TÉCNICO
LISBOA

Packing and Fusing Narrow-Width Vector Operations for Energy-Efficient SIMD

Miguel Eduardo Mateus Pinho

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Nuno Filipe Valentim Roma
Member of the Committee: Prof. Nuno Cavaco Gomes Horta

July 2020

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I want to start by thanking my family for their unwavering support and dedication: my parents, my brother, and my sister. I want to thank my supervisors, Prof. Nuno Roma and Prof. Pedro Tomás, for challenging me with this research opportunity, and for their constant feedback and advice, and thorough revisions. Had it not been for their demanding yet supportive attitude, this work would not be possible, and neither the personal growth it enabled. I had the luck of sharing the many challenges encountered in this thesis work with my colleague and friend João Mário, who always had some piece of advice. I also want to thank João Ramiro for his help in the thesis revision. The academic journey that now comes to an end would not have the same meaning without the friends from *SIIIIIIIIM* that accompanied me. I particularly treasure the companionship with Miguel Malaca and Pedro Mendes, with whom I overcame many a challenge (and who also helped with the revisions). This voyage was made even more significant by everyone I met at JUNITEC, where I found some of the most entrepreneur, dedicated, and motivated young people of my generation. Someone very special was also waiting there for me to meet, her name Sara Farias.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017, and by funds from the European Union Horizon 2020 research and innovation programme under grant agreement No. 826647.

Abstract

Application developers usually decide on the size of each variable data type by either considering its maximum range or simply by comfortably using larger data types. Since these represent maximum values (and not typical), the applications most often do not make full use of the bit-width offered by the processor integer arithmetic units. This wasted bit-width is especially relevant when using Single Instruction Multiple Data (SIMD) instructions, since the inefficient use of each arithmetic unit is multiplied by the number of vector elements. This (rather frequent) circumstance is herein exploited by proposing new run-time mechanisms to (i) efficiently handle narrow integer vector elements, by removing excess sign bits and packing these elements in a smaller vector, and (ii) agglomerate (fuse) multiple vector instructions pending in the execution queue of the processor, to simultaneously execute them on a single SIMD unit. When combined with clock and power gating techniques, the proposed approach provides a very significant reduction of the energy consumption in the SIMD units, by dynamically optimizing the execution of narrow-width integer vector computations, with low hardware overhead and no need for any changes in the application executable. Experimental results, based on a prototyping implementation supported on an ARM Cortex-A76 model, show a reduction of the dynamic and leakage energy consumption of the vector units of up to 54%, with either a negligible performance reduction or even some slight improvements of the execution time.

Keywords

Narrow-width, SIMD Units, Clock and Power Gating, Energy Efficiency, General-purpose Processors

Resumo

Os programadores de software normalmente decidem qual o tipo de dados de uma variável considerando o seu valor máximo esperado, ou simplesmente usando um tipo de dados com elevada precisão. Como os valores reais tendem a ser bastante inferiores a este valor máximo, grande parte da precisão das unidades aritméticas inteiras é desperdiçada. Em instruções vetoriais (ou *SIMD*) este desperdício da precisão do processador é ainda mais relevante, pois é multiplicado pelos vários elementos do vetor. Esta situação frequente é particularmente explorada neste trabalho através da proposta de novos mecanismos em *hardware* para (i) manipular elementos vetoriais inteiros de precisão reduzida, removendo os bits de sinal redundantes e compactando os elementos num vetor menor, e (ii) aglomerar (fundir) várias operações vetoriais à espera de execução na fila do processador, para execução simultânea numa única unidade de execução. Quando combinada com as técnicas de bloqueio do sinal de relógio e de alimentação, esta abordagem permite reduzir significativamente o consumo energético das unidades vetoriais, promovendo-se uma optimização dinâmica de cálculos vetoriais de inteiros, com uma penalização reduzida em termos de recursos de *hardware* e sem necessidade para quaisquer alterações nos ficheiros executáveis. Os resultados experimentais obtidos, baseados na prototipagem destes mecanismos com um modelo do processador ARM Cortex-A76, mostram uma redução do consumo de energia (dinâmica e estática) na unidade de execução vetorial de até 54%, com uma redução mínima no desempenho ou, até mesmo, em certas aplicações, com algumas melhorias ligeiras.

Palavras Chave

Precisão Reduzida, Unidades SIMD, Técnicas de Bloqueio de Relógio e de Alimentação, Eficiência Energética, Processadores de Uso Geral

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Objectives . . . 3

1.3 Thesis contributions . . . 3

1.4 Thesis outline . . . 4

2 Background and Related Work **6**

2.1 Contemporary GPP architectures . . . 7

2.2 Power efficiency in computer architectures . . . 10

2.2.1 Prevailing techniques for power efficiency . . . 11

2.2.2 Power gating unused functional units . . . 12

2.2.3 Reducing the SIMD units power dissipation . . . 14

2.3 Scalable width datapaths . . . 15

2.3.1 Software based techniques . . . 18

2.3.2 Dynamic hardware-based approaches . . . 19

2.3.3 Scalable width structures . . . 20

2.3.3.A Functional units . . . 21

2.3.3.B Register file . . . 22

2.3.3.C Caches . . . 24

2.4 Summary . . . 24

3 Narrow-width Opportunity in SIMD **25**

3.1 Defining narrow-width in SIMD computations . . . 25

3.2 Optimizing vector computations in out-of-order processors . . . 28

3.3 Profiling integer intensive applications . . . 29

3.3.1 Benchmarked applications . . . 30

3.3.2 SIMD unit usage analysis . . . 31

3.4 Envisaged energy savings . . . 34

3.5 Summary . . . 36

4 Architectural Mechanisms to Exploit Narrow-width 37
4.1 Proposed Mechanisms 38
4.1.1 Width encoding 38
4.1.2 Packing narrow-width vector operands 42
4.1.3 Fusing vector operations 45
4.1.4 Gating functional units 46
4.2 Integration in conventional processor architectures 48
4.3 Summary 50

5 Prototyping and Experimental Workflow 51
5.1 Architectural simulation tools 52
5.2 ARM ISA and the NEON vector extension 55
5.3 Implementation of the architectural changes 56
5.4 Traces and performance counters 58
5.5 Power modelling 60
5.6 Experimental workflow 61
5.7 Summary 62

6 Experimental Evaluation 64
6.1 Evaluation methodology 64
6.1.1 Considered configurations and benchmarks 64
6.1.2 Evaluated metrics 66
6.2 Experimental results 67
6.2.1 Energy and performance impact of the proposed mechanisms 67
6.2.2 Design parameters exploration 73
6.3 Summary 75

7 Conclusions and Future Work 76
7.1 Conclusions 76
7.2 Future Work 77

Bibliography 80

Appendix A Considered Benchmark Datasets 86

Appendix B ARMv8 NEON Instructions 88

List of Figures

1.1 Outline of the thesis work . . . 4

2.1 Microarchitecture of the Arm Cortex-A76 . . . 7
2.2 SIMD execution model . . . 9
2.3 Functional unit's power gating control mechanism . . . 13
2.4 Time-intervals in power gating . . . 14
2.5 Narrow-width integer representation . . . 15
2.6 Cumulative distribution of narrow-width values occurrence for SPEC2000 . . . 16
2.7 Distribution of integer operations grouped by classes, for SPECint2000 applications . . . 17
2.8 Operation fusing example . . . 19
2.9 Cluster partitioned approach for narrow-width exploitation . . . 21
2.10 Vectorized MAC design . . . 22
2.11 Implementing 32-bit, 16-bit, and 8-bit operation modes in a 64-bit scalar MAC unit . . . 23

3.1 Width of a vector operation . . . 26
3.2 Overview of vector operation modes in Arm NEON . . . 27
3.3 Details of the SIMD execution engine in the out-of-order microarchitecture . . . 29
3.4 Evaluation of the usage of the SIMD unit for the selected benchmarks . . . 32
3.5 Analysis of the width required by integer vector operations . . . 33
3.6 Execution samples for the mini-apps . . . 34
3.7 Maximum expected dynamic energy savings . . . 35

4.1 Examples of the widths detected with different width-block sizes . . . 38
4.2 Width encoding example . . . 39
4.3 Details of the width encoding procedure of an operand . . . 40
4.4 Width masks examples for different vector modes . . . 41
4.5 Vector operand packing in a SIMD Add . . . 43
4.6 Implementation of a irregular element size vectorized adder unit . . . 44

4.7 Selection of partial products for an irregular element size multiplication 45
4.8 Fusing two similar vector operations 46
4.9 Example of opportunities for clock and power gating vector functional units 47
4.10 Detailed changes proposed in the out-of-order execution engine 48

5.1 Comparison of different processor specification and modelling levels 52
5.2 Comparison of gem5 simulation modes and models 53
5.3 Modifications made to the internal structure of gem5's O3 CPU model 56
5.4 Block diagram of the McPAT framework 60
5.5 Description of the experimental workflow 62

6.1 Energy reduction and normalized execution time with the Cortex-A76 core 68
6.2 Energy reduction and normalized execution time with the High-performance core 69
6.3 Active rate of the SIMD units 71
6.4 Percentage of issued instructions that are fused 72
6.5 Impact of adding a new pipeline stage 73
6.6 Comparison of energy savings with different width-block sizes 75

7.1 Possible extension to the proposed architecture 78

A.1 Cumulative distribution function of the bit-width of the values in the random datasets . . . 87
A.2 Energy savings for different datasets 87

List of Tables

1.1 Number and width of integer SIMD units in recent out-of-order processors 2

3.1 Complete list of profiled benchmarks 31

4.1 Width-mask size trade-off 42

5.1 Baseline parameters of the CPU models 54

6.1 Considered simulation modes 65
6.2 Selected benchmarks for this evaluation 65
6.3 Best energy savings Cortex-A76 core 74
6.4 Best energy savings High-performance core 74

A.1 Sample values from each dataset 87

B.1 List of ARMv8 NEON vector instructions 88

Acronyms

ALU Arithmetic Logic Unit

CDB Common Data Bus

CLA Carry-Lookahead Adder

CPU Central Processing Unit

DSP Digital Signal Processor

FP Floating-Point

FU Functional Unit

GPP General Purpose Processor

GPU Graphics Processing Unit

HPC High-Performance Computing

ILP Instruction-Level Parallelism

IPC Instructions Per Cycle

ISA Instruction Set Architecture

MAC Multiply-Accumulator

MIMD Multiple Instruction Multiple Data

RAT Register Alias Table

RISC Reduced Instruction Set Computing

ROB Re-Order Buffer

ROI Region Of Interest

RTL Register-Transfer Logic

SIMD Single Instruction Multiple Data

SMT Simultaneous Multi-Threading

WAR Write After Read

WAW Write After Write

1

Introduction

Contents

1.1 Motivation 1
1.2 Objectives 3
1.3 Thesis contributions 3
1.4 Thesis outline 4

1.1 Motivation

The increasing number of transistors that implement today's microprocessors has not been accompanied by a corresponding increase in the power efficiency of each transistor, in what is known as the failing of Dennard scaling [1, 2]. As a result, power consumption has been the most limiting factor to increase the microprocessors' performance, as a result of the difficulty to dissipate the energy and fully utilize the chip at its maximum clock rate, without compromising its integrity. This power dissipation wall has been imposing a stagnation in the offered computer performance, leading to the need for a computer design shift towards power efficiency [3–6].

General Purpose Processors (GPPs), which will be the focus of this work, have been increasingly used for High-Performance Computing (HPC), such as multimedia, scientific, and engineering applications [6, 7]. Successive architectures in this segment have evolved to match the increased computing power demand by adopting an out-of-order architecture and by having wider issue widths, larger register files, and more functional units. A particular trend in modern Instruction Set Architectures (ISAs) is to include Single Instruction Multiple Data (SIMD) vector extensions [6–9], which allow the execution of

the same operation over a vector of several data values in parallel. These instructions provide considerable speed-ups in workloads with data-level parallelism, which are common in HPC. As these SIMD extensions have evolved to adapt to a broader range of applications, the supported vector width has increased to enable higher performance benefits. The recent trend has been to double the vector register size every four years [7–9]. For example, the successive Intel x86 vector extensions have gone from 64-bit vectors (MMX), to 128-bit (SSE) and 256-bit (AVX), to more recently 512-bit (AVX-512) [7, 10]. The ARM SVE extension is already designed to scale up to 2048 bits [9].

However, all these architectural features put an even higher strain on these chip's power limitations. In particular, the execution engine has become an increasingly important power drain in the processor cores, mainly due to the leakage power in its functional units [11, 12]. Moreover, as SIMD extensions became increasingly relevant in high performance and their vector length increased, several and wider integer vector units have been included per core, as presented in Table 1.1. Hence, these units have become a very significant source in power dissipation in processor cores, so they are a particularly relevant candidate for further improvements in power efficiency.

Table 1.1: Number and width of integer SIMD units in recent out-of-order processors

Microarchitecture	Domain	ISA (vector extensions)	Integer SIMD Units
ARM Cortex-A76/A77/A78	Mobile	ARM (NEON)	2 × 128-bit
Samsung Exynos M3/M4	Mobile	ARM (NEON)	3 × 128-bit
Apple A12 (Vortex) / A13 (Lightning)	Mobile	ARM (NEON)	3 × 128-bit
ARM Cortex-X1	Mobile	ARM (NEON)	4 × 128-bit
ARM Neoverse N1	Server	ARM (NEON)	2 × 128-bit
AMD Zen/Zen+	Desktop, Server	x86 (SSE, AVX)	4 × 128-bit
AMD Zen2	Desktop, Server	x86 (SSE, AVX)	4 × 256-bit
Intel Sunny Cove	Desktop, Server	x86 (SSE, AVX, AVX-512)	2 × 256-bit, and 1 × 512-bit (Server)

In this thesis, a still unexplored opportunity for reducing the power dissipation in SIMD units when handling integer computations is identified and evaluated. The conducted research arose from the observation that most integer computations do not require the whole word width, as their operands can be encoded with a lower number of bits, and can be executed using only a portion of the functional unit's width. In previous literature [13–16], these computations have been denoted as narrow-width, but only scalar instructions were exploited.

However, although vector extensions usually provide instructions to support several element data sizes, these instructions pose the constraint that all elements in a vector must have the same bit-width. Even though most integer element values can be represented using narrow data types (i.e. 8 or 16 bits), the occurrence of a small portion of wider values makes it necessary for the application programmer to use a larger element size, to avoid overflow. Hence, for a large portion of the vector lanes, the bit-resolution of the vector functional units is wasted, which means that a very significant portion of the SIMD unit does not perform any computation.

1.2 Objectives

The main objective of this research is to evaluate the opportunity to exploit narrow-width integer SIMD computations in out-of-order processors and to propose architectural mechanisms to increase power efficiency when handling intensive data-parallel integer workloads.

Achieving this goal first requires collecting and profiling a set of representative applications, which should be not only integer intensive, but also vectorizable, measuring relevant metrics about the width required by the operations and the degree of usage of the SIMD units. These two metrics will show, respectively, the existence and the relevance of this opportunity, and will provide insight on how to design mechanisms to exploit it.

The next step is to propose architectural changes to exploit the narrow-width opportunity, taking into account the characteristics and constraints imposed by out-of-order superscalar processors. The proposed mechanisms should be transparent to the compiler (alleviating the need for code recompilation and toolchain changes) and ISA agnostic (to be adaptable to different architectures and vector lengths). This design step must take into account the compromise between the expected power efficiency gains and the complexity and penalty of implementing the proposed mechanisms in hardware.

Afterwards, the proposed mechanisms should be evaluated using a state-of-the-art architectural simulation tool, making the necessary modifications to implement these architectural changes. Then, a relevant set of metrics to evaluate the impact in the processor operation should be identified and generated by the simulation. Finally, these metrics should be used to estimate the resulting impact in terms of performance and energy consumption, using an adequate power model.

1.3 Thesis contributions

The main novelty presented in this thesis consists in a viable exploitation of narrow-width in SIMD integer operations. This contrasts with other previous authors' works, who identified this narrow-width opportunity, but focused only on scalar instructions. On the other hand, the SIMD execution unit provides particular challenges for optimization, as the overhead of detecting the required bit-width for each vector element is multiplied by the number of elements. Moreover, the proposed scheme must be designed to handle the multiple element modes that vector extensions usually support (e.g. 64, 32, 16, or 8-bit elements). In accordance, the main contributions of this thesis are:

- The identification of a novel opportunity for reducing the SIMD unit power consumption, by exploiting narrow-width vector computations, and an evaluation of its relevance in a variety of applications;
- The proposal of low-overhead mechanisms for dynamic detection of the required operands' width for each SIMD lane (width encoding);

- A new scheme (operand packing) to efficiently pack integer vector operands, by discarding unnecessary sign bits between elements - these packed vectors can execute directly in an available portion of existing SIMD units (with minor changes);
- A new mechanism (operation fusing) to agglomerate multiple packed vector instructions for simultaneous execution in a single functional unit;
- A set of architectural modifications to the out-of-order vector execution pipeline of a modern processor to support the proposed packing and fusing mechanisms, as well as to provide the means for a more aggressive application of power and clock gating;
- An implementation of these modifications in the gem5 simulator, using the ARMv8 ISA and its NEON advanced vector extension as a proof of concept, and with processor models based on the ARM Cortex-A76 core;
- An estimation of the impact that the proposed architectural changes have in the energy consumption of the SIMD unit and in its execution time, by adding relevant performance counters and using the McPAT power modelling framework.

1.4 Thesis outline

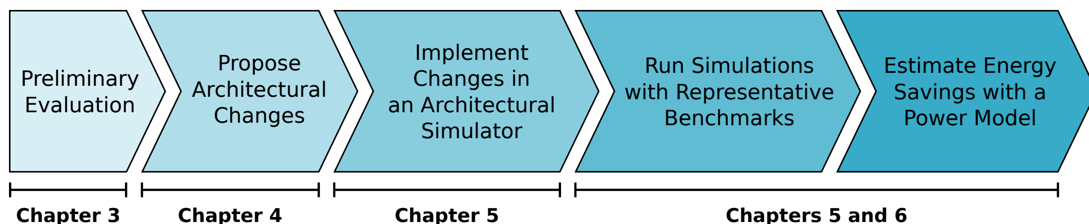


Figure 1.1: Outline of the thesis work, with the corresponding steps in the evaluation of the narrow-width opportunity in SIMD.

Chapter 2 starts by introducing the computer architecture concepts and the processor research background that are necessary to understand the work that is developed in this thesis. Then, the previous research works that exploited the narrow-width opportunity in the scalar integer pipeline are also reviewed. As depicted in Figure 1.1, Chapters 3 to 6 follow the typical steps in the evaluation of a new architectural mechanism. Chapter 3 presents a preliminary evaluation of the narrow-width opportunity in SIMD operations, using a modified model of the ARM Cortex-A76 processor core, which shows there are very relevant energy efficiency gains to be exploited. Then, Chapter 4 proposes architectural mechanisms to exploit this opportunity, taking into account the typical design constraints in an out-of-order processor but without tieing these mechanisms to a specific architecture or implementation. Chapter 5

specifies the prototyping architecture and core models, and details how the proposed changes were implemented in a state-of-the-art processor simulator. This chapter also presents the power models used to evaluate the impact of the proposed architectural changes. Finally, Chapter 6 starts by listing the configurations and applications that were considered in the evaluation and discusses and analyses the main results that were obtained. Lastly, Chapter 7 addresses the main conclusions and outlines possible future work opportunities.

2

Background and Related Work

Contents

2.1 Contemporary GPP architectures 7

2.2 Power efficiency in computer architectures 10

2.3 Scalable width datapaths 15

2.4 Summary 24

The main architectural features of general-purpose computer architectures are first briefly reviewed, with a particular focus on the superscalar and out-of-order architectures and on the Single Instruction Multiple Data (SIMD) vector extensions. This background is crucial to set the constraints and opportunities for new architectural mechanisms. The prevailing techniques and mechanisms for power efficiency are also introduced, with a particular focus on how to reduce the power dissipation of the execution unit.

Then, the previous approaches for taking advantage of narrow-width integer operations are reviewed. This opportunity is first clearly identified, and the two different approaches to explore it are presented: statically (at compile time) and dynamically (at run-time). After identifying the dynamic approach as the most promising alternative, the most relevant architectural mechanisms proposed in the literature are also described. This review includes the existing work on optimizing the datapath structures, with a particular focus on the execution unit, namely the research on the implementation of addition and multiplication units with variable width.

2.1 Contemporary GPP architectures

Over the past 20 years, most General Purpose Processors (GPPs) have adhered to the superscalar and out-of-order architecture paradigms [17–19]. These processors take advantage of the Instruction-Level Parallelism (ILP) existing in programs, where several instructions can be executed at the same time, to extract higher levels of performance. Figure 2.1 depicts the microarchitecture of a modern superscalar out-of-order core, the Arm Cortex-A76 launched in 2018, and is used to illustrate the computer architecture concepts and structures which are introduced in this section. This representative microarchitecture will be also used as an example for prototyping the work developed in the following chapters.

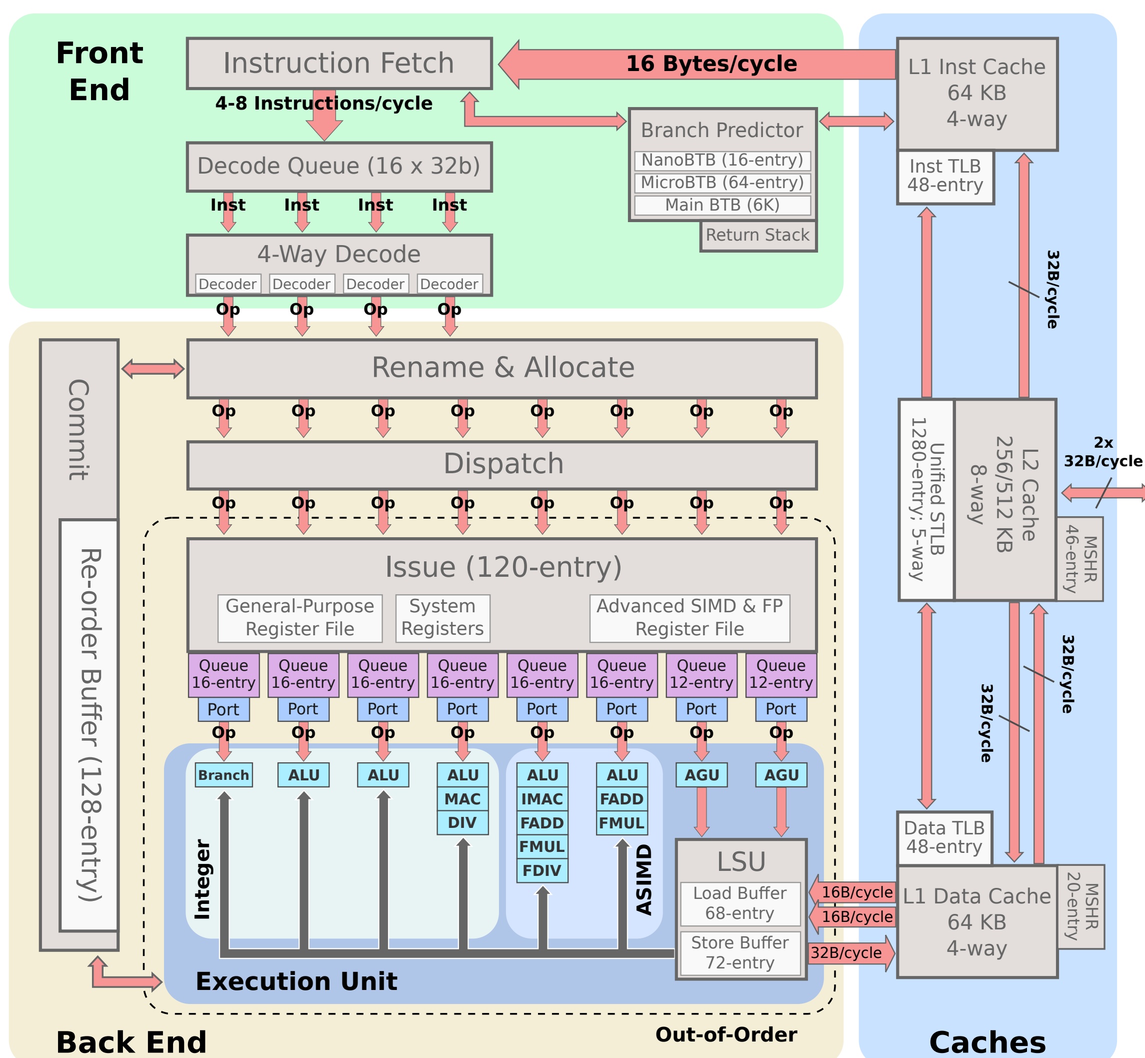


Figure 2.1: Microarchitecture of the Arm Cortex-A76, a modern 4-wide superscalar out-of-order core (based on the figure from `https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76`). Every cycle, this microprocessor can fetch and decode up to four new instructions, and can dispatch up to eight operations to the eight available pipelines, each with its independent issue queue.

A superscalar processor tries to fetch and dispatch several instructions in the same clock cycle and has multiple and different functional units to execute several instructions in parallel (see Fig. 2.1), as long as the original program dependencies are respected. Superscalar execution is usually complemented by out-of-order instruction scheduling, where instructions can be executed in a different order than the original program. Instructions fetched later are allowed to execute first, as long as all their dependencies with previous instructions have already been solved.

However, when instructions are executed out-of-order, the completion of a more recent instruction may overwrite a register operand that is still required by older instructions. Futhermore, if two instructions write to the same register in the wrong order, the final value in that register is incorrect. These two data hazards, denoted as Write After Read (WAR) and Write After Write (WAW), respectively, are usually solved by another mechanism typically associated with these processors: register renaming. Register renaming solves the reuse of the same architectural register, which is not an actual dependency, by allocating a new physical register to store the new value. The correspondence between the architectural registers (in the code) and the physical registers (in the hardware) is abstracted by using an intermediate table of pointers, which is usually called Register Alias Table (RAT) or rename map. Hence, a new rename stage is added after the instructions are decoded and before they are dispatched to an issue queue (see Fig. 2.1), which allocates a new register for the result and updates the corresponding pointer in the RAT.

However, the occurrence of branch instructions greatly limits the possibility of exploiting ILP, since which instructions should be executed after depends on the result of the branch condition. This control dependency is solved by predicting the result of the branch and fetching instructions accordingly. The superscalar out-of-order scheme is therefore complemented with speculative execution, where instructions resulting from branch prediction are allowed to execute but not to commit their results immediately, that is, to change any permanent state of the processor. This way, in the event of the prediction failing, the correct execution state can be recovered by deleting the instructions waiting to commit, without leaving any lingering effect. An auxiliary structure keeps track of the original instruction issue order, the Re-Order Buffer (ROB), so that the instructions are committed in order and only when they are no longer speculative.

These techniques allow for an increased performance, as the processor can maintain a higher instruction throughput and can more easily hide the latency of more time-consuming instructions (e.g. floating-point, SIMD and memory operations). However, this is traded for increased complexity in the processor control structures, as it must be able to fetch, decode, rename, execute, and commit several instructions at the same time, verifying and handling all the dependencies between them. A classical approach to handle all this complexity is to distribute the control of the instruction scheduling, in a scheme known as the Tomasulo's algorithm [17]. This approach relies on dispatching the decoded instructions to

one of the multiple issue queues available (also called reservation stations), each associated to a cluster of functional units (execution ports), depending on the instruction type (see Fig. 2.1). Each issue queue is responsible for monitoring when its instructions are ready for execution, by scanning the Common Data Bus (CDB) (shared with the other queues) for the needed operands and buffering them when they are produced. In each clock cycle, it selects as many ready instructions as possible and schedules them for execution, limited by the available functional unit resources.

To sum up, a modern superscalar speculative processor executes instructions out-of-order to exploit ILP, but has to operate in-order in the first pipeline stages (fetch, decode, rename, and dispatch) and in the commit step, so that an application is correctly executed. By following the usual terminology, the front-end is the part of the processor core responsible for fetching and decoding new instructions, and the back-end (or execution engine) is where these instructions are executed.

However, the amount of ILP that can be exploited is often limited [20]. As a result, and to better use the available resources, several modern processors also exploit another form of parallelism: multithreading. In multithreading, different threads that share the same process addressing space are executed concurrently, i.e. without the need for any process context switch. By using the register renaming and dynamic scheduling capabilities of a superscalar processor, instructions from different threads can be executed simultaneously (in the same core) in an approach called Simultaneous Multi-Threading (SMT) [21].

SIMD extensions

Another form of parallelism that can be exploited for increased performance is data-level parallelism, where the same operation is performed over several data elements in parallel, following the Single Instruction Multiple Data (SIMD) execution model (see Fig. 2.2). This type of parallelism was first exploited in graphics and multimedia applications, often supported in domain-specific architectures, Digital Signal Processors (DSPs), stream processors, and Graphics Processing Units (GPUs) [22]. However, SIMD architectures have since found application in other broader domains, such as scientific and engineering computing, and machine learning [6, 9].

