median, and RMS

values), while the FFT of the filtered signal is employed for frequency feature extraction. Finally, the cognitive workload is estimated using an SVM algorithm.

3.5 Experimental results

3.5.1 Performance on standalone kernels

FFT kernel

The performance and energy consumption results for various complex and real-valued FFT sizes are reported in Table 3.6 and Fig. 3.6, respectively. Two versions have been evaluated for the execution with the ARM Cortex-M4 (CM4) and the common microcontroller software interface standard digital signal processing (CMSIS-DSP) library: one using a 32-bit representation (q31) and another one using a 16-bit format (q15). The latter allows the digital signal processing extension of the CM4 to execute packet-single instruction multiple data (SIMD) operations and significantly improves the performance. However, the precision is lower, and its use depends on the application requirements. Both versions are reported to provide a complete view of the different possibilities.

The results show that both the FFT accelerator (FFT ASIC) and the VWR2A instance have similar performance and are $24.7\times$, respectively $26.2\times$, faster than the CM4-q31 on average. Compared to the CM4-q15, the FFT ASIC and the VWR2A instance are still $7.0\times$ and $7.4\times$ faster, respectively. The VWR2A instance is less performant for small FFT sizes because the programming of the DMA transfers and the kernel parameters has a slightly larger overhead than the FFT accelerator programming. As expected, Fig. 3.6 shows that the FFT accelerator is $4.9\times$ more energy-efficient on average than the VWR2A instance when the specific kernel it was designed for is considered in isolation. Nevertheless, our goal was to narrow as much as possible the energy gap between both implementations.

The FFT accelerator uses a mixed radix-2 and radix-4 implementation that depends on the actual FFT size, resulting in different performance and power consumption, while the VWR2A mapping is identical for all FFT sizes. This explains the variation of the energy consumption ratio in Fig. 3.6. Finally, this figure only considers the accelerator energy consumption. If the complete SoC is considered, the energy difference is between a factor $3.7\times$ to $4.1\times$. In any case, compared to an FFT using only the Cortex-M4 processor and the CMSIS-DSP library with 16-bit data in q15 format, both the FFT accelerator and our architecture produce average energy savings of $84.2\,\%$ and $39.1\,\%$, respectively.

Table 3.6: FFT kernel performance comparison for various sizes.

Cplx-valued	CM4+ CMSIS (q31) cycles	CM4+ CMSIS (q15) cycles	speed-up	FFT ASIC cycles	speed-up	VWR2A cycles	speed-up
512	142 497	47 926	3.0 ×	7099	20.1 ×	7125	20.0 ×
1024	279 576	84 753	3.3 ×	13 629	20.5 ×	12 405	22.5 ×
2048	670 594	219 667	3.1 ×	31 299	21.4 ×	30 217	22.2 ×
Real-valued							
512	102 148	24 927	4.1 ×	3523	29.0 ×	3666	27.9 ×
1024	233 705	62 326	3.7 ×	8007	29.2 ×	7133	32.8 ×
2048	459 784	113 489	4.1 ×	16 490	27.9 ×	14 427	31.9 ×

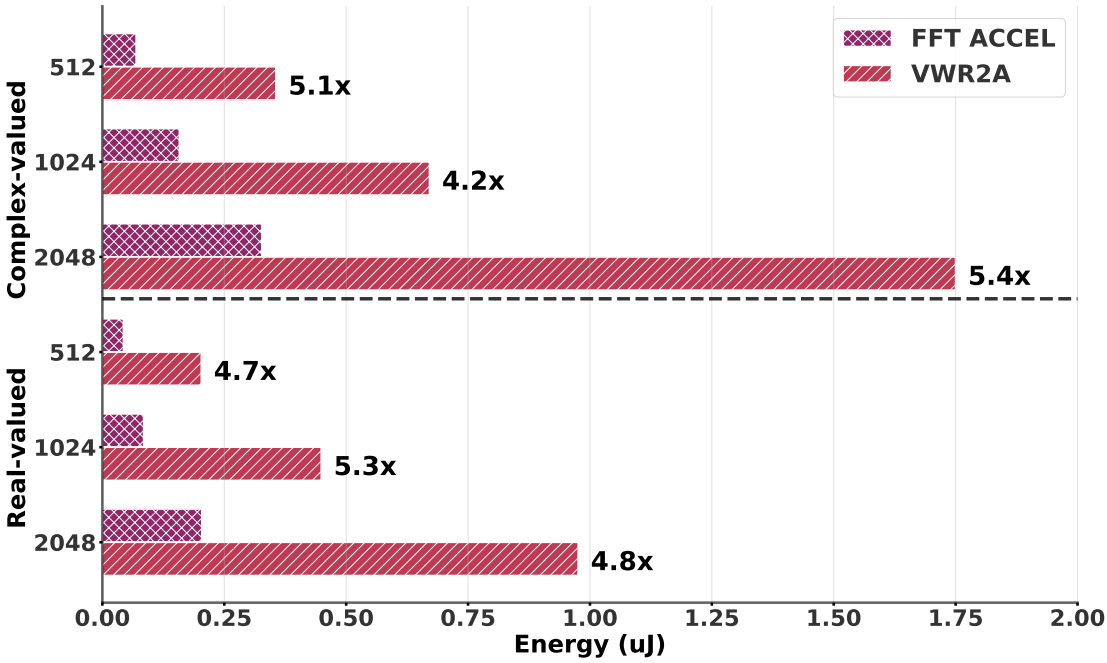


Figure 3.6: FFT kernel energy comparison for various sizes. Even if the performance of the VWR2A instance (VWR2A) is equivalent to that of the custom FFT accelerator (FFT ASIC), as expected, the gap in energy consumption is still significant in the case of isolated kernels.

Table 3.7: FFT accelerator and VWR2A power breakdown while executing a 512-point real-valued FFT.

	FFT ASIC		VWR2A		
Instance	Power (mW)	%	Power (mW)	%	ratio
DMA	1.07×10^{-2}	1 %	7.47×10^{-2}	2 %	7.0
Memories	6.68×10^{-1}	68 %	2.93×10^{0}	67 %	4.4
Control	6.25×10^{-2}	6 %	2.59×10^{-1}	6 %	4.2
Datapath	2.42×10^{-1}	25 %	1.08×10^{0}	25 %	4.5
Total	9.83×10^{-1}	100 %	4.34	100 %	4.4

Table 3.7 presents the power consumption breakdown per subcomponent, i.e., for the FFT accelerator and the VWR2A instance. It is interesting to see that both designs have an almost identical power distribution. The main contributors to our architecture are the memories and the datapath (i.e., the RCs). This means that the overhead of the instruction control, which is non-negligible in typical instruction-set processors [96], is removed in the VWR2A instance. The *Memories* category contains the VWR2A SPM (32 KiB), the VWRs (3 KiB), and the context memory (5.4 KiB), accounting for 36 %, 30 %, and 1 % of the total power, respectively. In contrast, the FFT accelerator has 17 KiB of memory in total. To build the SPM wide interface, smaller memory macros of the width supplied by the technology provider were concatenated. The VWRs were built using latches of the standard cell library. A custom design for these memories will undoubtedly reduce power consumption. Regarding the *Datapath* category, the available optimizations are more limited because the FFT accelerator is specialized for FFTs, with an 18-bit-wide datapath, while our RCs have a more general-purpose 32-bit ALU. One solution could be to have a 16-bit mode with two simultaneous 16-bit operations instead of one 32-bit operation (see Chapter 5.2.2).

Compared to the Ultra-Low Power Samsung Reconfigurable Processor (ULP-SRP) [84], a recent instantiation of the ADRES template that also uses the TSMC 40 nm LP technology, the VWR2A instance exhibits significant performance and energy gains. The authors reported an execution time of 839.1 µs and an energy consumption of 19.9 µJ for a 256-Point FFT,[1] while the VWR2A instance executes that same kernel in 35.6 µs and consumes 0.3 µJ. These numbers correspond to a factor of 23 × improvement in performance and a factor of 66 × in terms of energy. It is important to note that post-layout simulation has been done for the ULP-SRP, while I ran post-synthesis simulation, which can explain part of the significant difference in energy.

[1]The authors do not specify if they implement a complex-valued or a real-valued FFT. A 256-Point complex-valued FFT is considered, corresponding to the worst case for the VWR2A instantiation.

Table 3.8: FIR filter kernel performance and energy comparison for different numbers of points and 11 taps.

	CM4+CMSIS q31	**Matrix ASIC**		**VWR2A**	
	cycles	cycles	speed-up	cycles	speed-up
256 pts	21 269	11 598	1.8 ×	1866	11.4 ×
512 pts	42 298	23 118	1.8 ×	3280	12.9 ×
1024 pts	84 245	46 158	1.8 ×	6108	13.8 ×

FIR filter kernel

Table 3.8 reports the experimental results for three different input sizes for the FIR filter with 11 taps. I compared the performance of the processor (CM4) and the matrix processor (Matrix ASIC) with that of our VWR2A instance (VWR2A). All implementations use 32-bit data, and the CM4 uses the optimized CMSIS-DSP library.[2]

Table 3.8 shows that our accelerator is on average 12.7 times faster than the processor and 6.9 times faster than the matrix ASIC. In terms of energy, the VWR2A instance consumes 79.0 % less energy than the processor and 22.7 % less than the matrix ASIC. In this case, the DSIP instance is even better than the ASIC in terms of both throughput and energy. The main reason for the higher energy efficiency is the dedicated wide memory hierarchy of VWR2A, which has a lower energy per access cost than the SoC main memory accesses through the bus. As the matrix ASIC does not have any local data memory, it relies on the latter, translating to poor overall performance and energy consumption. The higher number of processing elements (8 RCs vs. one core) does not play a significant role as they translate to more transistors that consume more power, and the VWR2A instance consumes 14.0 × more power than the matrix ASIC. However, the energy is better as it is faster and consumes less energy per data access than the ASIC.

Compared to the FFT ASIC, which is still more energy efficient than the DSIP instance at the kernel level, the matrix ASIC is not. It is only more efficient in terms of hardware overhead: it uses 47.8 times less area than the VWR2A instance in a direct comparison (ASIC vs. DSIP), but only 1.1 times less area if the complete SoC is considered.[3] However, in the eventuality that the area overhead (i.e., the extra cost) of the VWR2A instance is too high, a slower instance of the template can be generated with fewer RCs or smaller

[2]The experiments using the CMSIS-DSP q15 format exhibited a much slower execution than the q31 implementation, but the reason could not be explained. Therefore, considering the opposite should be observed (q15 is faster than q31), only the results for the q31 version are presented.

[3]The area estimation is based on a post-synthesis netlist, and the net area is estimated using the Cadence Genus physical layout estimation (PLE) engine that uses the library lef files. Therefore, these numbers reasonably estimate the difference in the order of magnitude, but they are not accurate values.

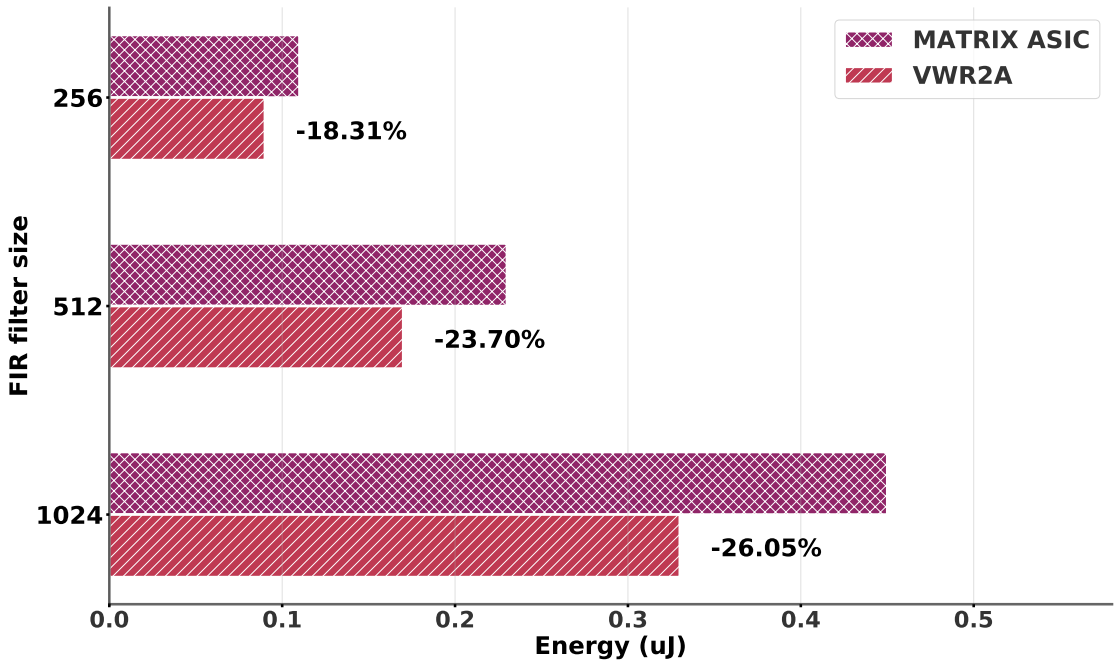


Figure 3.7: FIR filter kernel energy comparison for various sizes.

data memories (i.e., SPM and VWRs) for example (as long as the energy and throughput constraints allow it).

3.5.2 Performance on biosignal application

Table 3.9 reports the performance and energy consumption for the different steps of the application. Most of the application has been ported on the VWR2A instance, whereas the processor executes only the delineation step and manages the high-level control of the application.

Preprocessing

The CPU+VWR2A implementation is 13.2 times faster than the Cortex-M4 (CPU), which translates into energy savings of $61.8\,\%$. The CPU+ASICs version is 1.4 times faster than the CPU version because the matrix processor can execute the preprocessing FIR filter. These values are similar to the comparison at the kernel level (see Table 3.8) because the FIR filter represents most of the preprocessing step time, and the difference is due to the extra code present at the application level (e.g., function calls, data movements, execution

Table 3.9: Performance and energy comparison of the complete biomedical application for the ARM Cortex-M4 only (**CPU**), the ARM Cortex-M4 plus the FFT and Matrix ASICs (**CPU+ASICs**), and the ARM Cortex-M4 plus the VWR2A instance (**CPU+VWR2A**).

Cycles	**CPU**	**CPU + ASICs**	savings	**CPU + VWR2A**	savings
Preprocessing	49 760	34 402	30.9 %	3763	92.4 %
Delineation	32 716	32 716	0.0 %	32 716	0.0 %
Feat. extraction + SVM	70 639	54 255	23.2 %	9002	87.3 %
Total	153 115	121 373	20.7 %	45 481	70.3 %
Energy (µJ)					
Preprocessing	0.68	0.47	30.7 %	0.26	61.8 %
Delineation	0.54	0.54	0.0 %	0.54	0.0 %
Feat. extraction + SVM	1.1	0.98	10.9 %	0.40	63.2 %
Total	2.32	1.99	14.2 %	1.20	48.1 %

control). Still, the CPU+VWR2A implementation is 9.1 times faster and 44.8 % more energy efficient than the CPU+ASICs.

Delineation

The delineation step is not considered in this chapter, as its code structure (i.e., control-intensive code with complex if-else conditions) is very different from the other more data-intensive steps. This limits the gain of VWR2A at the application level because data have to be moved back and forth between the SoC SRAM (i.e., main memory) and the VWR2A SPM for the delineation step to be executed by the CPU. The possibility of executing this step and the impact at the application level are explored in Chapter 4.

Feature extraction and SVM prediction

The feature extraction and SVM prediction steps are grouped into a single step because their mapping on the VWR2A is a mix of multiple kernels without a clear distinction between the two steps. The FFT ASIC computes a real-valued 512-Point FFT during this step, which translates to a 10.9 % gain in energy compared to the CPU version. However, the FFT represents only a portion of the application code, of which the custom accelerator cannot execute anything else. In contrast, the VWR2A instance can execute all the code of this step (i.e., feature extraction step and SVM prediction), with a corresponding energy

saving of $63.2\,\%$ compared to the CPU version. It is also $6.0\times$ faster than the CPU+ASICs version and saves $58.7\,\%$ of energy. The VWR2A instance benefits from its large code coverage and the possibility to apply application-level optimizations. For example, the FFT ASIC has to copy the input data into its local memory and copy the complete output back to the system's main memory. For the VWR2A instance, this can be optimized thanks to its programmability. First, only part of the FFT output is used by the application, and the VWR2A instance can partially skip the unnecessary computations related to the unused output. In addition, as the rest of the code is also executed by the VWR2A instance, it copies only the estimated state by the SVM back to the system's main memory, while the FFT accelerator has to copy back the 512 FFT output values.

Discussion

Figure 3.8 summarize the energy results, normalized to the CM4 consumption, of both the kernels and the complete application. This figure supports the central claim of this work and shows that larger savings can be obtained from a reconfigurable architecture (i.e., a DSIP) with respect to custom accelerators (i.e., ASICs) when complete applications — rather than individual kernels — are considered. Although the FFT ASIC is $4.9\times$ **more** energy efficient on average than the VWR2A instance at the kernel level (i.e., executing FFTs), it is $2.4\times$ **less** energy efficient at the application level (considering only the feature extraction and SVM prediction step). For the matrix ASIC, which was already less efficient at the kernel level, the results at the application level are even worst because of the extra code required in an actual application.

At the complete application level, the CPU+ASICs is $1.3\times$ faster and $1.2\times$ more energy efficient than the CPU version. For the VWR2A instance, these values are higher, respectively $3.4\times$ faster and $1.9\times$ more energy efficient. This translates to an energy-delay product (EDP) improvement of $6.5\times$ and $4.4\times$ compared to the CPU (i.e., the Cortex-M4) and CPU+ASICs implementations, respectively.

Two essential features explain the better efficiency of the VWR2A instance: its low-power processing elements and its dedicated wide memory hierarchy. In particular, the specialized slots are responsible for an estimated speed-up factor of 2, while their power consumption represents less than $6\,\%$ of the total. In addition, the low-power design of the RCs enables a high throughput to relatively low power, translating to high energy efficiency. For example, one RC consumes $3.8\times$ less power than the CM4 for a 512-point 11-order FIR filter. The wide memory hierarchy allows fast parallel access to the data tailored to the RCs' throughput. Moreover, these data accesses are more energy efficient than accessing the system's main memory through the bus. On average, the VWR2A instance reduces the energy consumption of the main memory and the bus by $0.24\,\mu\text{J}$ when executing the FIR

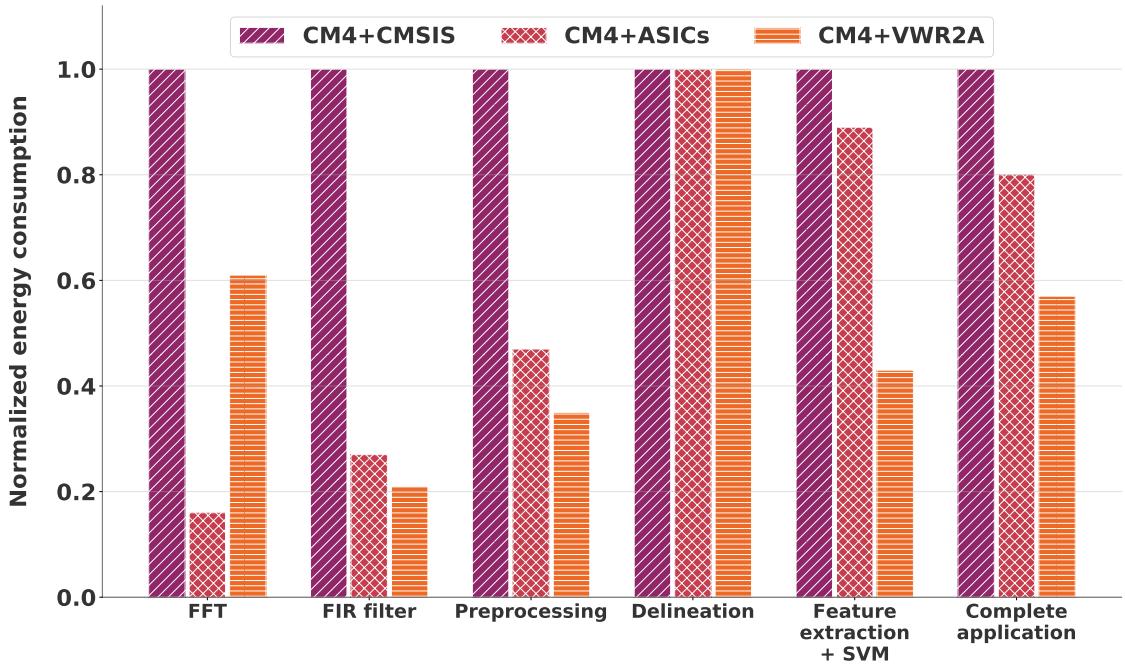


Figure 3.8: Normalized energy comparison for the kernels, the application steps, and the complete application.

filter kernels compared to the matrix ASIC. However, the wide memory hierarchy is only responsible for increasing the energy consumption of the VWR2A instance by $0.09\,\mu$J.