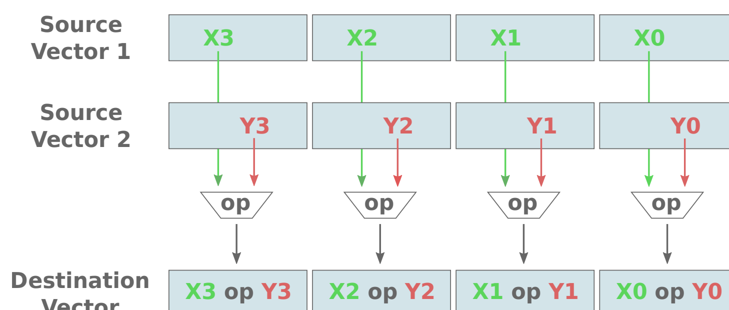


Figure 2.2: SIMD execution model [6].

As the relevance of these application domains increased, new instructions were gradually added to GPPs following the SIMD model, in what is usually called a vector extension in the Instruction Set Architecture (ISA). These SIMD instructions allow for the efficient manipulation (in parallel) of vectors of several data elements, reducing the number of instructions needed to process the input data. Initially, these instructions used the regular 64-bit registers and functional units, packing together several small width values, such as 8-bit and 16-bit [13], but eventually brought the appearance of dedicated vector registers and functional units, with larger widths. For example, the ARM Cortex-A76 microarchitecture, presented in Figure 2.1, includes a dedicated vector register file and two vector execution pipelines, which are shared with the Floating-Point (FP) operations. Since its version 7, the ARM ISA includes an Advanced SIMD extension, which has dedicated vector registers with 128-bit (in version 8), and whose implementation in ARM cores is called NEON (see Section 5.2 for more details).

The current trend is to increase the vector length with each new ISA generation to extract higher performance gains [7, 10]. This trend is exemplified by successive Intel architectures, where the MMX vector extension (1997) started by sharing the scalar 64-bit registers, and the SSE extension (1999) introduced dedicated 128-bit vectors registers, which were increased to 256-bit in AVX (2008) and then to 512-bit in AVX-512 (2013). ARM has taken a step further by introducing SVE, a SIMD extension that can scale to a vector length of up to 2048 bits [9]. As the vector registers increased in size, the datapath resources that are necessary to perform the vector operations have increased too, namely the SIMD register file and execution units. The drawback is the increase in power dissipation [11].

2.2 Power efficiency in computer architectures

The mechanisms presented in Section 2.1 have allowed new levels of performance to be attained but required a corresponding scaling of the available architectural resources. More functional units of each instruction type were added, the decode and commit stages were widened to support a larger instruction bandwidth, and the issue queues and ROBs sizes were increased (see the example microarchitecture in Fig. 2.1). The cost of these structures is paid in increased chip area and power dissipation [12, 18].

For many years, this performance scaling was supported by a steady advance in transistor integration technology, which was maintained at constant power density regimes. However, the reduction in transistor size that allowed for tighter integration had to be accompanied by a proportional reduction in their power dissipation through a downscaling in voltage and current. This is known as Dennard scaling and was valid for 30 years [23, 24]. Nevertheless, this scaling has broken down in the past two decades, as a consequence of the increasing sub-threshold leakage current in the transistors, and because their threshold voltage turns further reductions in the supply voltage unfeasible [1, 2]. These factors have been fixing the power per transistor, even when their dimensions are further reduced.

As a consequence, the overall power consumption by chip area has been steadily scaling upwards with each microprocessor generation. At the same time, the power that can be dissipated by the chip without compromising its integrity is limited, originating a ”Power Wall” that limits performance increases. Hence, power consumption became one of the major constraints in computer design, and architectural research has shifted towards power efficiency [3–5], which continues to be a focus in present-day research in all computing domains, from mobile to High-Performance Computing (HPC) [25–28].

Although superscalar processors provide enough architectural resources to sustain a high execution throughput (e.g. several functional units for each operation type and large physical register files), these resources are highly underused during large portions of the execution, namely in code regions with low ILP [29]. A research trend for increasing power efficiency is to implement mechanisms to turn off unneeded resources and cut their power waste [3,5,28,30,31]. A particular focus is given in the literature to optimizing the usage of functional execution units, as they represent a very significant fraction of the energy consumption [11,12,29,32–34]. Some of the most prevailing approaches will be briefly reviewed in the following subsections.

2.2.1 Prevailing techniques for power efficiency

The two principal sources of the microprocessor power consumption are dynamic power and leakage power [3, 30]. Dynamic power is caused by the charging and discharging of the transistors' capacitive load, when switching between states. In the CMOS technology, the dynamic power consumption of a logic block is a function of its activity, as a logic gate only dissipates dynamic power when it switches. This power component is approximated by

$$P_{dynamic} \simeq \alpha C {V_{DD}}^2 f, \tag{2.1}$$

where C is the equivalent capacitance of the load circuit, V_{DD} is the supply voltage, and f is the operating frequency. α is the activity factor, which varies between 0 and 1, and it is the average fraction of gates switching each cycle.

A significant portion of the dynamic power dissipation is caused by the clock signal tree, which switches at a high frequency and drives a high load. Hence, a common approach for reducing the dynamic power is to turn off the clock signal for unneeded logic blocks [3, 5, 30, 31]. This technique is called **clock gating** and is implemented by partitioning the clock network and by adding enable signals to toggle each portion.

The other main component of power dissipation is leakage (or static) power, which is caused by the transistors' leakage currents, which dissipate power even when the gate is not switching [3]. It is composed of two main components: sub-threshold leakage, which is due to weak inversion currents

across the device; and gate leakage, which is caused by tunnelling currents through the gate oxide. Hence, leakage power is given by

$$P_{leakage} = V_{DD} I_{leak} = V_{DD}(I_{sub} + I_{ox}), \tag{2.2}$$

where V_{DD} is the supply voltage, I_{sub} the sub-threshold leakage current, and I_{ox} is the gate leakage current. Due to the shrinking in transistor size and the consequent increase in leakage current, leakage power has become extremely relevant and is a major concern in microprocessor design [4, 32].

A common technique for reducing leakage is **power gating**, where the power supply for the idle logic blocks is cut off [32, 35–38]. Power gating suppresses both dynamic and leakage power consumption in that block, but has the drawback that it requires long wake-up delays when restoring the supply voltage, and has a significant energy overhead in the state transitions. Hence, its usage only outweighs the cost when the gated block is idle for an extended period. Power gating is implemented by placing a header transistor between the supply voltage and the supply of the circuit portion to be power gated, which is controlled by a sleep signal. As clock and power gating have different trade-offs, it can be advantageous to add support for both techniques in the microprocessor's components and to trigger them concurrently [36].

Another widely used power management technique is **dynamic voltage and frequency scaling**, which is similar to power gating in the sense that either the supply voltage is lowered to reduce leakage currents, or the frequency is lowered to reduce the switching rate. However, instead of completely cutting the supply voltage or the switching currents, they are simply reduced gradually [30, 35]. Hence, the targeted circuit block can still operate normally, and the decreased voltage/frequency results in a consequent reduction of dynamic and leakage power (see Equations 2.1 and 2.2). However, as reducing the supply voltage increases gate delays, the frequency must also be lowered for the circuit to function properly. Hence, varying the voltage and frequency during execution allows different trade-offs between power saving and performance reduction.

2.2.2 Power gating unused functional units

Functional units represent a very significant fraction of the microprocessor power dissipation, particularly in terms of leakage power, and several previous authors have focused on reducing their power dissipation when idle, using power gating [11, 12, 29, 32–34]. The crucial challenge in this approach is deciding when to gate a functional unit. Power gating techniques have a significant latency when waking up a gated unit, which cannot be used immediately [29]. If a functional unit is kept in the sleep mode for too long, it might significantly hinder performance. Moreover, turning off the functional unit has a transition delay: when the supply voltage is decreasing, the power dissipation gradually decreases [32], so the

earlier the power gating is initiated in an idle period, the better (see Fig. 2.4). Finally, the transitions when turning off and on this mechanism have a power overhead, so the idle period must be expected to last long enough to outweigh that cost.

Rele et al. [29] proposed a software-based approach, where the hardware gating mechanisms are controlled by new instructions, which hint on when to turn functional units off or on. The compiler is responsible for identifying regions where each functional unit type has low activity and trigger their sleep mode.

More recent literature focuses instead on dynamic hardware mechanisms, which detect the run-time usage of the functional unit and use predictors to decide on whether to trigger power gating during idle periods [12, 32, 33]. These authors propose the addition of an architectural structure associated with the functional unit that monitors its activity (idle signal) and uses some control logic to decide on when to generate a sleep signal. When the sleep signal is triggered, a leakage control mechanism power gates the functional unit into a low leakage mode (Figure 2.3).

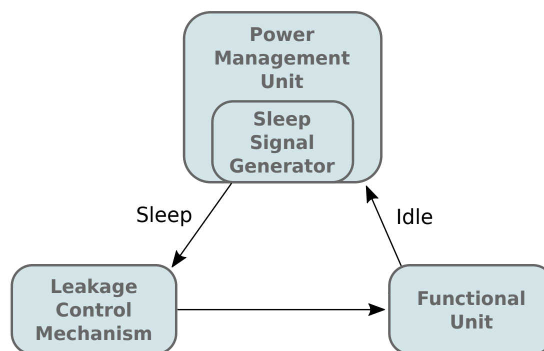


Figure 2.3: Functional unit's power gating control mechanism [12].

Hu et al. [32] proposed a model to evaluate the potential of power gating an idle functional unit, introducing three main intervals worth considering in a gating mechanism (see Fig. 2.4). The idle detect interval (A) begins when the functional unit becomes idle until the sleep signal is generated; the break-even interval (B) is the interval of time since the sleep signal is triggered until the leakage energy saved (E_{be}) outweighs the energy overhead in the sleep (E_{st}) and wake-up (E_{wt}) transitions; the wake-up interval (C) corresponds to the period from the moment when the wake-up signal is triggered until the functional unit is ready for issue.

Hu et al. [32] also proposed a mechanism for generating the sleep signal, consisting on a simple state machine which counts the number of idle cycles and triggers the sleep mode when a certain static threshold is reached. The threshold for each functional unit type is decided at design time, based on their model of the break-even interval and experimental testing. They reach a detection threshold of around 6 to 12 cycles for floating-point units, as a compromise between losing gating opportunities and mispredicting short idle periods. They reported that with the proposed mechanism, the floating-point

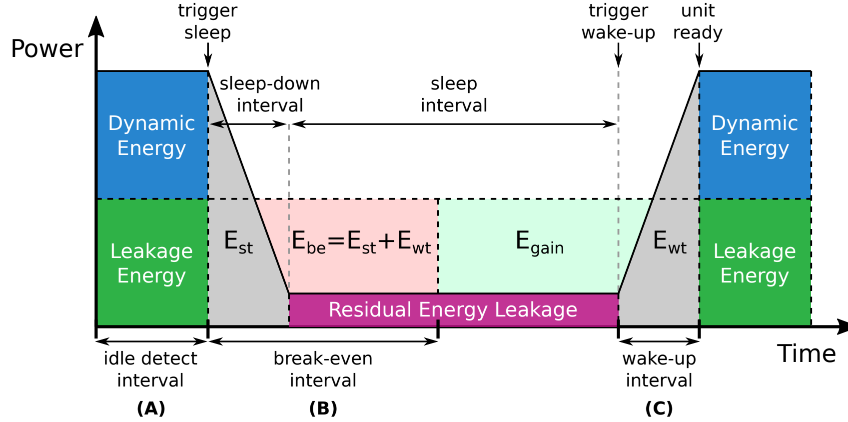


Figure 2.4: Time-intervals in power gating as described in [32] (also based on the figure in [39]). E_{be} is the break-even energy which is required to outweigh the overheads in the sleep transition, E_{st}, and in the wake-up transition, E_{wt}. The remaining leakage energy savings correspond to the actual energy gain, E_{gain}. The power curves in this diagram are simplified.

unit is power gated for up to 28% of the execution cycles, with only a 2% performance loss.

Youssef et al. [12] argued that the accuracy of this counter mechanism could be increased by allowing the sleep threshold to change dynamically, adapting to the currently running application. Their implementation raises the detection threshold when idle periods are frequently interrupted by functional unit usage spikes, and lowers the threshold when the functional unit usage is infrequent and the probability of saving energy when gating is high. They reported that their mechanism improved the accuracy of correctly detecting an idle period from an average of around $40-60\%$ to 98%. Lungu et al. [33] argued that even though these mechanisms can provide very relevant leakage power savings (as much as 99%), they can also cause significant increases in power consumption (up to 70%) when there is a systematic misprediction. They proposed improving the sleep control unit by adding a success monitor, which predicts whether a gating opportunity is likely to cause power loss based on previous occurrences, cancelling the sleep signal if that probability is high. They further proposed introducing a maximum power loss threshold which, when exceeded, aborts all sleep opportunities during a specified period.

2.2.3 Reducing the SIMD units power dissipation

SIMD functional units, due to their larger width, represent a significant contribution to power dissipation, namely leakage power, so they are a particularly relevant candidate for applying power efficiency techniques. Kumar et al. [11] tackled this problem in a complementary way to the previously discussed gating mechanisms, focusing on keeping the SIMD units idle for long periods. By increasing the duration of gating intervals and reducing the transitions between the sleep and ready states, the effectiveness of the gating mechanism is improved.

To attain this objective, they propose a mechanism that identifies short duration spikes in the SIMD

functional unit usage, and devectorizes the corresponding instructions, trading a slight performance penalty for more significant power benefits. However, their approach has a significant drawback, as the profiling and devectorization are performed dynamically at run-time, which requires an extra layer between software and hardware, with power and delay penalties.

Another shortcoming is the performance reduction from the devectorization. However, the authors propose an interesting solution for reaching a better compromise between the power dissipation reduction and this performance decrease, by performing partial devectorization. In regions with intermediate SIMD intensity, they propose gating only half of the SIMD unit and dynamically splitting vector operations into two, so that each half of the vector is computed separately. However, this approach increases even more the additional complexity in the dynamic translation between software and hardware.

2.3 Scalable width datapaths

Integer values are typically stored and computed using the processor's default bit-width, which for most modern GPP corresponds to 64 bits. However, most of the computed values are small when compared to the full integer range and can be encoded using only a small portion of those bits. The remaining bits that are not required to encode the value represent a waste of energy, as they are not required for computations or for storing that value. The values or computations that present this reduced width pattern have been labelled as narrow-width in the literature [13]. In particular, optimizing the architecture when handling computations over narrow-width operands is a very relevant opportunity for reducing power consumption in the execution unit.

Brooks and Martosini [13] and some later authors [14, 15] have defined the narrow-width representation of an integer value by removing its redundant sign bits on the left: the leading zero-bits or one-bits. This narrow-width definition is especially relevant and prevalent for integer values, where this reduction in the number of bits is done without losing any precision. The width of an integer value is then defined, in bits, as the index of the least significant sign bit (see Fig. 2.5).

Decimal	Binary	Width
25	...0000011001 (6 5 4 3 2 1)	6 bits
-8	...1111111000 (4 3 2 1)	4 bits
0	...0000000000	1 bit
-1	...1111111111	1 bit

Figure 2.5: Narrow-width integer representation, where redundant sign bits are removed. The index of the least significant sign bit corresponds to the value's width.

Furthermore, an operation itself can be labelled as narrow when the calculation involves two values

of limited width, hence requiring less logic to evaluate. Previous authors have differed slightly in how they classify an operation's width. Some authors only consider the width of the input operands [13], while others also take into account the resulting value [14, 40]. Although it is straightforward to consider the width of an operation as the largest width among its operands, Brooks et al. [13] note that when computing some addition operations, it may suffice to evaluate the portion corresponding to the narrower operand. If there is no carry from the least significant portion of the operation, the remaining bits of the wider operand can be propagated directly. However, this specific optimization is highly dependent on the operation type and the specific value of the operands.

Several authors have evaluated the relevance of exploiting narrow-width operations, using integer intensive or multimedia benchmarks. Based on the SPECint95 benchmarks and a modified SimpleScalar superscalar processor simulation environment, Brooks and Martonosi [13] reported that roughly 50% of the integer instructions had both operands within 16 bits. This ratio increased to a fraction between 80% to 90% for operands within 33 bits. Loh [14] obtained significantly lower fractions, around 30% and 50% for 16-bit and 33-bit, respectively, with the same simulator. They justified the new width distribution with the usage of an updated benchmark (SPEC2000) and because the operation result is also used to calculate the width. Ergin et al. [15] focused on the values stored in the register file, by using a superscalar cycle-accurate simulator based on SimpleScalar and the SPEC 2000 benchmark suite. They found that around 40% of the stored values have 16 bits or less, and 85% are covered by 32 bits. Özsoy et al. [41] obtained similar narrow-width statistics for the SPEC2000 but using the PTLsim simulator, which handles 64-bit x86 instructions. As shown in Figure 2.6, they reported that the 34-bit range covers more than 90% of the values in the profiled applications.

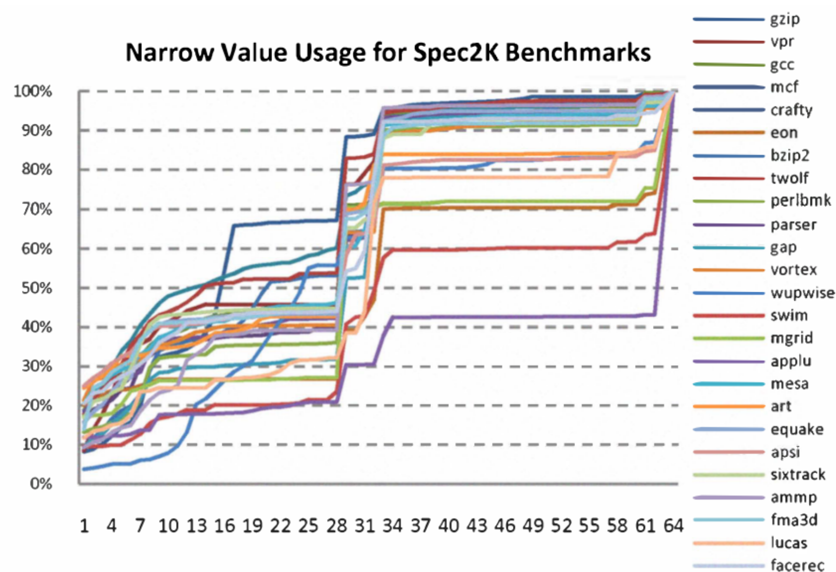


Figure 2.6: Cumulative distribution of narrow-width values occurrence, per benchmark of SPEC2000 [41].

Several authors have also expressed the notion of grouping values in different narrow-width classes, proposing static bit partitions based on their benchmark statistics. Most of the suggested partitions are among three classes (see Fig. 2.7), with 16 bits and 33/34 bits as boundaries [13, 14, 41]. The boundary around 33 bits is attributed to the memory addressing [14], which explains why it is more prone to change with specific benchmarks or even compilers. Grouping into a smaller number of classes introduces a coarser granularity but simplifies the control logic [41].

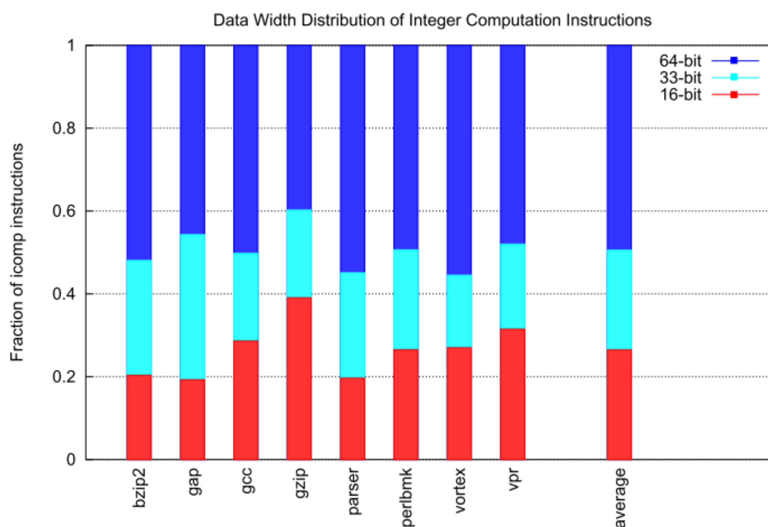


Figure 2.7: Distribution of integer operations grouped by classes (16-bit, 33-bit, and 64-bit), for several SPEC2000 applications [14].

In some of these analyses, it was also identified that not only narrow-width operations occur frequently, but they also show strong temporal locality. An instruction that recently operated on narrow-width values has more probability to perform narrow computations again, as it will likely manipulate the same type of input data. Loh [14] first found out that 90% of the instructions have the same width as the last three occurrences of the same instruction type, and 86% for the past seven instances. This temporal locality makes it possible to develop width prediction mechanisms that enable the detection of narrow-width instruction early in the pipeline [14, 15, 41], with low misprediction rates, even before the operands are known, or the result is computed.

Hence, narrow-width values are a highly relevant opportunity for microarchitectural optimizations, as the logic involved in performing calculations and storing these values is simplified. A narrow-width operation can be performed using a narrower Arithmetic Logic Unit (ALU), and its operands and result can be transferred in the datapath using a reduced bus-width and can be stored more compactly in the register file or even in memory.

2.3.1 Software based techniques

A possible approach to exploit narrow-width operands at the microarchitecture level is to explicitly provide the programmer with instructions to more efficiently manipulate smaller resolution operands. SIMD instructions are themselves an example of this, as they can operate on individual elements of up to 8 bits, in most ISAs [9,42,43]. In this case, the instruction itself encodes the operand's width, so the use of narrow-width operands has to be decided on compile-time. The performance efficiency of this approach relies on having architectural structures conveniently adapted to compute and store several values in parallel, but these values must all have the same width.

A different software-based technique was proposed by Canal et al. [44], where the binary instruction explicitly encodes the width required for an integer operation, and its execution is optimized by gating the unneeded datapath portion.

These static solutions have the advantage that the architectural optimizations can be performed with low control overhead, and can be identified very early in the pipeline, as soon as the instructions are decoded. However, these software-based approaches rely on compile-time analysis to estimate the width bounds for each operation, in order to identify which instructions require less resolution. The compiler must perform a conservative estimation, as no overflow should occur due to insufficient resolution, independently of the data input, which results in losing a significant portion of the narrow-width opportunities. In particular, for the SIMD instructions, even if only a small fraction of the elements requires a larger width, all operations have to be performed using this higher resolution.

A possible solution to reduce the number of missed opportunities is to allow the resolution encoded in these instructions to be speculative. This approach relaxes the constraints on the static analysis, which only needs to set the most probable width of the operand and result. Pokam et al. [16] proposed that the instruction encoding should be only a hint to the architecture and that there should be hardware mechanisms to recover from width overflow. If the hinted width is not enough to perform the computation, a replay trap should be set to restart the instruction with a wider datapath resolution.

However, these replay mechanisms come at a high performance and energy cost, as not only is the executed instruction wasted, but this may also interfere with the execution of the other instructions in the pipeline. Moreover, the compiler is still forced to be conservative to some degree in the estimation of the width bounds to keep the under-prediction rate low. Futhermore, the same program region can have different width patterns at different moments of execution, which makes the static analysis even more complicated and may require more complex program binaries.

Moreover, these approaches require making complex changes to the compilers and recompiling the applications to take advantage of the new optimized instructions. This increases the cost of deploying these software-based solutions.

2.3.2 Dynamic hardware-based approaches

As it was referred before, a considerable portion of narrow-width computations can only be detected at run-time, when the actual values are available. The width that is required by each value can be evaluated using leading zero or one-bit detectors. Hence, dynamic mechanisms implemented at the hardware level can better exploit narrow-width computations, for increased performance and power-efficiency, by conveniently scaling the datapath structures during run-time or by better utilizing the already available resources.

Most of the existing work in dynamic narrow-width optimization has been developed in out-of-order superscalar processors [13, 14, 40, 41]. Apart from its higher relevance in the GPP segment, this can be explained by the wider instruction window and higher amount of available datapath resources, which present more opportunities for width related optimizations.

Brooks and Martosini [13] suggested some optimizations to the scalar datapath at the level of the issue queue and the functional unit schedulers. By using leading zero and one detectors, the integer instructions waiting for execution are tagged according to the width of their operands, as soon as they are known (See Figure 2.8). They proposed two different optimization mechanisms when narrow instructions are issued to the corresponding functional unit:

- For power efficiency, they suggested clock-gating the unneeded resolution of the functional unit, when the instruction is issued;
- For performance, they proposed fusing multiple narrow operations of the same type for executing simultaneously, in the same ALU, in a form similar to auto-vectorization.

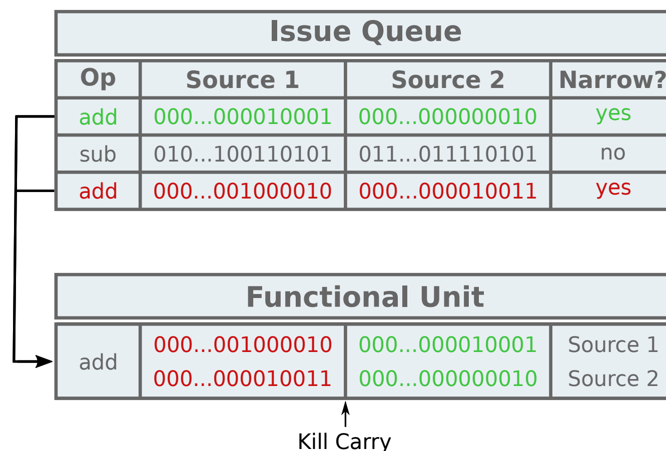


Figure 2.8: Example of an opportunity to fuse two narrow-width addition operations waiting for execution in the issue queue (adapted from [13])

For the fusing approach, they only consider simpler and more frequent arithmetic operations, such as additions, where the functional unit can be shared by cutting the carry chain and multiplexing the

operands to different bit ranges. The performance increase is achieved with the higher execution throughput using the same functional unit resources, as long as enough opportunities for fusing are encountered during the program execution.

They argue that the overhead in terms of circuit area and power consumption for their approaches is not significant, and it is mainly due to the leading bit detectors and width select multiplexers. For the power efficiency method, they estimated a reduction in power consumption of the integer unit of 54.1% for the SPECint95 benchmark suite. For the performance approach, they estimated an average speedup between 4.3% and 6.2%, with the same benchmarks.

Loh [14] presented a similar approach to increase the superscalar processor performance by fusing several one cycle integer narrow-width operations for execution in the same ALU. However, different operation types can be fused, in his approach, extending this technique from a SIMD to a Multiple Instruction Multiple Data (MIMD) approach. He argued that existing ALU designs that already support SIMD and multimedia extensions only have to be extended to have support for different opcodes for each sub-operation.

Rochecouste et al. [40] argued that the added complexity to handle instructions of different widths, in particular at the scheduler and in register file accesses, justify partitioning the superscalar microarchitecture into two clusters which process instructions of different width. These full and narrow-width cores have independent register files and datapaths, with the corresponding resolution, as illustrated in Figure 2.9. The narrow core only executes instructions where both the operands and the result are narrow, which they define as 16-bit or less. The narrow register file must be replicated in the full-width core, for the execution of instructions which have both types of operands or result. They argued that there are enough narrow and full instructions interleaved in the execution to keep both cores working at a good rate.

Islam and Stenstrom [45] focused on optimizing narrow-width memory instructions, proposing an additional cache for these narrow values. This narrow-width cache would be placed alongside the L1 data cache and connected directly to the Central Processing Unit (CPU) in parallel. This new cache would also be connected to the higher-level cache hierarchy through a narrow-width detection block, which would detect narrow operands closer to memory and fetch them directly.

2.3.3 Scalable width structures

These narrow-width optimization mechanisms require datapath structures that can adapt their bit-width during run-time, namely at the level of the Functional Units (FUs), register files, or caches. Several implementations of such structures have been proposed in the literature that either turn off part of their resolution when handling narrow-width values or that use their full width to pack several narrow-width values.

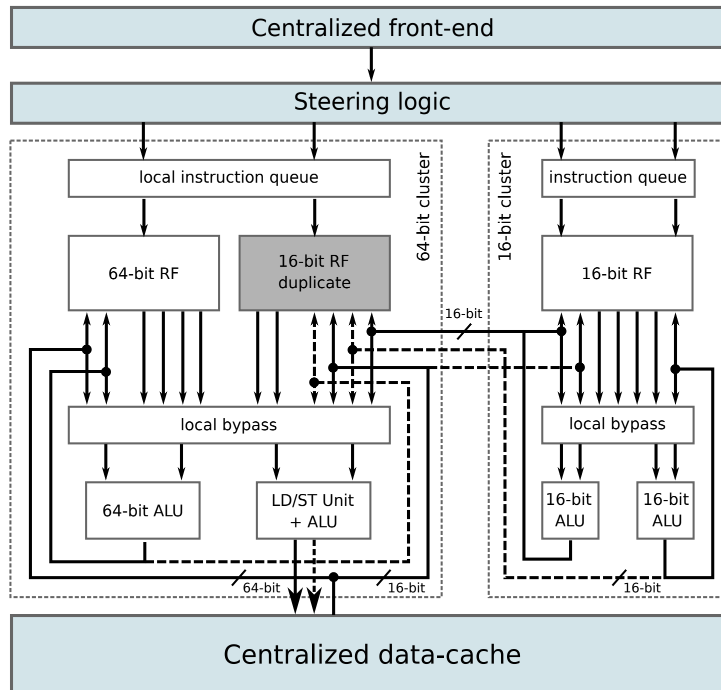


Figure 2.9: Cluster partitioned approach for narrow-width exploitation [40]

2.3.3.A Functional units

Several implementations have been proposed for integer FUs capable of supporting operands with multiple widths, mainly developed to provide support for SIMD ISA extensions. These implementations focus on multiplication and addition operations, although it is reasonable to assume that logical and shift operations are easier to implement.

Själander and Larsson-Edefors [46] proposed an integer multiplier design that adapts its bit-width to the operands. Their implementation also allows the multiplier logic to be efficiently shared for computing two narrow-width operations in parallel. They argued that the same could be done for other functional unit types, namely adders.

In the context of SIMD functional units, Balakrishnan and Nandy [47] and Karthikeyan and Ranganathan [48] went further and argued that the default element sizes (namely 8-bit, 16-bit, 32-bit, and 64-bit) miss some important media applications, such as medical imaging with 12-bit pixels and 9-bit encoded MPEG files. In accordance, they proposed changes in the SIMD FU for supporting arbitrary element sizes in addition and multiply-add operations, by modifying the carry-lookahead and Wallace tree algorithms, respectively, with a small hardware overhead.

For the efficient implementation of integer multiply and multiply-accumulation SIMD units, Danysh and Tan [49], and Krithivasan and Schulte [50] have proposed designs that allow the same multiplier hardware to be shared for different operation modes. The Multiply-Accumulator (MAC) design proposed

in [49] allows the execution of either one operation with 64-bit operands, two with 32 bits, four with 16 bits, or eight with 8 bits, both signed and unsigned. In both cases, the proposed designs are based on a similar scalar multiplier architecture, which is composed of a partial product generator, a Wallace reduction tree and a final carry-propagate adder (see Figure 2.10). The units are vectorized by selecting only the partial products corresponding to the elements in each mode and by killing the cross-products between boundaries of different elements (zeros in Figure 2.11). By aligning the partial products for each element so that there is no overlap with other elements (see Figure 2.11) the same reduction tree can be used for each mode.

The main difference between these two architectures is the implementation of the partial product generator, which is a Booth encoder in [49] and a matrix multiplier based on Baugh-Wooley technique in [50]. Using a Booth encoder reduces the number of partial products to half and allows for a smaller reduction tree. However, it requires more complex control logic for supporting multiple element sizes, as it requires suppressing carries between element boundaries in the reduction tree and final adder.

More recently, in the context of floating-point MAC architectures that support multiple precisions, several works [51–53] have proposed similar designs for implementing a mantissa multiplier (which is identic to an integer multiplication) that is shared between precision modes.

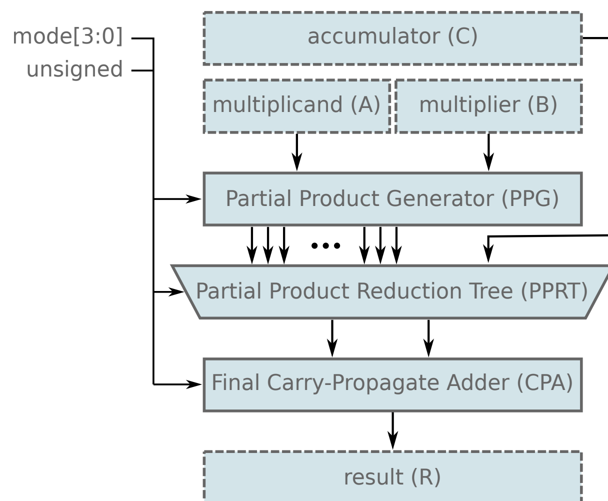


Figure 2.10: Vectorized MAC design proposed in [49].

2.3.3.B Register file

Large register files also play a fundamental role in sustaining the performance of a superscalar processor and represent a significant portion of dynamic and static power dissipation. Several authors have focused on taking advantage of the lower number of bits required by narrow-width values to use register logic more efficiently, either for increased performance, by compacting narrow values and increasing the

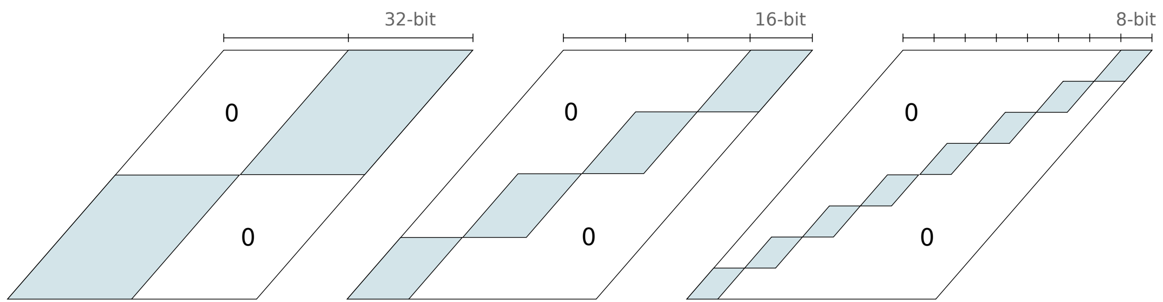


Figure 2.11: Selecting the partial products for 32-bit, 16-bit, and 8-bit operation modes, using the same hardware of a 64-bit scalar MAC unit.

number of free registers available, or for power-efficiency, by turning off unused portions of the register file.

However, register file optimization is quite challenging for a superscalar processor, mainly because of register renaming, as the registers are allocated before the value to be stored is known. To handle this, Ergin et al. [15] proposed conservative and speculative approaches. In the conservative approach, a full-size register is allocated during the renaming phase, and as soon as a narrow result is obtained, the empty space can be reallocated. In the speculative approach, the width of the instruction is predicted before renaming, and a register with an adequate size is allocated. If there is an over-prediction, the extra resolution can be either reallocated or wasted. Under-prediction is more difficult to handle, as it may lead to deadlocks if a suitable register is not found. The recovery mechanism depends on the details of the implementation but may require squashing more recent instructions to free-up registers.

Based on this same idea, two different techniques to optimize narrow-width value storage can be identified in the literature: packing and partitioning. In the packing technique, several narrow values share the same full-size register, and an extra pointer or mask is used to address the sub-registers. Ergin et al. [15] proposed a register packing approach, where four classes of values (16, 32, 48, and 64-bit) are stored in 8 byte physical registers. These values are stored in any of the 8 byte slots, not necessarily contiguously, by using a 4-bits mask to encode where each byte is stored. Each physical register is then addressed by a pointer and this mask. To manage the allocation and reallocation of these sub-registers, they also proposed independent lists of free registers for each class. Similarly to operation fusing, this register packing technique is more focused on performance, as it allows the existing resources to be better used. Although it does not reduce, by itself, the register file power consumption, it can be combined with other approaches, such as turning off unused parts of the register file.

In the register partitioning approach, the physical register file is partitioned in regions of different widths, where the unneeded resolution is either removed or gated, providing different register size classes. Özsoy et al. [41] proposed both a static scheme for this partition, where the number of registers for each class is fixed, and a dynamic scheme, which can change this partition in run-time, by using

gating mechanisms. In their implementation, they also resort to a width predictor to choose the register class in the renaming stage. In the case of a misprediction, if there are no free registers with enough size, the recovery mechanism is to flush the most recent instructions. In the dynamic partition, the registers' class configurations are updated periodically (e.g. every 1024 clock cycles), by using decision heuristics based on utilization statistics measured in that interval. They report power dissipation reductions in the register file in the order of 50% and 60%, for their static and dynamic partition approaches, respectively. They also report a minor performance degradation of 5% (on average) for the static approach, as the reduction in the number of full-size registers may lead to the program stalling more easily. For the dynamic approach, the reported degradation is negligible, and for some benchmarks there is even an increase in Instructions Per Cycle (IPC).

2.3.3.C Caches

The optimization in the storage of narrow-width values is not only relevant for the register file but also for the caches, in particular for the first level data cache connected to the datapath. Cache space is crucial for performance, as it allows accessing recently used values without the high latency of accesses to main memory (or other cache levels). Using the available space more efficiently, by packing narrow values, would allow for more new data values to be kept without the need for replacing older ones. Pujara and Aggarwal [54] proposed implementing these optimizations in the L1 data cache, obtaining an increase in its capacity of 50% and a reduction in the miss rate of 23%, on average.

2.4 Summary

This chapter presented a highly relevant research topic in computer architecture focused on increasing the energy efficiency of the processor's execution unit, in particular of the increasingly important SIMD unit. The main structures of modern superscalar processors were presented and the state-of-the-art techniques for reducing power dissipation were reviewed, in order to provide the necessary background for proposing new power-efficiency mechanisms. Then, the opportunity of exploiting narrow-width integer operations for reducing power and increasing performance was presented. Previous authors have shown that most integer values use only a fraction of the available ALU width, but they only proposed solutions for exploiting it in the scalar execution pipeline, disregarding the exploitation of these techniques in vector operations. The following chapter will evaluate the previously unexplored opportunity of optimizing the execution of narrow-width vector instructions, with the goal of reducing power consumption in the SIMD units.

3

Narrow-width Opportunity in SIMD

Contents

3.1 Defining narrow-width in SIMD computations 25
3.2 Optimizing vector computations in out-of-order processors 28
3.3 Profiling integer intensive applications 29
3.4 Envisaged energy savings 34
3.5 Summary 36

This chapter presents a preliminary evaluation of the opportunity to exploit narrow-width values in Single Instruction Multiple Data (SIMD) computations. Firstly, narrow-width vector computations are defined, and the out-of-order processor microarchitecture already presented in Section 2.1 is analyzed to identify how these operations can be exploited. Then, a variety of integer intensive applications are profiled to evaluate the prevalence of narrow vector computations. Finally, an initial estimation of the expected energy savings is performed to motivate the relevance of proposing new architectural mechanisms.

3.1 Defining narrow-width in SIMD computations

The narrow-width opportunity in SIMD computations can only be evaluated by first defining how the width of a vector operation is measured. The width of an integer value has been defined as the number of required bits to uniquely encode it, while excluding the redundant sign bits (see Figure 2.5). Due to the frequent usage of signed values (in two's complement) in most application domains, it is advantageous

to keep the least significant sign bit and to compute the operand width as its index in the bit-word. For positive (or unsigned) values, this representation wastes an extra bit, the leading zero bit. The alternative would be to discard all the leading zero bits and count the position of the most significant one bit (as some previous authors have done). However, this would result in the full architecture width being used for every negative value (as all the leading bits are one).

It is important to note that this optimized narrow representation is independent of whether this value will be considered as a signed or unsigned operand in an instruction, and is more related to how this value will be stored and transferred. When an operation (e.g. arithmetic, logic, or multiplication) is executed over narrow values (using the proposed representation), it may use a narrower Arithmetic Logic Unit (ALU). Naturally, this compressed representation does not result in any precision loss, as only redundant sign bits are discarded, and the original bit-arrays can be restored through sign bit extension.

A typical vector operation is composed of several sub-operations executed in parallel over the elements of the source vectors, which are usually called vector lanes. The (optimal) width of a vector operation is henceforth defined as the sum of the width required by each of these lanes (see Fig. 3.1). When a vector lane corresponds to an operation between two or more data elements, its width is considered as the maximum width between those elements. When there is only one operand, its width is used directly.

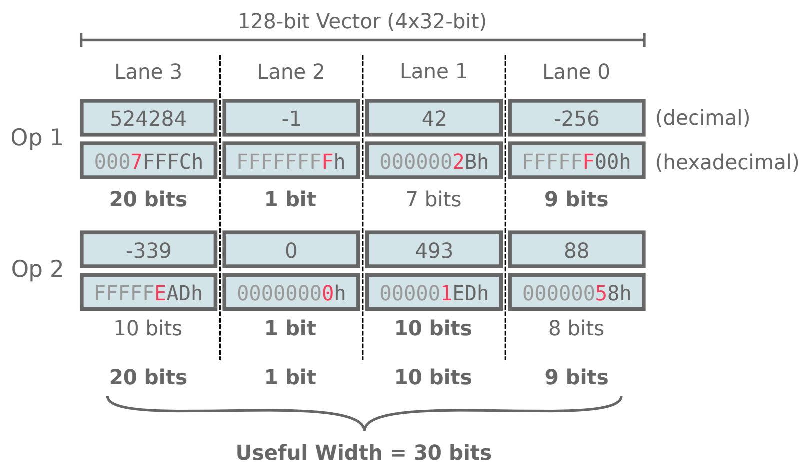


Figure 3.1: Width of an example 4x32-bit vector operation, in a 128-bit architecture. The width for each lane is the maximum between its sub-operands, and the operation width is the sum of all lanes. The hexadecimal digit with the leading sign bit is marked in red.

However, vector extensions usually support several different element modes (e.g. 64-bit, 32-bit, 16-bit, 8-bit), by varying the number and the size of the lanes in which vectors are divided. For the same contents in the vector operands, different modes change how the values are interpreted and computed, so the width of these values and of the whole vector computation is also changed. Hence, when computing the width of a vector instruction, it is necessary to take into account the element size.

Moreover, typical vector extensions provide more modes for combining operand elements than the regular SIMD model (see Fig. 2.2). Figure 3.2a depicts some of these different modes for the instructions in the ARM NEON extension. For example, the *pairwise* mode considers lanes as pairs of elements in the same vector, and the *across* mode makes a reduction over all the elements in a vector (e.g. ADDV sums all elements, and MAXV returns the maximum value in the vector). Some other instructions manipulate a single vector element, such as the UMOV and SMOV instructions, that move an element to a general-purpose (i.e. scalar) register. These different modes change how the width of an instruction is calculated, as they group vector elements into lanes differently. Most vector extensions also have specific instructions to deal with operations between vectors with different element sizes, or that store the result in wider or narrower elements (see Figure 3.2b). However, in this study, only the operand values are considered when calculating the width required by each lane.

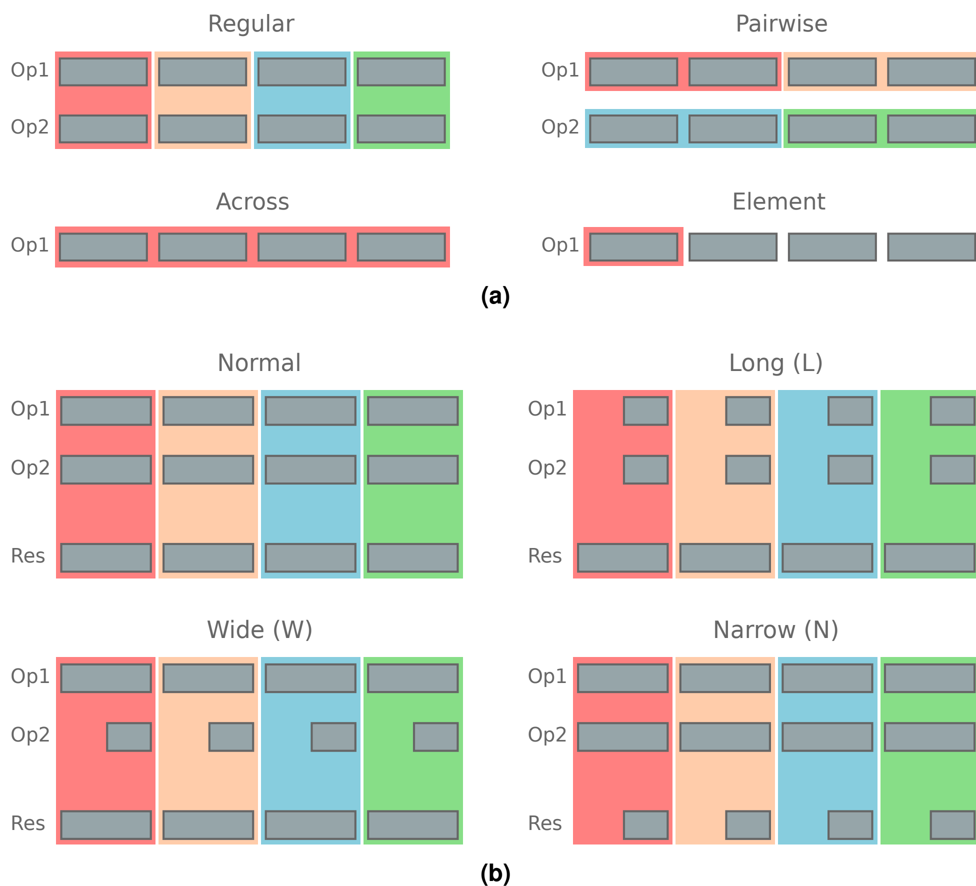


Figure 3.2: Overview of vector operation modes in Arm NEON, in terms of how the elements are combined (a), and the relative size between operand and result elements (b). In this example, each vector is considered to be composed of 4 data elements. Elements in the same lane are highlighted in the same colour.

3.2 Optimizing vector computations in out-of-order processors

When the elements in a vector operation do not require the whole bit-width assigned at compile-time, these elements can be narrowed by discarding redundant sign bits before the operation is computed. If the narrowed elements are packed into smaller width vectors, only part of the functional unit width is required to perform the computation, and the remaining portion of the unit can be either turned off or used for other computations. In either case, it is advantageous to perform these optimizations dynamically at run-time, when the actual values of the vector operands are known, and the width required can be detected.

Hence, narrow-width data can be exploited during the execution of vector operations. It should be recalled that a grater emphasis will be given to superscalar out-of-order processors, not only because this architecture topology has become dominant in General Purpose Processors (GPPs) but also because these cores present more opportunities for optimizing narrow-width operations in run-time. In fact, as explained in more detail in Section 2.1 (see Figure 2.1), these processors try to execute several independent operations in parallel, so there are several instructions (in particular vector operations) waiting for execution at a given time. As the instruction window increases (i.e. with a wider frontend, which allow more instructions to be fetched each cycle), more narrow-width opportunities can be uncovered simultaneously.

Moreover, modern superscalar cores have several functional units of each type, to allow multiple instructions to start executing at each clock cycle. In particular, recent designs have as many as two to four SIMD pipelines, as presented in Table 1.1. These additional units come at a high cost in power consumption, so it is essential to use them more efficiently, especially if some of the units can be turned off while maintaining an identic execution throughput. The existence of speculative execution in these cores is not an issue for the approaches that will be proposed. Even if a mispeculated narrow-width instruction is executed before being squashed, this still represents a reduction in the energy that would be wasted executing it.

In a typical out-of-order execution engine (see Figure 3.3), several instructions are dispatched at each clock cycle to one of the existing issue queues, accordingly to their operation type (B). In the issue queues, their dependencies are monitored, and ready operands are fetched and buffered (A). When all operands have been fetched, the instruction is scheduled (C) for execution in the corresponding functional unit (D), as soon as it is available. Several instructions of the same unit type are usually waiting for execution at the same time, and the scheduler is responsible for allocating the limited execution resources. When each instruction finishes execution, its result is written to the register file (E), and possibly fetched by other consuming instructions, directly from the Common Data Bus (CDB). Then, the instruction waits to be committed in the Re-Order Buffer (ROB).

Hence, while the instructions are waiting for execution in an issue queue, additional hardware could

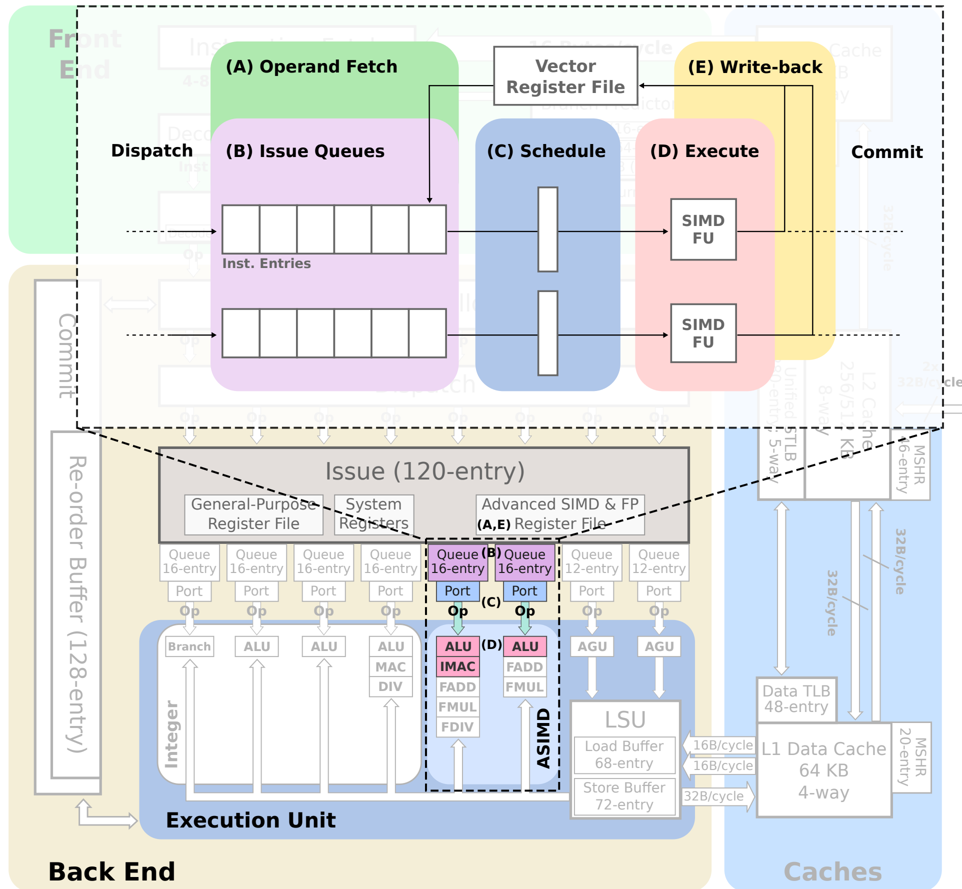


Figure 3.3: Details of the SIMD execution engine in the out-of-order microarchitecture. This work will focus on exploiting narrow-width in the SIMD execution pipelines, by modifying the integer vector units and their corresponding issue queues.

be added to identify which instructions have narrow operands and mark that information in the issue queue. Then, when the issue scheduler port chooses which operations are assigned to the available functional units, it can take that information into account to schedule the execution more efficiently, by assigning only a portion of the functional unit to packed narrow vector computations or by combining several of these operations for simultaneous execution.

3.3 Profiling integer intensive applications

Having introduced the notion of packing (compressing) narrow-width elements in vector computations for efficient execution, this section will evaluate the occurrence of these computations in several benchmarks. However, it should be noted that although several of the considered applications have a significant fraction of floating-point operations (or even floating-point SIMD operations), optimizing these computations is outside the scope of this study. Reducing the number of digits in a floating-point repre-

sentation is not as straightforward as with integer (or fixed-point) values, due to how the mantissa values are typically normalized (the most significant bit is an implicitly ”one”) and because it can result in a precision loss.

3.3.1 Benchmarked applications

The opportunity to exploit narrow-width in vector operations was evaluated by profiling several benchmark applications, with a particular focus on integer intensive applications. This profiling was performed using a modified version of the gem5 simulator [55], by considering a high-performance 8-wide Out-of-Order core model and using the ARMv8 Instruction Set Architecture (ISA) with the NEON (Advanced SIMD) vector extension, as further detailed in Chapter 5. Several applications of the SPLASH-2 [56], PARSEC 3.0 [57], and SPEC2006[1] benchmark suites were profiled (see Table 3.1), which cover a wide variety of computing domains (e.g. scientific computing, media processing, signal processing, and data mining). However, these benchmark suites are not well prepared for SIMD architectural exploration [58], as the degree of vectorization that can be achieved is low.

Hence, since this thesis work aims to propose mechanisms for optimizing the execution of intense loads of vector operations, an adequate evaluation can only be performed by finding applications that make significant use of the vector capabilities offered by modern processors. In fact, although modern compilers can perform auto-vectorization, the degree of vectorization that can be extracted in many applications is minimal, in great part because they are not implemented in a SIMD-friendly way. However, libraries of kernels optimized to take advantage of vector extensions were made available, as it is the case for the Arm NEON extension. In this thesis work, the Eigen[2], Arm Compute[3], and Ne10[4] libraries were used, which provide an implementation of a variety of algebra, signal, and image processing kernels. Several computationally intensive sample applications were prepared with these libraries' kernels, using sample images and randomly generated data as input.

These kernels are complemented by some real-world mini-applications, which are also optimized for Arm NEON:

- `IntegerNeuralNetwork`: An implementation[5] of a feed-forward neural network with one hidden layer and using integer weights and inputs, modified to use the Eigen library to vectorize linear algebra operations;
- `StreamVByte`: A fast byte-oriented integer compression library optimized using SIMD instructions[6],

[1] SPEC2006: `https://www.spec.org/cpu2006/`
[2] Eigen Library Version 3: `http://eigen.tuxfamily.org`
[3] Arm Compute Library: `https://developer.arm.com/ip-products/processors/machine-learning/compute-library`
[4] Ne10 Library: `https://projectne10.github.io/Ne10/`
[5] Integer Neural Net repository: `https://github.com/spolsley/integerNeuralNetwork`
[6] StreamVByte Repository: `https://github.com/lemire/streamvbyte`

which is used in databases (UpscaleDB) and in information retrieval systems (RediSearch and Trinity) [59].

- `Cartoon`: Cartoonify image application, by combining the Gaussian convolution and canny edge detector kernels. The implementation is based on the Arm Compute library.

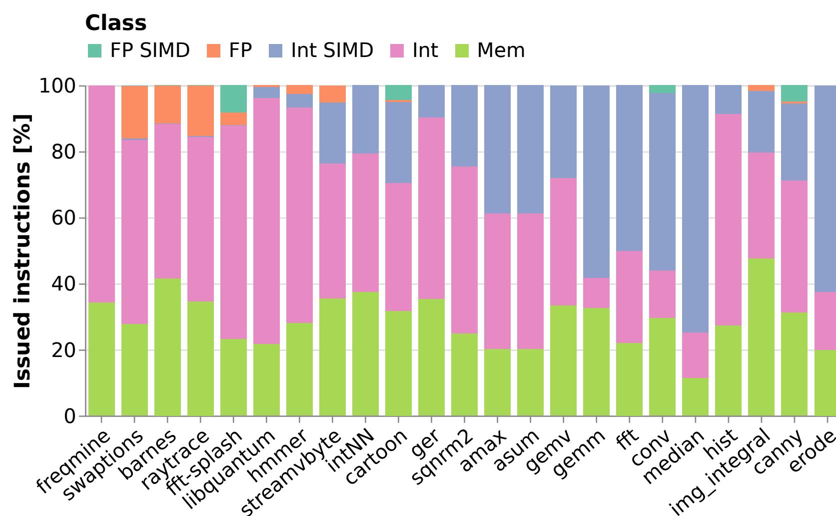
Table 3.1: Complete list of profiled benchmarks

Suite	Benchmark	Domain	Description
SPLASH-2	barnes raytrace fft-splash	Scientific Computing Graphics Signal Processing	N-body simulation 3D rendering Complex 1-D FFT
PARSEC3	freqmine swaptions	Big-Data Big-Data	Data mining Financial analysis
SPEC2006	libquantum hmmer	Scientific Computing Scientific Computing	Quantum computer simulation Gene sequencing
Mini-Apps	streamvbyte integerNN cartoon	Big-Data Machine Learning Image Processing	Integer compression Integer feed-forward neural network Cartoonify image
Kernels	sqnrm2 ger amax asum gemv gemm	Algebra	Squared vector l2 norm Rank-1 general matrix update Vector absolute max Vector absolute sum General matrix-vector multiplication General matrix multiplication
	fft conv median	Signal Processing	Fast fourier transform 2-D Convolution 2-D Median filter
	img_integral img_hist erode canny	Image Processing	Integral image Image grayscale histogram Image erosion Canny edge detector

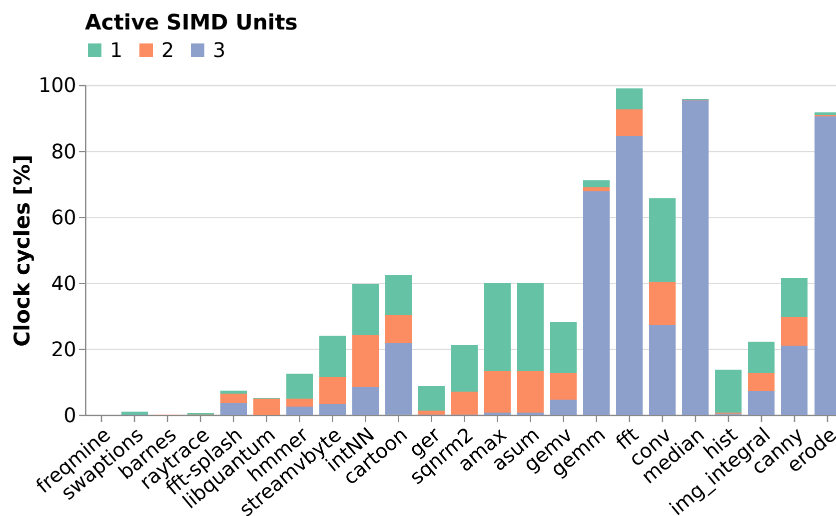
3.3.2 SIMD unit usage analysis

Figure 3.4a presents the fraction of issued instructions by operation type for each benchmark, where integer operations (including arithmetic, multiplication and SIMD instructions) represents between 50% (for the `img_integral` benchmark) and 90% (`median`) of the issued instructions. The focus of this work is on integer SIMD instructions, which ranges from almost no occurrence (`freqmine` and `swaptions`) to being the most frequent instruction type (e.g. in the `median` and `erode` kernels, where it represents 75.6% and 63.5% of the issued instructions). For a more detailed analysis, Figure 3.4b shows the number of active SIMD units used by percentage of clock cycles, for each application. From the obtained results, it can be observed that the activity in the SIMD unit is relevant for most of the mini-apps and kernels, which use up to three units in a significant fraction of the execution cycles. Each time these additional units are

required, it represents a significant energy consumption penalty, as the sleep state must be interrupted. Hence, it would be very advantageous if the same execution throughput could be maintained using fewer units.