The VWR2A template provides flexibility through its design parameters and allows the design of instances targeting different requirements. There are three main parameters to set: the total number of RCs, the data memory (SPM and VWRs), and the local program memory size. The total number of RCs and their organization (i.e., the number of columns) impact the throughput and the number of tasks that can run in parallel. In the VWR2A instance of this chapter, the RCs (of both columns) represent between $20\,\%$ to $40\,\%$ of the total power consumption. The rest is primarily shared between the SPM and the VWRs. Therefore, increasing the number of RCs in the columns improves performance and energy efficiency as long as the workload can be parallelized over the additional RCs. For example, doubling the RCs (i.e., 8 RCs per column) does not double the total power consumption but could potentially double, in the best case, the execution speed, hence the energy efficiency. However, this would increase the area (i.e., the cost). In this instance, the RCs of the two columns represent $8\,\%$ of the total area.

The data memory design (i.e., the wide memory hierarchy) is also fundamental. The main parameters are the width of the SPM and the VWRs, and the depth of the SPM. The width fixes the batch size of processed data and should be set according to the amount of

Table 3.10: Statistics of the units' local program memory considering all the data-intensive kernels.

	LCU	**LSU**	**MXCU**	**RC0**	**RC1**	**RC2**	**RC3**
avg instructions[4]	14	19	14	18	17	17	15
min instructions[5]	2	3	3	4	2	2	3
max instructions[6]	28	41	29	42	43	38	38
avg active instructions[7]	39.7 %	49.4 %	37.9 %	50.6 %	44.1 %	44.3 %	41.7 %
avg/min/max kernel size[8]	32 / 8 (mean) / 55 (rsp_params)						

data that has to be processed. As mentioned in Section 3.3.2, the 4096-bitwidth interface of this instance is a good performance-power tradeoff for the considered application (i.e., with a 5 Hz sampling frequency and a 1-minute and 40 seconds sampling window). A wider interface might be better for other scenarios with more data (i.e., with a higher sampling frequency and/or a shorter sampling window). However, one current limitation to the VWRs' width is their design based on standard cells that consume significant power. Therefore, it is usually better to pay a minor cycle penalty by reloading data from the SPM instead of increasing the VWRs' size. This conclusion is based on post-synthesis power and energy estimation, and after layout, it could change. The SPM depth defines how much data can be stored locally by a VWR2A instance. It should minimize the need for data transfer with the system's main memory to reduce the energy cost of data transfer through the long bus wires. For biomedical applications, the data are often processed in windows that can be discarded once processed. Therefore, the SPM should be large enough to contain all the data of one sampling window plus the extra data that are kept between two windows (e.g., some features, data that have not been processed, constant values such as the weights used by an FFT).

Finally, the units' local program memory size defines the maximum size of the kernels that can be executed. Table 3.10 shows the statistics of this memory for the different units. It shows a significant variation in kernel size, with less than five instructions for all the units at a minimum and up to 43 instructions for RC1 for one kernel. The smallest kernel (mean) computes the mean value of a vector and requires up to 8 instructions (i.e., the maximum PC address), while the largest one (rsp_params) needs 55 instructions. However, the latter could be decomposed into multiple kernels. The undividable kernel that forced the local program memory to 64 addresses is the FTT, which uses 39 instructions. This shows the importance of analyzing the type of kernels that will be encountered, especially the long ones that cannot be divided into smaller sub-kernels. Table 3.10 also shows that the different units have different needs in terms of maximum size. However, as the program counter is shared, all the units must have the same local program memory depth, while the LCU or the MXCU never have more than 28, respectively, 29 active instructions. This problem is discussed, and an alternative is proposed in Chapter 5.2.3.

3.6 Summary and conclusions

This chapter presented an instance of the VWR2A DSIP template proposed in Chapter 2 that focuses on data-intensive kernels of the biomedical domain. It features a 4-by-2 reconfigurable array organized in two columns, a $32\,\mathrm{KiB}$ SPM with a 4096-bitwidth interface toward the VWRs, enabling 128 words of 32-bit to be transferred in a single cycle. The optimized ISA of the RCs, in particular its single-cycle fixed point multiplication instruction, translate to high performance.

The VWR2A instance (referred to as VWR2A below) performance and energy consumption have been first compared against two ASICs at the kernel level. Compared to the FFT and matrix ASICs, VWR2A is $1.1\times$ and $6.7\times$ faster on average, respectively. This proves the feasibility of using a domain-specific programmable core as an accelerator instead of a fixed-function accelerator and achieving similar or even better performance. Regarding energy consumption, fixed-function accelerators typically perform better on specific tasks. The experiments at the kernel level showed that VWR2A was $4.9\times$ less energy efficient than the FFT ASIC, but $1.3\times$ **more** efficient than the matrix ASIC.

The full potential of VWR2A is best seen at the application level, where it significantly surpasses GPPs and ASICs in both performance and energy with an EDP improvement of $6.5\times$ and $4.4\times$, respectively. In detail, the evaluated VWR2A instance showed an $3.4\times$ speed-up and $48.1\,\%$ energy consumption reduction compared to an ARM Cortex-M4 GPP and an $2.7\times$ speed-up and $39.5\,\%$ energy consumption reduction compared to an ARM Cortex-M4 GPP augmented with two ASICs (i.e., the FFT and matrix ASICs) at the application level. The reason is that more code is eligible for acceleration with a flexible architecture, which leads to better overall performance and energy efficiency, as long as the energy gap with respect to the fixed-function accelerator has been sufficiently reduced. In addition, programmable architectures can also accommodate application-specific optimizations that are impossible with custom accelerators. These results show the benefit of designing a single DSIP instance covering a large domain of applications and the higher performance improvement gained at the application level compared to ASICs.

[4]Static average of instructions that are not a nop.
[5]Static minimum number of instructions that are not a nop.
[6]Static maximum number of instructions that are not a nop.
[7]Average percentage of instructions that are not nops based on each kernel length (not the total program memory size of 64 instructions).
[8]This value correspond to the highest PC address in which an active instruction is stored, not to the maximum numbers of active instruction in a unit.

4 Acceleration of Control-intensive Applications for Embedded Systems

4.1 Introduction

Computing at the edge is challenging because of the limited resources environment and the always-increasing workload of new applications. The biomedical domain, particularly wearable biosignal-processing devices, is an excellent example of edge computing revolutionizing our daily life, but where the required energy efficiency is not yet accomplished. Embedded systems have to be optimized all across the stack in order to reach this energy efficiency.

At the hardware level, the recent trend in research is towards heterogeneous platforms containing multi-core architectures with heterogeneous processors and hardware accelerators. These systems can adapt their performance and power consumption based on their current operating state (e.g., their workload and battery load), enabling high-performance and energy-efficient execution. In this context, co-processors or hardware accelerators have become a standard in state-of-the-art platforms [22, 32, 45] because of their better efficiency at executing repetitive operations compared to a general-purpose processor. The flexibility-performance tradeoff of hardware accelerator design has led to two categories of accelerators:

a) domain-specific instruction-set processors (DSIPs) or programmable cores (e.g., coarse-grained reconfigurable arrays (CGRAs) [45]).

b) application-specific integrated circuits (ASICs) or custom accelerators (e.g., fast Fourier transform (FFT)[32], convolutional neural network (CNN) [97]).

However, most of the literature focuses on evaluating the performance and energy tradeoffs for data-intensive kernels (e.g., FFTs, finite impulse response (FIR) filters, mean

or RMS computation), as done in Chapter 3. This approach is valid only if the considered workload is dominated by such code. Besides data-intensive code, applications usually have control-intensive kernels. Control-intensive kernels contain a significant number of operations related to control (e.g., loop, if-else structure) compared to arithmetic operations performed on data (some examples of such code are shown in Figures 4.4 and 4.6 and discussed later in Sections 4.4 and 4.5.3). Data-dominated kernels usually have multi-level nested loops and minimal data dependencies enabling the parallel computation of different data points (e.g., FIR, FFT, convolution). Conversely, control-intensive code originates from algorithms that have data-dependent paths. An extreme example is the usual implementation of a decision tree based on a tree of if-else conditions. Therefore, parallelization is often not possible at the data level, and other mechanisms are required to improve the execution of such code. The proportion in execution time of data- and control-intensive code is essential to optimize the hardware.

4.1.1 Low-power architectures for control-intensive code and limitations

Data-intensive code often represents most of the execution time, which is why the literature focuses on it. Nonetheless, this can limit the performance improvement of hardware accelerators at the application level [85] in two cases: (a) the application is dominated by control-intensive code by nature, or (b) most of its data-intensive kernels have already been accelerated by the hardware. In both cases, the non-accelerated code (i.e., the control-intensive kernels), which represents a significant portion of the total execution, is executed by a general-purpose processor (GPP). Therefore, energy-efficiency can only be enhanced further by improving the execution of the control-intensive code.

The biomedical domain is a perfect illustration of this potential limitation. In Table 3.9 of Chapter 3, the original application (i.e., executed by the GPP) has data- and control-intensive code representing $79\,\%$ and $21\,\%$, respectively, of its processing time. In this case, optimizing the execution of data-intensive code is the best option to increase overall efficiency. However, once this code is accelerated with the proposed DSIP of Chapter 3, for example, it represents only $28\,\%$ of the total execution time, while the control-intensive code now represents $72\,\%$. In this specific example, the remaining code is a delineation algorithm, shown in Figure 4.1, which extracts the main characteristics point of a biosignal.

The literature does not focus much on control-intensive code and its energy-efficient execution on low-power architectures. Some techniques have been proposed to increase the performance of GPPs executing such code. For example, zero-overhead loops [56] or branch prediction [98] can improve the performance of processors, but they do not always translate into energy savings [52] if they increase the circuit area too much.

Additionally, for control-intensive kernels, workload parallelization at the data level is often not possible, and other mechanisms, such as instruction level parallelism (ILP), are required to improve the execution of such code. Very-long instruction word (VLIW) processors can exploit code ILP and increase instructions-per-cycle (IPC) with a higher energy efficiency compared to superscalar GPPs because part of the work is relayed to the compiler rather than to runtime logic [85]. In [84], the authors combined a VLIW processor with a CGRA to optimize performance on both data- and control-intensive code. However, such a design is not the most optimal in terms of area as it contains a VLIW processor and a CGRA. Therefore, replacing these two elements with a single one would be more area- and cost-effective and save energy because of the reduced wiring.

4.1.2 Contributions and outline of the chapter

This chapter explores the performance and energy efficiency of accelerators based on domain-specific programmable cores for control-intensive code on energy-constrained embedded systems. Based on the DSIP architecture template proposed in Chapter 2 and the variation presented in Chapter 3 for data-intensive biomedical applications, I propose a new instance focusing on control-intensive code for the biomedical domain. Its performance and energy consumption on control-intensive code are compared to two general-purpose baseline processors: an ARM Cortex-M4 [99] and a RISC-V based lowRISC Ibex [35] (formerly known as the Zero-riscy [53]). The Cortex-M4 is a middle-class processor with simple branch prediction (branch forwarding) and DSP extensions, while the Ibex is optimized for control-dominated code [53]. The main contributions of this chapter are:

- An instantiation of the DSIP template proposed in Chapter 2 targeting control-intensive biomedical kernels and applications.
- An analysis of the target domain and of the template design parameters used for the VWR2A instance.
- An evaluation of its performance and energy consumption on control-intensive kernels and applications.
- A comparison with two general-purpose processors, one of which is optimized for executing control-intensive code.

The rest of this chapter is organized as follows. First, the existing solutions and their limitations are discussed in Section 4.2. Then, I present the architecture features for control-intensive code execution of the DSIP template instance in Section 4.3 and evaluate

its power consumption on control-intensive kernels in Section 4.4. The experimental setup to evaluate the architecture is described in Section 4.5, and the results are presented and analyzed in Section 4.6. Finally, the main conclusions of this chapter are drawn in Section 4.7.

4.2 Related work

4.2.1 Low-power architecture for control-intensive code

Hardware accelerators are primarily designed for data-intensive code, as it usually represents most of the execution time. However, once this code is accelerated, control-intensive code or control flow (e.g., branches, conditions) becomes predominant, limiting the impact of the accelerator [85], especially at the application level. Reconfigurable architectures are usually preferred as their flexibility can offer good performance for multiple kernels. In particular, CGRAs are often used in low-power platforms because of their proven performance and energy efficiency [33, 45, 100]. However, the traditional modulo scheduling (software pipelining) [101] for kernel mapping fails to convert nested loops and complex if-else structures to static code for CGRAs. This situation implies a CGRA has to rely on its host processor for complex control code (e.g., if-else blocks) and outer loop control (i.e., nested loops). Some software-pipelining solutions or mapping techniques have been proposed to partially solve this problem [102, 103], but this work focuses on solutions at the architecture level.

At the system level, different solutions have been proposed. For example, the MorphoSys platform [68] combines a reduced instruction set computer (RISC) processor and a CGRA. Despite the excellent performance of the CGRA on data-intensive kernels, the control code (inherent to any application) is left to the RISC processor, limiting the performance gain at the application level. The authors of [85] proposed the ADRES framework to solve this problem. ADRES combines a VLIW and a CGRA to efficiently execute control-intensive and data-intensive code, respectively. While the authors of ADRES improved the overall performance, they did not focus on energy consumption. VWR2A, the proposed DSIP template in Chapter 2 is a CGRA-like architecture augmented with low-power memory structures, made of a scratchpad memory (SPM) and very-wide registers (VWRs), and integrating VLIW concepts, such as specialized slots (e.g., a load-and-store unit (LSU)). The VWR2A variation of Chapter 3 has shown better energy efficiency on data-intensive code than ULP-SRP [84], an instantiation of the ADRES framework.

The authors of [100] proposed an Integrated Programmable Array (IPA): a 4x4 reconfigurable array of reconfigurable cells (RCs). Similar to the VWR2A template, some RCs

have an LSU, but it is connected to a shared multi-bank tightly coupled data memory (TCDM) through a logarithmic interconnect (while VWR2A proposes a low-energy wide memory hierarchy). Additionally, the RCs are augmented with jump and conditional branch instructions to execute nested loop structures. VWR2A integrates a loop-control unit (LCU) for the same reason, and because the RCs do not handle these instructions, the performance should be better. Both IPA and VWR2A demonstrated good energy efficiency for data-intensive kernels with nested loops (e.g., FIR, FFT, convolution), but the former has not been evaluated on control-intensive kernels.

SNAFU [104], an ultra-low-power CGRA generation framework, proposes an architecture with heterogeneous processing elements that are specialized for specific tasks, similar to the specialized slots of VWR2A. SNAFU introduces the spatial vector-dataflow execution model to reduce energy consumption. This model is conceived to map the complete data flow graph (DFG) of an innermost kernel to the complete accelerator array, configuring each processing element (PE) to perform one single operation. This model, conceptually closer to that of a super-systolic array, contrasts with the programming model of VWR2A, in which each PE has a small instruction memory to implement a small program.

VWR2A and SNAFU present two additional significant differences: first, SNAFU uses a traditional memory hierarchy with multiple master ports connected to the platform's bus and main memory. On the contrary, VWR2A has a wide memory hierarchy that offers a high bandwidth and low energy solution compared to SNAFU. SNAFU implements scratchpad memories to limit costly accesses to the platform's main memory (in latency and energy), particularly between reconfigurations of the array. The latter is due to the second main difference: VWR2A has a specialized slot, namely an LCU, enabling the mapping of multi-level nested loops. As this allows VWR2A to cover longer fragments of code independently, the GPPs present in the system can sleep for longer periods or perform other unrelated tasks. In comparison, SNAFU is limited to the innermost loop, relying on the GPP and possibly multiple reconfigurations for the outer loops within one kernel. Finally, similar to the IPA, SNAFU has demonstrated chiefly its efficiency on data-intensive kernels (e.g., FFTs, DWT) but not for control-intensive code.

Other proposals offer architecture solutions, but they target high performance rather than energy efficiency. For example, Elastic CGRAs [105] introduce elasticity in the PEs' interconnection network to enable efficient use of PEs with operations that have varying latency (e.g., memory access vs. ALU operation). However, for low-power CGRAs, the environment is typically more constrained, and the latency of each operation is known at compilation time, allowing efficient scheduling. Techniques for conditionals, partial and full predication have been proposed for CGRAs [106, 107]. However, while such techniques improve performance, they are usually worse in energy [52, 53]. Moreover,

these methods have been applied in the context of modulo scheduling, therefore, limited to the condition(s) present in the innermost loop of a kernel.

This chapter evaluates the extensions introduced explicitly in a VWR2A instance to improve the execution of control-intensive kernels and applications and assess its resulting higher efficiency compared to a central processing unit (CPU) for that type of code. Two state-of-the-art baseline GPPs for low-power applications are considered. The first one is an ARM Cortex-M4 processor [99], the host processor of the system-on-chip (SoC) used for the experiment in Chapter 3 and in which a VWR2A instance was integrated. However, this middle-class processor may not be the most energy-efficient for control-dominated code. Therefore, for generality, I also compare the architecture with a lowRISC Ibex processor [35] optimized for control-intensive kernels [35, 53]. The Ibex implements the RISC-V RVC32IM instruction set architecture (ISA) with a two-stage pipeline.

4.2.2 Biomedical applications

Wearable bio-signal processing devices often have at least one control-intensive step called delineation[1]. In this step, the characteristic points of a bio-signal are extracted to provide direct information to a specialist (e.g., a doctor) or to extract features for autonomous diagnosis from the device (see Chapter 3.2.1). Figure 4.1 shows the filtered respiration bio-signal of a person, its delineation, and the corresponding C code made of complex if-else structures used to perform this delineation (instructions inside the conditions are removed for clarity). It illustrates the intrinsic control nature of delineation. This step is highly dependent on the nature of the signal and a person's physiologic characteristics. Designing an ASIC for such a code would make it very efficient, but such a circuit would not be a viable commercial solution as a different circuit is required for different bio-signals and sometimes even for different people. Therefore, relying on programmable hardware is the only solution. Delineation is a control-intensive step particular to biomedical applications, but more generic examples (e.g., median, morphological filtering (MF)) are shown and discussed in Sections 4.4 and 4.5.3.

4.3 Very-Wide Register Reconfigurable-Array architecture

The VWR2A instance proposed for control-intensive code for the biomedical domain is shown in Figure 4.2. It is very similar from a high-level perspective to the instance proposed for data-intensive code in Chapter 3. The modified hardware blocks to enable

[1]Delineation often refers to the extraction of the P, QRS, and T waves of an ECG signal, but it is used here to refer to any bio-signal characteristic points extraction.