(a) Fraction of instructions issued by operation class



(b) Number of active SIMD units per percentage of clock cycles

Figure 3.4: Evaluation of the usage of the SIMD unit for the selected benchmarks

This thesis will focus only on applications with high SIMD unit activity, where its power consumption is more relevant and there is an opportunity to increase the efficiency. Figure 3.5 presents the fraction of issued vector computations grouped in ranges defined by the width they require, for a subset of the considered benchmarks with a significant degree of vectorization. As explained in Section 3.1, the width of a vector instruction is evaluated as the sum of the bit-width required by each operand lane. For these applications, between 29% (for the `img_integral` kernel) and near 100% (for the `integerNN`, `ger`,

sqnrm2, gemm, and hist benchmarks) of the issued SIMD instructions can be executed using only half or less of the total vector width, if the vector computations are packed to discard unneeded sign bits. Hence, there is an opportunity for optimizing narrow-width vector computations, enabling a significant reduction in power consumption.

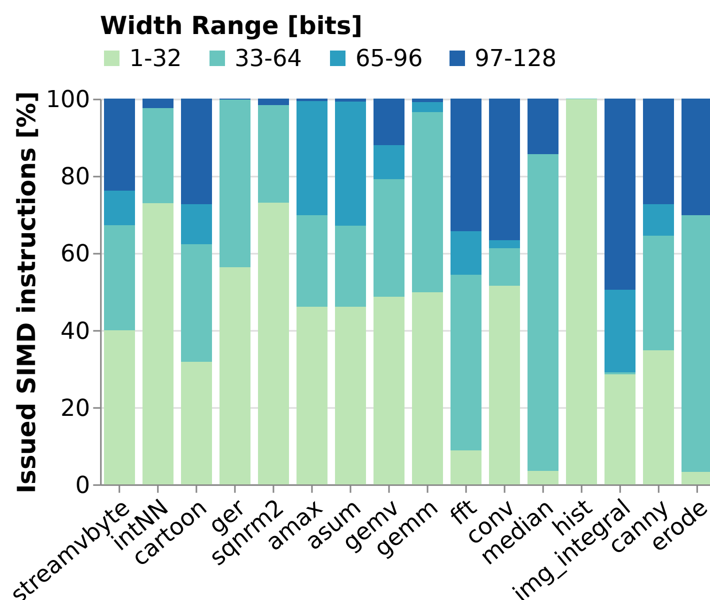


Figure 3.5: Analysis of the width required by integer vector operations for a subset of applications with significant SIMD unit activity.

A more detailed cycle-by-cycle tracing of the SIMD units' activity, together with the measurement of the width required by each vector operation, was performed for the considered mini-applications. The observed sample execution intervals are presented in Figure 3.6. Although several of the three available SIMD units are in use in most cycles, the combined (i.e. summed) width required by all issued vector operations is only a smaller fraction of the total width provided (i.e. $3 \times 128 = 384\,bits$), for most cycles. In the cartoon sample (Fig. 3.6a), in the cycles where all the three units are in use (18.5% of the sample), the average used width is 154 bits, which is only 40% of the full width. For the whole streamvbyte sample (Fig. 3.6b), the maximum required width is only 216 bits, which means that the third unit could be kept turned off for the whole interval. For the integerNN sample (Fig. 3.6c), the third unit is only required in 0.55% of the cycles, while without optimized execution it is used in 9.2%. Even when a single unit is active, the average width usage is 56%, 48%, and 36%, for the cartoon, streamvbyte, and integerNN samples, respectively. Hence, the active integer SIMD units are significantly underused. If these operations were compressed to discard the unnecessary bit-width and packed together for execution on the same unit, the third or even the second unit would no longer be necessary for most of the cycles. These units could be switched off without a significant performance impact.

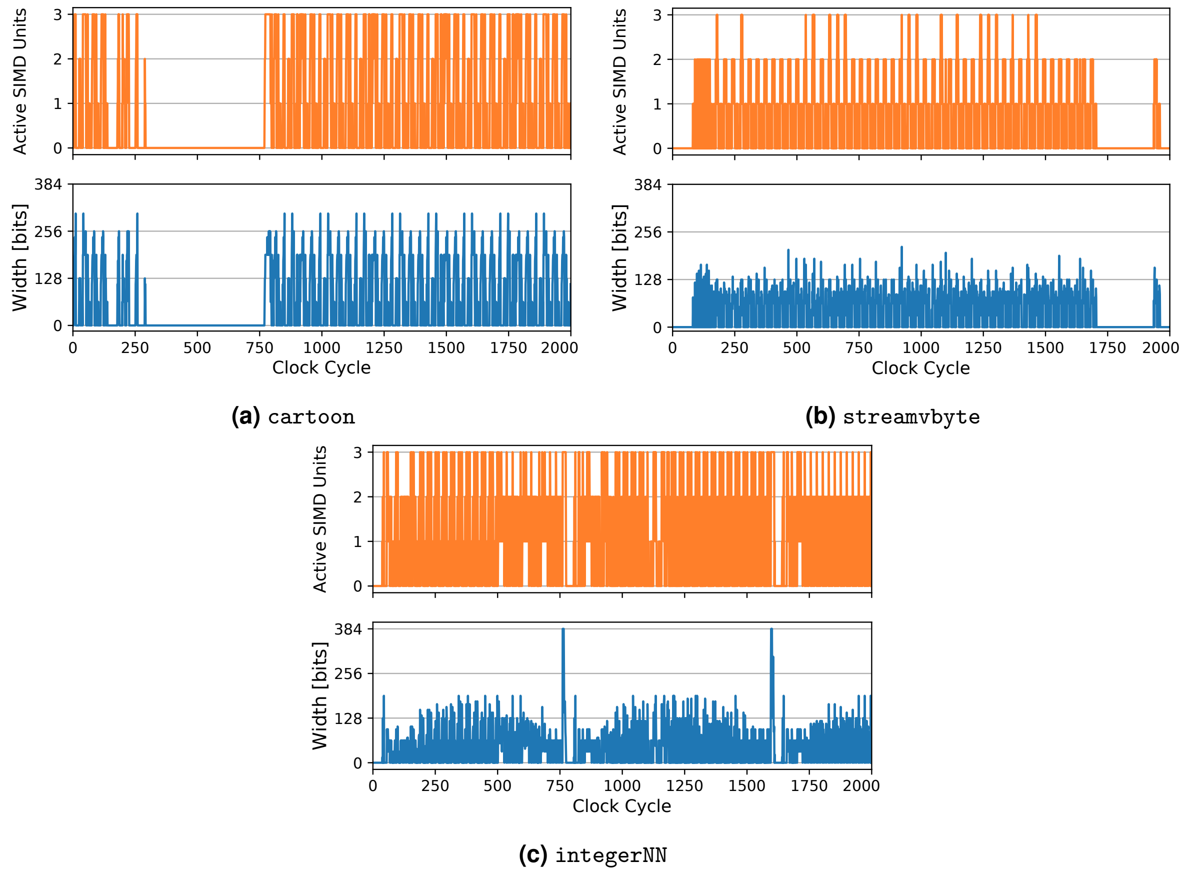


Figure 3.6: Number of active SIMD units in each clock cycle when compared with the combined width of the corresponding (packed) vector operations, for sample intervals of the mini-apps.

3.4 Envisaged energy savings

As it was referred before, this thesis aims to exploit narrow-width operations for reducing the energy consumption of vector SIMD units. As it was presented in Section 2.2, power consumption can be divided into two main components, dynamic power and leakage power. As given by Equation 2.1, the dynamic power consumption of a component is approximately proportional to its average activity. For the case a functional unit, the activity is highly correlated with the average number of accesses to that time unit. Each clock cycle a functional unit (or a portion of one) is idle represents a saving in dynamic energy, as the corresponding logic blocks are not switching. The dynamic power reduction is even higher if the clock signal of these idle blocks is turned off, with clock gating, as it represents a large portion of the dynamic dissipation. In current designs, clock gating can often be triggered with finer granularity, both during short periods (e.g. a single clock cycle) and in specific circuit blocks (e.g. a single functional unit, or even a portion of one).

Hence, narrow-width vector operations represent an opportunity for reducing the dynamic energy consumption, as the clock signal of the ALU blocks corresponding to unneeded width in the functional

unit can be turned off, reducing the activity fraction (α) in the dynamic power Equation 2.1. To obtain a preliminary estimate of the maximum dynamic energy savings that can be expected in the profiled applications, the fraction of each unit that is active in each operation was calculated, by dividing the width used by the full vector register size. The new activity fraction, $\hat{\alpha}$, of these units is given by:

$$\hat{\alpha} = \alpha \left(\frac{1}{N} \sum_{i=0}^{N} \frac{Width_{Used}}{Width_{Register}} \right) = \alpha \left(\frac{\sum_{i=0}^{N} Width_{Used}}{N\, Width_{Register}} \right) = \alpha\beta, \tag{3.1}$$

where α is the original unit's activity, β is the average fraction of width that is actually used, N is the number of operations executed, and $Width_{Used}$ and $Width_{Register}$ are measured in bits. Then, the relative energy savings, in fraction, are obtained from:

$$E_{savings} = \frac{P_{dynamic}T - \hat{P}_{dynamic}T}{P_{dynamic}T} = 1 - \frac{\hat{\alpha} C {V_{DD}}^2 f}{\alpha C {V_{DD}}^2 f} = 1 - \beta, \tag{3.2}$$

where T is the execution time (which is considered constant); $P_{dynamic}$ and $\hat{P}_{dynamic}$ are the original and new dynamic power consumptions, respectively. The estimated dynamic energy savings for each considered benchmarks are presented in Figure 3.7, representing an average reduction as high as 56.4%. Although these values represent a rather optimistic scenario, that would require suppressing all the dynamic power consumption in the unused portions of the ALU, they show that there are very interesting efficiency gains that can be obtained from this opportunity.

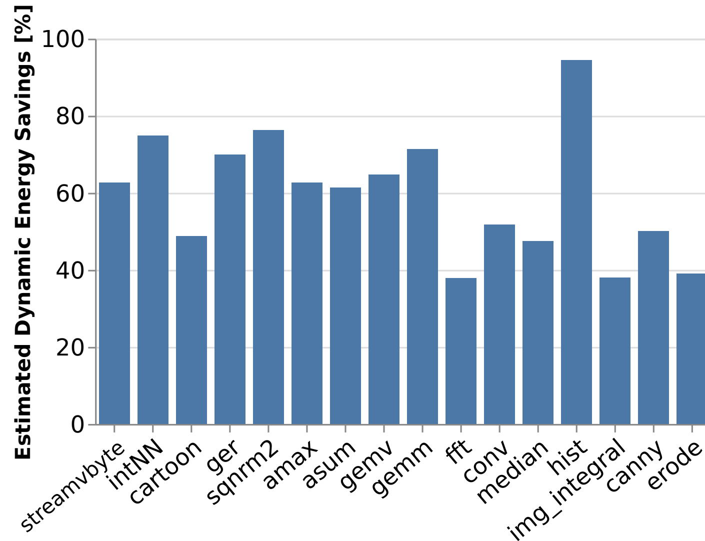


Figure 3.7: Maximum expected dynamic energy savings in the SIMD unit if each vector instruction is executed using the optimal width.

Leakage power is an increasingly significant fraction of the power dissipation in microprocessors,

due to the reduction in the transistors' size. Since it is always dissipated, even when a functional unit is idle, reducing the units' activity is not a solution. However, leakage power can be reduced by turning off the power supply of unneeded logic blocks (as explained in Section 2.2), with a mechanism called power gating.

Power gating a functional unit is only advantageous when that unit is kept turned off for extended periods (e.g. around the tens or hundreds of cycles), as there is a high energy and delay penalty when transitioning between sleep and active states (recall Fig. 2.4). On the other hand, when executing operations with narrow operands, the unused width in each unit could be used to execute additional instructions. This way, the same execution throughput could be sustained using fewer units, and the remaining could be power gated for longer periods of time (or even removed from the design) without a significant performance impact. The leakage power increases with the number of logic gates with power supply in the circuit. Hence, the leakage dissipation in the execution unit is correlated with the number of available functional units. Each vector unit that is removed from the design or power gated corresponds to a fraction of leakage energy that is saved every cycle.

3.5 Summary

This chapter showed that for a variety of applications with significant activity in the integer SIMD unit, most vector computations make use of only a fraction of the with of the available vector units. This opportunity will be exploited for optimizing the vector execution unit, by detecting when ready instructions in the issue queues have narrow operands and packing their vector elements to discard the unneeded sign bits. On average, 56.4% of the width in the SIMD unit is wasted (in the profiled applications), so this a very interesting opportunity for reducing energy consumption. The following chapter will address some architectural mechanism that allow an efficient exploitation of narrow-width operations, in order to achieve these envisaged energy gains.

4

Architectural Mechanisms to Exploit Narrow-width

Contents

4.1 Proposed Mechanisms 38

4.2 Integration in conventional processor architectures 48

4.3 Summary 50

After identifying the opportunity to optimize the execution of narrow-width integer vector operations, three complementary mechanisms for exploiting it are proposed in this chapter: width encoding, operand packing, and operation fusing. **Width encoding** consists in detecting the width required by each lane in a vector operation and efficiently encoding this information in a width mask. The **operand packing** mechanism compacts the vector operands efficiently, by discarding the unnecessary bit-width in each vector element, reducing the total width required to execute an operation. **Operation fusing** consists in simultaneously issuing several independent instructions to the same functional unit when each operation uses only a portion of the available width.

Then, this chapter details the necessary architectural changes to implement these mechanisms in the out-of-order microarchitecture presented in Chapter 2. These include changes to the existing functional unit structures (i.e. issue queues, Single Instruction Multiple Data (SIMD) functional units, and issue schedulers) and the addition of new structures, to calculate the width required by vector operations and execute them efficiently.

4.1 Proposed Mechanisms

4.1.1 Width encoding

The proposed optimization of the execution of narrow-width operations starts by first detecting the width required by each sub-operation, at run-time, i.e. when the actual values are available. Such a procedure is implemented by adding an extra step in the execution of vector instructions after their operands are fetched, which is henceforth called **width encoding** (see also Section 4.2). In this step, the width required by each lane in a vector operation is determined and encoded in a bit-array, that it is transferred along the execution pipeline with the corresponding instruction.

The encoded width values are restricted to multiples of a fixed **width-block** (e.g. 8 bits), rounding values up. For example, in an architecture with an 8-bit width-block, a value with a bit-width of 12 bits is detected as a 16 bits operand ($16 = 2 \times 8\,bits \geq 12\,bits$). This minimum width-block parameter (that will be also denoted as w) is set at design time and allows for a portion of wasted width to be traded for a much simpler implementation, as it allows the width of each lane to be encoded with fewer bits. Figure 4.1 depicts the wasted bit-width when comparing the original width values (which correspond to a 1-bit width-block) to designs with a 4-bit and 8-bit width-block. Naturally although using a larger width-block results in more bit-width being wasted in each computation, it significantly simplifies the implementation of this detection and encoding mechanism, as will be shown in the following paragraphs.

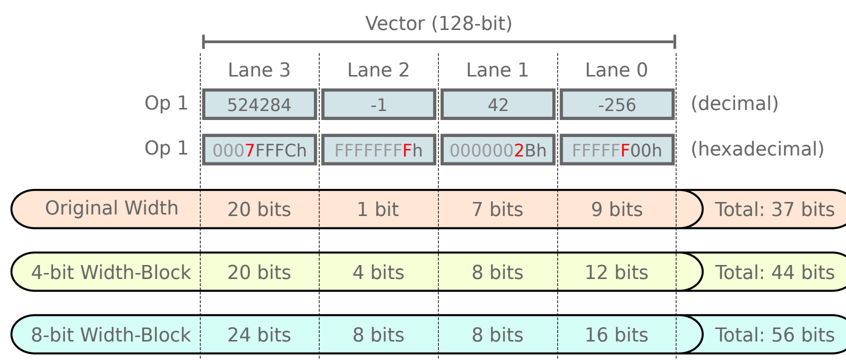


Figure 4.1: Examples of the widths detected with different width-block sizes, in an architecture with 128-bit vectors and an operand in the 4×32-bit vector mode. Each vector element is presented in decimal and hexadecimal notation, and in the hexadecimal representation, the digit with the sign bit is highlighted in red. The total width required in each case is the sum of the width values for each lane.

Figure 4.2 presents the proposed width encoding stage for an example instruction with two 4×32-bit vector operands, written both in decimal and hexadecimal formats (see parts (A) and (B)). At this step, the width required by each vector lane is detected and encoded in a **width mask** (D). This mask is composed of a group of bits per vector lane, where each bit-set encodes the number of bits actually used by the corresponding lane, i.e. after removing excess sign bits. In this example architecture, with 128-bit vectors and an 8-bit width-block, each lane in the 4x32-bit vector is encoded using only 2 bits,

for a total mask size of 8 bits.

This width encoding stage is divided into two steps: a first step (I) where the masks for each vector operand (C) are encoded separately; and a second step (II) where the masks for the two operands are combined to generate the mask for the whole instruction (D). In this second step, the code for each lane in the combined mask is the maximum code of the corresponding lanes in the two operand masks.

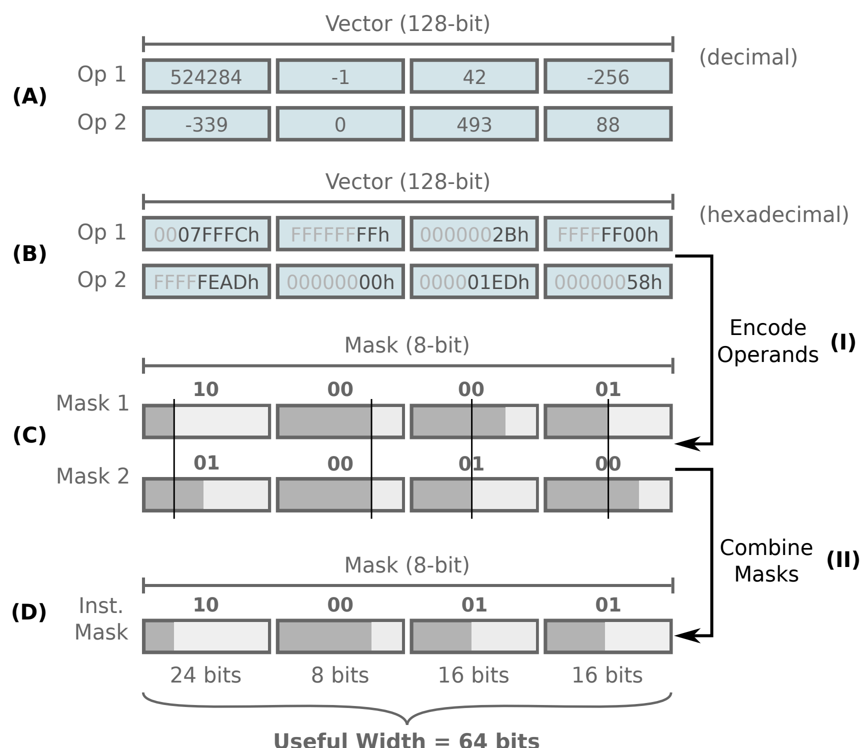


Figure 4.2: Width encoding for an example with a 4×32-bit vector operation with two vector operands, considering an 8-bit width-block. In the hexadecimal representation (B), the unneeded bit-blocks are represented in light grey. In the width masks in (C) and (D), the colouring in the symbol below each 2-bit set indicates the bit-blocks which are useful in the corresponding operand (light-grey), and those that are wasted (darker grey).

The process of computing the width mask for each operand is divided into two parts, the block reduction (I.a) and the encoding (I.b) steps, which are presented with more detail in Figure 4.3. Firstly, in the block reduction step (I.a), the vector bit-array ((A) in Fig. 4.3) is divided into blocks with the width-block size (B), that are processed independently. In this example, when considering an 8-bit width-block ($w = 8$), each 32-bit lane is represented as four groups of 8 bits, as shown in (B), where each bit in these w-bit blocks is denoted as b_i. Each block is then reduced to a 2-bit pre-mask representation (x_1, x_0), where one bit (x_1) corresponds to the block leading bit ($x_1 \leftarrow b_{w-1}$). The other bit (x_0) states whether the remaining bits are equal to the sign bit ($x_0 \leftarrow x_1 \; if \; \forall \, i < w-1, b_i = b_{w-1}$), or are different ($x_0 \leftarrow \overline{x}_1 \; if \; \exists \, i < w-1 : b_i \neq b_{w-1}$), as in (C). These two bits have the minimum information required to determine if the leading sign bit is in the corresponding block, by encoding both the sign of the block (x_1),

and whether that sign is propagated to the next block (x_0). Note that using one bit to encode whether all bits are zero or one is not enough, as it would not handle the case where a block with all ones is followed by all zeros, and vice versa. It is worth noting that this reduction step is agnostic to the element mode, and can be implemented using few logic gates, as it is a logic function with a small number of entries (e.g. 8 bits) and two outputs.

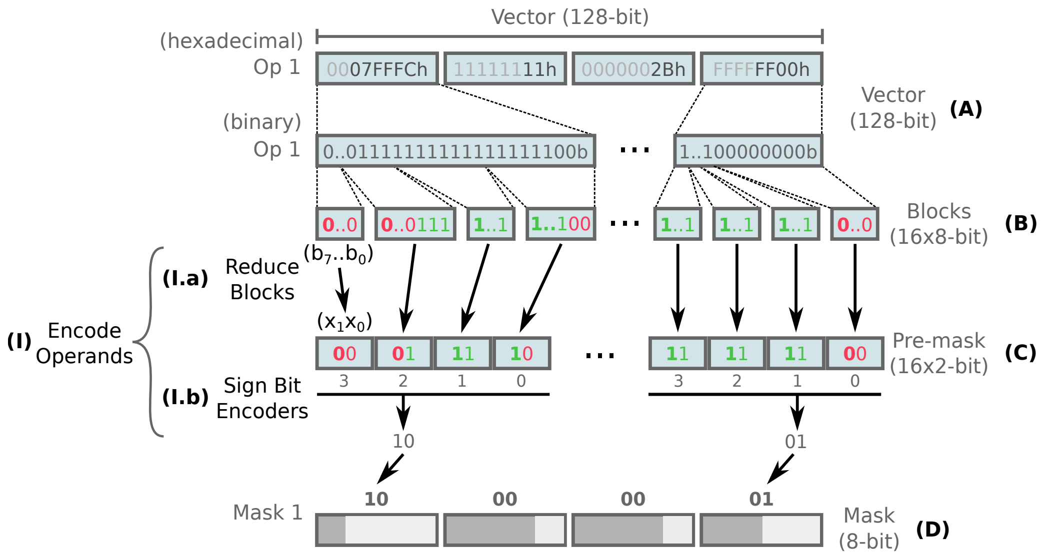


Figure 4.3: Details of the width encoding procedure of a single operand, using the first vector operand from the example in Fig. 4.2. In steps (B) and (C), the 0 and 1 bits are highlighted in opposite colour (red and green, respectively), so that it is easier to visualise how the bits (x_1, x_0) in the pre-mask are generated.

In the second step ((I.b) in Fig. 4.3), a priority encoder identifies the width-block containing the leading sign bit, for each group of blocks that corresponds to a lane (D). In the considered example (with the 4×32-bit mode), each lane corresponds to a group of four blocks in the pre-mask, and the index of the sign bit can be encoded using 2 ($\log_2 4$) bits. The resulting codes are concatenated to obtain the complete width mask, which in this example has 8 bits.

It should be noted that this encoding step depends on the considered vector mode (e.g. 8, 16, 32, or 64 bits) since it determines the size and the number of encoders used (i.e. the number and size of the lanes). However, the encoders used by different modes can share the same logic, as a larger priority encoder can be implemented using smaller ones. Moreover, the same 8 bits are enough to store the width-mask for all the vector modes that are supported, only changing how these bits are grouped (see Fig. 4.4). The 16-bit and 64-bit modes would require 8 and 6 bits, respectively, whereas the 8-bit mode is not supported with an 8-bit width-block (all the width in each lane is always used). Also note from this example that the total width usage, for the same vector, varies depending on which mode is considered, as the width mask itself also changes.

Hence, this width mask can be computed using an array of leading sign bit detectors, one for each

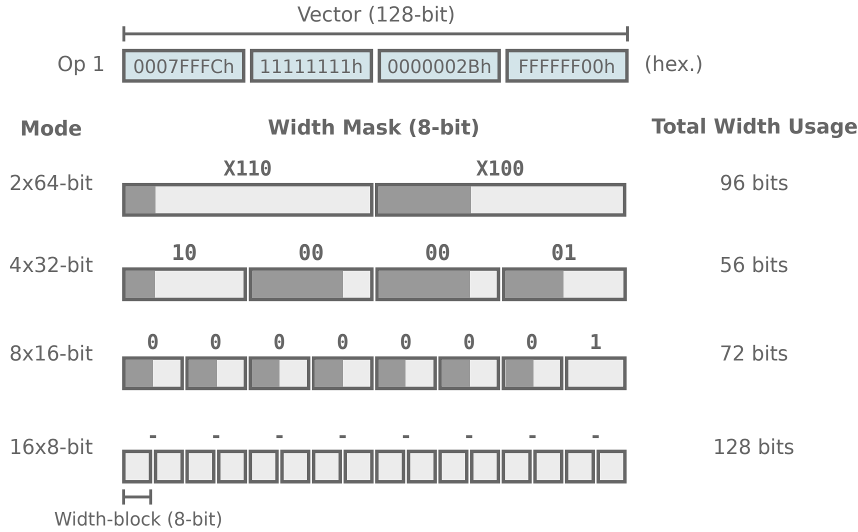


Figure 4.4: Width masks examples for different vector modes, using the same Operand 1 from Figures 4.2 and 4.3. The same 8 bits are grouped differently for the different vector modes. For the 2×64-bit mode, only 3 bits are required per lane ($\log_2(\frac{64}{8}) = 3$), so $1 + 1$ bits in the mask are not used (marked as `X`). The 16×8-bit mode is not supported with this width-block, and so the full width is always used (128-bit).

vector element. Since the width values only have to be determined with the granularity defined by the width-block, only the index of the block with the leading sign bit has to be detected, and not the index of the bit itself. Therefore, the detection step can happen after the blocks are reduced, as 2 bits per block are enough to determine if it contains the leading sign bit. This greatly decreases the number of entries and the complexity of the detectors.

Notice that the number of bits per lane of the width mask, which depends on the vector mode, can be determined as $\log_2(\frac{m}{w})$, where m represents the number of bits per lane in that mode and w is the width-block size. For an n-bit vector, the total size of the width mask (all lanes) corresponds to $\frac{n}{m}\log_2(\frac{m}{w})$, when considering that n, m, and w are powers of 2. Naturally, it only makes sense to consider vector mode values which are greater than the width-block ($m > w$) for the width encoding step, as otherwise, the width for each element is always the maximum (when rounding towards the width-block). For example, an 8-bit element always uses all the bits when choosing an 8-bit width-block. When considering an architecture supporting multiple vector modes, but sharing the width mask bits across all of them, the mode that requires the larger number of bits is the one with the smallest element size, which corresponds to $m = 2w$. Hence, the upper bound for the mask size required for a given architecture is obtained from

$$\frac{n}{2w}\log_2(\frac{2w}{w}) = \frac{n}{2w}. \tag{4.1}$$

As we expected, this expression shows that the impact of generating, transferring, and storing the width mask can be significantly reduced by increasing the width-block parameter (w).

The actual number of bits required for the width mask also depends on which modes are supported

Table 4.1: Width-mask size trade-off for an architecture with 128-bit vectors ($n = 128$), for different width-block sizes (w) and with the typical vector mode sizes (m). The upper bound is determined by using the expression $\frac{n}{2w}$, and the maximum corresponds to the width required by the chosen modes.

Width-block [bits]	$m = 8$	$m = 16$	$m = 32$	$m = 64$	Upper Bound [bits]	Maximum Mask Size [bits]	Fraction of Vector Size [%]
	Mask Size [bits]						
$w = 1$	48	32	20	12	64	48	37.5
$w = 2$	32	24	16	10	32	32	25.0
$w = 4$	16	16	12	8	16	16	12.5
$w = 8$	—	8	8	6	8	8	6.25
$w = 16$	—	—	4	4	4	4	3.13
$w = 32$	—	—	—	2	2	2	1.56

in that architecture, and typically $m \in \{8, 16, 32, 64\}$. Table 4.1 presents the size of the mask in the case of a 128-bit vector ($n = 128$) when considering these modes, for several different width-block sizes (w), considering that it is not useful to have width masks for vector modes lower or equal to the width-block. This table confirms that increasing the width-block parameter greatly reduces the overhead of the width mask. In this example, when a width-block of 8 bits ($w = 8$) is applied to an architecture with 128-bit vector registers, the width mask of an operation requires 8 bits, corresponding to 6.25% of the vector size (∼3% when considering the size of two vector operands). When the original width value is encoded directly (i.e. $w = 1$), the width mask requires 48 bits, which corresponds to 37.5% of the vector size and represents a very high overhead.