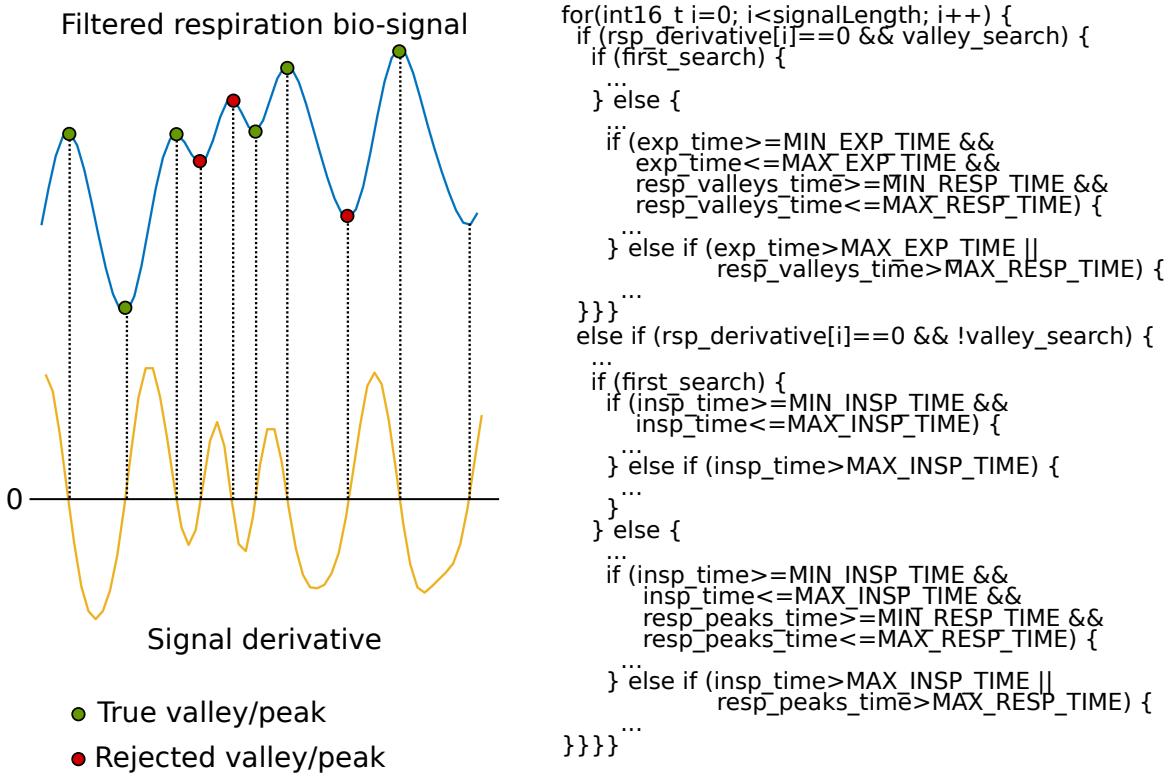


Figure 4.1: Respiration bio-signal delineation (left) and if-else C code structure for valid valley-peak pairs detection (right).

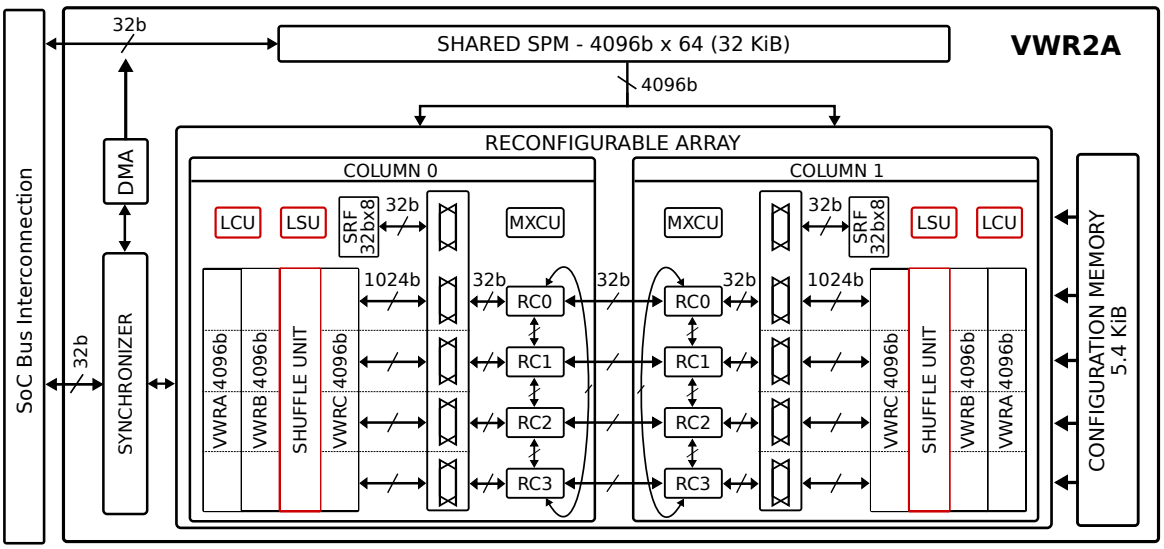


Figure 4.2: VWR2A instance for control-intensive biomedical applications and used for experiments. The modified hardware blocks (compared to the instance of Chapter 3 focusing on data-intensive code) are highlighted in red (LCU, LSU, and the shuffle unit).

efficient mapping of control-intensive code (compared to the instance of Chapter 3) are highlighted in red in Figure 4.2. It features a 4x2 reconfigurable array organized in two columns of 4 RCs. The specialized slots and the RCs of a column are all programmed in parallel at the beginning of a kernel execution by loading a maximum of 64 instruction words from the context memory to their internal instruction memory.

The same memory hierarchy design as Chapter 3.3 is reused because it targets the same domain. It has a 32 KiB shared SPM between the columns with a 4096 bits width (128 words of 32 bits). Each column has three VWRs with a dual interface: 4096 bits on the SPM side and 128 bits on the RCs' side (i.e., one 32-bit word per RC). This enables the transfer of a wide line (i.e., 4096 bits) between the SPM and the VWRs in one cycle, whereas the RCs can consume data in 32-bit words. These transfers are controlled by the LSU with an optimized ISA for this task. The VWRs are divided into four slices to enable concurrent access from the RCs of one column. The details of the SPM, the VWRs, and the scalar register file (SRF) are not discussed further in this chapter as they are less relevant for control-intensive code, and the same design has been presented in Chapter 3. However, their low-power features and single-cycle transfer play an essential role in the overall performance and energy efficiency of the architecture.

The specialized slots LCU and multiplexer-control unit (MXCU) were included in the VWR2A architecture template specifically to tackle control-intensive code. These slots enable separating the control instructions from the main stream of instructions, which exposes more parallelism to the platform in general and to the RCs in particular.

4.3.1 Reconfigurable array

In this specific case, the target domain is the same as Chapter 3 (i.e., the biomedical domain), and the ideal DSIP should be able to execute both types of code. Therefore, the reconfigurable array is similar to Chapter 3. Nevertheless, the possibility of optimizing the architecture for one or the other type of code (i.e., control- or data-intensive) is discussed in Section 4.6.7. The RC configuration words have no immediate field but hardcoded values for two multiplexer inputs: 0 and 1, reducing the instruction width to 17 bits (see Table 3.1). A small register file of two entries per RC is enough, and a datapath width of 32 bits to stay generic and compatible with standard processors is implemented. A minor change compared to Chapter 3 is the addition of two output flags directly connected to the LCU. These are the equal and greater-or-equal flags based on the results of the four RCs and OR-ed to produce a single flag for both conditions. This enables fast and efficient mapping of data-dependent conditions on the architecture (see the LCU section below for more details).

The performance of the RCs is not crucial for control-intensive code. For example, the RCs implement a one-cycle signed or fixed-point multiplier; however, control code usually consists of simpler operations, such as addition and subtraction. For a different target domain, having a multi-cycle multiplier or even no multiplier might be a better option, particularly if control-intensive kernels are dominant. These optimizations are further discussed in Section 4.6.7. Two essential operations for control-intensive code are operand selection based on the sign and the zero flags. These operations select one of the two arithmetic logic unit (ALU) input operands as output based on the value of the sign flag —movs.sf instruction— or the zero flag —movs.zf instruction. Similar to the RCs' interconnection for data, the RCs can select their internal flags or those from their neighboring RCs. The flag value refers to the previous cycle operation result in both cases. These instructions enable efficient mapping of simple if-else conditions (i.e., without any dependencies on data outside an RC) to be executed locally by the RCs.

4.3.2 Specialized slots for control-intensive code

The specialized slots are optimized for a subset of tasks and can run in parallel to take advantage of the ILP inherent to any code, particularly control code. Each slot is optimized for its specific set of tasks with a custom ISA, datapath, and register file. These slots are inspired by the brain's regions that are optimized for certain tasks (e.g., vision, motor control) and allows the efficient execution of different tasks in parallel. For a VWR2A instance, this translates to performant and energy-efficient execution of different types of kernels. Similarly to the rest of the VWR2A instance of this chapter, the specialized slots are the same as Chapter 3 with a few changes to better support the execution of control-intensive code.

Loop-Control Unit

The primary task of the LCU is to handle loop control (e.g., counter increments, branches) and conditions (e.g., if-else blocks). The design proposed in Chapter 3 supports conditional branch, signed addition/subtraction, logical bit operations (and, or, xor), and left/right logical and arithmetic shift. The branch-if-greater-or-equal operation implements pre-decrementation of the counter by one, enabling a single-cycle counter update and branch. The conditional branch instructions update the program counter (PC) register shared by all the units in a column (i.e., the four RCs and the specialized slots). While this design is sufficient for simple data-independent conditions, it is not the case for more complex conditions, which may require multiple data-dependent comparisons, for example. To accommodate for such a scenario, the LCU can execute a conditional branch based on the RCs' condition flags (i.e., equal and greater-or-equal flags). The flags of the four RCs of a

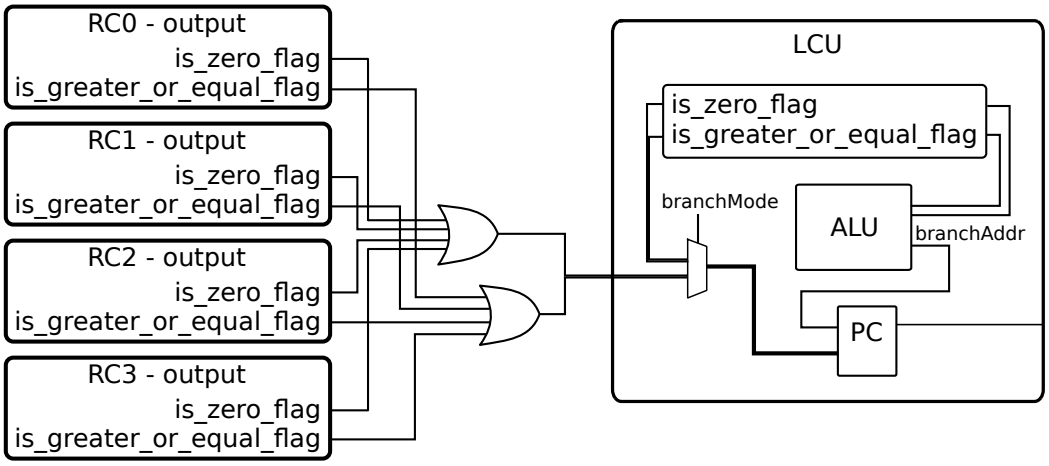


Figure 4.3: RCs' flags OR-ed and LCU conditional branch flags selection (branchMode).

column are OR-ed to produce a single flag for both conditions that are connected to the LCU, as discussed above. Figure 4.3 shows the block diagram of these connections.

In addition, the LCU is augmented with a jump instruction (without condition) that uses the sum of the two ALU operands as the destination PC address. This dynamic address generation helps map control-intensive code characterized by irregular data access patterns on the single-ported VWRs. For example, the kernels that implement a queue (see Section 4.5.3) benefit from it by jumping dynamically to the address that uses the correct VWRs' slice and RC to access specific data in the queue. Table 4.1 shows the LCU configuration word format with the additional *branchMode* field of 1 bit to select between the LCU's and the RCs' flags.

Without this simple feature, some kernels can still be executed but have to use the shared SRF to transmit data between the RCs —where the data are compared— and the LCU —where the branch decision is taken. This results in penalty cycles that would increase, for example, the execution time of the *Median* and *Delineation* kernels (see Section 5) by $6\,\%$ and $28\,\%$, respectively. For the MF kernels (see Section 4.5), it is even more critical as they could not be mapped on the VWR2A instance without these features. In this case, the additional instructions required to use the shared SRF do not fit in the internal instruction memory of the LCU. For the biomedical domain, these extra features are a better alternative in terms of performance and energy, but if such solutions are not conceivable, increasing the size of the units' local program memory is possible. However, it would represent a significant power overhead.

Table 4.1: LCU configuration word format and size updated for control-intensive code (see Figure 2.9 for the LCU internal architecture).

Field	muxAsel	muxBsel	aluOp	branchMode	rfWe	rfSel	immediate
Bits	19:17	16:14	13:10	9	8	7:6	5:0
Configuration word width							20 bits

Table 4.2: MXCU configuration word format and size (see Figure 2.8 for the MXCU internal architecture).

Field	muxAsel	muxBsel	aluOp	rfWe	rfSel
Bits	26:23	22:19	18:16	15	14:12
Field	srfWe	srfSel	srfWmux	vwrSel	vwrWe
Bits	11	10:9	8:6	5:4	3:0
Configuration word width					27 bits

MultipleXer-Control Unit

The MXCU computes the addresses of the VWRs' words passed to the RCs. While many data accesses with very regular patterns are usually seen for data-intensive code, the inverse is often true for control-intensive code: few data accesses with irregular patterns. In both scenarios, the number of instructions of the innermost loop mapped on the architecture is critical for high performance, and the MXCU plays a crucial role.

The MXCU has an instruction width of 27 bits, as shown in Figure 4.2, and an eight-entry register file. The instructions do not have an immediate field, but the ALU's input multiplexers have entries hardcoded to 0, 1, and 2. These values are enough to create most of the access patterns. The ALU can execute addition/subtraction, logical bit operations (and, or, xor), and left/right logical bit shift. The datapath width of 5 bits is fixed by the address range of the VWRs: $2^5 = 32$ (i.e., the VWRs word width, 128, divided by four slices, one per RC). These parameters are the same as Chapter 3 and proved to be enough for typical control-intensive kernels of the biomedical domain.

Load-and-Store Unit

The LSU is less critical for control-intensive kernels as they usually require less frequent data movements. The only change related to the LSU is the addition of one shufling operation to the existing *words interleaving*, *bit-reversal*, and *even/odd index pruning*:

- *Circular shift*: the concatenation of VWRs A and B is shifted by 32 words up in a circular manner (i.e., the upper 32 words are moved to the lower 32 words). The

Table 4.3: LSU configuration word format and size (see Figure 2.7 for the LSU internal architecture).

Field	muxAsel	muxBsel	aluOp	rfWe	rfSel	vwr/srfOp	vwr/srfSel
Bits	19:16	15:12	11:9	8	7:5	4:3	2:0
Configuration word width							20 bits

result is twice the size of a VWR, and the upper or lower half can be selected as the output.

This shuffling operation helps execute code that cannot be parallelized at the data level, such as during delineation. Because each RC can access only one-fourth of the VWRs, the *circular shift* enables the sequential processing of all the elements inside a VWR with the same instruction flow. Table 4.3 shows the configuration word format of the LSU. No extra bit is required to support the additional shuffling operation because the corresponding field (i.e., vwr_op field in Table 4.3) is already large enough to support it.

4.4 Illustrative kernel mapping analysis

The specialized slots are the main reason for the architecture's performance and energy efficiency. They remove the latency of control-related instructions from the RCs, increasing the ILP of the architecture. Their design relies on the fact that many kernels can be divided into four —usually independent— tasks: execution control, data loading and storing, data address update, and data computation. These tasks are mapped on the LCU, the LSU, the MXCU, and the RCs, respectively, maximizing the architecture ILP.

To illustrate the use of the architecture, the mapping of an ascending sorting algorithm that is part of the median kernel discussed in Section 4.5 is analyzed. This example highlights the key features of the architecture and helps to understand the performance results of Section 4.6. The detailed analysis at the instruction level is done in Appendix A.

Figure 4.4 shows the C code of the ascending sorting algorithm and its high-level mapping on the architecture. The LCU manages the 2-level nested loop. The counting direction is reversed to take advantage of the single-cycle counter decrement and branch (see 4.3.2). Moreover, the size of the loops is divided by two because the sorting is parallelized over the RCs of one column. This allows using all the four available RCs, therefore improving performance. The MXCU generates the addresses for accessing the i-th element and the j-th element of the data array. In this particular example, the MXCU only needs to increase the address by one for every inner loop iteration. The LSU loads the input data array to the

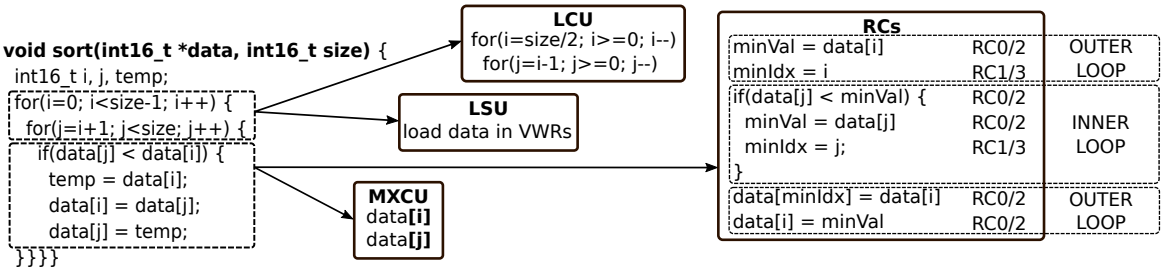


Figure 4.4: Sorting algorithm C code (left) and its high-level mapping on VWR2A (right).

foreground memory (i.e., the VWRs). Depending on the size of the array, the LSU may be used more than once.

Figure 4.4 details how the code is adapted for the architecture. Because the condition is simple, it can be executed locally by the RCs (i.e., without the LCU) by using the movs.sf instruction (see Section 4.3.1). The RCs assigned to each line of code are given on the right side of the block. For example, RCs 0 and 2 start by loading the data[i] value (minVal) in one of their internal registers, while RCs 1 and 3 store its index (minIdx). The array is split in two: RCs 0 and 1 sort the lower part of the array, and RCs 2 and 3 the upper part. The performance improvement of the parallelization offsets the overhead required at the end to recover the complete output. This is not always the case and should be evaluated for every kernel.

Table 4.4 compares the outer and inner loop size of the ascending sorting algorithm for different architectures: a Cortex-M4 processor (CM4), an Ibex processor, and the VWR2A instance. For the inner loop, the two cases for a *true* and a *false* if-condition are evaluated. The outer and inner loops are highlighted in the RCs block of Figure 4.4. While the outer loop is two instructions longer for the VWR2A instance than for the Cortex-M4 and the Ibex processors, the inner loop is much faster. In terms of performance, the latter is usually the most important. The performance improvement of the VWR2A instance comes from the instruction parallelization over the specialized slots (ILP) as depicted in Figure 4.4. Moreover, as long as there are no data dependencies over the iterations, the RCs can execute the condition independently using the mov.sf instruction (see Section 4.3.1).

[2]The number of iterations is reduced by 1 for every outer loop iteration.

Table 4.4: Sorting algorithm outer- and inner-loop length (in cycles) comparison between ARM Cortex-M4 (CM4), Ibex, and VWR2A architectures.

		CM4	**Ibex**	**VWR2A**
outer loop	#instructions	6	6	8
	#iterations	31	31	15
inner loop	#instructions true	10	9	2
	#instructions false	8	7	2
	#iterations[2]	30	30	15

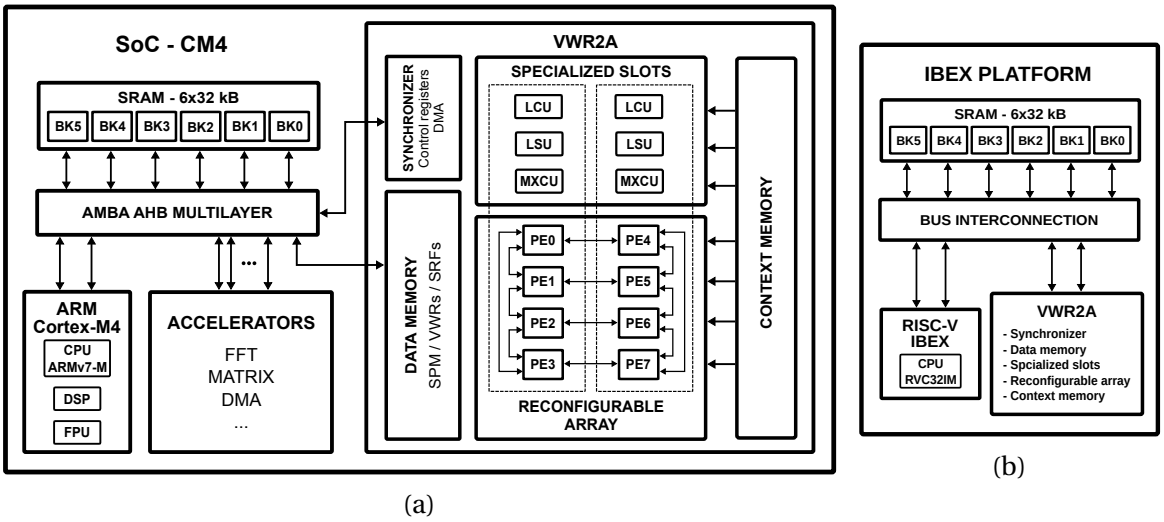
4.5 Experimental setup

4.5.1 Ultra-low power embedded platform

The same experimental setup as Chapter 3 is used: the VWR2A instance is integrated into an ultra-low power SoC, as shown in Figure 4.5a, intended for biomedical applications [32] (referred to as the ARM Cortex-M4 SoC in this chapter). The main features of that SoC are an ARM Cortex-M4 processor and 6 static random access memory (SRAM) banks of 32 KiB (192 KiB in total) that can be accessed in parallel and individually power gated. The platform has multiple custom accelerators (e.g., FFT, matrix processor); however, such fixed-function accelerators do not have the flexibility to execute control-intensive code.

In order to better demonstrate the efficiency of a programmable accelerator architecture, I also implemented a platform featuring an Ibex core, 192 KiB of SRAM divided into six banks, and the VWR2A instance of this chapter (Figure 4.5b) for comparison with the ARM Cortex-M4 SoC. The performance of the different implementations can be directly compared as the platforms are simulated at the cycle-accurate RTL level. However, the Ibex platform is simplified (i.e., not an actual SoC), and the energy numbers presented in Section 4.6 are lower-bound values. Therefore, a direct comparison between the ARM Cortex-M4 SoC and the Ibex platform is not completely fair. The main difference between the Ibex platform and the ARM Cortex-M4 SoC is the consumption of the bus interconnection. While the Ibex platform has a custom 3-masters and 7-slaves interconnection bus, the ARM Cortex-M4 SoC has an 19-masters and 14-slaves AMBA AHB multilayer interconnection bus.

Table 4.5 compares the main features of the Cortex-M4 and the Ibex processors. These are two representative cases of state-of-the-art programmable processors that target low-power devices, with the Ibex targeting specifically control-intensive workloads. As long as the code is dominated by control and the Ibex core can execute as fast as the Cortex-M4 processor, it will be more energy efficient because of its simpler architecture (even if the overhead in power mentioned previously is removed).



(a) (b)

Figure 4.5: (a) VWR2A instance implementation and integration inside a low-power SoC for biomedical applications [32]. (b) Ibex platform augmented with the VWR2A instance used for comparison.

Table 4.5: Main features comparison between the ARM Cortex-M4 and the RISC-V Ibex processors.

	ARM Cortex-M4	**RISC-V Ibex**
ISA	ARMv7-M	RVC32IM
pipeline	3 stages	2 stages
32-bit integer multiplier	1 cycle	3-4 cycles
32-bit integer divider	2-12 cycles	37 cycles
branch prediction	target forwarding	pipeline stall
FPU	hardware	NA (software)
Extension(s)	SIMD, 1 cycle MAC	NA

4.5.2 Performance and energy evaluation methodology

The ARM Cortex-M4 SoC and the Ibex platform, both including the VWR2A instance, have been synthesized with the TSMC $40\,\mathrm{nm}$ LP CMOS technology at $80\,\mathrm{MHz}$ (the original frequency of the ARM SoC). I ran post-synthesis simulations with Cadence Incisive Verification Platform [76] to get cycle-accurate execution times and measure the cell's switching activity used for power estimation with Synopsys PrimePower tool [78].

4.5.3 Representative set of software benchmarks for the biomedical target domain

In this Section, the link between the characteristics of the control-intensive benchmarks analyzed in Section 4.6 and the architectural features of the VWR2A instantiation are discussed. The software benchmarks are divided into standalone kernels and applications. Standalone kernels are extracted from existing biomedical applications to evaluate the architectures at the kernel level. The impact at the application level is evaluated on three biosignal applications. Only the processing steps are considered, not the acquisition phase. The standalone kernels are generic and can be included in applications of various domains, not just the biomedical one. For the evaluation of the standalone kernels on the VWR2A instance, the overhead time to transfer the data to its internal SPM (before kernel execution) and back to the SoC SRAM (after the execution) is accounted for, as well as the reconfiguration time (i.e., loading the corresponding kernel instructions from the context memory to the RCs and the specialized slots).

Control-intensive kernels spend most of their time executing control-related instructions. Table 4.6 shows this by comparing control- and data-intensive kernels. The proportion of arithmetic and logic operations (ALU column in Table 4.6) is not enough for distinguishing control-intensive from data-intensive code. However, the analysis of the assembly code produced for each kernel shows that most of these operations are devoted to controlling tasks (e.g., counter incrementation, conditions) for the former and direct computation on data to produce an output (e.g., input data multiplication, addition) for the latter. Load and store instructions hint at the type of code executed, and data-intensive kernels usually have a significant proportion of them. However, some control-intensive kernels, such as the median, can also have a high proportion of them. In the end, the best indicator is the number of branch instructions executed (e.g., branch-if-equal, jump, branch-if-not-equal). Table 4.6 shows a clear difference between control-intensive and data-intensive kernels. For this analysis, control-intensive code is defined as having more than $25\,\%$ of branch instructions at execution time.

Table 4.6: Control-intensive and data-intensive kernels' instructions profiling using the RVC32IM ISA.

Kernel		INSTRUCTION		
		ALU	**Load/Store**	**Branch**
Control-intensive	median	29 %	30 %	41 %
	delineation	54 %	19 %	27 %
Data-intensive	FFT radix-2	48 %	42 %	10 %
	FIR Filter	51 %	31 %	18 %

Three examples of control-intensive code in C language are shown in Figure 4.6. The respiration signal delineation (Figure 4.6a) and the morphological low-pass filter (Figure 4.6b) examples only show part of the code for conciseness. The FILT_WIN and WIN_SIZE variables of the morphological low-pass filter and dilation queue examples (Figures 4.6b, and 4.6c) are the sizes of the structuring elements: 5 and 75 elements, respectively.

Similarly to the sorting algorithm code analyzed in Section 4.4, these examples can be divided into four tasks: execution control, data loading and storing, data address update, and data computation. Each of these tasks corresponds to one specialized slot of the VWR2A architecture, which improves the ILP. Moreover, the RCs enable parallelization at the data level when possible. For example, lines 14 and 15 of the respiration delineation code (Figure 4.6a), which compute the input signal derivative, take two clock cycles as instructions can be parallelized over the slots. Moreover, in this particular case, the derivative computation can be parallelized by unrolling the loop over the 8 RCs as this kernel uses both columns.

Standalone kernels

The first kernel, *Median*, takes an input array of up to 28 values, sorts them in ascending order, and obtains the middle value (i.e., the median). As discussed in Section 4.4, this kernel is dominated by control instructions to reorder the array. The second kernel is *Delineation* (Figure 4.6a), which is a typical step of many biomedical applications that extracts the fiducial points of a signal (e.g., ECG, EEG). These points are later used to extract features. Delineation is a highly irregular code. It includes complex nested if-else structures that are not predictable because of their data dependency. Therefore, the workload cannot be parallelized, and the data are processed one by one. The *Median* and *Delineation* kernels are extracted from a cognitive workload estimation application [81], from the respiration (RSP) signal features extraction and delineation steps, respectively.

```
void rsp_delineation(int16_t *data) {
00:  static int16_t first_win=1, lastValleyIndexRSP=0;
01:  static int16_t lastSigDiff=1, lastVal=0;
02:  static int16_t sig_diff[DATA_LENGTH];
03:
04:  if (first_win==1) {
05:    sig_diff[0]=0;
06:  } else {
07:    sig_diff[0]=data[0]-lastVal;
08:  }
09:
10:  if(sig_diff[0]==0) {
11:    sig_diff[0]=lastSigDiff;
12:  }
13:
14:  for(int16_t i=1; i<DATA_LENGTH; i++) {
15:    sig_diff[i] = data[i]-data[i-1];
16:  }
17:
18:  for(int16_t i=1; i<DATA_LENGTH; i++) {
19:    if(sig_diff[i]>0) {
20:      sig_diff[i] = 1;
21:    } else if (sig_diff[i]<0) {
22:      sig_diff[i] = -1;
23:    } else {
24:      sig_diff[i] = sig_diff[i-1];
25:    }
26:  }
27:
28:  first_win=0; lastSigDiff=sig_diff[DATA_LENGTH-1];
29:  last_val=data[DATA_LENGTH-1];
30:
 ...
}
```

(a)

```
void mf_lp_filter(int16_t *data) {
00:  static int16_t B[FILT_WIN*FILT_WIN] = {
01:    0, 0, 1, 5, 1,
02:    1, 0, 0, 1, 5,
03:    5, 1, 0, 0, 1,
04:    1, 5, 1, 0, 0,
05:    0, 1, 5, 1, 0
06:  };
07:  static int16_t original[HF_FILT_250HZ_WIN];
08:  static int16_t dilation[HF_FILT_250HZ_WIN];
09:  static int16_t erosion[HF_FILT_250HZ_WIN];
 ...    ...
17:  for(int16_t i=0; i<DATA_LENGTH; i++) {
18:    int16_t* Bi = B + FILT_WIN*i;
19:    original[i] = data[i];
20:    dilation[i] = original[0] + Bi[0];
21:    erosion[i]  = original[0] - Bi[0];
22:    for (int16_t j=1; j<FILT_WIN; j++) {
23:      if (original[j] + Bi[j] > dilation[i]) {
24:        dilation[i] = original[j] + Bi[j];
25:      }
26:      if (original[j] - Bi[j] < erosion[i]) {
27:        erosion[i] = original[j] - Bi[j];
28:      }
29:    }
 ...    ...
40:  }
}
```

(b)

```
void mf_dilation_queue(int16_t *data) {
00:  static int16_t q_val[WIN_SIZE], q_pos[WIN_SIZE];
01:  static int16_t front=0, back=0;
02:
03:  for (int16_t i=0; i<DATA_LENGTH; i++) {
04:    while (front != back && data[i] > q_val[back]) {
05:      back = (back-1)&(WIN_SIZE-1);
06:    }
07:    back = (back+1)&(WIN_SIZE-1);
08:    q_pos[back] = i;
09:    q_val[back] = data[i];
10:    if (WIN_SIZE <=
11:          ((i - q_pos[(front+1)&(WIN_SIZE-1)])&(WIN_SIZE-1))) {
12:      front = (front+1)&(WIN_SIZE-1);
13:    }
14:    input = q_val[(front+1)&(WIN_SIZE-1)];
15:  }
}
```

(c)

Figure 4.6: Examples of control-intensive code in C language. (a) Respiration delineation (partial code). (b) Morphological filtering low-pass filter (partial code). (c) Morphological filtering dilation queue.

Morphological filters are used in many signal processing applications [88, 108]. In the biomedical domain, MF is used for two purposes: biosignal baseline removal and filtering (e.g., low-pass filter) [108]. Biosignals often contain low-frequency harmonics known as the baseline. MF can be used to remove this baseline by sequentially applying an erosion, a dilation (Figure 4.6c), and again an erosion filter. The output is the baseline and can be subtracted from the original input. The implementation uses three queues (one per morphological filter), which require many control instructions. The data-dependent nature of a queue does not allow efficient parallelization at the data level. Therefore the input data are processed sample by sample. The analyzed VWR2A instance can manage the three queues on its own, thanks to its flexibility. This data-structure based functionality is quite unusual for hardware accelerators but is used in many applications, which creates huge opportunities for programmable architectures such as VWR2A.

The second example code using MF is a low-pass filter that uses closing and opening morphological operators to produce a low-pass filter (Figure 4.6b) [108]. Due to the small size of the structure element (i.e., the weights matrix), the implementation does not use queues and can be parallelized at the data level with a small computation penalty to recover the correct output. The two MF kernels are extracted from a heartbeat classifier application [108].

Biosignal applications

The first application is a heartbeat classifier that acquires a 3-leads ECG signal and classifies the beats into normal and abnormal to detect arrhythmia [108]. The beat classification is performed on a single lead. If a heartbeat is classified as abnormal, the two additional leads are also processed. Therefore, the application processes either a 1-lead or a 3-leads ECG signal, referred to as *1-lead* and *1+2-leads*, respectively, in Section 4.6. Depending on the health condition of the monitored patient, the second scenario (*1+2-leads*) is activated more or less often. This application uses the MF baseline removal and the MF low-pass filter kernels to preprocess the ECG signals before executing the rest of the application (classifier, features extraction, ...). The results are shown for the processing time of one window of 768 samples (3 seconds of ECG signal).

The second application processes the complete set of 12-leads ECG signals as required for medical standards (e.g., devices in hospitals). This application applies the MF baseline removal and low-pass filtering kernels before combining the leads and extracting the fiducial points. The results are also shown for the processing time of one window of 768 samples.

The last application, which contains the *Median* and the *Delineation* kernels, is a multi-signal cognitive workload estimation application [81]. For this chapter, I focused on analyzing the performance and energy consumption of these two control-intensive kernels at the application level. The performance of the related VWR2A instance on the data-intensive kernels has already been presented in Chapter 3. The results correspond to the processing of a window of 512 samples (102 seconds of respiration signal).

4.6 Experimental results

4.6.1 Architecture power analysis

The power breakdown of the architecture is first evaluated, notably the power consumption of the specialized slots optimized for control-intensive code in order to assess their low-power nature and understand better the architecture's performance and energy efficiency. Table 4.7 shows the average and standard deviation of the power breakdown of the VWR2A components when executing the standalone kernels that use two columns of the architecture (i.e., *Median 2x*, *Delineation*, *MF – Baseline removal 2-leads*, and *MF – Low-pass filter 2-leads*). The RCs and the data memories consume $87.5\,\%$ of the total power on average, while the specialized slots (i.e., LCUs, LSUs, and MXCUs) account only for $7.9\,\%$. The energy overhead of the specialized slots is minimal, therefore justifying their

Table 4.7: Average power consumption and breakdown of kernels that use two columns for the components of the VWR2A instance focusing on control-intensive.

VWR2A Component	Power (µW) avg	%	σ
Context memory (5.4 KiB)	64	2.7 %	±1.3 %
Synchronizer	45	1.9 %	±14.2 %
LCUs	46	2.0 %	±34.0 %
LSUs	84	3.6 %	±21.7 %
MXCUs	55	2.3 %	±25.0 %
RCs	407	17.4 %	±28.2 %
ALUs	(182)	(7.8 %)	(±34.8 %)
instruction memories	(123)	(5.2 %)	(±30.7 %)
RCs interconnection	(69)	(3.0 %)	(±34.7 %)
register files	(33)	(1.4 %)	(±34.1 %)
Data memory (35 KiB)	1644	70.1 %	±21.5 %
SPM (32 KiB)	(636)	(27.1 %)	(±8.8 %)
VWRs (3 KiB)	(1000)	(42.7 %)	(±31.5 %)
SRFs (64 B)	(8)	(0.3 %)	(±106.8 %)
Total	2344	100 %	±20.6 %

use as they drastically increase the performance by improving ILP. Without them, the RCs would have to handle also their instruction flow, increasing the overall number of cycles to execute a kernel.

In addition, the specialized slots are more efficient than the RCs. Table 4.8 compares the average power consumption of different processing elements while executing control-intensive code. First, this table shows that RCs offer a high power efficiency compared to GPPs as their total power consumption is 9.3 × lower than the ARM Cortex-M4 (CM4), for example. The specialized slots further improve the power efficiency of VWR2A compared to a traditional CGRA by providing processing elements more energy efficient than the RCs to execute specific tasks. The LCU, LSU, and MXCU consume on average 2.2 ×,1.2 ×, and 1.2 × less power than an RC, and 20.4 ×,11.3 ×, and 11.3 × less power than the CM4.

The RCs power consumption is divided between the ALUs, the internal instruction memories —64 words of 17 bits per RC—, the reconfigurable array interconnection, and the internal register files —2 words of 32 bits per RC. The datapaths (i.e., ALUs, interconnection, and register files) account for 69.8 % of the RCs power consumption. This shows that most of the RCs' power is consumed for actual data computation (i.e., the datapaths).

Table 4.8: Average power consumption comparison of the ARM Cortex-M4 (CM4), an RC, and the specialized slots (LCU, LSU, and MXCU) considering the *Median 2x*, *Delineation*, *MF – Baseline removal 2-leads*, and *MF – Low-pass filter 2-leads* kernels.

	CM4	**RC**	**LCU**	**LSU**	**MXCU**
Average Power (uW)	471[3]	51[4]	23[5]	42[6]	42[7]
Power reduction	$1.0\times$	$9.3\times$	$20.4\times$	$11.3\times$	$11.3\times$

The context memory, which holds the kernel configuration words, is implemented using standard-cell flip-flops to perfectly match the bitwidth of the units' configuration words (e.g., 27 bits for the MXCUs). These flip-flops are active only during the configuration phase of the columns and clock-gated during the execution of a kernel. The latter takes at least a few thousand cycles, while the configuration phase takes a maximum of 64 cycles. Therefore, the context memory power consumption is very small.

The data memory hierarchy consumes $70.1\,\%$ of the total power on average. While the SPM uses memory macros, the VWRs are implemented with standard cell libraries' latches. This partially explains why the VWRs represent $42.7\,\%$ of the total power consumption while the SPM accounts only for $27.1\,\%$. A custom design where larger cells and a pragma-guided regular floorplan are imposed, as proposed in [48, 72, 109], would undoubtedly reduce their power consumption further. The power consumption of the SRFs is almost negligible because of their relative small size compared to the SPM and VWRs.

The average power of each component as shown in Table 4.7 presents a marked variability, which is due to the specific instructions executed by each kernel on each slot. For example, the duty-cycle of the LCU is $64\,\%$ for the *Median 2x* kernel, whereas it reaches $76\,\%$ for the *MF – Baseline removal 2-leads* kernel. This difference translates into a higher relative average power for the LCU when executing the latter. Nevertheless, the absolute average power of the specialized slots is so small that the variation on the total system power is very small.

The SRFs have the highest variability, $\pm 106.8\,\%$, because some kernels, like the *Delineation*, rely mostly on the SRF of one column only. However, the power consumption of the SRFs has almost no impact at the system level because it represents only $0.3\,\%$ of the total power on average. The $\pm 20.6\,\%$ variation at the architecture level is mainly dependent on the variability of the VWRs power consumption. Some kernels, like the *Median 2x*, do not

[3] Only the ARM Cortex-M4 processor, not the full SoC.
[4] RCs average power consumption of Table 4.7 (407) divided by 8 RCs.
[5] LCUs average power consumption of Table 4.7 (46) divided by 2 LCUs.
[6] LSUs average power consumption of Table 4.7 (84) divided by 2 LSUs.
[7] MXCUs average power consumption of Table 4.7 (55) divided by 2 MXCUs.

access the VWRs so much, while others heavily use them, like the *MF – Baseline removal 2-leads* that stores not only the input data but the queues as well.