4.1.2 Packing narrow-width vector operands

Having encoded the width required for each vector sub-operation, one of the possible optimizations in the execution unit is to clock-gate unneeded portions of the functional units, to reduce the waste of dynamic power with unneeded circuit activity. However, although this power efficiency technique can already be performed with fine granularity (i.e. to specific logic blocks), it still requires an overhead in the control logic that is necessary to trigger it. Applying it to specific element portions of each vector unit, that change every cycle, can degrade the benefit it would provide.

Hence, a new mechanism is proposed to simplify this clock gating control: **operand packing**. This mechanism consists in allowing the elements in a vector operand to have different sizes, by compressing the vector to remove the unneeded bits in each data element. This way, only a fraction of the available functional unit's width is required to execute that instruction, and the unused blocks are contiguous and can be clock gated with simpler control logic. An example of a 4x32-bit integer vector addition in a processor with a 128-bit vector size is presented in Figure 4.5, where each sub-operation can be performed using a smaller width than the 32-bit element size. If the operands are all packed to fit in the least significant bits of the vector, this instruction can execute using only 56 bits of the unit, instead of

128 bits. Furthermore, with this mechanism, the activity of the functional units will tend to concentrate in the lower portion of the unit (for example), and the higher portion will tend to remain clock gated for several contiguous clock cycles. Since, triggering and recovering from clock gating also generates transients and has an energy penalty, even if much smaller than power gating. This packing allows the clock gating to be interrupted less frequently, making the power efficiency gains higher.

Vector operands are packed using the information in the operation's width mask. When there are two or more vector operands, each vector lane must be properly aligned (reserving the size used by the widest element, as it is illustrated in Figure 4.5), so that the operation can still be performed using the existing vector arithmetic units. In the write-back stage, after the operation is executed, the result vector can be unpacked using the same width mask, recovering the original result.

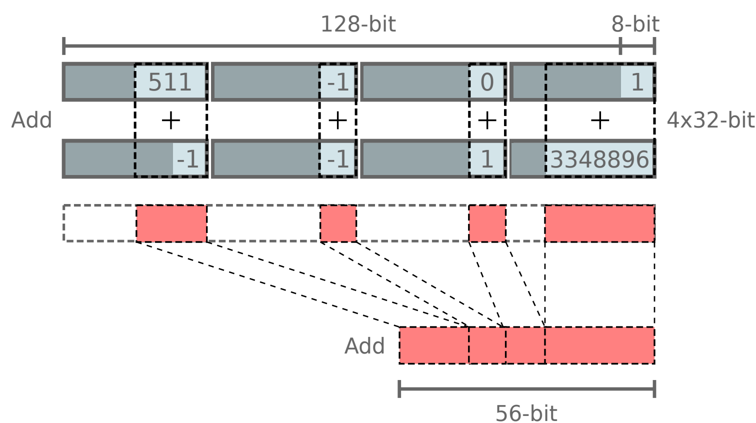


Figure 4.5: Vector operand packing in a SIMD Add operation. With an 8-bit width-block, this 128-bit (4×32-bit) operation can be compressed to use only 56 bits of the SIMD ALU. The bit-width portion required in each operand lane (of the 32 bits) is represented in light grey.

The packing and realignment of vector operands take advantage of the width-block concept already introduced, as a coarser granularity simplifies the additional logic that is necessary. In particular, if the chosen width-block is a multiple of the smallest vector element size supported in that architecture (typically 8 bits), the overhead in the functional units is minimal. Futhermore, adding support for the execution of packed instructions with irregular element sizes can take advantage of the hardware that already enables vector functional units to support different element size modes. In the designs proposed in the literature for these vectorized units (see Subsection 2.3.3), the same hardware is shared for operations of different element size by dynamically changing the element boundaries accordingly to the mode signal. Supporting more elaborate element patterns is mostly a matter of extending the control signals that change these boundaries to use the width mask as an additional input.

The proposed implementation of a vector adder unit with support for irregular element sizes is based on the 128-bit carry-propagate adder design presented in [49]. This unit consists of several intercon-

nected 4-bit Carry-Lookahead Adder (CLA) blocks and should be similar to the SIMD units' implementations in current processor designs. The element boundaries are changed by inhibiting (killing) the corresponding carries between blocks through the addition of an additional 2-input AND gate to the 4-bit CLA blocks (see Figure 4.6), which the authors argue does not increase the critical path delay. This design can be modified to support irregular element boundaries by extending the control circuit that generates the carry-kill signals. Whereas in their design the carry-kills only depended on the element mode, these can be extended to also take into account the width mask.

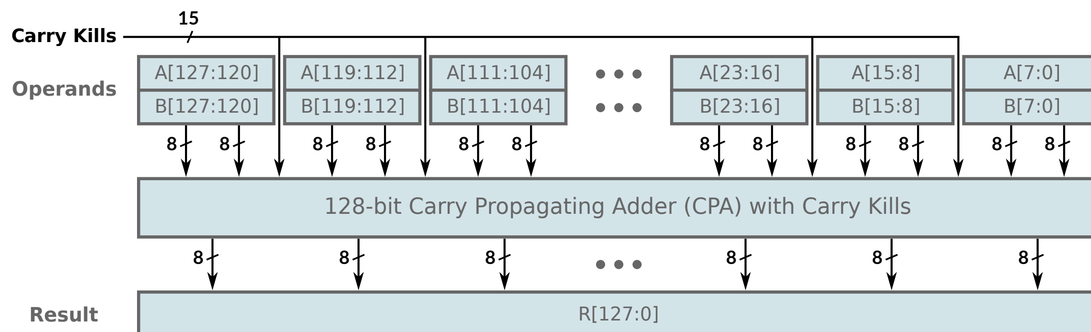


Figure 4.6: Implementation of a 128-bit vectorized adder unit, which allows for irregular element boundaries, with an 8-bit width-block.

For the multiply and multiply-accumulate operations, the architectures presented in [49] and [50] can also be extended to support irregular element sizes, by modifying the control logic that selects which partial products are added, while maintaining most of the existing multiplier hardware. As long as the partial products for each element do not overlap (see Figure 4.7), the same hardware can be shared for different element boundaries. Supporting mixed element sizes is a matter of allowing a finer granularity of control over which partial products are suppressed, and making sure there are no carries in the addition stage between different elements. The base architecture, proposed in [50], which uses an array multiplier for the partial product generation, is the most interesting for this implementation. This design needs less control logic for changing the element boundaries, as it does not require suppressing the carries at boundaries in the reduction tree and final addition steps.

For correctness, the execution of packed instructions must also support the overflow of sub-operations with a lower width than the original element size, as this is not an actual overflow. For the addition operation, the carries between element boundaries are verified, despite not being propagated. In case of an overflow, the encoding of the corresponding element in the width mask is increased, and the element's size is expanded to fit the additional bit. For the multiplication, the bits corresponding to the higher half of each element's multiplication are already generated when reducing the partial products (e.g. an 8-bit by 8-bit multiplication generates a 16-bit result in the final adder). When an overflow fits in the original width, these bits must be multiplexed to the result vector, and the corresponding width mask code is increased. A significant portion of the extra logic that is required to multiplex the most significant part of

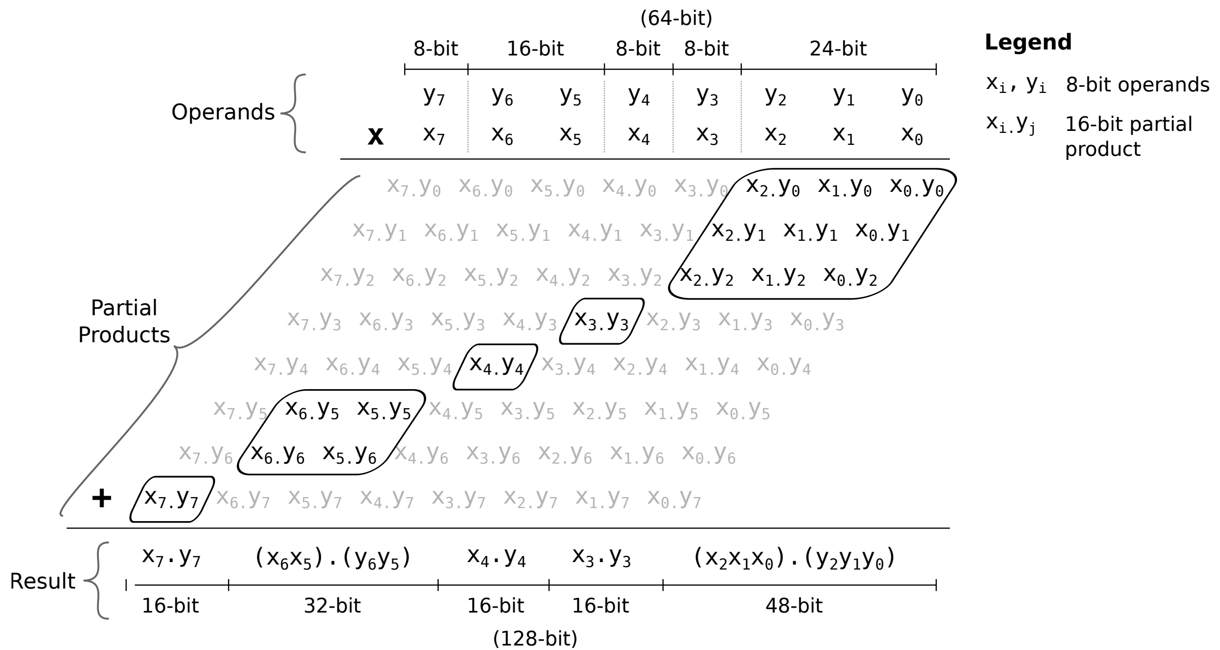


Figure 4.7: Selection of partial products for an irregular element size multiplication, using the existing hardware in an example 64-bit SIMD unit (compare with Fig. 2.10). The partial products between blocks from different elements (marked in light grey) are discarded.

the multiplication result should already be present in SIMD functional unit implementations. In fact, there are instructions in most vector extensions that keep the entire multiplication result, normally by increasing the element size of the result. For example, in the ARM NEON extension, one of these instructions is `SMULL`, signed multiply long (vector).

4.1.3 Fusing vector operations

As it was referred before, the main goal of packing vector instructions is to reduce the energy consumption of the execution unit, by switching-off unneeded portions of the Arithmetic Logic Unit (ALU). However, if several (independent) packed instructions are simultaneously ready for execution in the processor's issue queues, whenever two or more of these instructions fit in the full vector width of the ALU, they can be issued together to the same arithmetic unit. This mechanism will henceforth be denoted as **operation fusing** (see Fig.4.8). It should be noted that while these instructions are combined during the execution stage, they continue to be regarded as independent instructions, writing to different registers and committing (or even being squashed) independently.

Hence, if a significant fraction of the instructions can be fused for execution in the same unit, the execution throughput can be sustained using fewer SIMD functional units. The remaining units can be completely clock gated, or, whenever they are not needed for long execution intervals, they can even be power gated or removed from the microprocessor design. Power gating also reduces leakage

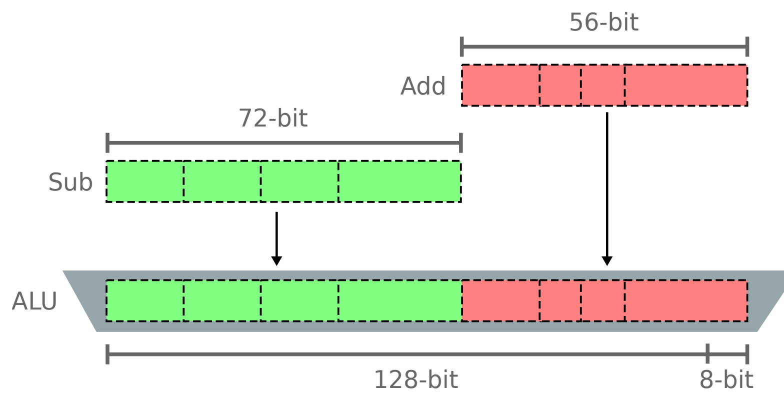


Figure 4.8: Fusing two similar vector operations (an addition and a subtraction) that fit in the 128-bit ALU.

dissipation but has the disadvantage of introducing a high state transition overhead and only provides relevant energy savings when it is not interrupted frequently.

On the other hand, in execution intervals with very intensive SIMD unit usage, instead of reducing the number of functional units, this mechanism can be leveraged to increase the processor performance, by increasing the maximum throughput of the vector pipelines. Whenever this approach significantly reduces the execution time, it can also result in energy savings, as energy is the product of power and time ($E = PT$). Moreover, as the fusing mechanism also reduces the total number of accesses to the SIMD units (each pair of fused instructions is one less cycle in which that unit is active), it can allow for a very interesting combination of execution time reduction and energy efficiency.

One possible implementation of this fusing mechanism is to execute together only instructions with the same arithmetic operation, which does not add extra complexity to the existing ALU design and can be seen as a type of auto-vectorization. However, it can still be advantageous to allow for different (but similar) operation types to be fused (see Fig. 4.8), in order to increase the number of opportunities found for multiple issuing, even if at the cost of a slight increase in the ALU complexity.

4.1.4 Gating functional units

As it was already explained, the proposed operand packing and operation fusing mechanisms can be translated into dynamic energy reduction when combined with clock gating techniques, either by switching off the clock signal for unused portions of the functional units, or even the entire unit. Clock gating the whole unit, as frequently allowed by operation fusing, can be more advantageous than gating two half units, as a result of operation packing, as the whole scheduling and control logic in that unit can be switched off. However, there might not be enough instructions to fuse in a given cycle (or their width might not be compatible), so even with operation fusing it is useful to enable partial clock gating of a unit.

As it was also already presented, when operation fusing allows the average number of needed functional units to be reduced, and for long execution periods, it can also be combined with power gating to

enable for leakage power reduction, without significant performance penalties. Hence, the best results enabled by the proposed mechanisms should be obtained by a combining the usage of clock and power gating techniques in the vector execution units, as illustrated in Figure 4.9.

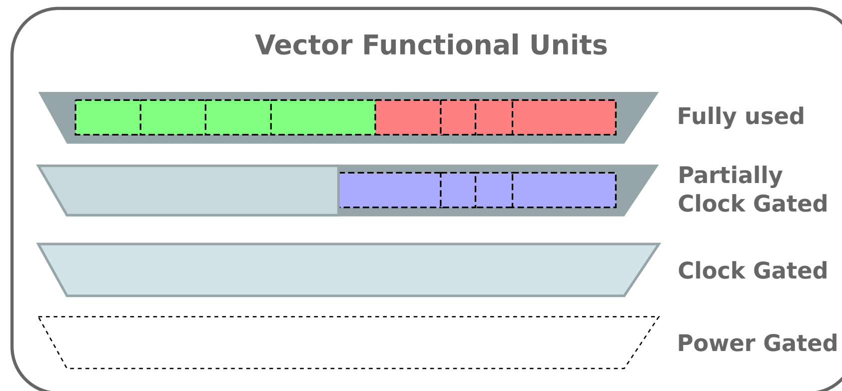


Figure 4.9: Example of opportunities for clock and power gating vector functional units. For an execution engine with four units, the three vector instructions issued in this cycle were packed and executed using only two functional units, one of which has a portion that can be clock gated. The third unit is not active in this cycle and can be clock gated, and the fourth has not been needed for a longer period and was power gated.

Dynamically triggering the power gating of some vector units according to the current execution workload (namely depending on the occurrence rate of vector instructions and on the width of the integer data) requires a decision mechanism similar to those proposed in [12, 32, 33]. However, implementing this decision mechanism is out of the scope of this thesis, and is left as future work (see Section 7.2). Nevertheless, this study will show that the proposed fusing mechanism facilitates the application of power gating. The main difficulty when choosing the correct instant to trigger the sleep signal is the performance impact of shutting down the unit too soon. In case the unit is required shortly after, there is a heavy penalty because of the delay in powering it up. As the fusing mechanism allows an increased throughput with the existing units, it amortizes the impact of not having the gated unit ready and can even postpone the need for waking it up. This allows a higher energy saving margin to be obtained, with simpler decision mechanisms.

Finally, the fact that functional units (especially vector units) are almost always pipelined, and that the different stages in the pipeline may be in different usage states, has not been mentioned so far. However, this only implies that the gating mechanisms (in particular, clock gating) should be implemented to allow different portions of the pipeline to be in different states at the same time (isolated by the pipeline stage transition registers). This is achieved by also pipelining the gating trigger signals. Even more, the existence of the execution pipeline allows the gating control mechanism to look ahead, and to start preparing state transitions earlier.

4.2 Integration in conventional processor architectures

Implementing the set of optimization mechanisms proposed in the previous section only requires some minor architectural changes in the processor execution engine. In the particular case of an out-of-order core, it requires some adaptations in the vector pipelines and, in particular, in their issue queues, SIMD functional units, and issue schedulers.

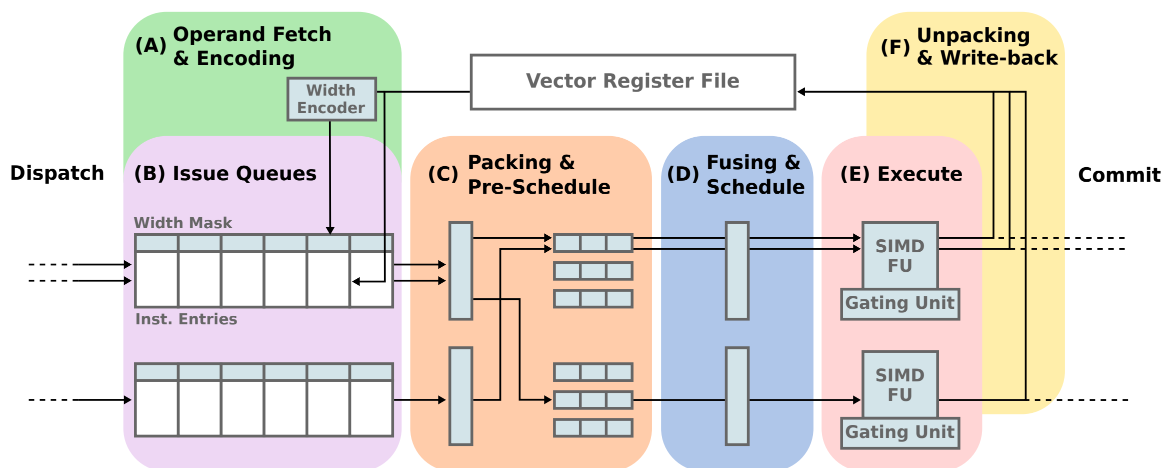


Figure 4.10: Detailed changes proposed to the execution pipeline of vector instructions, in the out-of-order execution engine presented in Fig. 3.3. Apart from the additional width encoding step in the operand fetch (A), operation fusing in scheduling (D), and unpacking in write-back (F), a new stage is added for packing and pre-schedule by width buffers (C).

Figure 4.10 depicts the proposed modifications to the vector execution engine. The width encoding step is performed at the same time as each operand is fetched into the issue queue (A), and its result is used to update the width mask for that instruction in (B) (using the max operator). Thus, the width information required for each vector instruction can be encoded using a single additional field in the issue queue (B). For the particular example of a design with an 8-bit width-block and 128-bit vectors, this field only has 8 bits, which is minimal when compared to the size of the vector operands that are stored in the queue. Hence, adding support for width encoding is simply a matter of extending the operand fetch mechanism (part (A) in Fig. 4.10), whether the values are fetched from the physical register file or directly from the Common Data Bus (CDB). This step should not cause a significant increase in the critical path, as the encoding process is simple (recall Fig. 4.2), and can be performed as soon as the previous (source) instruction writes on the CDB.

The, the vector operands can only be packed when the width mask for the instruction has been fully updated, i.e. when all the operands have been fetched (and that instruction is ready to issue). In order to avoid any penalization in terms of the critical path, with a consequent degradation in the processor performance, a new pipeline stage is added to the vector unit between issuing and execution (C). In

this new stage, the operands for ready instructions are packed, and the instructions are reorganized in buffers according to their width (and instruction type) so that they can be easily fused and issued. The consequent increase of the vector pipeline latency (by one cycle) is mostly hidden by the out-of-order execution. Hence, the resulting performance penalty is not significant (sometimes it is even mitigated by the fusing mechanism) and is largely outweighed by the consequent energy savings.

Operation fusing is accomplished by implementing an extension to the issue queue scheduler, in order to monitor the width masks of ready instructions, trying to issue more than one at the same time whenever possible (D). The vector operands of these additional instructions are then shifted (or multiplexed), so that there are no overlaps in the functional unit, being issued using the remaining bandwidth in the issue port. The width masks of the fused instructions are used to encode the boundaries between elements (in the same instruction or between fused ones) for packed execution in the ALU (E), which is extended to support irregular element sizes. After the instruction is executed, in the write-back stage (F), the resulting vector is unpacked with the information in the width mask.

Since each execution port usually has its independent issue queue, some fusing opportunities might be lost because those instructions are in different pipelines. This problem is circumvented by the new packing and pre-issue stage, which is also used to exchange instructions between pipelines, when organizing them in buffers according to their width (part (C) in Fig. 4.10), and thus allows more pairs of compatible instructions to be found.

An essential remark in this modified issuing stage is that some degree of priority should be given to the execution of instructions that were fetched earlier, even if this means missing some opportunities to fuse later instructions. Otherwise, an instruction that does not have a fusing candidate might not find an opportunity for execution, stalling the pipeline. This can be done by respecting the order by which the instructions were inserted in the pre-issue buffers (C), and by using a round-robin mechanism when choosing from which buffers to schedule (D).

The clock or power gating mechanisms of the processor are triggered by the gating control units, which are added to each vector functional unit (see part (E) in Fig. 4.10). Each control unit monitors the activity of its vector unit and **clock gates** the entire unit (or portions of it) when it is idle. These control units also decide when to trigger **power gating**, whenever the unit is expected to remain idle for a long period (namely by using a decision mechanisms inspired by [12, 32, 33]). While a vector unit is in the sleep mode (i.e. power gated), the instructions in its issue queue are handled by the remaining units, thanks to the new shared stage in the issuing step. This way, its reactivation is postponed until a significant increase in throughput is required.

4.3 Summary

This chapter described how an out-of-order microarchitecture can be extended to efficiently execute narrow-width SIMD operations, with limited overhead and using most of the hardware already existing in these processors. The proposed design efficiently encodes the width required by a vector instruction in a compact mask (e.g. that takes up only $\sim 3\%$ of the bits required by the operands) and propagates this information so that these instructions are efficiently executed in the corresponding units. The microarchitecture adaptation proposed in this chapter is also transparent to existing applications and libraries, increasing its applicability, and no changes in the compiler are required.

5

Prototyping and Experimental Workflow

Contents

5.1 Architectural simulation tools 52
5.2 ARM ISA and the NEON vector extension 55
5.3 Implementation of the architectural changes 56
5.4 Traces and performance counters 58
5.5 Power modelling 60
5.6 Experimental workflow 61
5.7 Summary 62

Following the proposal of the mechanisms and architectural changes to exploit narrow-width in vector operations, this chapter describes how these changes were implemented and evaluated using a state-of-the-art computer architecture simulation framework, gem5, combined with a power, area and timing modelling framework, McPAT.

Section 5.1 introduces the architectural simulator and the baseline processor models, and Section 5.2 presents the Instruction Set Architecture (ISA) that will be used for prototyping. Section 5.3 details how the original model was modified, and Section 5.4 presents the new performance metrics which were added to evaluate the impact of this Thesis' proposal. Then, Section 5.5 explains how these metrics were used to model the power efficiency of the modified core. Finally, Section 5.6 provides an overview of the experimental workflow.

5.1 Architectural simulation tools

Processor simulation is widely used in the initial ideation and evaluation stages of new optimization opportunities and architectural features, both in the academic and in the industry backgrounds. It allows for fast prototyping of new ideas and for extensive design space exploration (i.e. different configurations can be evaluated easily).

When comparing different simulation levels (see Fig. 5.1), the most accurate evaluation is usually obtained through Register-Transfer Logic (RTL) specifications such as VHDL or Verilog. However, RTL simulation is very limited in terms of design space exploration, and it is highly time-consuming, both in terms of specification and simulation time [60]. Alternative processor models include mathematical estimation models (e.g. implemented in Python or MATLAB), functional executable models (e.g. QEMU or gem5 in a functional mode), and cycle-accurate models (e.g. gem5 in a cycle-accurate mode). These more abstract models trade different levels of accuracy for faster evaluation and exploration times (as depicted in Fig. 5.1). Consequently, they are a better fit for an early evaluation of new architectural ideas, as it is the case of exploiting narrow-width in vector operations. Moreover, there are no publicly available complete RTL specifications of modern out-of-order processors, which are the focus of this work. In contrast, very accurate models are available at higher abstraction levels.

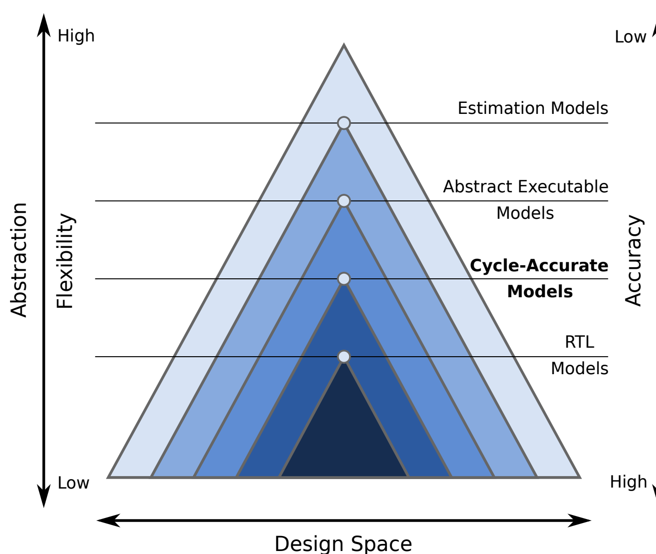


Figure 5.1: Comparison of different processor specification and modelling levels [60]. Models that are lower in the pyramid are more accurate, but are less flexible to changes, and are more constrained in terms of the available design space (i.e. the pyramid base in the figure is narrower). For this work, a cycle-accurate model was used.

Gem5 is a highly configurable, modular, and extensible open-source computer system simulation framework [55]. It is prepared to emulate a variety of processor models and ISAs (e.g. ARM, ALPHA, and x86), with varying degrees of detail and accuracy. It is actively developed by leading industry

and academic partners, such as ARM, AMD, Princeton, and MIT, and has been used in hundreds of publications. In particular, this simulator provides a very detailed out-of-order processor model, *O3* (**o**ut-**o**f-**o**rder), which adequately models the superscalar pipelines with cycle-accurate timing, at the cost of long simulation time. For example, *O3* models the issue queues and functional unit states, and it accurately handles dependencies between instructions. However, its implementation is more complex than the remaining in-order models, such as (see Fig. 5.2): *AtomicSimple*, a minimal functional emulator; *TimingSimple*, an extension of atomic with timing in memory references; *InOrder*, a pipelined in-order cycle-accurate model.

Gem5 provides two simulation modes (see Fig. 5.2): *system-call emulation mode* (SE), where the external devices and the operating system are not modelled, and system calls are emulated by the host system; and *full-system mode* (FS), where both user-level and kernel-level instructions are executed, and the complete architecture is simulated (i.e. with an operating system and IO devices). Full-system emulation allows for Linux images to be booted, which enables a wider variety of applications and benchmarks to be run, and provides more realistic and accurate results, at the cost of quite longer simulation times and a more complex simulation setup. However, gem5 provides the ability to record snapshots of the processor state in interesting instants (e.g. after the system boots), which saves time by resuming execution from that checkpoint.

		CPU Model			
		Atomic Simple	Timing Simple	In Order	O3
System Mode	System-Call Emulation	Speed			
	Full-System				Accuracy

Figure 5.2: Comparison of gem5 simulation modes and models, in terms of the speed-accuracy compromise [55]. In this work, the evaluation was performed using full-system simulation with the O3 model.

In accordance, the evaluation of the narrow-width opportunity in vector computations, presented in Chapter 3, was performed using the gem5 simulator framework. This simulator was also used to estimate the impact of the optimization mechanisms and architectural changes proposed in Chapter 4. Futhermore, the experimental work was performed with the *O3* CPU model, as it is an accurate model of a superscalar out-of-order core. The full-system simulation mode was also used, both because it generates more realistic results than system-call emulation, and because it is required by several of the

profiled applications (e.g. PARSEC and SPEC2006 suites).

The changes to the vector pipeline were prototyped using the ARMv8 Advanced SIMD (NEON) extension (see Section 5.2 for more details), because of the maturity of this extension (i.e. when compared to ARM SVE). At the time the experimental work was performed, there was a wider variety of applications and libraries vectorized for this extension, better compiler support, and full implementation in gem5.