4.6.2 Standalone kernels results

Tables 4.9 and 4.10 show the performance and energy consumption for the four evaluated architectures on seven computational kernels present in biomedical applications. The VWR2A instance exhibits similar performance on both the Cortex-M4 SoC and the Ibex platform as their memory and bus architecture are similar (only the processors differ).

Median

The mapping of the *Median* kernel uses only one column of the reconfigurable array. Two cases are evaluated: 1 median executed in one column (the other one is idle) and two medians computed in parallel over two columns. The latter is the best case for the VWR2A instance as the execution time is almost similar to computing one median, improving its energy efficiency compared to the processors.

The good performance of the VWR2A architecture is mainly due to the improvements in the sorting algorithm discussed in Section 4.4. Table 4.4 shows that the outer and inner loops of the sorting algorithm have two times fewer iterations than the Cortex-M4 and the Ibex implementations, entirely due to the architectural innovations of the VWR2A template (i.e., for the same algorithm). Moreover, the inner loop size is only two instructions, while the Cortex-M4 and the Ibex use at least 8 and 7 instructions, respectively. This should give approximately a $16\times$ performance improvement for the VWR2A instance. However, the overhead of the acceleration request from the host processor and the data transfer limit its overall performance. Moreover, the parallelization of the sorting algorithm on the VWR2A instance requires a recovery step (executed on the VWR2A instance) to get the correct output, reducing the performance improvement to a final factor of $8.5\times$ and $9.6\times$ compared to the Cortex-M4 and the Ibex, respectively, for the *Median 2x* case.

Delineation

This kernel uses the two columns of the accelerator. The first step of the kernel consists of the input signal derivative computation and extraction of local minimums and maximums. This step is easily parallelizable at the algorithmic and instruction level, which translates to a $19\times$ performance improvement compared to the Cortex-M4. The second step consists of analyzing the local minimums and maximums and selecting the valid ones. This is done

Table 4.9: Standalone kernels performance and energy comparison for the ARM Cortex-M4 SoC (**CM4**) and the ARM Cortex-M4 SoC including the VWR2A instance (**CM4+VWR2A**).

Cycles	**CM4**	**CM4 + VWR2A**	**CM4 vs. CM4+VWR2A**
Median 1x	5015	1098	$4.6\times$
Median 2x	10 007	1183	$8.5\times$
Delineation	32 113	2723	$11.8\times$
MF – Baseline removal 1-lead	98 228	30 664	$3.2\times$
MF – Baseline removal 2-leads	205 768	31 577	$6.5\times$
MF – Low-pass filter 1-lead	96 784	8642	$11.2\times$
MF – Low-pass filter 2-leads	191 515	10 121	$18.9\times$
Average speed-up			$\mathbf{9.2}\times$
Energy (µJ)			
Median 1x	8.23×10^{-2}	3.05×10^{-2}	$-62.9\,\%$
Median 2x	1.70×10^{-1}	3.97×10^{-2}	$-76.7\,\%$
Delineation	5.74×10^{-1}	1.30×10^{-1}	$-77.3\,\%$
MF – Baseline removal 1-lead	1.72×10^{0}	8.79×10^{-1}	$-48.9\,\%$
MF – Baseline removal 2-leads	3.70×10^{0}	1.35×10^{0}	$-63.5\,\%$
MF – Low-pass filter 1-lead	1.70×10^{0}	3.10×10^{-1}	$-81.8\,\%$
MF – Low-pass filter 2-leads	3.36×10^{0}	5.28×10^{-1}	$-84.3\,\%$
Average energy saving			$\mathbf{-70.8}\,\%$

through a three-level nested if-else structure. This structure selects the next valid point based on data-dependent conditions (see Figure 4.1), which limits parallelization at the algorithmic level. However, this step can take advantage of the ILP of the architecture and achieves an $8\times$ performance improvement compared to the Cortex-M4. This translates to an overall $11.8\times$ performance improvement at the kernel level compared to the Cortex-M4 (Table 4.9). For the Ibex platform, the results are very similar.

MF - Baseline removal

The morphological filter for baseline removal is the kernel in which the VWR2A instance obtains the lowest speed-up with respect to the two CPUs. The reason is that the implementation of the queues is only parallelized at the instruction level, as the algorithm cannot be parallelized at the data level without significant overhead. The mapping is limited to one column, so two executions can run in parallel, improving performance. The two scenarios, 1-lead and 2-leads, are reported in Tables 4.9 and 4.10, using one, or two columns, respectively. As this kernel is the worst in terms of speed-up, it also gives

Table 4.10: Standalone kernels performance and energy comparison for the Ibex platform (**Ibex**) and the Ibex including the VWR2A instance (**Ibex+VWR2A**).

Cycles	**Ibex**	**Ibex + VWR2A**	**Ibex vs. Ibex+VWR2A**
Median 1x	6101	1087	5.6 ×
Median 2x	12 169	1264	9.6 ×
Delineation	32 194	2713	11.9 ×
MF – Baseline removal 1-lead	103 288	30 448	3.4 ×
MF – Baseline removal 2-leads	208 611	31 212	6.7 ×
MF – Low-pass filter 1-lead	152 333	8523	17.9 ×
MF – Low-pass filter 2-leads	304 649	9783	31.1 ×
Average speed-up			**12.3** ×
Energy (µJ)			
Median 1x	8.00×10^{-2}	2.53×10^{-2}	−68.4 %
Median 2x	1.60×10^{-1}	3.51×10^{-2}	−78.0 %
Delineation	4.74×10^{-1}	9.03×10^{-2}	−81.0 %
MF – Baseline removal 1-lead	1.52×10^{0}	6.85×10^{-1}	−54.9 %
MF – Baseline removal 2-leads	3.08×10^{0}	9.79×10^{-1}	−68.2 %
MF – Low-pass filter 1-lead	1.63×10^{0}	2.24×10^{-1}	−86.3 %
MF – Low-pass filter 2-leads	3.27×10^{0}	3.57×10^{-1}	−89.1 %
Average energy saving			**−75.1** %

the lowest energy savings for the VWR2A instance compared to the ARM Cortex-M4 SoC and the Ibex platform: 48.9 % and 54.9 %, respectively, for the 1-lead case, and 63.5 % and 68.2 %, respectively, for the 2-leads case.

MF - Low-pass filter

This kernel also uses a single column, and the results for the cases of 1-lead, using one column, and 2-leads, using two columns, are reported (Tables 4.9 and 4.10). This kernel is the best in terms of performance and energy savings for the VWR2A instance because it contains more computation on data than the other kernels, although it is still dominated by control instructions. Moreover, because of the small size of the morphological structuring elements, an implementation using queues is not the most efficient. Therefore, a straightforward implementation that can be parallelized with a small overhead on the VWR2A instance is used. This translates into energy savings up to 84.3 % and 89.1 % for the 2-leads case compared to the ARM Cortex-M4 SoC and the Ibex platform, respectively.

4.6.3 Standalone kernels analysis

The VWR2A instance is the fastest implementation for all the kernels. This corroborates that an adequately designed reconfigurable architecture can also accelerate control-intensive kernels, in this case, when optimized for the biomedical domain. In terms of energy, although the VWR2A instance uses more power than the two considered processors, its speed-up translates to significant savings in energy consumption, particularly when the complete architecture is used (i.e., the two columns).

While the increase in energy consumption is equal to the increase in execution time for the CM4 and the Ibex (e.g., executing two leads requires two times more cycles and energy), it is not valid for the VWR2A instance. If the extra work (e.g., a second lead) can be executed in parallel by another column, the energy consumption will not increase by a factor of two. The reason is that some elements are shared, such as the SPM. Moreover, the leakage power of both columns is always present even when a single column is used. Utilizing the second column only adds its dynamic power to the overall consumption. These reasons explain why the CM4 SoC and the Ibex platform double their energy consumption when executing the *MF – Baseline removal 2-leads* and *MF – Low-pass filter 2-leads* kernels compared to the single lead versions, while the CM4+VWR2A only increases its energy consumption by a factor $1.6\times$.

As only the kernels themselves are studied here, the performance and energy consumption of the VWR2A instance can be analyzed in isolation and compared directly with the execution on the two CPUs. Compared to the Cortex-M4 processor (not the SoC, only the CPU), the VWR2A instance alone uses $5.8\times$ more power but is $9.2\times$ faster on average, hence it is more energy efficient. Moreover, the VWR2A instance uses its internal memory structure to access the data (i.e., SPM and VWRs), which is more energy-efficient than the SoC AMBA AHB interconnection. This leads to a $70.8\,\%$ energy saving on average at the SoC level, which is combined with the higher performance leading to an energy-delay product (EDP) improvement of $43.8\times$.

Compared to the Ibex core (only the CPU), the VWR2A instance alone requires $8.7\times$ more power but is $12.3\times$ faster, leading to an average energy saving of $75.1\,\%$ at the platform level (Ibex+VWR2A vs. Ibex). This translates to an EDP improvement of $81.2\times$ for the Ibex+VWR2A platform compared to the Ibex platform. Interestingly, although the Ibex processor is optimized for control-intensive code and has better energy efficiency than the ARM Cortex-M4 SoC, it is on average $1.2\times$ slower, translating to an EDP decrease of $1.1\times$ for our biosignal processing target domain.

4.6.4 Biosignal applications results

The performance and energy consumption of the three evaluated designs are shown in Tables 4.11 and 4.12. The code executed on the VWR2A instance was limited to the control-intensive kernels presented in Tables 4.9 and 4.10 to evaluate the impact of control code acceleration at the application level. However, it has been substantiated in Chapter 3 that a VWR2A instance can also execute most of the data-intensive kernels of the evaluated applications. Therefore, the numbers reported in Tables 4.11 and 4.12 are the minimum energy savings that can be achieved using the VWR2A instance of this chapter (Section 4.6.6 evaluates the case where both data- and control-intensive kernels are executed by a VWR2A instance).

Heartbeat classifier

Two scenarios have been considered: 1-lead and 1+2-leads. The second one is the optimal case for the VWR2A instance, as it can execute the two leads in parallel. For the first case (1-lead), only half of the VWR2A architecture is used (i.e., one column), reducing its energy efficiency. The execution is dominated by the MF —baseline removal and low-pass filter— of the leads, which takes more than $65\,\%$ and $75\,\%$ of the total processing time, for the 1-lead and 2-leads cases, respectively (for both the ARM Cortex-M4 SoC and the Ibex platform). At the application level, gains are limited compared to those at the kernel level (see Tables 4.9 and 4.10). The reason is that the lead window size is bigger than the one used for the standalone kernels, and the SPM of the VWR2A instance is not large enough to hold all the data simultaneously, requiring some backup to the platform SRAM (using a double-buffering strategy). This increases the bus interconnection energy consumption and limits the gain of the VWR2A instance compared to both processors. Moreover, the acceleration is limited to the control-intensive kernels of the application while the processors execute the remaining part, limiting the overall savings of the VWR2A instance.

Medical standard 12-leads

Like the previous application, executing the morphological filter for 12 leads requires some adaptation as the VWR2A internal memory is not large enough to store all the values (input data and kernel parameters) at once. Therefore, additional data transfers are needed to back up these values in the ARM Cortex-M4 SoC and the Ibex platform SRAM. This induces a penalty in performance and energy but using the VWR2A instance is still more energy-efficient. This application is even the best regarding energy savings: the VWR2A

Table 4.11: Biosignal applications performance and energy comparison for the ARM Cortex-M4 SoC (**CM4**) and the ARM Cortex-M4 SoC including the VWR2A instance (**CM4+VWR2A**). The VWR2A instance is limited to execute only control-dominated kernels.

Cycles	**CM4**	**CM4 + VWR2A**	**CM4 vs. CM4+VWR2A**
Heartbeat classifier 1-lead (ECG)	463 025	208 525	2.2 ×
Heartbeat classifier 1+2-leads (ECG)	1 278 191	465 366	2.7 ×
Medical standard 12-leads (ECG)	4 261 659	894 567	4.8 ×
Cognitive workload est. 1-lead (RSP)	153 115	123 122	1.2 ×
Average speed-up			**2.7** ×
Energy (µJ)			
Heartbeat classifier 1-lead (ECG)	4.4	3.9	−11.6 %
Heartbeat classifier 1+2-leads (ECG)	10.9	8.6	−20.7 %
Medical standard 12-leads (ECG)	40.3	25.1	−37.7 %
Cognitive workload est. 1-lead (RSP)	2.3	1.9	−17.7 %
Average energy saving			**−21.9** %

instance consumes 37.7 % and 71.6 % less energy than the ARM Cortex-M4 SoC and the Ibex platform, respectively. This is due to the even number of leads, which maximizes the VWR2A utilization and, therefore, its energy efficiency.

Cognitive workload estimation

This application is the worst in terms of overall performance and energy savings, particularly compared to the Ibex platform, because it is dominated by data-intensive kernels (as discussed previously) and, given the focus of this chapter, here our study is limited to accelerate only the control-intensive kernels (*Median* and *Delineation kernels*). These kernels account for less than 5 % of the total processing time on the Ibex platform, limiting the impact of the VWR2A instance. For the ARM Cortex-M4 SoC, these kernels represent 21 % of the processing time, which translates to higher savings for the VWR2A instance. For such application, having a programmable accelerator capable of accelerating both data-intensive and control-intensive kernels would be the best. This scenario is evaluated later in this chapter to show further the benefit of having a DSIP with broad kernel coverage of a domain. Nevertheless, the VWR2A instance still exhibits 17.7 % and 3.7 % lower energy consumption than the ARM Cortex-M4 SoC and the Ibex platform, respectively.

Table 4.12: Biosignal applications performance and energy comparison for the Ibex platform (**Ibex**) and the Ibex including the VWR2A instance (**Ibex+VWR2A**). The VWR2A instance is limited to execute only control-dominated kernels.

Cycles	**Ibex**	**Ibex + VWR2A**	**Ibex vs. Ibex+VWR2A**
Heartbeat classifier 1-lead (ECG)	672 639	289 881	2.3 ×
Heartbeat classifier 1+2-leads (ECG)	1 773 784	579 513	3.1 ×
Medical standard 12-leads (ECG)	10 132 440	1 003 344	10.1 ×
Cognitive workload est. 1-lead (RSP)	645 677	616 570	1.0 ×
Average speed-up			**4.1** ×
Energy (µJ)			
Heartbeat classifier 1-lead (ECG)	7.4	3.7	−50.5 %
Heartbeat classifier 1+2-leads (ECG)	20.4	8.6	−58.0 %
Medical standard 12-leads (ECG)	78.4	22.2	−71.6 %
Cognitive workload est. 1-lead (RSP)	8.4	8.1	−3.7 %
Average energy saving			**−46.0** %

4.6.5 Biomedical applications analysis

Compared to the results of the standalone kernels, the savings at the application level are more limited. When complete applications are considered, there are inevitable data-intensive steps that significantly worsen the performance and energy efficiency, particularly for the Ibex core, which is optimized for low-power execution of control code. This is particularly true for the *Cognitive workload estimation* application that is dominated by data-intensive kernels, such as an FFT and a FIR filter. At the kernel level, the Ibex core is slower but more energy efficient than the ARM Cortex-M4, but at the application level, it is both slower and less energy efficient with an EDP 3.7 × lower.

Similar to the case of standalone kernels, the VWR2A instance is the best implementation in terms of performance and energy. The VWR2A instance is 2.7 × and 4.1 × faster than the ARM Cortex-M4 SoC and the Ibex platform, respectively, on average. In terms of energy, the VWR2A instance consumes 21.9 % and 46.0 % less energy on average than the ARM Cortex-M4 SoC and the Ibex platform, respectively. The performance and energy combined give the VWR2A instance an improvement in EDP of 3.8 × and 12.2 × compared to the ARM Cortex-M4 SoC and the Ibex platform, respectively.

Table 4.13: Biosignal application performance and energy comparison when a VWR2A instance is used for all the possible data- and control-intensive kernels.

Cycles	**CM4**	**CM4 + ASICs**	savings	**CM4 + VWR2A**	savings
Preprocessing	49 760	34 402	30.9 %	2569	94.8 %
Delineation	32 716	32 716	0.0 %	2723	91.7 %
Feat. extraction + SVM	70 639	54 255	23.2 %	7915	88.8 %
Total	153 115	121 373	20.7 %	13 207	91.4 %
Energy (µJ)					
Preprocessing	0.68	0.47	30.7 %	0.15	77.3 %
Delineation	0.54	0.54	0.0 %	0.13	75.9 %
Feat. extraction + SVM	1.1	0.98	10.9 %	0.38	65.5 %
Total	2.32	1.99	14.2 %	0.66	71.4 %

4.6.6 Experiment including both data- and control-intensive code

The *Cognitive workload estimation* application (also used in Chapter 3) has been entirely ported to the VWR2A instance of this chapter to show the full potential of such an architecture. In this case, all the application was executed by the VWR2A instance, further improving the savings at the application level, as shown in Table 4.13. The experiments have been done only with the ARM Cortex-M4 SoC, as the application is dominated by data-intensive kernels (79 % of the total execution time using only the ARM Cortex-M4 core). Executing this application on the Ibex platform will undoubtedly translate to poor performance and energy efficiency (for the Ibex, not the VWR2A instance). Compared to the ARM Cortex-M4 SoC (i.e., using its complete set of ASICs), the VWR2A instance is $9.2\times$ faster and consumes 66.7 % less energy. This translates to an improvement in EDP of $27.6\times$. Compared only to the ARM Cortex-M4 processor (i.e., without the ASICs), the improvement in EDP is $40.5\times$.

These results show the importance of evaluating architectures at the application level and the advantages of the flexibility of a reconfigurable accelerator, allowing both data-intensive and control-intensive code to be executed more efficiently, in terms of performance and energy, than GPPs and their potential set of ASICs.

4.6.7 Template instance optimization

The VWR2A instance used in this chapter is very similar to the one proposed in Chapter 3. The specific features added in this chapter significantly impact the architecture capacity and efficiency to execute control-intensive kernels, increasing its code coverage while having a negligible impact on the overall power consumption. First, this shows the advantage of a programmable architecture that can adapt to different types of code with minor changes. The instance of Chapter 3 is already capable of executing some of the control-intensive kernels presented here, but not all of them (i.e., the MF kernels), and not as efficiently as the instance proposed in this chapter. Therefore, these features should be included inside an instance focusing on data-intensive code as they enlarge the covered domain of applications with negligible hardware overhead.

However, the opposite is not necessarily true. For example, the single-cycle multiplier's huge area and leakage power consumption (not used in any control-intensive kernels) can be removed in an instance targeting control-dominated applications only. Such an instance would still support multiplication but with a higher cycle penalty, hence the execution of data-intensive kernels but with lower energy efficiency. This design choice depends on a deeper study of the application domain and a use case analysis to ensure such an instance would cover enough scenarios. This thesis does not provide this analysis. Nonetheless, the main parameters that can be optimized for an instance of the VWR2A instance focusing on control-dominated applications are discussed below.

As shown in Table 4.7, the RCs and the data memories, particularly the VWRs and the SPM, account for most of the power consumption of the VWR2A instance. Therefore, changing the template's parameters related to them will impact the power consumption most. One option is to reduce the SPM size, but the minimum size depends on the actual requirement of the applications. For the applications considered in this chapter, the SPM is actually too small, and a double-buffering scenario is needed for the *Heartbeat classifier* and the *Medical standard 12-leads* applications. In particular, the queues of a morphological filter require a significant amount of memory, which is multiplied by the number of leads. Therefore, if applications with a high number of leads are considered (e.g., in medically certified devices), a bigger SPM may be a better option. In this case, as discussed in Chapter 2, when a platform integrating a VWR2A instance is built from scratch, it is essential to properly size the system's main memory and the SPM size of the VWR2A instance. In this chapter, the SoC used as a baseline has been originally designed with a $192\,\mathrm{KiB}$ main memory, and no co-optimization of the complete SoC has been done with the integration of the VWR2A instance. Considering the VWR2A instance, it would be better to increase the SPM size (e.g., to $64\,\mathrm{KiB}$) and reduce the main memory (e.g., to $160\,\mathrm{KiB}$). Moreover, building the SPM with multiple banks and enabling power-gating

Table 4.14: Statistics of the units for all the control-intensive kernels.

	LCU	**LSU**	**MXCU**	**RC0**	**RC1**	**RC2**	**RC3**
avg instructions[8]	26	10	17	19	16	13	15
min instructions[9]	3	3	2	4	4	4	4
max instructions[10]	51	18	50	36	32	32	37
avg active instructions[11]	57.1 %	24.3 %	39.8 %	49.4 %	42.1 %	34.1 %	38.6 %
avg/min/max kernel size[12]	44 / 22 (mf_baseline_removal) / 64 (mf_erosion)						

(which has not been considered in the experiments) would allow more flexibility and adaptability to the requirements of various applications.

The other optimizable parameter with a significant impact on power is the width of the VWRs. For example, dividing the VWRs width by two is possible for control-intensive applications with a negligible impact on performance because the data transfer frequency between the VWRs and the SPM is very low. This would reduce power and energy consumption (as the impact on performance is small). However, depending on the complete set of targeted applications, changing the VWRs' size will impact their performance if data-intensive kernels are present. The final choice must consider the complete set of applications targeted by a specific instance of the VWR2A template.

Table 4.14 shows some statistics related to the instructions and program memory of the units. This table provides insights into the important template's parameters to optimize and their impact on performance. One crucial parameter is the operations supported by the ISAs of the processing elements (i.e., the specialized slots and the RCs). One example is the selection operation of the RCs that uses the status flags to select one of the input operands. The number of RCs is also important, but not as much as for data-intensive kernels, and having more than 4 RCs per column is irrelevant. The average active instructions[13] row of Table 4.14 shows that the RCs mostly have nop instructions, particularly RC2 and RC3, as the workload is primarily mapped into RC0 and RC1. Moreover, most control-intensive kernels use only one column, while the opposite is true for the data-intensive ones (mainly to increase the number of RCs, hence the computation resources). However, having multiple columns is still interesting for control-intensive code, especially for multi-lead applications that can be parallelized per lead. A more efficient design, in

[8]Static average number of instructions that are not a nop.
[9]Static minimum number of instructions that are not a nop.
[10]Static maximum number of instructions that are not a nop.
[11]Average percentage of instructions that are not nops based on each kernel length (not the total program memory size of 64 instructions).
[12]This value correspond to the highest PC address in which an active instruction is stored, not to the maximum numbers of active instruction in a unit.
[13]This row is not a profiling of the execution but just a measure of the content of the units' program memory. Therefore, a low percentage does not necessarily mean that the unit *executes* nops most of the time.

this case, would probably be an instance with more columns and potentially fewer RCs per column, for example, a two-by-four (i.e., four columns of two RCs).

Table 4.14 shows that the MXCU and the LCU are also crucial for control-intensive kernels, while the LSU is less used. Therefore, optimizing the parameters related to the LCU and MXCU have the most effect on performance. The *branchMode* control bit proposed in this chapter (see 4.3.2) is one example. Moreover, as shown in Table 4.7, the specialized slots represent a tiny portion of the total power consumption but provide a significant performance benefit. Therefore any feature (that does not incur a notable power overhead) or operation increasing their performance should also improve the overall energy efficiency.

4.7 Summary and conclusions

This chapter has evaluated the performance and savings in energy consumption of an instance of the VWR2A template with two general-purpose baseline processors for control-intensive kernels and applications. The VWR2A instance is similar to the one proposed in Chapter 3 but has been augmented with some features to execute control-intensive code better. This perfectly illustrates the purpose of the template. The latter defines the high-level features that a DSIP targeting the biomedical domain should have, but detailed optimizations of the architecture that depend on specific requirements are done at the instance level.

The results show that the DSIP instance is consistently faster and more energy-efficient than two representative processors of state-of-the-art programmable processors targeting low-power devices —an ARM Cortex-M4 processor and a RISC-V lowRISC Ibex processor— for kernels and applications. The specialized slots play a crucial role, particularly the MXCU and the LCU, and significantly improve the architecture performance on control-dominated kernels by increasing the ILP of the VWR2A instance. In detail, the VWR2A instance is $2.7\times$ faster and consumes $21.9\,\%$ less energy than the ARM Cortex-M4 SoC. Compared to the Ibex platform, it is $4.1\times$ faster and consumes $46.0\,\%$ less energy. This translates to an improvement in EDP of $3.8\times$ and $12.2\times$ compared to the ARM Cortex-M4 SoC and the Ibex platform, respectively.

The improvements above are lower bound values as the kernels executed on the VWR2A instance have been limited to the control-intensive ones. One application has been entirely ported, both the data- and control-intensive kernels, to the VWR2A instance to show its full potential. In this case, the savings are even higher, with an execution $9.2\times$ faster while consuming $66.7\,\%$ less energy compared to the ARM Cortex-M4 SoC (using its complete

set of ASICs). This translates to an improvement in EDP of $27.6\times$. Compared only to the ARM Cortex-M4 processor (i.e., without the ASICs), the improvement in EDP is $40.5\times$.

These results, similar to the previous chapters, strongly support the proposal of this thesis to move toward DSIP architectures in order to achieve higher performance and energy efficiency at the edge. In particular, the broad code coverage of DSIPs is the key to improving energy efficiency in future heterogeneous embedded systems.

5 Conclusions and Future work

5.1 Summary and contributions

In this thesis, I advocate for the design of domain-specific instruction-set processors (DSIPs) to improve embedded systems' energy efficiency and lower their development costs. DSIPs are fully programmable heterogeneous processors that do not contain any customized components and remain fully programmable for any source code that could be targeted. The main difference with a general-purpose processor (GPP) is that the compiled assembly code runs more efficiently for the target domain than for code outside this domain. A DSIP integrated inside a system-on-chip (SoC) as a co-processor allows large portions of applications within its target domain to be accelerated efficiently. This translates to higher savings at the application level than more customized circuits, such as application-specific integrated circuits (ASICs).

In Chapter 1, I have contextualized this work into the biomedical domain and showed how actual problems, such as noncommunicable diseases (NCDs) monitoring and prevention, could benefit from the architecture exploration presented in this thesis. It also summarizes the main contributions of all the chapters and provides an outline of this thesis to help readers navigate this work.

In Chapter 2, I have proposed VWR2A: a very-wide-register reconfigurable-array template to build DSIPs for the biomedical domain. The main architectural features of the template are based on the state-of-the-art in different domains: the memory hierarchy (from many studies in embedded systems), computational density (from coarse-grained reconfigurable arrays (CGRAs)), instruction decoding efficiency (from CGRAs), and increased instruction parallelism (from very-long instruction word (VLIW) processors). In particular, these features are:

- A compute fabric core based on the CGRA template that uses reconfigurable cells (RCs) with a high computational density.

- Specialized slots, which are loosely inspired by the brain regions and their focus on specific tasks (e.g., motor control, vision) and similar to the functional units (FUs) of a VLIW processor to increase the instruction level parallelism (ILP) by freeing the RCs from executing simple instructions unrelated to the actual data processing (e.g., loop control, branches). These specialized slots can perform the simple operations for which they are designed using less energy than the RCs.
- An energy-efficient instruction decoding scheme based on the CGRA template with a context memory containing pre-decoded instructions loaded to local program memories close to the processing elements (i.e., the RCs and the specialized slots).
- A low-energy wide memory hierarchy including:
    1. A scratchpad memory (SPM) to reduce energy consumption by targeting most data accesses to a small memory closer to the processing elements (i.e., the RCs).
    2. Wide memories —an SPM and very-wide registers (VWRs)— with reduced decoding energy, high bandwidth, and low switching in large output transistors.
    3. Single-ported memories (the SPM and VWRs) to reduce their energy consumption to a minimum.

The principal parameters of the template have been presented and illustrated with concrete examples. The kernel mapping process and the constraint of the memory hierarchy have been discussed with a simple two-vector element-wise addition example. A more complex example using a fast Fourier transform (FFT) kernel, typical in biomedical applications, is also evaluated. This example demonstrated the proposed template's higher efficiency compared to two existing state-of-the-art architectures targeting the biomedical domain: an ARM Cortex-M4 SoC [32] and a CGRA architecture based on the designs proposed in [33, 34]. Moreover, it justifies the design choices of VWR2A, such as the integration of specialized slots. Compared to the ARM Cortex-M4 and the baseline CGRA, the VWR2A instance showed an average energy-delay product (EDP) improvement of $104.8\times$ and $19.8\times$, respectively, while executing FFTs.

In Chapter 3, I have presented one instance of the VWR2A template targeting data-intensive kernels of the biomedical domain. To demonstrate the advantage of a DSIP architecture, I compared the VWR2A instance to an SoC targeting the biomedical domain composed of an ARM Cortex-M4 GPP and two ASICs: an FFT accelerator and a matrix processor. At the kernel level, the VWR2A instance showed similar or better performance than the ASICs compared to the execution on the ARM Cortex-M4 GPP. On average, the FFT ASIC consumes $4.9\times$ less energy than the VWR2A instance while executing FFTs. Compared to the matrix processor ASIC, the VWR2A instance consumes, on average, $1.3\times$ less energy while executing a finite impulse response (FIR) filter.

At the application level, the experiments showed that the VWR2A instance is always the fastest and most energy-efficient solution compared to the ARM Cortex-M4 SoC (using its ASICs whenever possible). The results showed an EDP improvement of $6.5\times$ and $4.4\times$ compared to an execution using only the ARM Cortex-M4 core and the ARM Cortex-M4 SoC (including the ASICs), respectively. These results show that ASICs are suboptimal solutions when complete applications are considered and that DSIPs enable higher energy efficiency. In addition, ASIC accelerators have a very targeted specific context that inevitably reduces their target market domain, reducing the yearly production volumes of the integrated circuits (ICs) containing these accelerators, and increasing their costs. On the contrary, DSIPs cover a large domain of applications, hence their target market domain, making them a viable solution, particularly for the modern scaled technology nodes with huge nonrecurring engineering (NRE) costs.

In Chapter 4, I have further demonstrated the efficiency of DSIPs by considering control-intensive kernels and proposing an instance of the VWR2A template optimized for such code. Hardware accelerators are primarily designed for data-intensive code because it usually represents most of the execution time. However, once this code is accelerated, control-intensive code (e.g., branches, conditions) becomes predominant, limiting the impact of the accelerator [85], especially at the application level. Because of the complex nature of control-intensive code, its execution is usually left to GPPs, which translates to poor performance. The programmability of DSIPs, compared to the fixed nature of ASICs, allows them to execute such code. The specialized slots of VWR2A, which increase the ILP, play a crucial role in efficiently executing control-intensive kernels. Two state-of-the-art GPPs have been considered to compare the performance of a VWR2A instance. The first is an ARM Cortex-M4 processor (used in Chapters 2 and 3) and the second is a RISC-V Ibex processor that is optimized for control code. In all the considered scenarios —both at the kernel and application level— the VWR2A instance has shown better performance and energy efficiency. In detail, it demonstrated an improvement in EDP of $3.8\times$ and $12.2\times$ compared to the ARM Cortex-M4 SoC and the Ibex platform, respectively, at the application level (without accelerating data-intensive kernels on VWR2A). Finally, an entire application —both the data- and control-intensive kernels— has been mapped on the VWR2A instance to demonstrate its full potential. In this case, the savings are even higher, with an execution $9.2\times$ faster while consuming $66.7\,\%$ less energy compared to the ARM Cortex-M4 SoC (using its complete set of ASICs). This translates to an improvement in EDP of $27.6\times$. Compared only to the ARM Cortex-M4 processor (i.e., without the ASICs), the improvement in EDP is $40.5\times$.

All the experimental results of these chapters strongly support the initial hypothesis of this thesis that proposed the development of DSIPs to improve the energy efficiency of embedded systems and reduce the production cost of modern low-power ICs. To conclude, the main contributions of this thesis are listed below:

1. Showing the benefit of using programmable architectures for embedded biosignal processing platforms.

2. VWR2A: a very-wide registers and reconfigurable array template for DSIPs targeting the biomedical domain.

3. A detailed analysis and justification through experiments of the architecture features introduced in VWR2A.

4. The comparison of one instance of VWR2A with two ASICs at the kernel level to show how judicious features at the architecture level can close or narrow the performance gap between DSIPs and ASICs.

5. The comparison of one instance of VWR2A to an SoC containing an ARM Cortex-M4 processor and ASICs targeting the biomedical domain to showcase the superiority of programmable architectures compared to custom circuits.

6. The evaluation of one instance of VWR2A on control-intensive code and its comparison with two state-of-the-art GPPs: an ARM Cortex-M4 and a RISC-V Ibex to demonstrate the large code coverage of a DSIP and its higher performance on its domain of applications compared to GPPs.

Publications. This work has resulted in three publications: the initial design space exploration using a CGRA design (i.e., the baseline CGRA used in Chapter 2) was presented at the Embedded Systems Week (ESWEEK) conference [34]. This work allowed me to discover the bottleneck at the architecture level that limits traditional CGRA design performance and explore various solutions to improve it. This led to the publication of a paper at the Design Automation Conference [110] presenting the DSIP instance focused on data-intensive code (Chapter 3 of this thesis). Finally, the DSIP instance for control-intensive code proposed in Chapter 4 is currently under revision for publication in the IEEE Transactions on Computers journal [111].

5.2 Future work

In this thesis, I have proposed solutions and opened new questions and challenges to investigate. The most interesting topics that could be explored are listed below, and hints are given to provide starting ideas for those who may want to explore them.

5.2.1 Very-Wide-Register and wide memory hierarchy layout optimization

Table 4.7 shows that the SPM and the VWRs represent, on average, $69.8\,\%$ of the total power consumption for control-intensive kernels, while Table 3.7 shows a power consumption accounting of $64.0\,\%$[1] for a 512-point real-valued FFT (a data-intensive kernel). Therefore, future efforts should focus on them to improve overall energy efficiency. This requires multiple problems to be solved. First, the wide SPM is built from concatenated memory macros because no existing memory compiler could produce its wide interface. Compared to the current solution, the improved efficiency of such a compiler is hard to evaluate, and a preliminary evaluation should be done to assess the potential benefits.

For the VWRs, a similar approach has been used but using cells from the standard library –latches– provided by the technology supplier as no memory compiler exists for such memories. The problem with this approach is the long wires produced during the placement and routing of the cells, which significantly increases power consumption. Although the experiments of this thesis have been done with a post-synthesis netlist (i.e., no place and route), the estimated wires' load during synthesis impacts the sizing and type (i.e., HVT, RVT, LVT) of the cells. Additionally, the Cadence Genus tool's physical layout estimation (PLE) engine has been used (instead of the traditional wireload models) to provide more accurate results. PLE uses lef files to estimate the cells' placement during synthesis and better evaluate the required speed and drive strength of the cells. The team who proposed the VWRs has already witnessed this problem. They have shown that this problem can be solved during the place and route (using the TSMC $90\,\mathrm{nm}$ technology) by guiding the tool to align VWRs cells to the wide SPM pitch, effectively reducing the wires' length to a minimum [72]. Figure 5.1 shows this long wire problem for one instance of VWR2A that I have placed and routed with the TSMC $40\,\mathrm{nm}$ LP CMOS technology. Quantifying the benefit of an optimized place and route of VWRs using standard-cells for advanced technology nodes ($<40\,\mathrm{nm}$) would assess the energy efficiency of the VWR approach for such technologies. Another solution would be to have a full-custom design. While it would probably give the best energy efficiency, this solution is not easily portable to new technologies or changeable for another width (e.g., for a new template instance).

5.2.2 Optimization of the RC datapath

The second largest contributors to total power consumption are the RCs. The main reason is the presence of a single cycle 16-bit by 16-bit to 32-bit multiplier (per RC). Its implementation is left to the synthesis tool, and no optimization is done. Exploring other multiplier

[1] This value also contains the context memory and the scalar register files (SRFs), but they account only for a tiny amount.

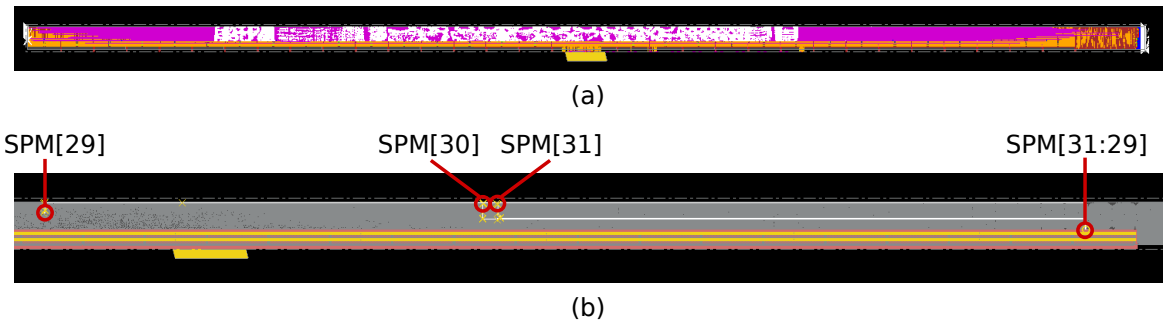


Figure 5.1: Place and route of the VWR2A instance of Chapter 3. The layout is very wide as all the SPM macros are aligned to produce the wide 4096b interface. (a) Complete layout displayed with the VWRs cells highlighted in white (24576 latches from the TSMC 40nm LP CMOS library). (b) Layout zoomed to show the long wires of the SPM pins 29 to 31. The corresponding latches should be placed as close as possible to the SPM macro pins to reduce the wires' lentght to a minimum.

designs and the power vs. performance tradeoff of a multi-cycle multiplier would provide more guidance in designing a new instance of the VWR2A template.

Another possible optimization is the support for single instruction multiple data (SIMD) operations. All the instances of this thesis have used a 32-bit datapath and data size. However, many domains, such as the biomedical one, can use smaller representations without losing accuracy. Adding SIMD support does not represent a massive power and area overhead, as most of the existing hardware can be reused. For example, a 32-bit adder can easily be transformed into four 8-bit adders with negligible hardware addition (only the relevant carry bits have to be masked).

5.2.3 Instruction memories optimization

The local program memories of the RCs represent 7 % of the total power consumption on average. This is relatively small in the current instances. However, if the efficiency of the SPM and the VWRs is improved or the local program memory size is increased (e.g., for a specific domain), it will become more critical. In this case, considering the implementation of a distributed loop buffer organisation [96] could be interesting. A distributed loop buffer organization removes the need for explicit nop instruction by using an activation trace (AT) table to activate or de-activate the units. This allows the units to have program memories of different sizes but still controlled by a common PC. In addition, it allows the compression of the kernel instructions, which increases the number of concurrent kernels supported by the architecture. The efficiency of such a design depends on the number of nops inside these program memories. Table 5.1 shows the minimum and average percentage of nops for all the units considering all the kernels

Table 5.1: Minimum and average percentage of nops for all the units (with a program memory depth of 64 configuration words) considering all the kernels of this thesis.

	LCU	LSU	MXCU	RC0	RC1	RC2	RC3
min % nops	20.3 %	35.9 %	21.9 %	34.4 %	32.8 %	35.9 %	35.9 %
avg % nops	70.2 %	74.5 %	75.2 %	69.5 %	73.1 %	74.9 %	75.3 %

presented in this thesis. The high average of nops means that most of the time, only a few instructions inside the units' program memory are actual instructions (i.e., not nops). However, all the units require at least 69.5 % of the program memory (i.e., 44 instructions) at least for one kernel, as shown by the minimum percentage of nops row in Table 5.1. Considering only program memories with a power-of-two size, it is impossible to reduce any of the units' program memory in this case. Therefore, a loop buffer would probably not reduce the power consumption (at least not significantly).