Two different out-of-order CPU configurations (based on the *O3* model) were used for the experimental simulations, whose baseline parameters are presented in Table 5.1: a model of an ARM's 4-wide **Cortex-A76** core; and a model of an 8-wide **High-performance** core model, which is a scaled version of the A76 model with the functional units pool configuration based on public information of the 8-wide Apple Vortex Core[1].

Table 5.1: Baseline parameters of the CPU models

	Cortex-A76	High-performance
Frequency	2.0 GHz	
Fetch Width	4 insts/cycle	8 insts/cycle
Dispatch/Issue Width	8 insts/cycle	12 insts/cycle
Issue Queue	120 entries	180 entries
Load Queue	68 entries	68 entries
Store Queue	72 entries	72 entries
ROB	128 entries	192 entries
Integer Reg. File	256 registers	384 registers
FP/SIMD Reg. File	256 registers	384 registers
Functional Units	3 Int ALU (1 cycle) 1 Int Mul/Div (2/12 cycles) 2 FP/SIMD units: - SIMD ALU (2 cycles) - SIMD Mul (4 cycles) - FP ALU (2 cycles) - FP Mult (3 cycles)	6 Int ALU (1 cycle) 2 Int Mul/Div (4/8 cycles) 3 FP/SIMD units: - SIMD ALU (2 cycles) - SIMD Mul (3 cycles) - FP ALU (3 cycles) - FP Mult (4 cycles)
Private L1 ICache	64KB / 4-way (8 MSHRs) 1-cycle latency	
Private L1 DCache	64KB / 4-way (20 MSHRs) 2-cycle latency	
Shared L2 Cache	256KB / 8-way (46 MSHRs) 9-cycle latency	
DRAM	DDR3 1600 MHz 8x8GB	

[1] Apple A12: `https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets`

5.2 ARM ISA and the NEON vector extension

ARMv8-A is the latest generation of the general-purpose ARM Instruction Set Architecture (ISA), which is implemented in ARM's Cortex-A processor series, as well as by several other microprocessor cores from sublicensee companies, such as Qualcomm's Falkor, Samsung's M4, and Apple's A12 Vortex and Tempest. ARMv8-A is a 64-bit ISA, with a wide physical and virtual addressing space, and a large architectural register bank, with thirty-one 64-bit general-purpose registers. The ARM ISA was initially designed as a Reduced Instruction Set Computing (RISC) architecture, with a simpler but more efficient set of instructions, but has since evolved to include a wide variety of specialized features, namely vector extensions. The ARM ISA is currently the most widely used, in great part due to its ubiquity in the mobile and embedded sectors, but it is slowly entering the desktop and server markets. More powerful cores are available each generation, and there is an increasing support in terms of available toolchains and libraries.

ARM NEON is the advanced SIMD vector extension for the Cortex-A series processors. NEON is a packed vector extension, which (for the ARMv8-A version) uses 32 quadword (128-bit) architectural register, accommodated in a separate register bank that is shared with floating-point registers[2]. For each NEON instruction, the input register operands are considered as a vector of data elements of the same type, which can be signed or unsigned integers with 64, 32, 16, or 8 bits, or single or double precision floating-point. This vector extension is designed for a wide range of application domains, from high-performance computing to multimedia.

In general, NEON vector instructions perform the same operation in parallel over all lanes, but there are specific instructions that either operate over only one element, perform reductions across the whole vector, or handle pairs of side-by-side elements (see Fig. 3.2a). Most instructions have the same element size for the two source vector operands, but there are special instructions where these sizes differ, namely long, wide, and narrow instructions (see Fig. 3.2b). NEON also includes some instructions which handle vector elements in more elaborate ways, as is the example of the table vector lookup instruction (`TBL`), which uses the values of one source vector as indexes to select values from up to three other vectors, and the `ZIP` instruction which mixes alternate elements from two source vectors. A comprehensive list of the instructions in the NEON extension is provided in Appendix B, with a focus on the subset of instructions with integer vector operands that were given support for the proposed mechanisms in the presented prototype.

[2]ARM Neon programming quick reference: `https://community.arm.com/developer/tools-software/oss-platforms/b/android-blog/posts/arm-neon-programming-quick-reference`

5.3 Implementation of the architectural changes

Gem5 (in particular its *O3* CPU model) is implemented in C++ and Python, following the object-oriented programming paradigm [55]. Its architectural components, such as functional units, instruction queues, and register files, are represented by interconnected objects (see the object diagram in Fig. 5.3, which mimics the example microarchitecture presented in Fig. 2.1). Gem5 is a discrete-event simulator, where time (in clock cycles) is simulated as a series of discrete events that instantaneously change the system state. These events (e.g. accesses to memory, transitions of instructions between stages, etc.) are scheduled accordingly to their dependencies in a priority queue, by increasing order of trigger instant. As an example, the parameterization of the execution latency of each operation class provides the delay between the issuing and write-back events for an instruction of that class.

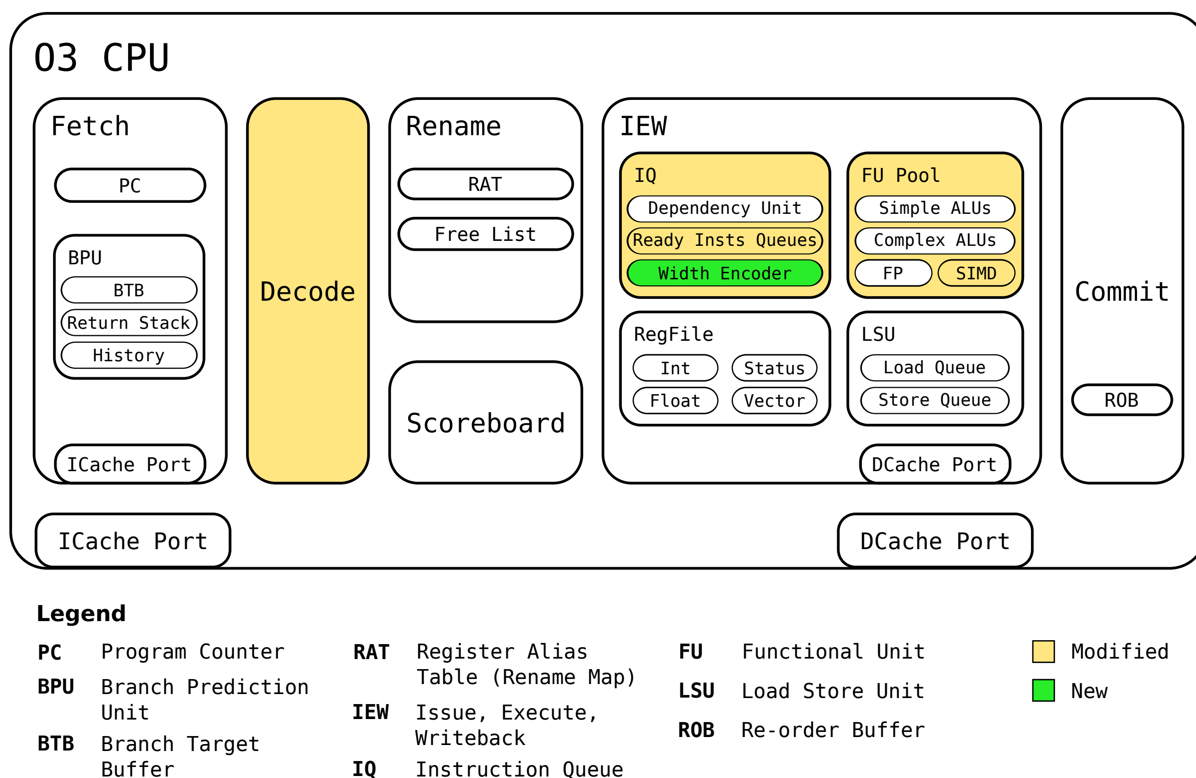


Figure 5.3: Modifications made to the internal structure of gem5's O3 (**O**ut-**o**f-**O**rder) CPU model. Modified components are highlighted in yellow, and new structures in green.

Each instruction is represented by two objects: the `StaticInst` and the `DynamicInst`. The `StaticInst` keeps the immutable state for each different decoded instruction, such as the operation code and the architectural registers' names, and is shared by all instructions with the same machine code (for an efficient memory usage). The `DynamicInst` object stores the dynamic state for each different instruction instance, such as the renamed physical registers and the current pipeline stage.

Every simulation object is implemented by two classes: a wrapper in Python, that specifies its parameters and interface, which is instantiated in the configuration scripts; and the actual definition in C++, which defines the object behaviour and stores its state.

Although gem5 is designed to be extensible, implementing a radically new mechanism, such as the proposed instruction issuing dependent on the operands' width, requires changing several components at various abstraction layers of the simulator. In the particular case of the presented work, prototyping the width encoding, operand packing, and operation fusing mechanisms required editing around 40 source files, with more than 5000 lines of code changed or added[3]. This represents 10% of the 50000 lines that implement the O3 CPU model, most of which had to be read to understand how the model operates.

The main modifications to the simulator were the following (they are highlighted in Figure 5.3):

- Extension of the **decoding** stage, to extract the information that is required to compute the instruction width, i.e. element size, number of vector operands, and vector addressing mode;
- Extension of the **vector issue queues**, to introduce additional fields with the new decoded information and to store the instruction width mask;
- Addition of a new component, the **width encoder**, which uses the decoding fields in the instruction queue and the operand register values to compute the width mask. This structure also aligns the packed elements in the operands' registers and assigns the ready vector instructions to the adequate width queue;
- Addition of new queuing structures, the **queues by width**, which buffer the ready vector instructions, organized by the required Arithmetic Logic Unit (ALU) width, so that they can be efficiently issued by the scheduler;
- Extension of the **vector execution scheduler**, to optimize the issuing of vector instructions, trying to perform operation fusing according to the type and width of the instructions. Additional control logic was also added to keep track of which portions of the vector functional unit were already used in that cycle;
- Modification of the **vector functional units**, so that a packed vector instruction can be executed using only a portion of the unit's width, and each unit executes additional instructions if possible.

The changes in the decoding stage are the ones that represent the lowest abstraction levels, as they are dependent on the specific ISA, and require modifications in each particular instruction group. Most of the instructions in the ARM NEON vector extension were extended to add support for width optimizations. The complete list of supported instructions is highlighted in Appendix B. As already explained (see

[3]The modified source code is available at `https://github.com/miguelpinho/gem5-thesis`

Section 3.1), the width of a vector instruction depends not only on the actual operand values but also on the vector element size (i.e. 32-bit, 16-bit. . .) and operation mode (e.g. regular, pairwise. . .). This extra information has to be propagated to the execution engine, so new fields are added to the instruction queues. In the actual simulator, this was accomplished by extending the `StaticInst` and `DynamicInst` classes.

A new component, the `WidthEncoder`, uses this new decoding information and the operand values (as soon as they are available) to generate the width mask. The mask is stored as a new object, the `WidthMask`, in the `DynamicInst` object. The `WidthEncoder` was included in the `InstructionQueue` component, which is responsible for monitoring dependencies and issuing ready instructions. It already uses a set of priority queues to organize ready instructions by operation type, so these are extended to also group instructions by operation width. The issue scheduler logic in the `InstructionQueue` was also extended to try to fit several compatible instructions in the same functional unit, by using new state variables in the functional units, which record the remaining width.

Gem5 also allows adding new parameters to its components by extending the Python wrappers and the configuration scripts. The *O3* CPU implementation was extended to add the new width-block parameter. The corresponding integer value (in bits) is passed as an argument when initializing the `WidthEncoder` component, and it is used when computing the width mask.

5.4 Traces and performance counters

To evaluate the gains obtained by exploiting narrow-width in vector operations and to estimate the impact of the proposed architectural changes, both in terms of power efficiency and performance, gem5 was also extended to record several new relevant execution metrics and counters. The gem5 simulation framework is very extendable in terms of measured statistics and provides several default object types to represent common metrics, such as histograms, averages, and formulas. Each component has a specific interface for defining its metrics, which can then be updated in the component's behaviour description. Unfortunately, these statistics are not already outputted in a structured format (e.g. JSON or CSV) and are instead dumped into a text file, requiring additional effort for parsing them.

The statistics are automatically outputted at the end of the simulation (corresponding to the whole execution interval), but they can also be reset and dumped at specific instants during execution. This can be done with custom shell commands (in the full-system mode), or by using special reserved machine code instructions, which can be added through assembly macros in the application source code. A common practice is to reset the statistics at the start of the Region Of Interest (ROI) in the code (after the code segment with the initialization of variables and input reading), and to dump them at the end (before output generation), so that only the relevant application portion is evaluated.

In accordance, several new statistics were added in the scope of this work, focusing on the width distribution of vector elements and operations, on the functional unit usage, and on the operation fusing effectiveness:

- histograms of the number of decoded and issued vector instructions, by instruction type and element size;
- histograms of operand's width, both per individual elements and whole packed vectors, grouped by instruction type and element size;
- histograms of the number of active vector functional units at each clock cycle, and the width used in the units;
- counter of the number of instructions that were fused for execution in the same functional unit;
- counter of the number of otherwise compatible instructions that could not be fused because there was not enough width remaining in a vector functional unit.

By default, gem5 also generates other needed statistics, such as the simulated execution time, queue states, and register and functional unit accesses counters, which are also used for power and energy consumption estimation.

For debugging and more detailed execution analysis, gem5 also provides tracing capabilities. Traces differ from the gem5 statistics, as they correspond to the state in specific instants and not to the summary over large intervals. These traces are generated by adding specific macros in the simulator code (similar to a print function), which are associated to a flag. These flags allow a fine degree of control over which information should be outputted, and are usually related to specific components (e.g. the `InstructionQueue`). For more complete traces, gem5 also provides compound flags, e.g. a flag for all components related to the execution stage. The trace output is a text file with a simple format, where each line stores the clock tick and the message that was generated.

However, as trace messages are usually outputted at each cycle, they can easily reach tens or hundreds or GiB, even for small applications. Their size can be reduced by generating traces only for precise simulation intervals (e.g. some hundreds of cycles before a clock tick where there was an error) and by compressing the output.

In the presented work, these traces were mostly used for debugging changes in the simulator. For example, a trace was generated each time the width mask for an instruction is created, including the values and widths for each operand element. Similarly, snapshots of the functional unit state were taken each cycle, recording how many units were used and what portion of their width is required. During the debugging stage, I usually used the System-call Emulation mode (*SE*), which allows for quicker testing.

These traces were also used in the results section to analyze small intervals of specific applications (see Section 6.2), to obtain a time-series view of the width and unit usage.

5.5 Power modelling

The McPAT power, area, and timing modelling framework [61] was used to estimate the impact of the proposed changes in the dynamic and leakage energy consumption of the Central Processing Unit (CPU) core's components, in particular in the Single Instruction Multiple Data (SIMD) unit. McPAT can model the microarchitectural components of out-of-order processor cores, such as functional units, issue queues, and register files. Two distinct out-of-order architecture models are available to be used with McPAT: one based on distributed reservation stations, where values are fetched directly from the Common Data Bus (CDB); and another where values are fetched from the physical register file.

McPAT is prepared to receive low-level configuration details and activity statistics through a flexible XML interface (see Fig. 5.4). This interface allows this framework to be used with a variety of performance simulators, in particular with gem5.

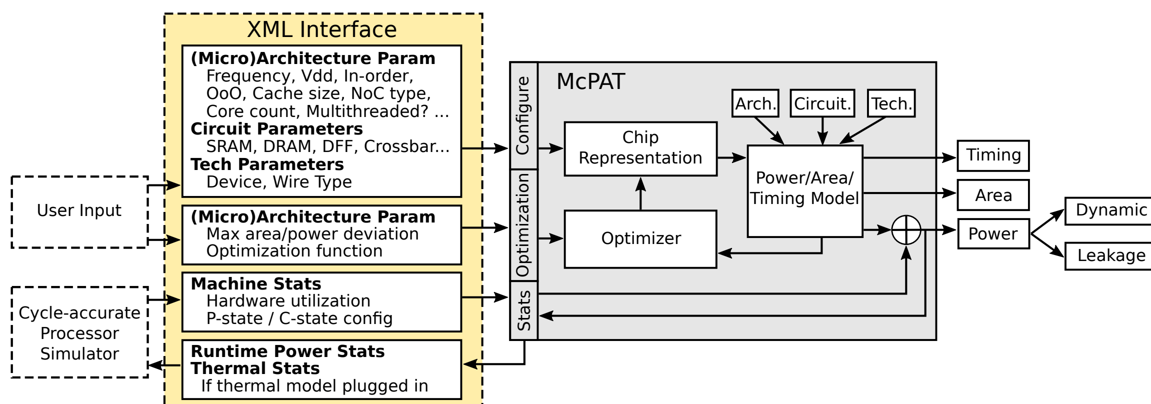


Figure 5.4: Block diagram of the McPAT framework [61]

To compute the timing, area, and power of the design, McPAT uses detailed models that simultaneously integrate these three parameters, to ensure that the results are consistent. These models are hierarchically organized in three levels with increasing detail: architectural, circuit, and technology. At the architectural level, the microprocessor is divided into major components, such as cores, caches, and clock. At the circuit level, the architecture components are mapped into wires, arrays, logic blocks, and clock networks. Finally, at the technology level, the physical parameters of devices and wires are computed, namely resistances, capacitances, and current densities.

The dynamic power consumption for each component is modelled using Equation 2.1 [61], where the supply voltage and frequency are specified in the configuration input, and the load capacitance is calculated using models of the corresponding circuit blocks. The activity factor is computed using the

access statistics for that component, which is coherent with the estimation model deduced in Section 3.4. The two parcels of the leakage current, sub-threshold and gate leakage currents (recall Equation 2.2), are computed based on the transistors width and device state, using detailed models for the CMOS technology. McPAT also models power-saving techniques, namely clock gating and power gating. Clock gating is modelled using the activity factor of each component and the corresponding clock network parameters, and power gating is computed using the circuit state statistics passed by a performance simulator.

In the conducted experimental work with gem5, an XML file was generated for each simulation run using a base template. The template is filled in with the configurations for the emulated core (e.g. the number of functional units and the fetch width) and with the run-time statistics obtained from the simulator (i.e. the performance counters). Apart from these parameters, the McPAT model was configured with a 28 nm technology process and an operating temperature of 340 K. The out-of-order model where the values are fetched from the physical register file was chosen, has it better resembles the Cortex-A76 architecture.

The number of accesses to the SIMD unit is one of the most relevant run-time metrics for this analysis, as each fused instruction counts as one less access to the unit, reducing the dynamic energy consumption. In turn, changes in the fetch/issue width and in the functional unit pool configurations significantly impact the leakage energy and circuit area results. Since the considered model does not include a SIMD pipeline, the register file and functional unit configurations and the accesses to these units were added to the corresponding floating-point parameters instead. However, twice the number of units and accesses had to be considered, since the prototyped ISA extension considers a vector length (i.e. 128-bit) that is twice the double-precision floating-point format.

5.6 Experimental workflow

The full experimental procedure using gem5 and McPAT is complex and error-prone, which is aggravated by the large number of benchmarks and configurations that were evaluated. Near 4000 individual simulation runs were necessary to obtain the final results. Hence, most of the process had to be automated using a collection of scripts[4] in Bash, AWK, and Python, as presented in Figure 5.5. These scripts include more than 10000 lines of code and automate tasks such as running gem5 with each configuration, parsing the inputs and outputs from McPAT, and analysing the final results.

Firstly, each full-system simulation in gem5 requires preparing a disk image for the emulated system ((A) in Fig. 5.5), similarly to a virtual machine. For this work, a disk image was prepared with a slightly modified version of Ubuntu 14.04 LTS for ARM, and the needed benchmarks and libraries were compiled

[4]These scripts are available at `https://github.com/miguelpinho/thesis-scripts`

natively using the QEMU emulator and the gcc 7.4.0 compiler toolchain. The full-system simulations also require preparing a boot script for each different benchmark (B), which runs the application with the desired arguments after the emulated system finishes booting. Similarly, each different configuration and parameter variation is specified using a Python module (C). These boot scripts and configuration files were generated automatically from templates.

The statistics and system description files created by gem5 (E) for each run are parsed into an XML file for input in McPAT (F), using a specific template for each CPU model (D). The statistics outputted by gem5 and McPAT (E, G), in text format, are parsed into a collection of CSV files with the needed results (H, I), e.g. the execution time and average power consumption. Finally, these results are analysed and summarised in plots and tables (J) using a collection of Jupyter notebooks and the *pandas* library [62]. The full workflow itself is automated using Bash and Python scripts (K) and parallelized using the very powerful GNU Parallel utility [63].

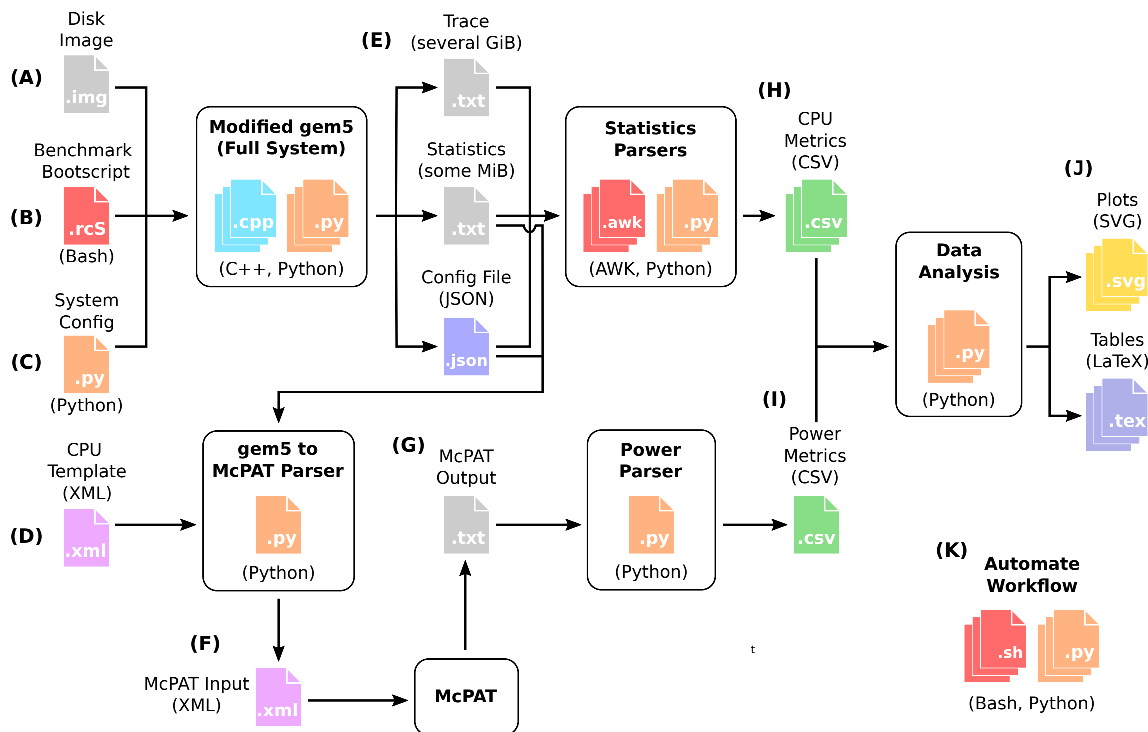


Figure 5.5: Description of the experimental workflow

5.7 Summary

This chapter presented a comprehensive description of the prototyping and evaluation environment that was used to obtain th profiling study presented in Chapter 3 and the evaluation results that will be

presented in Chapter 6. This description included the necessary context on which simulation tools, ISA, processor and power models were used and on how they were connected in the experimental workflow.

Hence, whereas the previous chapters had been mostly agnostic to the specific architecture and ISA, this chapter concretized the used prototyping environment, namely the CPU models and the ISA. However, it is worth noting that this evaluation represents only a proof of concept, since the proposed mechanisms could be equally applied to other architectures.

6

Experimental Evaluation

Contents

6.1 Evaluation methodology 64

6.2 Experimental results 67

6.3 Summary 75

This chapter presents and discusses the results of the conducted experimental evaluation to assess the impact of the proposed mechanism, in terms of energy consumption in the Single Instruction Multiple Data (SIMD) units and execution time. Several different configurations are evaluated, by varying the number of units and the mechanisms that are enabled, using several representative benchmarks and two processor models. The impact of changing the width-block parameter introduced in the proposed architecture is also explored.

6.1 Evaluation methodology

6.1.1 Considered configurations and benchmarks

In order to evaluate the impact of the proposed mechanisms in terms of the functional unit usage, energy consumption, and execution time, the **Cortex-A76** and the **High-performance** core models were simulated by considering several different execution modes, as labelled and presented in Table 6.1. In particular, the core models were simulated without any of the proposed mechanism enabled (`Original`), only with the operand packing enabled (`Packing`), and with packing and fusing enabled (`Fusing`). The proposed packing and fusing modes were also evaluated with an additional latency of one clock cycle in the

Table 6.1: Considered simulation modes: the `Original` mode is the baseline.

Label	Mode
Original	No mechanisms enabled.
Packing	Operation **packing** enabled.
Fusing	Operation **packing** and **fusing** enabled.
PackingExtraStage	Operation **packing** enabled, with an **extra latency** cycle.
FusingExtraStage	Operation **packing** and **fusing** enabled, with an **extra latency** cycle.

SIMD pipeline to implement these mechanisms (labelled PackingExtraStage and FusingExtraStage, respectively), to model the impact of this new stage in the resulting performance.

Each core model and proposed mechanism were evaluated with a different number of integer SIMD units, to measure the impact of power gating units (both in terms of energy savings and execution time). Each different simulated configuration is labelled as x-y-z, where $x \in \{A76, HP\}$ refers to the core, y is the mechanism mode, and $z \in \{1FU, 2FU, 3FU\}$ is the number of considered units. The A76-Original-2FU and HP-Original-3FU labels denote the baseline configurations for the Cortex-A76 and High-performance cores, respectively.

Several values of the width-block parameter were also evaluated to explore its impact on the energy savings: 1-bit, 2-bit, 4-bit, 8-bit, 16-bit, and 32-bit. The 1-bit width-block corresponds to the case where no width is wasted, and only the required bits are used for each operation. When not stated, the default width-block parameter for all configurations is 8 bits.

By following the same profiling procedure that was performed in Section 3.3, the complete list of benchmarks selected for this evaluation is presented in Table 6.2. The amax and asum kernels are very similar, and the results obtained for both were almost identical, so only the amax kernel will be presented.

Table 6.2: Selected benchmarks for this evaluation

Mini-Apps	Kernels		
	Algebra	Signal	Image
streamvbyte	sqnrm2	fft	hist
integerNN	amax	conv	img_integral
cartoon	gemv	median	canny
	gemm		erode

For the mini-apps, the considered input datasets correspond to real-world data. In particular, an already trained neural network was considered for integerNN, a large list of integers for streamvbyte, and a collection of large photos for cartoon. The same photos were used for all the image kernels, and for the conv and median kernels. For the algebra and the fft kernels, the input data was generated randomly, as detailed in Appendix A. In this appendix, the impact in the results of different input data distributions is also analysed.

For each benchmark, a Region Of Interest (ROI) in the application code was chosen, by discarding the code regions where the parameters and input files are read, and the output is generated. The results presented in the next section correspond only to this compute-intensive code section.

In addition to enabling and disabling the proposed mechanisms and varying the number of functional units, the impact of varying other design parameters was also evaluated. In particular, by also including two CPU models with different front-end and back-end widths (i.e. fetch and dispatch/issue width, respectively) it was possible to evaluate the impact in the fusing mechanism of increasing the instruction window size. Moreover, several different width-block sizes were experimented for the packing mechanism, with the only restriction of choosing powers of two: 1, 2, 4, 8 (default), 16, and 32 bits. As already explained, larger width-block values (coarser granularity) trade wasted operation width (and hence wasted energy) for less complex control logic and hardware overhead. A 1-bit width-block is the optimal case when strictly considering the minimum width required for each sub-operation. In contrast, a 32-bit block ignores the optimization mechanisms for vector element modes of 8, 16, and 32 bits.

6.1.2 Evaluated metrics

The most important metrics to evaluate the impact of the proposed narrow-width optimization mechanisms are the energy consumed in the SIMD unit, which is split into the dynamic and leakage energy, and the execution time in the ROI. These metrics were accurately benchmarked for each configuration and application through simulation with the gem5 and McPAT frameworks. The average power estimates, P, obtained from McPAT are used to compute the energy spent in that interval, E:

$$E = Pt, \tag{6.1}$$

where t is the execution time in the ROI estimated by gem5.

For a meaningful comparison of these metrics between different benchmarks, the energy and execution values are normalized against the reference configuration for that CPU model, which corresponds to the baseline configurations on Table 5.1 without any optimization mechanism enabled (the `Original` mode in Table 6.1). This normalization is crucial for a correct interpretation of energy values, as the obtained absolute values are highly dependent on the particular McPAT model, whereas the normalized values give a more meaningful relative comparison. For the normalized execution times, a value higher than 1 represents a slow-down compared to the baseline, while a lower corresponds to a speed-up.