5.2.4 Architectural template for various domains

The VWR2A template has been proposed for the biomedical domain. It has not been evaluated outside this domain, but the efficiency shown on some generic kernels, such as the FFT, median, or FIR filter ones, shows the potential of such design applied in other areas. In particular, domains using digital signal processing algorithms and requiring low power and high performance should benefit from the high-level features proposed in the VWR2A template. In this context, using VWR2A as a baseline architecture is definitely a good approach. For example, digital signal processing algorithms, such as the FFT and FIR filter kernel evaluated in this thesis, are used in multimedia applications [112, 113] and the wireless communication domain [114, 115]. On the contrary, domains with random memory access over a large memory address space and highly irregular code will certainly not benefit from VWR2A features. Additionally, applications that process data one by one —as opposed to data processed in batch— will not benefit from the wide memory hierarchy proposed in VWR2A.

It could also be transposable to domains further than low-power applications, such as high-performance computing (HPC). While energy efficiency is mandatory for embedded systems, it was not the case for HPC until recently. The growing environmental crisis requires a more sustainable society, which forces HPC toward higher energy efficiency. One example is the square kilometre array (SKA) project [116], which favored the energy-to-completion rather than the time-to-completion to limit the system's total energy consumption. The interesting point about the SKA project is that FFTs, mainly 2D partial FFTs, represent a considerable percentage of all the kernels executed. Considering the higher performance requirement and all the types of kernels of the SKA project, the VWR2A

template cannot be directly applied to this domain. However, its efficiency in computing FFTs and programmability (i.e., capacity to potentially execute any kernel of a domain) could inspire the design of an architecture for the astronomical domain.

5.2.5 Compiler support

The main drawback of exploration at the architecture level is the lack of compiler support. All the evaluated kernels have been manually mapped, and this long process limits the experiments to a few examples. Some compilers for CGRAs have been proposed, but none is easily re-usable and extendable to an architecture like VWR2A. The mapping of instructions on the RCs and the specialized slots is probably more an engineering effort than an academic one. On the other hand, the data placement constraints caused by the single-port nature of VWRs and their integration inside a compiler are more challenging and require innovative solutions.

Bibliography

[1] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. "Internet of things for smart cities". In: *IEEE Internet of Things journal* 1.1 (2014), pp. 22–32.

[2] Tiziana Campisi, Alessandro Severino, Muhammad Ahmad Al-Rashid, and Giovanni Pau. "The development of the smart cities in the connected and autonomous vehicles (CAVs) era: From mobility patterns to scaling in cities". In: *Infrastructures* 6.7 (2021), p. 100.

[3] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. "A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges". In: *IEEE Internet of Things Journal* (2022).

[4] Daniel Hernández, Juan-Carlos Cano, Federico Silla, Carlos T. Calafate, and José M. Cecilia. "AI-Enabled Autonomous Drones for Fast Climate Change Crisis Assessment". In: *IEEE Internet of Things Journal* 9.10 (2022), pp. 7286–7297. DOI: 10.1109/JIOT.2021.3098379.

[5] V Jagadeeswari, V Subramaniyaswamy, R Logesh, and Varadarajan Vijayakumar. "A study on medical Internet of Things and Big Data in personalized healthcare system". In: *Health information science and systems* 6.1 (2018), pp. 1–20.

[6] Anne-Katrin Witte and Rüdiger Zarnekow. "Transforming personal healthcare through technology-a systematic literature review of wearable sensors for medical application". In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 2019.

[7] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646.

[8] World Health Organization. Accessed on: Nov. 17, 2022. URL: https://www.who.int.

[9] Organisation for Economic Co-operation and Development (OECD). Accessed on: Nov. 17, 2022. URL: https://stats.oecd.org/.

[10] United States - Energy Information Administration. 2018 Commercial Buildings Energy Consumption Survey. Accessed on: Nov. 17, 2022. URL: https://www.eia.gov/consumption/commercial.

[11] Grégoire Casimir Joseph Surrel. "Low Power Sensing and Processing in Wearable Biomedical Devices for Personalized Health Monitoring". EPFL, 2019. DOI: 10.5075/epfl-thesis-9721.

[12] Haya Elayan, Moayad Aloqaily, and Mohsen Guizani. "Digital twin for intelligent context-aware IoT healthcare systems". In: *IEEE Internet of Things Journal* 8.23 (2021), pp. 16749–16757.

[13] Ying Liu, Lin Zhang, Yuan Yang, Longfei Zhou, Lei Ren, Fei Wang, Rong Liu, Zhibo Pang, and M Jamal Deen. "A novel cloud-based framework for the elderly healthcare services using digital twin". In: *IEEE access* 7 (2019), pp. 49088–49101.

[14] Philips. ePatch. Accessed on: Nov. 17, 2022. URL: https://www.philips.com.

[15] Smartcardia. Accessed on: Nov. 17, 2022. URL: https://www.smartcardia.com.

[16] Francisco Rincon, Julien Pidoux, Srini Murali, and Jean-Jacques Goy. "Performance of the new SmartCardia wireless, wearable oximeter: a comparison with arterial SaO2 in healthy volunteers". In: *BMC anesthesiology* 22.1 (2022), pp. 1–7.

[17] Karim Bayoumy, Mohammed Gaber, Abdallah Elshafeey, Omar Mhaimeed, Elizabeth H Dineen, Francoise A Marvel, Seth S Martin, Evan D Muse, Mintu P Turakhia, Khaldoun G Tarakji, et al. "Smart wearable devices in cardiovascular care: where we are and how to move forward". In: *Nature Reviews Cardiology* 18.8 (2021), pp. 581–599.

[18] Elisabetta De Giovanni, Adriana Arza Valdes, Miguel Peon-Quiros, Amir Aminifar, and David Atienza. "Real-Time Personalized Atrial Fibrillation Prediction on Multi-Core Wearable Sensors". In: *IEEE Transactions on Emerging Topics in Computing* 9.4 (2020), pp. 1654–1666.

[19] Ying Wei, Jun Zhou, Yin Wang, Yinggang Liu, Qingsong Liu, Jiansheng Luo, Chao Wang, Fengbo Ren, and Li Huang. "A review of algorithm & hardware design for AI-based biomedical applications". In: *IEEE transactions on biomedical circuits and systems* 14.2 (2020), pp. 145–163.

[20] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[21] maxim integrated - ANALOG DEVICES. Accessed on: Oct. 17, 2022. URL: https://www.maximintegrated.com.

[22] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2018, pp. 1–4. DOI: 10.1109/ASAP.2018.8445101.

[23] Greenwaves technologies. Accessed on: Oct. 17, 2022. URL: https://greenwaves-technologies.com/.

[24] Canaan. Kendryte K210. Accessed on: Nov. 1, 2022. URL: https://canaan.io/product/kendryteai.

[25] Michael L Rieger. "Retrospective on VLSI value scaling and lithography". In: *Journal of Micro/Nanolithography, MEMS, and MOEMS* 18.4 (2019), p. 040902.

[26] Arm Limited. Accessed on: Oct. 17, 2022. URL: https://www.arm.com/.

[27] Davide Rossi, Igor Loi, Francesco Conti, Giuseppe Tagliavini, Antonio Pullini, and Andrea Marongiu. "Energy efficient parallel computing on the PULP platform with support for OpenMP". In: *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*. 2014, pp. 1–5. DOI: 10.1109/EEEI.2014.7005803.

[28] Amara Amara, Frédéric Amiel, and Thomas Ea. "FPGA vs. ASIC for low power applications". In: *Microelectronics Journal* 37.8 (2006), pp. 669–677. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2005.11.003. URL: https://www.sciencedirect.com/science/article/pii/S0026269205003927.

[29] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[30] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. "You Cannot Improve What You Do Not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference". In: *ACM Trans. Reconfigurable Technol. Syst.* 11.3 (Dec. 2018). ISSN: 1936-7406. DOI: 10.1145/3242898. URL: https://doi.org/10.1145/3242898.

[31] 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". In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3357375. URL: https://doi.org/10.1145/3357375.

[32] Shuang Song, Mario Konijnenburg, Roland Van Wegberg, Jiawei Xu, Hyunsoo Ha, Wim Sijbers, Stefano Stanzione, Dwaipayan Biswas, Arjan Breeschoten, Peter Vis, et al. "A 769 μW battery-powered single-chip SoC with BLE for multi-modal vital sign monitoring health patches". In: *IEEE Transactions on Biomedical Circuits and Systems* 13.6 (2019), pp. 1506–1517.

[33] Loris Duch, Soumya Basu, Rubén Braojos, David Atienza, Giovanni Ansaloni, and Laura Pozzi. "A multi-core reconfigurable architecture for ultra-low power bio-signal analysis". In: *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2016, pp. 416–419. DOI: 10.1109/BioCAS.2016.7833820.

[34] Elisabetta De Giovanni, Fabio Montagna, Benoît W. Denkinger, Simone Machetti, Miguel Peón-Quirós, Simone Benatti, Davide Rossi, Luca Benini, and David Atienza. "Modular Design and Optimization of Biomedical Applications for Ultralow Power Heterogeneous Platforms". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3821–3832. DOI: 10.1109/TCAD.2020.3012652.

[35] lowRISC – Ibex. Accessed on: Apr. 4, 2022. URL: https://lowrisc.org/our-work/.

[36] Transforma Insights. Accessed on: Apr. 4, 2022. [Online]. Available: https://transformainsights.com/news/iot-market-24-billion-usd15-trillion-revenue-2030. URL: https://transformainsights.com/news/iot-market-24-billion-usd15-trillion-revenue-2030.

[37] Jun Sato, Masaharu Imai, Tetsuya Hakata, Alauddin Y Alomary, and Nobuyuki Hikichi. "An integrated design environment for application specific integrated processor". In: *1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society. 1991, pp. 414–415.

[38] Alauddin Alomary, Takeharu Nakata, Yoshimichi Honma, Jun Sato, Nobuyuki Hikichi, and Masaharu Imai. "PEAS-I: A hardware/software co-design system for ASIPs". In: *Proceedings of EURO-DAC 93 and EURO-VHDL 93-European Design Automation Conference*. IEEE. 1993, pp. 2–7.

[39] Diksha Moolchandani, Anshul Kumar, and Smruti R Sarangi. "Accelerating cnn inference on asics: A survey". In: *Journal of Systems Architecture* 113 (2021), p. 101887.

[40] Xueyu Han, Jiajia Chen, Boyu Qin, and Susanto Rahardja. "A novel area-power efficient design for approximated small-point FFT architecture". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4816–4827.

[41] Mahadev Satyanarayanan, Nathan Beckmann, Grace A Lewis, and Brandon Lucia. "The role of edge offload for hardware-accelerated mobile devices". In: *GetMobile: Mobile Computing and Communications* 25.2 (2021), pp. 5–13.

[42] A Balamanikandan and K Krishnamoorthi. "Low area ASIC implementation of LUT–CLA–QTL architecture for cryptography applications". In: *Wireless Networks* 26.4 (2020), pp. 2681–2693.

[43] L. Shang and N.K. Jha. "Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs". In: *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design.* 2002, pp. 345–352. DOI: 10.1109/ASPDAC.2002.994946.

[44] Onur Ulusel, Christopher Picardo, Christopher B. Harris, Sherief Reda, and R. Iris Bahar. "Hardware acceleration of feature detection and description algorithms on low-power embedded platforms". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL).* 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577310.

[45] Loris Duch, Soumya Basu, Rubén Braojos, Giovanni Ansaloni, Laura Pozzi, and David Atienza. "HEAL-WEAR: An ultra-low power heterogeneous system for biosignal analysis". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.9 (2017), pp. 2448–2461.

[46] Michael De Nil, Lennart Yseboodt, Frank Bouwens, Jos Hulzink, Mladen Berekovic, Jos Huisken, and Jef van Meerbergen. "Ultra Low Power ASIP Design for Wireless Sensor Nodes". In: *2007 14th IEEE International Conference on Electronics, Circuits and Systems.* 2007, pp. 1352–1355. DOI: 10.1109/ICECS.2007.4511249.

[47] Jin Soo Kim and Myung H. Sunwoo. "Three low power ASIP processor designs for communications, video, and audio applications". In: *2007 International Conference on Design & Technology of Integrated Systems in Nanoscale Era.* 2007, pp. 241–244. DOI: 10.1109/DTIS.2007.4449529.

[48] Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Francky Catthoor, Diederik Verkest, and Henk Corporaal. "Very Wide Register: An Asymmetric Register File Organization for Low Power Embedded Processors". In: *2007 Design, Automation Test in Europe Conference Exhibition.* 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364435.

[49] Ciji Isen, Lizy K. John, and Eugene John. "A Tale of Two Processors: Revisiting the RISC-CISC Debate". In: *Computer Performance Evaluation and Benchmarking.* Ed. by David Kaeli and Kai Sachs. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–76. ISBN: 978-3-540-93799-9.

[50] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. "ISA Wars: Understanding the Relevance of ISA Being RISC or CISC to Performance, Power, and Energy on Modern Architectures". In: *ACM Trans. Comput. Syst.* 33.1 (Mar. 2015). ISSN: 0734-2071. DOI: 10.1145/2699682. URL: https://doi.org/10.1145/2699682.

[51] David Patterson. “50 Years of computer architecture: From the mainframe CPU to the domain-specific TPU and the open RISC-V instruction set”. In: *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. 2018, pp. 27–31. DOI: 10.1109/ISSCC.2018.8310168.

[52] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. “Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. Saint-Malo, France: ACM, 2010, pp. 26–36. ISBN: 9781450300537. DOI: 10.1145/1815961.1815967.

[53] 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: *27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[54] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[55] François Botman, Julien de Vos, Sébastien Bernard, François Stas, Jean-Didier Legat, and David Bol. “Bellevue: A 50MHz variable-width SIMD 32bit microcontroller at 0.37V for processing-intensive wireless sensor nodes”. In: *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2014, pp. 1207–1210. DOI: 10.1109/ISCAS.2014.6865358.

[56] Gang-Ryung Uh, Yuhong Wang, David Whalley, Sanjay Jinturkar, Chris Burns, and Vincent Cao. “Techniques for Effectively Exploiting a Zero Overhead Loop Buffer”. In: *Compiler Construction*. Ed. by David A. Watt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 157–172. ISBN: 978-3-540-46423-5.

[57] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. “Understanding Sources of Inefficiency in General-Purpose Chips”. In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 37–47. ISSN: 0163-5964. DOI: 10.1145/1816038.1815968. URL: https://doi.org/10.1145/1816038.1815968.

[58] Francky Catthoor, Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Angeliki Kritikakou, and Javed Absar. “Energy Consumption Breakdown and Requirements for an Embedded Platform”. In: *Ultra-Low Energy Domain-Specific Instruction-Set Processors*. Dordrecht: Springer Netherlands, 2010, pp. 33–81. ISBN: 978-90-481-9528-2. DOI: 10.1007/978-90-481-9528-2_3. URL: https://doi.org/10.1007/978-90-481-9528-2_3.

[59] Ahmed Yasir Dogan, Jeremy Constantin, David Atienza, Andreas Burg, and Luca Benini. "Low-power processor architecture exploration for online biomedical signal analysis". In: *IET Circuits, Devices & Systems* 6.5 (Sept. 2012), pp. 279–286. ISSN: 1751-8598. DOI: 10.1049/iet-cds.2012.0011. URL: https://doi.org/10.1049/iet-cds.2012.0011.

[60] Microchip. Accessed on: Oct. 17, 2022. URL: https://www.microchip.com/.

[61] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing". In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1970–1981. DOI: 10.1109/JSSC.2019.2912307.

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

[63] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33. DOI: 10.1147/rd.111.0025.

[64] Herb Schorr et al. "Design principles for a high-performance system". In: *Proceedings of the Symposium on Computers and Automata*. 1971.

[65] Tilak Agerwala and John Cocke. *High performance reduced instruction set processors*. IBM Watson Research Center, 1987.

[66] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. "An analysis of accelerator coupling in heterogeneous architectures". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744794.

[67] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. "Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 1027–1040. DOI: 10.1109/ISCA52012.2021.00084.

[68] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications". In: *IEEE Transactions on Computers* 49.5 (2000), pp. 465–481. DOI: 10.1109/12.859540.

[69] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*. 2002, pp. 73–78. DOI: 10.1145/774789.774805.

[70] Victor Zyuban and Peter Kogge. "The energy complexity of register files". In: *Proceedings of the 1998 international symposium on Low power electronics and design*. 1998, pp. 305–310.

[71] Boris Hübener, Gregor Sievers, Thorsten Jungeblut, Mario Porrmann, and Ulrich Rückert. "CoreVA: A configurable resource-efficient VLIW processor architecture". In: *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE. 2014, pp. 9–16.

[72] Francky Catthoor, Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Angeliki Kritikakou, and Javed Absar. "An Asymmetrical Register File: The VWR". In: *Ultra-Low Energy Domain-Specific Instruction-Set Processors*. Dordrecht: Springer Netherlands, 2010, pp. 199–222. ISBN: 978-90-481-9528-2. DOI: 10.1007/978-90-481-9528-2_8. URL: https://doi.org/10.1007/978-90-481-9528-2_8.

[73] Texas Instruments. Arm-based microcontrollers. Accessed on: Nov. 1, 2022. URL: https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/arm-based-microcontrollers/overview.html.

[74] ST. STM32 32-bit Arm Cortex MCUs. Accessed on: Nov. 1, 2022. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.

[75] Silicon Labs. 32-bit EFM32 Cortex M. Accessed on: Nov. 1, 2022. URL: https://www.silabs.com/mcu.

[76] Cadence Genus. v18.1. Accessed on: Nov. 1, 2022. URL: https://www.cadence.com.

[77] TSMC. 40nm Low Power (LP) process. Accessed on: Nov. 1, 2022. URL: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_40nm.

[78] Synopsys. Q-2019.12. Accessed on: Oct. 17, 2022. URL: https://www.synopsys.com.

[79] Sridhar Krishnan and Yashodhan Athavale. "Trends in biomedical signal feature extraction". In: *Biomedical Signal Processing and Control* 43 (2018), pp. 41–63. ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2018.02.008. URL: https://www.sciencedirect.com/science/article/pii/S1746809418300399.

[80] Ajit P. Yoganathan, Ramesh Gupta, and William H. Corcoran. "Fast Fourier transform in the analysis of biomedical data". In: *Medical and biological engineering* 14 (1976), pp. 239–245. ISSN: 1746-8094. DOI: https://doi.org/10.1007/BF02478755.

[81] Fabio Dell'Agnola, Una Pale, Rodrigo Marino, Adriana Arza, and David Atienza. "MBioTracker: Multimodal Self-Aware Bio-Monitoring Wearable System for Online Workload Detection". In: *IEEE Transactions on Biomedical Circuits and Systems* 15.5 (2021), pp. 994–1007.

[82] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2003354 (visited on 10/17/2022).

[83] ARM. CMSIS-Core (Cortex-M) v4.30. Accessed on: Nov. 17, 2022. URL: https://arm-software.github.io/CMSIS_5/Core/html/core_revisionHistory.html.

[84] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. "ULP-SRP: Ultra Low-Power Samsung Reconfigurable Processor for Biomedical Applications". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.3 (2014), pp. 1–15.

[85] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. "Architecture exploration for a reconfigurable architecture template". In: *IEEE Design Test of Computers* 22.2 (2005), pp. 90–101. DOI: 10.1109/MDT.2005.27.

[86] Robert Fasthuber, Francky Catthoor, Praveen Raghavan, and Frederik Naessens. "Energy-efficient communication processors". In: *Springer, New York, NY* 10 (2013), pp. 978–1.

[87] Safwat Mostafa Noor, Eugene John, and Manoj Panday. "Design and Implementation of an Ultralow-Energy FFT ASIC for Processing ECG in Cardiac Pacemakers". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.4 (2019), pp. 983–987. DOI: 10.1109/TVLSI.2018.2883642.

[88] John L. Semmlow. *Biosignal and medical image processing*. 2nd ed. Boca Raton: CRC Press, 2009. ISBN: 9781420062304.

[89] Elisabetta De Giovanni, Tomas Teijeiro, Grégoire P. Millet, and David Atienza. "Adaptive R-Peak Detection on Wearable ECG Sensors for High-Intensity Exercise". In: *IEEE Transactions on Biomedical Engineering* (2022), pp. 1–12. DOI: 10.1109/TBME.2022.3205304.

[90] M. Murugappan, M. Rizon, R. Nagarajan, S. Yaacob, D. Hazry, and I. Zunaidi. "Time-Frequency Analysis of EEG Signals for Human Emotion Detection". In: *4th Kuala Lumpur International Conference on Biomedical Engineering 2008*. Ed. by Noor Azuan Abu Osman, Fatimah Ibrahim, Wan Abu Bakar Wan Abas, Herman Shah Abdul Rahman, and Hua-Nong Ting. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 262–265. ISBN: 978-3-540-69139-6.

[91] Ercan Gokgoz and Abdulhamit Subasi. "Comparison of decision tree algorithms for EMG signal classification using DWT". In: *Biomedical Signal Processing and Control* 18 (2015), pp. 138–144. ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2014.12.005. URL: https://www.sciencedirect.com/science/article/pii/S1746809414002006.

[92] U. Rajendra Acharya, Hamido Fujita, Oh Shu Lih, Yuki Hagiwara, Jen Hong Tan, and Muhammad Adam. "Automated detection of arrhythmias using different intervals of tachycardia ECG segments with convolutional neural network". In: *Information Sciences* 405 (2017), pp. 81–90. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2017.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0020025517306539.

[93] Wei-Long Zheng and Bao-Liang Lu. "Investigating Critical Frequency Bands and Channels for EEG-Based Emotion Recognition with Deep Neural Networks". In: *IEEE Transactions on Autonomous Mental Development* 7.3 (2015), pp. 162–175. DOI: 10.1109/TAMD.2015.2431497.

[94] Ying Wei, Jun Zhou, Yin Wang, Yinggang Liu, Qingsong Liu, Jiansheng Luo, Chao Wang, Fengbo Ren, and Li Huang. "A Review of Algorithm & Hardware Design for AI-Based Biomedical Applications". In: *IEEE Transactions on Biomedical Circuits and Systems* 14.2 (2020), pp. 145–163. DOI: 10.1109/TBCAS.2020.2974154.

[95] Aishwarya Raamkumar. "Hardware Acceleration Evaluation and Compiler Integration for an in-house design". MA thesis. Eindhoven University of Technology, 2020.

[96] Francky Catthoor, Praveen Raghavan, Andy Lambrechts, Murali Jayapala, Angeliki Kritikakou, and Javed Absar. "Ultra-low energy domain-specific instruction-set processors". In: 1st ed. Springer Netherlands, 2010. Chap. An asymmetrical register file: the VWR, pp. 199–222. ISBN: 978-90-481-9528-2.

[97] Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. "XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37.11 (2018), pp. 2940–2951. DOI: 10.1109/TCAD.2018.2857019.

[98] J. Lee and A. Smith. "Branch Prediction Strategies and Branch Target Buffer Design". In: *Computer* 17.01 (Jan. 1984), pp. 6–22. ISSN: 1558-0814. DOI: 10.1109/MC.1984.1658927.

[99] ARM Cortex-M4 – Reference manual. Accessed on: Apr. 4, 2022. URL: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4.

[100] Satyajit Das, Kevin J. M. Martin, Davide Rossi, Philippe Coussy, and Luca Benini. "An Energy-Efficient Integrated Programmable Array Accelerator and Compilation Flow for Near-Sensor Ultralow Power Processing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.6 (2019), pp. 1095–1108. DOI: 10.1109/TCAD.2018.2834397.

[101] B. Ramakrishna Rau. "Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: ACM, 1994, pp. 63–74. ISBN: 0897917073. DOI: 10.1145/192724.192731.

[102] Manupa Karunaratne, Cheng Tan, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. "Dnestmap: Mapping Deeply-Nested Loops on Ultra-Low Power CGRAs". In: *Design Automation Conference (DAC)*. San Francisco, California: ACM, 2018. ISBN: 9781450357005. DOI: 10.1145/3195970.3196027. URL: https://doi.org/10.1145/3195970.3196027.

[103] Shouyi Yin, Pengcheng Zhou, Leibo Liu, and Shaojun Wei. "Acceleration of nested conditionals on CGRAs via trigger scheme". In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2015, pp. 597–604.

[104] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. "Snafu: An ultra-low-power, energy-minimal CGRA-generation framework and architecture". In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 1027–1040.

[105] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. "Elastic CGRAs". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2013, pp. 171–180.

[106] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. "Power-efficient predication techniques for acceleration of control flow execution on CGRA". In: *ACM Transactions on Architecture and Code Optimization* 10.2 (May 2013). ISSN: 1544-3566. DOI: 10.1145/2459316.2459319.

[107] Kyuseung Han, Jong Kyung Paek, and Kiyoung Choi. "Acceleration of control flow on CGRA using advanced predicated execution". In: *2010 International Conference on Field-Programmable Technology*. 2010, pp. 429–432. DOI: 10.1109/FPT.2010.5681452.

[108] Rubén Braojos, Giovanni Ansaloni, and David Atienza. "A methodology for embedded classification of heartbeats using random projections". In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 899–904. DOI: 10.7873/DATE.2013.189.

[109] Adam Teman, Davide Rossi, Pascal Meinerzhagen, Luca Benini, and Andreas Burg. "Controlled placement of standard cell memory arrays for high density and low power in 28nm FD-SOI". In: *The 20th Asia and South Pacific Design Automation Conference*. 2015, pp. 81–86. DOI: 10.1109/ASPDAC.2015.7058985.

[110] Benoît W. Denkinger, Miguel Peón-Quirós, Mario Konijnenburg, David Atienza, and Francky Catthoor. "VWR2A: A Very-Wide-Register Reconfigurable-Array Architecture for Low-Power Embedded Devices". In: DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 895–900. ISBN: 9781450391429. DOI: 10.1145/3489517.3530980. URL: https://doi.org/10.1145/3489517.3530980.

[111] Benoît W. Denkinger, Miguel Peón-Quirós, Mario Konijnenburg, David Atienza, and Francky Catthoor. "Acceleration of Control Intensive Applications on Coarse-Grained Reconfigurable Arrays for Embedded Systems". In: *publication pending in IEEE Transactions on Computers* ().

[112] Jerry D Gibson. "Speech compression". In: *Information* 7.2 (2016), p. 32.

[113] G Ganesh Kumar, Subhendu K Sahoo, and Pramod Kumar Meher. "50 years of FFT algorithms and applications". In: *Circuits, Systems, and Signal Processing* 38 (2019), pp. 5665–5698.

[114] Ganesh Kumar Ganjikunta and Subhendu Kumar Sahoo. "An area-efficient and low-power 64-point pipeline Fast Fourier Transform for OFDM applications". In: *Integration* 57 (2017), pp. 125–131. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2016.12.002. URL: https://www.sciencedirect.com/science/article/pii/S0167926016301833.

[115] Taesang Cho, Hanho Lee, Jounsup Park, and Chulgyun Park. "A high-speed low-complexity modified radix-2 5 FFT processor for gigabit WPAN applications". In: *2011 IEEE international symposium of circuits and systems (ISCAS)*. IEEE. 2011, pp. 1259–1262.

[116] SKAO. Square Kilometre Array Observartory. Accessed on: Dec. 2, 2022. URL: https://www.skao.int/.

A Sorting kernel mapping assembly code analysis

This appendix details the execution of the sorting algorithm discussed in Chapter 4.4 at the instruction level for the ARM Cortex-M4 processor, the Ibex processor, and the VWR2A. Figure A.1 shows the original C code of the sorting algorithm and the assembly instructions for the three implementations. The ARM Cortex-M4 (Figure A.1b) and the Ibex (Figure A.1c) assembly codes are very similar. The main difference is the single-cycle compare and branch instruction feature of the Ibex platform (i.e. –*bge a6,a2,.L4* in this case–, while the ARM processor first compares the values, *cmp r6,r5*, and then branches depending on the output, *itt lt*. This explains the smaller inner loop instruction size of the Ibex compared to the ARM Cortex-M4 processor, with 9 and 10 instructions executed when the if-condition is true, respectively. For the VWR2A, Figure A.1d shows the assembly for all the different units (i.e., LCU, LSU, MXCU, and the RCs). The inner loop takes two instructions per unit, ten instructions in total (without counting the *nop* instructions). As all the units execute in parallel, the inner loop takes only two clock cycles, which is $5\times$ better than the processors.

Here, a detailed cycle-level walk-through of the VWR2A mapping is given to help the reader understand the mapping better. The mapping over the RCs is adapted to the architecture constraints and parallelization. The original data array is divided into two sub-arrays that are sorted by RC0-RC3, and RC1-RC2. The following line numbers correspond to Figure A.1d. Lines 00 to 02 are initialization instructions where, for example, the data are loaded into VWR 0 (LSU - line 02: *ld.vwr #0, [r7]*). At line 03, RC0 and RC1 load the current data (data[i] in Figure A.1a) from VWR 0 into r0 (RC0/1: *movs r0, vwr #0*). At the same time, RC3 stores in r1 the value coming from RC2 (RC3 - *movs r1, rct*, where rc[t/b] is for processing the top / bottom element). At line 04, RC0 and RC1 store a backup of data[i] to r1, while RC2 and RC3 initialize r0 with the array index of the current data (*i-th* element index). The MXCU increments by one the address of the VWRs (MXCU - line 04: *adds r0, r1, #1)*). At line 05, the inner loop starts with RC0 and RC1 comparing the current minimum value —data[i] stored in r0— with the next entry of the array —data[j] from VWR

```
void sort(int16_t *data, int32_t size){
 int i, j, temp;
 for(i=0; i<size-1; i++) {
  for(j=i+1; j<size; j++) {
   if(data[j] < data[i]) {
    temp = data[i];
    data[i] = data[j];
    data[j] = temp;
   }
  }
 }
}
```

(a)

```
void sort(int16_t *data, int32_t size)
    push    {...}
    movs    r2, #0
    mov     r4, r0
    subs    r7, r1, #1
.L0:cmp     r7, r2
    bgt.n   .L1
    pop     {...}
.L1:adds    r2, #1
    mov     r3, r2
.L2:cmp     r3, r1
    bne.n   .L3
    adds    r4, #2
    b.n     .L0
.L3:ldrsh.w r6, [r0, r3, lsl #1]
    ldrsh.w r5, [r4]
    cmp     r6, r5
    itt     lt
    strhlt  r6, [r4, #0]
    strhlt.w r5, [r0, r3, lsl #1]
    adds    r3, #1
    b.n     .L2
```

(b)

```
void sort(int16_t *data, int32_t size)
    li      a5,0
    addi    a7,a1,-1
.L0:blt     a5,a7,.L1
    ret
.L1:addi    a5,a5,1
    mv      a4,a0
    mv      a3,a5
.L2:bne     a3,a1,.L3
    addi    a0,a0,2
    j       .L0
.L3:lh      a6,2(a4)
    lh      a2,0(a0)
    bge     a6,a2,.L4
    sh      a6,0(a0)
    sh      a2,2(a4)
.L4:addi    a3,a3,1
    addi    a4,a4,2
    j       .L2
```

(c)

void sort(int16_t *data, int32_t size)

LCU:	LSU:	MXCU:	RC0:	RC1:	RC2:	RC3:
00: movs r0, #16	00: ld.srf #3, [r7]	00: movs r0, #0	00: nop	00: nop	00: nop	00: nop
01: subs r0, #1	01: movs r7, SRF[2]	01: movs r1, #0	01: nop	01: nop	01: nop	01: nop
02: nop	02: ld.vwr #0, [r7]	02: movs r2, #0	02: nop	02: nop	02: movs r1, #0	02: nop
03: L0:movs r1, r0	03:.L0:nop	03:.L0:nop	03:.L0:movs r0, vwr #0	03:.L0:movs r0, vwr #0	03:.L0:nop	03:.L0:movs r1, rct
04: nop	04: nop	04: adds r0, r1, #1	04: movs r1, vwr #0	04: movs r1, vwr #0	04: movs r0, rcb	04: movs r0, r1
05:.L1:nop	05:.L1:nop	05:.L1:nop	05:.L1:cmp vwr #0, r0	05:.L1:cmp vwr #0, r0	05:.L1:nop	05:.L1:adds r1, r1, #1
06: bgepd r1, #1, .L1	06: nop	06: adds r0, r0, #1	06: movs.sf r0, vwr #0, r0	06: movs.sf r0, vwr #0, r0	06: movs.sf r0, rcb, r0, rct	06: movs.sf r0, r1, r0, rcb
07: nop	07: nop	07: movs r0, r1	07: nop	07: nop	07: nop	07: nop
08: nop	08: nop	08: adds r1, r1, #1	08: movs vwr #0, r0	08: movs vwr #0, r0	08: adds r0, #0	08: adds r0, #0
09: nop	09: nop	09: nop	09: movs SRF[5], rct	09: adds rout, rcb	09: nop	09: nop
10: nop	10: nop	10: movs r0, SRF[5]	10: movs SRF[5], rcb	10: nop	10: nop	10: nop
11: nop	11: nop	11: movs r0, SRF[5]	11: movs vwr #0, r1	11: nop	11: nop	11: nop
12: bgepd r0, #1, .L0	12: nop	12: movs r0, r1	12: nop	12: movs vwr #0, r1	12: adds r1, r1, #1	12: nop
13: ret	13: str.vwr #0, [r7]	13: nop	13: nop	13: nop	13: nop	13: nop

(d)

Figure A.1: Sorting algorithm assembly code. Red dashed (- - -) boxes highlight outer loop instructions, and green dashed-doted (-·-·-) boxes inner loop instructions. (a) Original C code. (b) ARM Cortex-M4 assembly code. (c) RISC-V Ibex assembly code. (d) VWR2A assembly code for one column.

0—, and RC3 increments the next entry address by one to keep track of the j index locally. Based on the sign flag resulting from the previous comparison, RC0 and RC1 update their local minimum value store in r0 (RC0 - line 06: *movs.sf r0, vwr #0, r0, self*). RC2 and RC3 do the same, but keep track of the minimum value index by using the sign flag (movs.sf) of rct (i.e., RC1) and rcb (i.e., RC0), respectively. At the same time, the MXCU updates the address of the VWRs for the next iteration comparison, and the LCU decides whether or not to branch for another iteration of the inner loop or exit it (LCU - line 06: *bgepd r1, #1, .L1*).

The actual element swapping inside the array takes place only in the outer loop for VWR2A, while this is done every time a new minimum value is found for the ARM Cortex-M4 and the RISC-V Ibex (as shown in the C source code in Figure A.1a). The VWR2A mapping only swaps the elements and their index in the RCs' local register file (i.e., r0). Once the inner loop is finished, RC0 and RC1 first store their local minimum value back to VWR 0 (RC0/1 - line 08: *movs vwr #0, r0*). Then, their local backup, in r1, of the value previously stored at this location is stored at the old index of the new minimum value. As RC0 and RC1 process their half of the input array, two, often different, old indexes are held by RC3 and RC2, respectively. These indexes are passed to RC0, which stores them, one after the other, to the SRF (only RC0 can write into it). The MXCU updates the VWRs address with these two values (MXCU - line 10-11), one after the other, and RC0 and RC1 store their local backup of the minimum value back to the VWR 0 (RC0/1 - line 11/12: *vwr #0, r1*).

Finally, the MXCU increments the array address by one at line 08 and updates the VWRs address with it at line 12 for the next iteration. This corresponds to incrementing the *i* variable by one to access the next i-th element of the input data array (outer loop in Figure A.1a). RC2 also keeps a local track of this address. It updates it on line 12 and shares it with RC3 on line 03. RC0 and RC1 load the current data (data[i] in Figure A.1a) VWR 0 into r0 (RC0/1 - line 03: *movs r0, vwr #0*) and another inner loop can start. Once the outer loop reaches its limit, the LSU stores the sorted array back into the SPM at line 13. Dividing the original array into two sub-arrays allows the parallelization of the sorting algorithm on the RCs. However, it requires an extra step to recover the complete sorted array. In this specific case, the sorting algorithm is used to compute a median; therefore, only half of the complete sorted array is necessary to obtain the median. This limits the recovery step on the VWR2A. This step is not shown in Figure A.1d for conciseness.

The outer loop size for the VWR2A is larger compared to the ARM Cortex-M4 and the Ibex processors because the RCs have to pass the index of the smallest values to the MXCU through the single-port SRF. However, the impact of the outer loop on the overall performance is limited. Moreover, the number of outer loop iterations is divided by two, compared to the processors, thanks to the parallelization of the algorithm.

BENOIT DENKINGER

Engineer and Doctor in Electrical Engineering & Computer Science

+41 78 827 87 90

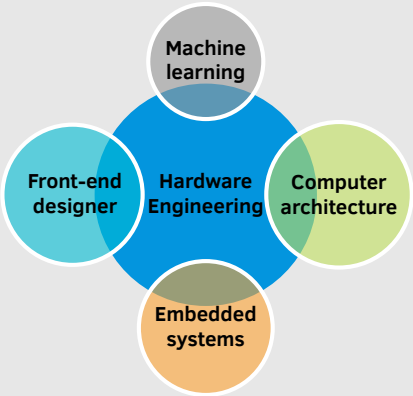
benoit.denkinger@gmail.com

/in/benoît-denkinger

benoitdenkinger

Technical Skills

Overview



Programming

SystemVerilog • Verilog • VHDL • Tcl

C • C++

Python • Matlab • LaTeX

Tools

Design Compiler • Genus • Innovus
Questasim • Vivado
Github/Gitlab • Tensorflow • LabView
GNU/Linux • Mac OS X • Microsoft Windows

Languages

French: native speaker
English: professional knowledge

Education

MSc., Micro-engineering
Robotics and Autonomous Systems
EPFL | 2014 - 2017 | Lausanne, CH

Experience

Sep 2017 - Present **Doctorate in EPFL, Embedded Systems Laboratory** ESL

- **Thesis title**: Exploring brain-inspired multi-core heterogeneous hardware templates for low-power biomedical embedded systems
- **Teaching**: Lab in EDA based design, Design and Optimization of Internet-of-Things Systems, IC design I, Lab on hardware-software digital systems codesign
  Tools: Cadence (Genus, Innovus, Virtuoso), Synopsys (Design compiler, Primepower), Siemens (Questasim), Xilinx, Vivado, Github, C, C++, Tensorflow

Sep 2016 - Jan 2017 **Internship at Nestlé Nespresso SA in Lausanne** Nespresso

- Next-generation coffee-machine development
- **Tools**: LabView
- **Patent**: Beverage preparation machine with enhanced pump control

Jun 2016 - Aug 2016 **Internship at Productize in Brussels** Productize

- IoT devices prototyping
  - Developed IoT demos based on LoRa, from PCB to software
- **Tools**: KiCad, mbed, C, C++

Education

2014 - 2017 **Master of Science in Microengineering** EPFL
Major: Robotics and Autonomous Systems
Minor: Space Technologies

Master project title: Development of walking assistive strategies for a lower limb exoskeleton device
- Implemented transparency of the 6 DOFs of AUTONOMYO exoskeleton
- Designed walking assistive strategies based on impedance control considering states depending on the different gait phase
- Implemented and experimented with a sample of selected strategies

- **Tools**: C, C++, Matlab, Webots
- **Patent**: Bio-inspired adaptive impedance based controller for human-robot interaction and method

Publications

- *An Active Impedance Controller to Assist Gait in People with Neuromuscular Diseases: Implementation to the Hip Joint of the AUTONOMYO Exoskeleton*, IEEE Biorob, 2018
- *Impact of Memory Voltage Scaling on Accuracy and Resilience of Deep Learning Based Edge Devices*, IEEE Design & Test, 2020
- *Modular Design and Optimization of Biomedical Applications for Ultralow Power Heterogeneous Platforms*, IEEE TCAD, 2020
- *VWR2A: A Very-Wide-Register Reconfigurable-Array Architecture for Low-Power Embedded Devices*, DAC, 2022
- *A hardware/software co-design vision for deep learning at the edge*, IEEE Micro, 2022
- *Acceleration of Control Intensive Applications on Coarse-Grained Reconfigurable Arrays for Embedded Systems*, IEEE Transactions on Computers, pending