For each configuration and benchmark, several simulation runs were performed using different (but identic) input sets, in order to reduce the bias from a particular run of a benchmark. The normalization of each metric was performed against the reference value obtained with the same input set. To summarize the results for each benchmark and configuration, the geometric average was used instead of

the arithmetic average. As these are normalized values, the arithmetic mean has no meaning, while the geometric mean is the more adequate measure [17, 64]. This is mainly due to the multiplicative property of the geometric mean, which states that the mean of the products equals the product of the means. Moreover, the geometric mean gives the same results regardless of which configuration is chosen as reference for the normalization. For example, the normalized energy for each benchmark and configuration (over n runs) is given by:

$$E = \sqrt[n]{\prod_{i=1}^{n} \frac{E_i}{E_{ref_i}}}, \tag{6.2}$$

where E_i is the energy estimated for each run i, and E_{ref_i} is the energy obtained for the reference configuration with the same input set.

6.2 Experimental results

6.2.1 Energy and performance impact of the proposed mechanisms

Figures 6.1 and 6.2 present the obtained energy saving results when evaluating the proposed packing and fusing mechanisms with the considered set of benchmarks, for the Cortex-A76 and High-performance cores, respectively. The estimated energy consumption is broken down in terms of its dynamic and leakage parcels. They also depict the execution time for each benchmark and configuration, normalized against the baseline configuration for each core model, namely `A76-Original-2FU` for the Cortex-A76 and `HP-Original-3FU` for the `HP` core.

When maintaining the original number of functional units, the packing mechanism reduces the total energy consumption in the SIMD unit by 21.5% and 20.6% (on average), for the `A76` and `HP` cores, respectively. These energy savings are slightly increased to 23.3% and 22.4%, respectively, when the fusing mechanism is also enabled. Since the number of switched-on units is maintained, these savings are mostly provided by a reduction in dynamic power, as idle and unused portions of the functional units are clock gated.

However, an interesting aspect is observed with the three benchmarks which use the SIMD unit more intensively: `gemm`, `median`, and `erode`. Since they are bounded by the SIMD units throughput, the fusing mechanism allows for an average speed-up of 26.9% and 29.0%, for the Cortex-A76 and `HP` cores, respectively, by allowing the execution of additional narrow-width instructions in the same units. For these benchmarks, this increase in the maximum throughput (using the same number of units), with a consequent reduction of the execution time, also contributes to a leakage energy decrease, resulting in a higher total energy reduction of 37.4% and 38.1% (which explains why the average energy savings with fusing are slightly higher).

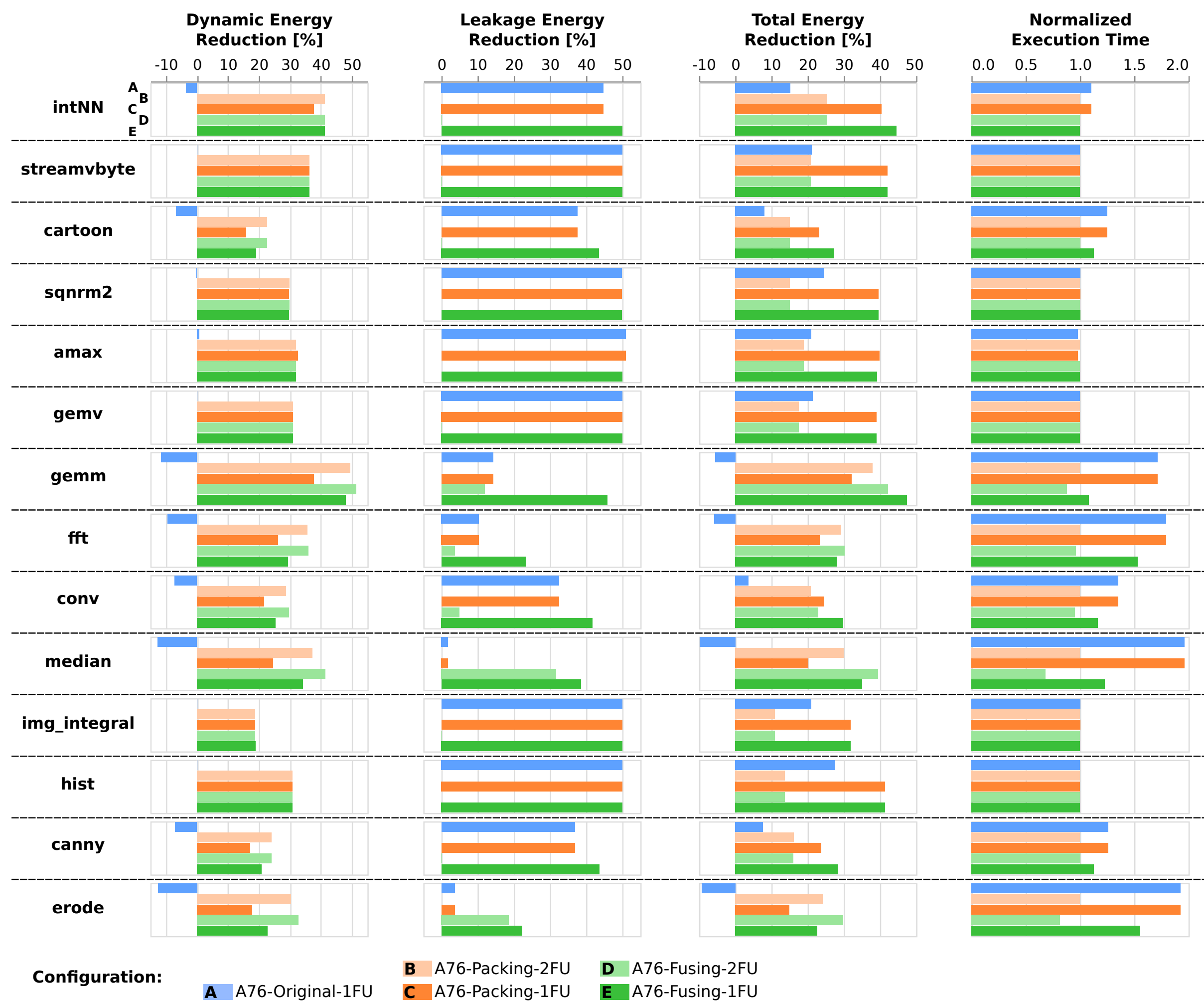


Figure 6.1: Energy reduction (broken down in the dynamic and leakage parcels) in the SIMD unit and normalized execution time for each benchmark with the Cortex-A76 core configurations. The baseline configuration, `A76-Original-2FU`, is used as a reference and is not included.

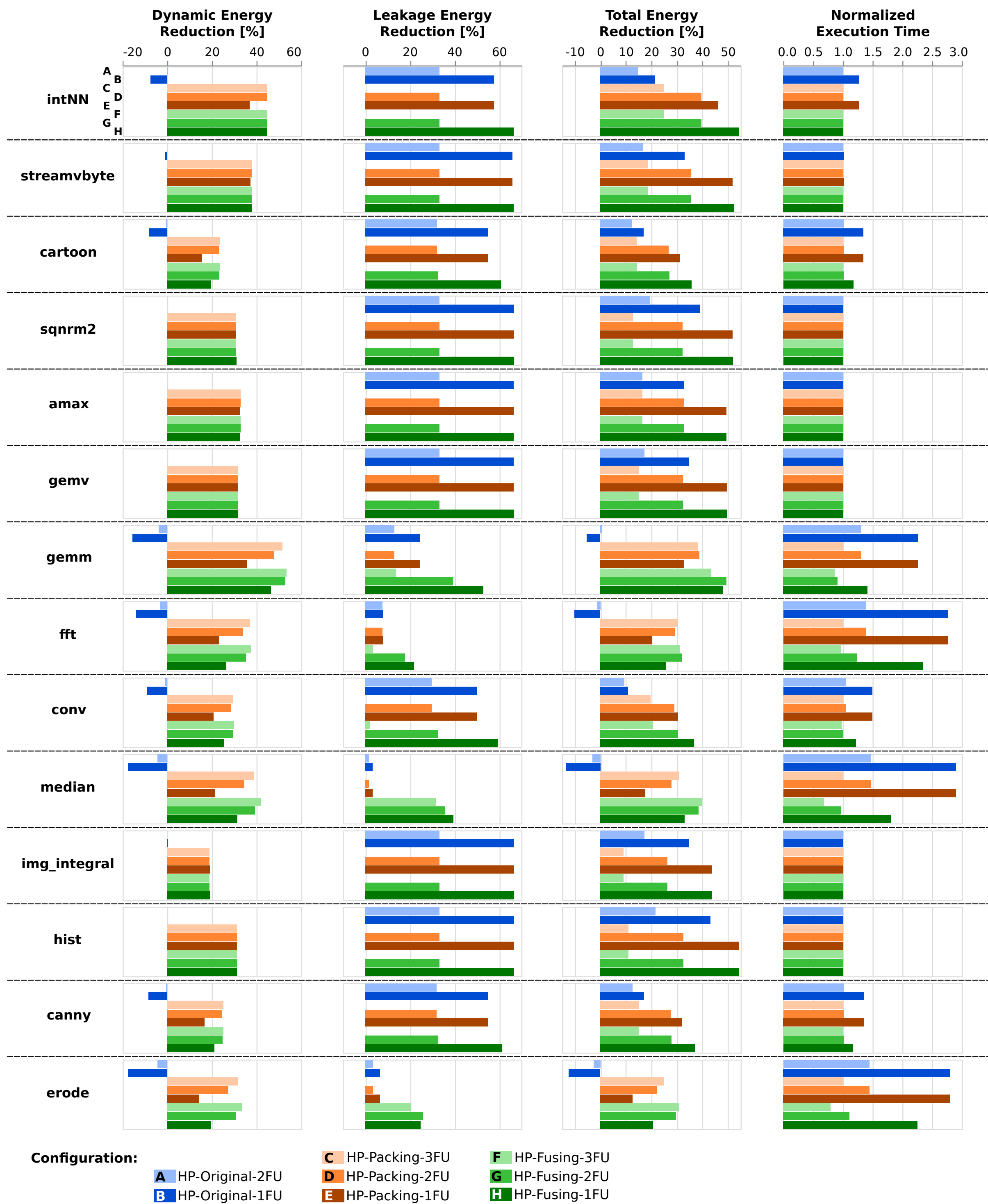


Figure 6.2: Energy reduction in the SIMD unit and normalized execution time for each benchmark with the High-performance core configurations. The baseline configuration, `HP-Original-3FU`, is used as a reference and is not included.

As it was discussed before, switching off functional units can further reduce the leakage power in the SIMD execution unit. In fact, by enabling the operation fusing mechanism, more units can be power gated (or removed), while maintaining a sufficient execution throughput. This is particularly interesting both to limit the number of units in the design (thus saving area), and to power off some units on a non-SIMD application phase (where most SIMD units can be turned off). In the later case, there is a costly delay when units need to be powered-on again, which is compensated by the proposed mechanisms. Moreover, the need for reactivating units can be even postponed. For the most intensive benchmarks (i.e. `gemm`, `median`, and `erode`), when one of the three SIMD units in the `HP` core is removed, the total energy consumption is reduced by 39.7%, while there is still an average speed-up of 1%. Without the fusing mechanism, removing one unit leads to a slow-down of 41%. For the remaining benchmarks, two units can be removed (in the `HP` core) with a slow-down of only 13%, on average, when it would be 23% without fusing, resulting in a total energy reduction in the SIMD engine of 45.3%.

Before concluding, it is worth noting that the presented results for the dynamic energy savings are significantly lower than the optimal values envisaged in Section 3.4, as the average savings are 34.3% when compared to 56.4%. However, these are still significant reductions, especially when also considering the savings in leakage power. Firstly, part of the difference can be explained by the width-block simplification in the width encoding step, as these results are for an 8-bit width-block. With an ideal (but unrealistic) 1-bit width-block, the average reduction would be 43.1% (see Subsection 6.2.2). Moreover, the initial analysis did not take into account other components of the dynamic power consumption of the execution unit, that are now modelled with the McPAT simulator, such as dissipation in interconnects, pipeline buffers, and the unit's scheduler. For the case of the `hist` kernel, with the highest discrepancy (94.5% compared to 31.2%), it is important to observe that the activity of the SIMD unit is quite low for this specific benchmark (see Fig. 6.3), so the weight of the power in these components should be very significant. Finally, as explained in Section 5.5, the SIMD and floating-point units are shared in the McPAT power model. Hence, the energy results presented in this chapter also include the execution of floating-point operations, which are not optimized in the proposed architecture.

Effectiveness of the fusing mechanism

Figure 6.3 depicts the active rate of the processor SIMD units during the execution of the considered benchmarks either when fusing is disabled (`Original`) and enabled (`Fusing`). In particular, it is represented by the percentage of the execution time (clock cycles) when 1, 2, or all 3 units are active. Naturally, there will be some fractions of time when none of the SIMD units are active, for example in parts of the code where there is no vectorization being exploited, or whenever the SIMD issue queues have already been emptied. The presented results show that the fusing mechanism significantly reduces the average number of SIMD units active in each cycle. The average usage of the second SIMD unit in

the Cortex-A76 core is reduced from 38.4% to 28.9% when the fusing mechanism is enabled. For the HP core, enabling the fusing mechanism reduces the usage of the third unit from 30.9% to 22.3%, and of the second unit from 37.8% to 29.8%. Hence, these results confirm that the operation fusing mechanism increases the average number of idle cycles of the SIMD units, reducing the activity factor (recall Equation 2.1) and consequently the dynamic energy consumption in these units. Moreover, as fewer units are needed (on average), some units can be power gated more frequently without significantly affecting the performance, which also reduces the leakage energy consumption.

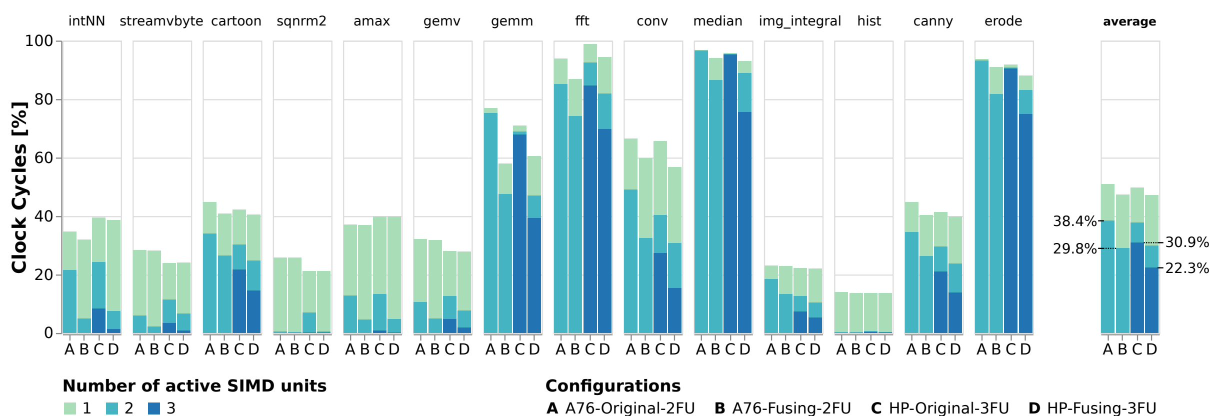


Figure 6.3: Active rate of the processor SIMD units during the execution of the considered benchmarks for the `Original` and `Fusing` configurations of the Cortex-A76 and High-performance cores. The empty portion of each bar in the plot corresponds to when all the units are idle.

As it can be observed in Figure 6.4, in most benchmarks, the fusing mechanism allows for a very significant fraction of the vector instructions to be issued and executed together with another instruction. In particular, the fusing effectiveness is very high for the `integerNN`, `gemm`, and `median` benchmarks, where (on average) 66.6% and 71.1% of the vector instructions are fused, for the Cortex-A76 and HP cores respectively.

The average fraction of fused instructions is slightly higher in the HP core (43.2% compared to 37.1% in the Cortex-A76, using the baseline number of SIMD units). In particular, for the `sqnrm2` benchmark, the fusing percentage goes from 1.2% in the Cortex-A76 core to 46.9% in HP. This increased effectiveness of the fusing mechanism in the wider HP model can be explained by the larger instruction window available, which allows more vector instructions to be waiting for execution in the same cycle, making fusing opportunities more likely. In fact, there is also a slight average increase in the fusing percentage when the number of units is reduced: from 37.1% to 39.3% in the Cortex-A76, and 43.2% to 45.6% in HP. A likely explanation is that the reduction in the execution throughput causes the issue queues to become fuller, so there is a higher chance that compatible instructions are waiting for execution in the same cycle.

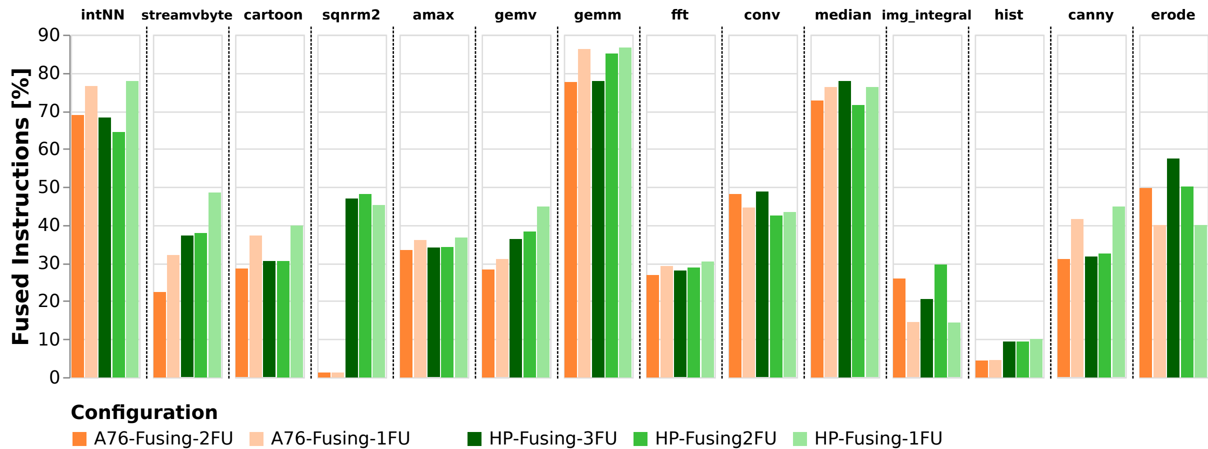


Figure 6.4: Percentage of issued instructions that are fused, for the `Fusing` configurations.

Impact of the extra pipeline stage

Figure 6.5 shows the impact of adding a new pipeline stage in the vector units to account for the additional steps in the packing and fusing mechanisms. As it can be observed, the energy savings are mostly unchanged, while this extra latency does not cause a significant slow-down for most benchmarks, except for the `fft` and `conv` benchmarks. In the `HP` core and only with the operand packing mechanism (`PackingExtraStage` configuration), there is a slow-down of 19.5% and 21.1% for these benchmarks, respectively. However, the fusing mechanism (`FusingExtraStage`) tends to slightly minimize the impact of the additional latency by allowing an increase in throughput. In these benchmarks, it reduces the slow-down to 18.6% and 20.5%.

Best configurations

Tables 6.3 and 6.4 report the best total energy reduction results obtained when varying the number of available units, for each operation mode and benchmark, and with the Cortex-A76 and `HP` cores, respectively. Only the configurations with a slow-down lower than or equal to 10% are considered, in order to limit the degradation in the execution time, unless none of the configurations satisfies this requisite (in which case the lowest execution time is reported). These tables confirm that the operation packing mechanism allows for significant energy savings in all benchmarks, with average consumption reductions of 31.5% and 40.1%, in the Cortex-A76 and `HP` cores, respectively. The operation fusing mechanism enables higher energy reductions of 34.9% and 43.2% (on average). In the particular example of the `integerNN` benchmark and the `HP` core, this is achieved by allowing an additional unit to be power gated without any performance penalty, which increases the energy reduction from 39.6% to 54.4%. In the `HP` core, the fusing mechanism allows a very significant reduction in execution time for the `median` and `erode`, with speed-ups of 46.8% and 25.9%, respectively.

Figure 6.5: Impact of adding a new pipeline stage in the vector unit in terms of the total energy reduction and normalized execution time, for the packing and fusing modes, and the Cortex-A76 and High-performance cores.

The presented results also further demonstrate that the impact of the additional latency cycle in the vector pipeline is not significant, in most cases. In the `HP` core, the average energy reduction in the SIMD unit only slight decreases to 37.4% and 40.9%, in the `PackingExtraStage` and `FusingExtraStage` modes, respectively. These tables also confirm that there is only a significant execution time increase in two benchmarks, `fft` and `conv`.

6.2.2 Design parameters exploration

The proposed architecture introduced the width-block design parameter, which significantly changes the energy reductions that can be obtained. A higher width-block simplifies the necessary control logic but causes more bit-width to be wasted in each computation. Figure 6.6 presents the energy reduction (broken down by dynamic and leakage components) for different width-block values, with the `HP-Fusing-3FU` configuration and using some example benchmarks and the overall average.

As it was predicted before, for most benchmarks, the dynamic energy savings significantly decrease

Table 6.3: Best energy savings obtained for each mode in the Cortex-A76 core, while not allowing a significant performance degradation.

	Original			Packing			Fusing			PackingExtraStage			FusingExtraStage		
	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time
integerNN	2	0.0	1.00	2	25.2	1.00	1	44.6	1.00	2	23.5	1.03	1	43.4	1.03
streamvbyte	1	21.2	1.00	1	42.1	1.00	1	42.1	1.00	1	41.8	1.00	1	41.9	1.00
cartoon	2	0.0	1.00	2	15.1	1.00	2	15.0	1.00	2	12.3	1.06	2	12.3	1.06
sqnrm2	1	24.5	1.00	1	39.6	1.00	1	39.6	1.00	1	40.4	0.99	1	40.4	0.99
amax	1	21.0	0.98	1	39.9	0.98	1	39.1	1.00	1	38.6	1.01	1	38.7	1.01
gemv	1	21.5	1.00	1	39.0	1.00	1	39.0	1.00	1	38.8	1.01	1	38.8	1.01
gemm	2	0.0	1.00	2	37.9	1.00	1	47.5	1.08	2	37.8	1.00	1	47.4	1.09
fft	2	0.0	1.00	2	29.2	1.00	2	30.2	0.96	2	28.9	1.01	2	29.6	0.98
conv	2	0.0	1.00	2	20.9	1.00	2	23.0	0.95	2	14.1	1.16	2	15.1	1.14
median	2	0.0	1.00	2	30.0	1.00	2	39.5	0.68	2	29.9	1.00	2	37.0	0.77
hist	1	27.7	1.00	1	41.4	1.00	1	41.4	1.00	1	41.1	1.00	1	41.1	1.00
img_integral	1	21.0	1.00	1	31.8	1.00	1	31.9	1.00	1	31.1	1.02	1	31.2	1.02
canny	2	0.0	1.00	2	16.2	1.00	2	16.1	1.00	2	13.0	1.06	2	13.0	1.06
erode	2	0.0	1.00	2	24.2	1.00	2	29.9	0.81	2	24.0	1.01	2	27.4	0.89
average	—	10.5	1.00	—	31.5	1.00	—	34.9	0.96	—	30.4	1.03	—	33.6	1.00

Table 6.4: Best energy savings obtained for each mode in the High-performance core, while not allowing a significant performance degradation.

	Original			Packing			Fusing			PackingExtraStage			FusingExtraStage		
	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time	Units	Saved Energy [%]	Norm. Time
integerNN	2	14.9	1.00	2	39.6	1.00	1	54.4	1.00	2	37.6	1.04	1	53.0	1.05
streamvbyte	1	33.0	1.02	1	51.8	1.02	1	52.4	1.00	1	51.5	1.03	1	52.0	1.01
cartoon	2	12.4	1.02	2	26.7	1.02	2	27.0	1.01	2	23.8	1.09	2	24.1	1.08
sqnrm2	1	38.9	1.00	1	51.8	1.00	1	51.9	1.00	1	52.0	1.00	1	52.0	1.00
amax	1	32.8	1.00	1	49.3	1.00	1	49.3	1.00	1	48.8	1.02	1	48.9	1.02
gemv	1	34.6	1.00	1	49.7	1.00	1	49.8	1.00	1	49.5	1.01	1	49.4	1.01
gemm	3	0.0	1.00	3	38.4	1.00	2	49.3	0.91	3	38.2	1.00	2	48.8	0.93
fft	3	0.0	1.00	3	30.4	1.00	3	31.3	0.96	3	25.2	1.21	3	25.3	1.21
conv	2	9.3	1.05	2	29.0	1.05	2	30.5	1.01	3	10.9	1.19	3	11.3	1.19
median	3	0.0	1.00	3	30.9	1.00	3	39.7	0.68	3	30.8	1.01	2	39.1	0.94
hist	1	43.2	1.00	1	54.2	1.00	1	54.2	1.00	1	54.1	1.00	1	54.1	1.00
img_integral	1	34.7	1.00	1	43.7	1.00	1	43.7	1.00	1	43.7	1.00	1	43.7	1.00
canny	2	12.6	1.02	2	27.6	1.02	2	27.9	1.01	3	10.6	1.08	2	24.6	1.09
erode	3	0.0	1.00	3	24.9	1.00	3	30.7	0.79	3	24.5	1.01	3	26.0	0.96
average	—	20.6	1.01	—	40.1	1.01	—	43.2	0.95	—	37.4	1.05	—	40.9	1.03

with the increase of the width-block, as more width is wasted in each operation. On average, a 1-bit block allows for a reduction in 43.1% of the dynamic energy, whereas with 8-bits it is slightly lower at 34.3%, and with 32-bits it is only 13.2%. The leakage energy is mostly unchanged in Fig. 6.6 with different width-blocks because the number of units is not changed (three SIMD units), and that is the most significant factor. However, for the `gemm` kernel, the fusing mechanism is also responsible for significant leakage energy savings, due to a noticeable reduction in execution time. As a result, for this benchmark, when the width-block is increased so much that its instructions are no longer optimized, there is no longer

a reduction in leakage energy or execution time. This happens with a 32-bit width-block because this kernel uses mostly 4×32-bit vector instructions, which are no longer optimized with this block size.

Hence, a larger width-block minimizes the overhead of implementing the new mechanisms but implies that more bits will be wasted in each operation, limiting the resulting energy savings. The analysis of Figure 6.6 indicates that an 8-bit width-block seems to be a good compromise for the considered processor architecture (i.e. ARM NEON). In fact, an 8-bit block not only allows for significant energy savings, but also corresponds to the smallest element size already supported in this vector extension, which simplifies adding hardware support for the proposed mechanisms (as explained in Subsection 4.1.3).

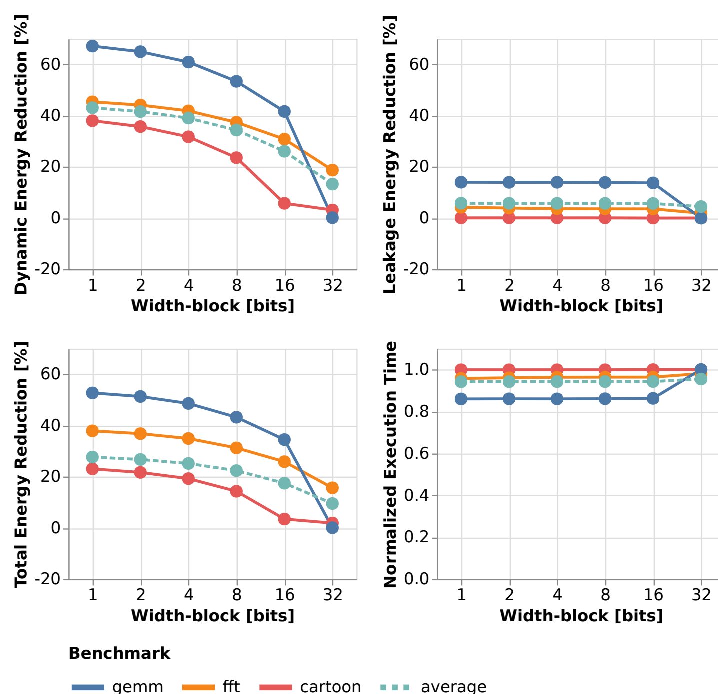


Figure 6.6: Comparison of energy savings (broken down into dynamic and leakage) and execution times with different width-block sizes, for the `HP-Fusing-3FU` configuration.

6.3 Summary

This chapter presented a comprehensive evaluation of the proposed mechanisms and architecture and showed that very relevant (dynamic and leakage) energy savings in the SIMD execution unit can be obtained while having a reduced or even negligible overhead. This increase in power efficiency is attained without significantly compromising the performance, or even with considerable speed-ups, for very computation-intensive kernels. After performing a design space exploration, by considering a prototype based on the Cortex-A76 microarchitecture, it was concluded that an 8-bit width-block seems to be the compromise for this parameter in this processor.

7

Conclusions and Future Work

Contents

7.1 Conclusions 76
7.2 Future Work 77

7.1 Conclusions

This thesis explored a previously ignored but highly relevant opportunity to exploit narrow-width vector computations, in the context of the increasingly predominant Single Instruction Multiple Data (SIMD) extensions in General Purpose Processors (GPPs). Several kernels and applications were profiled to support this claim, observing that although a wide data type is often assigned at compile-time for vector elements (e.g. 64 or 32-bit modes), for a large portion of vector operations, the actual values fit in a much narrower width (e.g. 16 or 8-bit). The performed preliminary evaluation showed that, on average, 56.4% of the total vector width is wasted in vector computations, when summing the unused portion of all vector elements, in the profiled benchmarks.

By following this preliminary assessment, two complementary mechanisms to exploit narrow-width in vector operations for energy efficiency were proposed: **operand packing**, and **operation fusing**. In these mechanisms, the elements in each vector operand are narrowed by discarding redundant sign bits and packed in a smaller width vector. Then, these narrower vector operands open up the opportunity to issue more than one instruction to the same functional unit (fusing), by using the remaining (available) width of the unit.

A modified superscalar out-of-order architecture is proposed, implementing these mechanisms dy-

namically (at run-time) in the execution engine, with a low hardware overhead and transparently to applications and libraries.

The prototyping and implementation of these mechanisms showed that they allow the usage of the SIMD units to be dynamically optimized to the size of the input data, reducing the dynamic and leakage energy consumption when combined with clock and power gating techniques. Furthermore, the fusing mechanism allows an adequate execution throughput to be maintained when the number of units is reduced. This is particularly advantageous in highly vectorized code regions and kernels, where the proposed technique allows reducing the number of active units with no significant performance losses, thus contributing to a reduction in energy consumption. Moreover, it is also advantageous in other code regions, since it amortizes the performance penalty caused by the delay in powering up gated SIMD units, thus facilitating the exploitation of power gating mechanisms.

The conducted experimental evaluation considered a prototyping architecture based on the ARM Cortex-A76 out-of-order core (with the ARM NEON vector extension). It demonstrated that energy savings in the SIMD unit as high as 54% could be achieved with the proposed implementation, while having an average slow-down of 3%, with even some slight speed-ups in compute-intensive kernels. Moreover, in a microarchitecture with three SIMD units, the propose mechanisms allow one or two units to be removed (e.g. by power gating) for most applications without a significant performance impact.

7.2 Future Work

The presented study proposed a lightweight scheme for packing narrow-width vector operands and operations, using a compact width mask for encoding the element size required by each lane. This initial proposal exploited narrow-width SIMD vector values only in the execution unit, but it might be very advantageous to extend this efficient vector representation to other portions of the datapath. In particular, the register file and data cache are responsible for a very significant fraction of the processor power consumption, so very relevant energy savings might be further obtained by optimizing these structures. Packed narrow vectors require less storage space in the register file and data cache, and unneeded portions could be gated to reduce energy consumption.

In the architecture that was proposed and evaluated in this study, the operands are packed before each instruction is executed, and the result is unpacked when writing-back to the register file (recall Fig. 4.10). Hence, the envisaged extension to this architecture would directly store the packed results in the register file, with the corresponding width mask, as is presented in Figure 7.1. The packed vector values would allow a large portion of the register file to be turned off, reducing its power consumption. The extra register space that would be required to store the width mask (e.g. 8 bits) would be largely outweighed by the savings enabled.

Moreover, if the width mask is directly fetched from the register file, it does not have to be computed in the operand fetch step ((A) in Fig. 7.1). Hence, as the vector operands are already packed, the new issuing stage (C) would be simplified, as it would only be required to realign the lanes between vector operands. Therefore, the critical path overhead in these stages would be reduced. During write-back (F), before storing the packed vector result, it would be advantageous to detect if the width of a lane was reduced by the operation and to adjust the vector and the mask accordingly.

In a limit situation, only immediate operands and memory values would need to be explicitly encoded and packed, but that would no longer interfere with the critical path of the execution engine. The immediate operands are available early in the decoding stage, and the values loaded from memory can be packed along the memory hierarchy (e.g. as early as the memory controller).

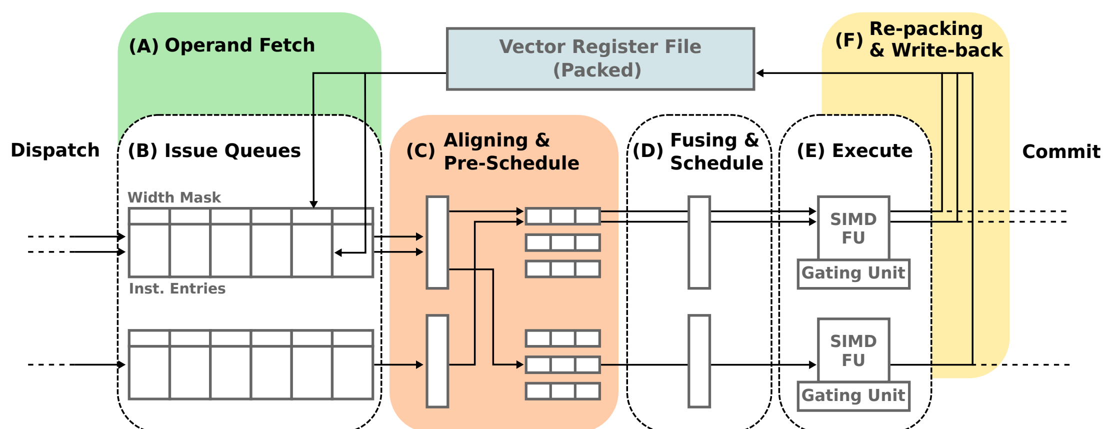


Figure 7.1: Possible extension to the proposed architecture, focusing on the changes to the execution engine (compare with Fig. 4.10). The packed vector is directly fetched from the register file with its width mask (A). Since the operand vectors are already packed, they only have to be re-aligned (C). During the write-back, the result vectors are re-packed only if the encoding of any lane has changed (F).

Hence, this architecture extension would not only increase the energy efficiency gains that can be obtained from vector packing but would also reduce the overhead of this approach (in particular, the critical path increase). However, some challenges would have to be addressed, namely those related to the different element modes that vector operations support. As was already explained, the same vector value has several different representations, depending on the vector mode. Hence, if there is a mismatch between the element size of the producer and consumer instructions, that packed vector can not be used directly and would have to be unpacked first. Moreover, in typical vector extensions, memory operations are agnostic in terms of element size, so they do not give any indication of the vector mode of the consumer instruction.

Another interesting extension that could be considered is to design and evaluate dynamic gating decision mechanisms, that also take into account the width of the current data when deciding when to turn off and on SIMD units. Tables 6.3 and 6.4 show that the number of vector units that allows for

the best compromise between energy savings and performance depends on the specific application. Moreover, the execution traces in Figure 3.6 show that the number of required units varies during the execution of an application, as code regions with intensive usage of vector computations alternate with non-SIMD phases, and as the width of the input data varies. By dynamically varying the number of units, better energy savings and performance compromises would be possible. However, the mechanism designed for deciding when to turn units on and off should not trigger these changes too often, due to the energy overhead and performance penalty of powering-up units.

Moreover, it would be interesting to prototype these architectural mechanisms with the novel ARM Scalable Vector Extension (ARM SVE) [9]. This extension not only brings larger vector sizes and new scatter/gather instructions but also introduces predication registers. Predicate registers are masks that select which elements should be computed in a vector. Elements that are not required for a given computation do not need to be fetched from the register file and can be exploited to further pack vector operands, by extending the proposed scheme. At the time this work was developed, the support for SVE in gem5 was still in development, and few libraries were optimized for this extension. However, this situation is gradually changing.

Finally, it would be interesting to explore similar run-time mechanisms to support dynamic precision in floating-point (scalar and vector) operations. However, this would bring new challenges, namely on how to shorten the floating-point representations without having significant precision losses during computations.

Bibliography

[1] M. Taylor, “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,” in *Design Automation Conference*. IEEE, 2012, pp. 1131–1136.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012.

[3] T. Mudge, “Power: a first-class architectural design constraint,” *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[4] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, “Leakage Current : Moore’s law meets static power,” *Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[5] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, “Reducing power in high-performance microprocessors,” in *Design Automation Conference*. ACM Press, 1998, pp. 732–737.

[6] M. Hassaballah, S. Omran, and Y. B. Mahdy, “A review of SIMD multimedia extensions and their usage in scientific and engineering applications,” *The Computer Journal*, vol. 51, no. 6, pp. 630–649, 2008.

[7] D. Y. Hong, S. Y. Fu, Y. P. Liu, J. J. Wu, and W. C. Hsu, “Exploiting longer SIMD lanes in dynamic binary translation,” in *International Conference on Parallel and Distributed Systems*. IEEE, 2016, pp. 853–860.

[8] A. Barredo, J. M. Cebrian, M. Valero, M. Casas, and M. Moreto, “Efficiency analysis of modern vector architectures: vector ALU sizes, core counts and clock frequencies,” *The Journal of Supercomputing*, pp. 1–20, 2019.

[9] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, “The ARM scalable vector extension,” *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[10] D. Habich, P. Damme, A. Ungethüm, and W. Lehner, "Make larger vector register sizes new challenges?: Lessons learned from the area of vectorized lightweight compression algorithms," in *Workshop on Testing Database Systems*. ACM Press, 2018, pp. 1–6.

[11] R. Kumar, A. Martínez, and A. González, "Dynamic selective devectorization for efficient power gating of SIMD units in a HW/SW Co-designed environment," in *International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2013, pp. 81–88.

[12] A. Youssef, M. Anis, and M. Elmasry, "Dynamic standby prediction for leakage tolerant microprocessor functional units," in *International Symposium on Microarchitecture*. IEEE, 2006, pp. 371–381.

[13] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *International Symposium on High-Performance Computing*. IEEE, 1999, pp. 13–22.

[14] G. H. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," in *International Symposium on Microarchitecture*. IEEE, 2002, pp. 395–405.

[15] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *International Symposium on Microarchitecture*. IEEE, 2004, pp. 304–315.

[16] G. Pokam, O. Rochecouste, A. Seznec, F. Bodin, G. Pokam, O. Rochecouste, A. Seznec, and F. Bodin, "Speculative software management of datapath-width for energy optimization," in *Conference on Languages, Compilers, and Tools*, vol. 39, no. 7. New York, New York, USA: ACM Press, 2004, p. 78.

[17] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach*, 6th ed., Elsevier, Ed., 2017.

[18] S. Carroll and W. Lin, "A queuing model for CPU functional unit and issue queue configuration," *Simulation Modelling Practice and Theory*, vol. 87, pp. 327–342, sep 2018.

[19] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[20] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge, "The limits of instruction level parallelism in SPEC95 applications," *ACM SIGARCH Computer Architecture News*, vol. 27, no. 1, pp. 31–34, 1999.

[21] J. J. Sharkey and D. V. Ponomarev, "Efficient instruction schedulers for SMT processors," in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2006. IEEE, 2006, pp. 293–303.

[22] I. Kuroda and T. Nishitani, "Multimedia processors," *Proceedings of the IEEE*, vol. 86, no. 6, pp. 1203–1221, jun 1998.

[23] R. Dennard, F. Gaensslen, W.-N. Yu, L. Rideout, E. Bassous, and A. Le Blanc, "Design of Ion-Implanted Small MOSFET ' S Dimensions with Very Small Physical Dimentions," *IEEE Journal of Solid State Circuits*, vol. 9, no. 5, pp. 257–268, 1974.

[24] B. Davari, R. H. Dennard, and G. G. Shahidi, "CMOS Scaling for high performance and Low Power next ten years," in *Proceedings of the IEEE*, 1995, pp. 595–606.

[25] S. Li and S. Mishra, "Optimizing power consumption in multicore smartphones," *Journal of Parallel and Distributed Computing*, vol. 95, pp. 124–137, 2016.

[26] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. De Supinski, "Practical resource management in power-constrained, high performance computing," in *International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 121–132.

[27] W. Lin, W. Wu, H. Wang, J. Z. Wang, and C. H. Hsu, "Experimental and quantitative analysis of server power model for cloud data centers," *Future Generation Computer Systems*, vol. 86, pp. 940–950, 2018.

[28] E. Ozen and A. Orailoglu, "The Return of Power Gating: Smart Leakage Energy Reductions in Modern Out-of-Order Processor Architectures," in *International Conference on Architecture of Computing Systems*. Springer, 2019, pp. 253–266.

[29] S. Rele, S. Pande, S. Onder, and R. Gupta, "Optimizing static power dissipation by functional units in superscalar processors," in *Conference on Compiler Construction*, vol. 2304. Springer, Berlin, Heidelberg, 2002, pp. 261–275.

[30] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Computing Surveys*, vol. 37, no. 3, pp. 195–237, 2005.

[31] J. Kathuria, M. Ayoubkhan, and A. Noor, "A Review of Clock Gating Techniques," *MIT International Journal of Electronics and Communication Engineering*, vol. 1, no. 2, pp. 106–114, 2011.

[32] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural Techniques for Power Gating of Execution Units," in *International Symposium on Low Power Electronics and Design*. ACM Press, 2004, pp. 32–37.

[33] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin, "Dynamic power gating with quality guarantees," in *International Symposium on Low Power Electronics and Design*, 2009, pp. 377–382.

[34] X. Wang and W. Zhang, "Execution units power-gating to improve energy efficiency of GPGPUs," in *International Conference on Internet of Things and International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing*. IEEE, 2019, pp. 711–718.

[35] B. Calhoun, J. Kao, and A. Chandrakasan, "Power Gating and Dynamic Voltage Scaling," in *Leakage in Nanometer CMOS Technologies*, 2006, pp. 41–75.

[36] L. Bolzani, A. Calimera, A. Macii, E. Macii, and M. Poncino, "Enabling concurrent clock and power gating in an industrial design flow," in *Design, Automation and Test in Europe*, 2009, pp. 334–339.

[37] J. Seomun, I. Shin, and Y. Shin, "Synthesis of active-mode power-gating circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 391–403, 2012.

[38] N. Wang, W. Zhong, S. Chen, Z. Ma, X. Ling, and Y. Zhu, "Power-gating-aware scheduling with effective hardware resources optimization," *Integration*, vol. 61, pp. 167–177, mar 2018.

[39] C. Cortes, H. Amano, and N. Yamasaki, "Break even time analysis using empirical overhead parameters for embedded systems on SOTB technology," *Conference on Design of Circuits and Integrated Systems*, vol. 2017-Novem, pp. 1–6, 2018.

[40] O. Rochecouste, G. Pokam, and A. Seznec, "A case for a complexity-effective, width-partitioned microarchitecture," *ACM Transactions on Architecture and Code Optimization*, vol. 3, no. 3, pp. 295–326, 2006.

[41] M. Özsoy, Y. O. Koçberber, M. Kayaalp, and O. Ergin, "Dynamic register file partitioning in superscalar microprocessors for energy efficiency," in *IEEE International Conference on Computer Design*. IEEE, oct 2010, pp. 515–520.

[42] C. Lomont, "Introduction to Intel advanced vector extensions," Tech. Rep., 2011.

[43] A. Waterman and K. Asanovi, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Document Version: 2.2," Tech. Rep., 2017.

[44] R. Canal, A. González, and J. E. Smith, "Software-controlled operand-gating," in *Symposium on Code Generation and Optimization, CGO*. IEEE, 2004, pp. 125–136.

[45] M. M. Islam and P. Stenstrom, "Characterization and exploitation of narrow-width loads: the narrow-width cache approach," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press, 2010, pp. 227–236.

[46] M. Sjalander and P. Larsson-Edefors, "Multiplication acceleration through twin precision," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 9, pp. 1233–1246, sep 2009.

[47] S. Balakrishnan and S. K. Nandy, "Arbitrary precision arithmetic - SIMD style," in *Conference on VLSI Design*. IEEE, 1998, pp. 128–132.

[48] P. S. Karthikeyan and P. S. Ranganathan, "More on arbitrary boundary packed arithmetic," in *International Conference on High Performance Computing*. IEEE, 1998, pp. 19–24.

[49] A. Danysh and D. Tan, "Architecture and implementation of a vector/SIMD multiply-accumulate unit," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 284–293, 2005.

[50] S. Krithivasan and M. J. Schulte, "Multiplier architectures for media processing," in *Asilomar Conference on Signals, Systems & Computers*, vol. 2, 2003, pp. 2193–2197.

[51] H. Libo, S. Li, D. Kui, and W. Zhiying, "A new architecture for multiple-precision floating-point multiply-add fused unit design," in *Symposium on Computer Arithmetic*, 2007, pp. 69–76.

[52] H. Zhang, D. Chen, and S. B. Ko, "Efficient Multiple-Precision Floating-Point Fused Multiply-Add with Mixed-Precision Support," *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1035–1048, 2019.

[53] ——, "New Flexible Multiple-Precision Multiply-Accumulate Unit for Deep Neural Network Training and Inference," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 26–38, 2020.

[54] P. Pujara and A. Aggarwal, "Restrictive compression techniques to increase level 1 cache capacity," in *IEEE International Conference on Computer Design*, vol. 2005. IEEE Comput. Soc, 2005, pp. 327–333.

[55] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[56] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs," pp. 24–36, 1995.

[57] G. Southern and J. Renau, "Analysis of PARSEC workload scalability," in *International Symposium on Performance Analysis of Systems and Software*. IEEE, 2016, pp. 133–142.

[58] J. M. Cebrián, M. Jahre, and L. Natvig, "Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks," in *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 66–75.

[59] D. Lemire, N. Kurz, and C. Rupp, "STREAM VBYTE: Faster byte-oriented integer compression," *Information Processing Letters*, vol. 130, pp. 1–6, 2018.

[60] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," in *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip*. IEEE, jul 2012, pp. 1–7.

[61] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture*. ACM Press, 2009, pp. 469–480.

[62] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[63] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011.

[64] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986.

Considered Benchmark Datasets

The integer input data for the `fft` and the algebra (i.e. `sqnrm2`, `amax`, `gemv`, and `gemm`) kernels was generated randomly by following tree different distributions:

- `8bit`, a discretized Normal distribution ($\mathcal{N}(0, 65^2)$) where around 95% of the values use 8-bits or less;
- `12bit`, a discretized Log-normal distribution ($Lognormal(4, 2.2^2)$) where around 95% of the values fit in 12-bits or less;
- `16bit`, a discretized Log-normal distribution ($Lognormal(5, 3.3^2)$) where around 95% of the values fit in 16-bits or less.

The Normal distribution is ubiquitous in scientific, media, and other applications, and in contrast with the Log-normal also generates negative values. However, its values tend to concentrate in lower widths (further supporting the narrow-width relevance), so to obtain datasets with a wider spread the Log-normal distribution was also used. All values from these datasets fit in a 32-bit integer data element, which is used in these applications. Figure A.1 presents the cumulative distribution function for the width of the values in these datasets, and Table A.1 presents some sample values from each dataset. The results presented for these kernels in Section 6.2 correspond to the `16bit` dataset, except for the `fft` kernel, where the `12bit` dataset is used.

The impact of these different input datasets in the results obtained for these kernels was also evaluated, and Figure A.2 depicts the total energy savings obtained for each dataset with these kernels, in the `HP-Fusing-3FU` configuration. As expected, higher energy savings are possible with narrower data, as there is more wasted width to exploit, but for most kernels (with the exception of `fft`) the difference is not very significant. This is because even in the `16bit` dataset most of the values use only half of the 32-bit data elements.

Table A.1: Sample values from each dataset

8bit	12bit	16bit
96	24	1
-78	2	89650
130	264	0
-62	79	208
-50	406	1
-42	5	18203
39	15	0
1	4	367
-183	135	1
-177	25	5225

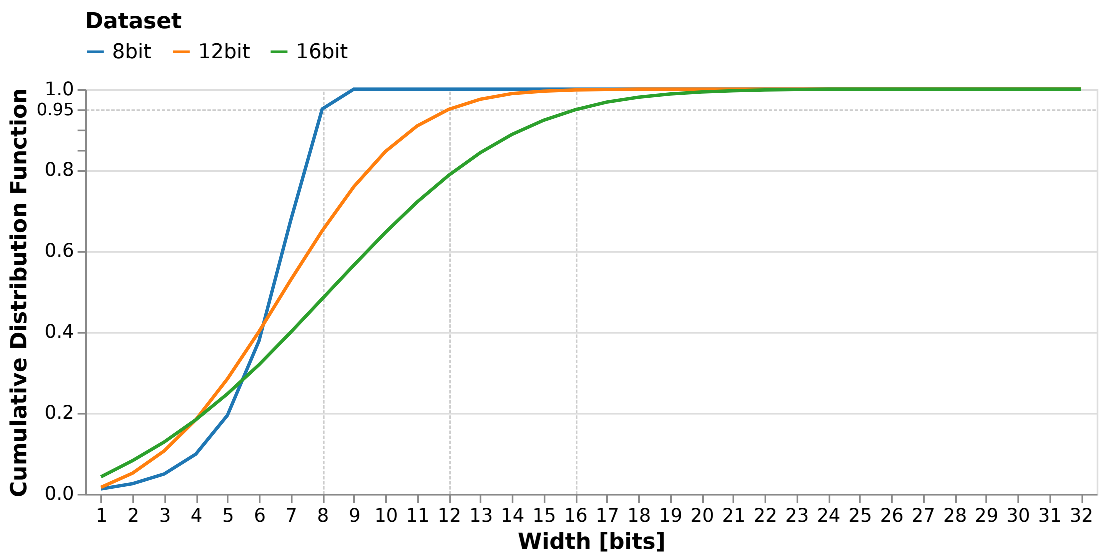


Figure A.1: Cumulative distribution function of the bit-width of the values in the random datasets

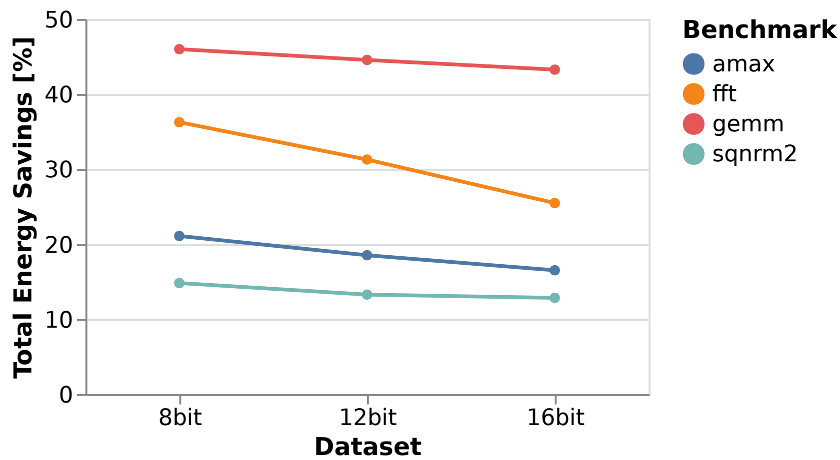


Figure A.2: Energy savings for different datasets, for the `HP-Fusing-3FU` configuration

B

ARMv8 NEON Instructions

Table B.1 presents a comprehensive list of the instructions included in the ARMv8 NEON vector extension. Instructions for which support was added (in gem5) to prototype the proposed mechanisms are marked in colour. The support for these instructions was gradually added as they appeared in the compiled binaries of the considered benchmarks, but it ended up covering a major fraction of the NEON extension, in particular when focusing on instructions with integer vector operands. A noteworthy exclusion is the instructions that perform multiplication (and accumulation) between a vector and a scalar element (e.g. MUL (by element)), but these were infrequent in the chosen applications.

Table B.1: List of ARMv8 NEON vector instructions, with modified and prototyped instructions marked in colour

Acronym	NEON Instruction	Mode
ADD	Add	Regular
SUB	Subtract	Regular
SABD	Signed Absolute Difference	Regular
UABD	Unsigned Absolute Difference	Regular
SABA	Signed Absolute Difference and Accumulate	Regular
UABA	Unsigned Absolute Difference and Accumulate	Regular
ADDP	Add Pairwise	Pairwise
ADDV	Add Across Vector	Across Vector
SQADD	Signed Saturating Add	Regular
UQADD	Unsigned Saturating Add	Regular
SQSUB	Signed Saturating Subtract	Regular
UQSUB	Unsigned Saturating Subtract	Regular
SHADD	Signed Halving Add	Regular
UHADD	Unsigned Halving Add	Regular
SRHADD	Signed Rounding Halving Add	Regular
URHADD	Unsigned Rounding Halving Add	Regular
SHSUB	Signed Halving Subtract	Regular
UHSUB	Unsigned Halving Subtract	Regular

Continued on next page

Acronym	NEON Instruction	Mode
SADDW	Signed Add Wide	Wide
UADDW	Unsigned Add Wide	Wide
SSUBW	Signed Subtract Wide	Wide
USUBW	Unsigned Subtract Wide	Wide
SADDL	Signed Add Long	Long
UADDL	Unsigned Add Long	Long
SSUBL	Signed Subtract Long	Long
USUBL	Unsigned Subtract Long	Long
ADDHN	Add High Narrow	Narrow
SABAL	Signed Absolute Difference Accumulate Long	Long
UABAL	Unsigned Absolute Difference Accumulate Long	Long
SUBHN	Subtract High Narrow	Narrow
SABDL	Signed Absolute Difference Long	Long
UABDL	Unsigned Absolute Difference Long	Long
SADDLP	Signed Add Long Pairwise	Pairwise
UADDLP	Unsigned Add Long Pairwise	Pairwise
SUQADD	Signed Saturating Accumulate of Unsigned Value	Regular
SQABS	Signed Saturating Absolute	Regular
SQNEG	Signed Saturating Negate	Regular
MUL	Multiply	Regular
MLA	Multiply-Add	Regular
MLS	Multiply-Subtract	Regular
MUL (by element)	Multiply by Element	Broadcast
MLA (by element)	Multiply-Add by Element	Broadcast
MLS (by element)	Multiply-Subtract by Element	Broadcast
SMULL	Signed Multiply Long	Long
UMULL	Unsigned Multiply Long	Long
SMLAL	Signed Multiply-Add Long	Long
SMLSL	Signed Multiply-Subtract Long	Long
UMLAL	Unsigned Multiply-Add Long	Long
UMLSL	Unsigned Multiply-Subtract Long	Long
SQDMULL	Signed Saturating Doubling Multiply Long	Long
SQDMLAL	Signed Saturating Doubling Multiply-Add Long	Long
SQDMLSL	Signed Saturating Doubling Multiply-Subtract Long	Long
SMULL (by element)	Signed Multiply Long by Element	Broadcast, Long
SMLAL (by element)	Signed Multiply-Add Long by Element	Broadcast, Long
SMLSL (by element)	Signed Multiply-Subtract Long by Element	Broadcast, Long
UMLAL (by element)	Unsigned Multiply-Add Long by Element	Broadcast, Long
UMLSL (by element)	Unsigned Multiply-Subtract Long by Element	Broadcast, Long
SQDMULL (by element)	Signed Saturating Doubling Multiply Long by Element	Broadcast, Long
SQDMLAL (by element)	Signed Saturating Doubling Multiply-Add Long by Element	Broadcast, Long
SQDMLSL (by element)	Signed Saturating Doubling Multiply-Subtract Long by Element	Broadcast, Long
PMUL	Polynomial Multiply	Regular
PMULL	Polynomial Multiply Long	Long
SMAX	Signed Maximum	Regular
UMAX	Unsigned Maximum	Regular

Continued on next page

Acronym	NEON Instruction	Mode
SMIN	Signed Minimum	Regular
UMIN	Unsigned Minimum	Regular
SMAXP	Signed Maximum Pairwise	Pairwise
UMAXP	Unsigned Maximum Pairwise	Pairwise
SMINP	Signed Minimum Pairwise	Pairwise
UMINP	Unsigned Minimum Pairwise	Pairwise
SMAXV	Signed Maximum Across Vector	Across Vector
UMAXV	Unsigned Maximum Across Vector	Across Vector
SMINV	Signed Minimum Across Vector	Across Vector
UMINV	Unsigned Minimum Across Vector	Across Vector
CMEQ	Compare Bitwise Equal	Regular
CMGE	Compare Signed Greater Than or Equal	Regular
CMGT	Compared Signed Greater Than	Regular
CMHI	Compare Unsigned Higher	Regular
CMHS	Compare Unsigned Higher or Same	Regular
CMEQ (zero)	Compare Bitwise Equal to Zero	Regular
CMGE (zero)	Compare Signed Greater Than or Equal to Zero	Regular
CMGT (zero)	Compare Signed Greater Than Zero	Regular
CMLE (zero)	Compare Signed Less Than or Equal to Zero	Regular
CMLT (zero)	Compare Signed Less Than Zero	Regular
CMTST	Compare Bitwise Test Bits Nonzero	Regular
ABS	Absolute	Regular
NEG	Negative	Regular
CLS	Count Leading Sign Bits	Regular
CLZ	Count Leading Zero Bits	Regular
EOR	Bitwise Exclusive OR	Regular
AND	Bitwise AND	Regular
BSL	Bitwise Select	Regular
BIC	Bitwise Bit Clear	Regular
BIT	Bitwise Insert if True	Regular
ORR	Bitwise Inclusive OR	Regular
BIF	Bitwise Insert if False	Regular
ORN	Bitwise Inclusive OR NOT	Regular
SHL	Shift Left	Regular
SSHR	Signed Shift Right	Regular
USHR	Unsigned Shift Right	Regular
SQSHLU	Signed Saturating Shift Left Unsigned	Regular
SQSHL	Signed Saturating Shift Left	Regular
UQSHL	Unsigned Saturating Shift Left	Regular
SRSRA	Signed Rounding Shift Right and Accumulate	Regular
RSHRN	Rounding Shift Right Narrow	Narrow
USHLL	Unsigned Absolute Difference	Regular
DUP (general)	Duplicate General-purpose Register to Vector	Element
DUP (element)	Duplicate Element	Element
UMOV	Unsigned Move Element to General-purpose Register	Element
SMOV	Signed Move Element to General-purpose Register	Element
INS (general)	Insert Vector Element from General-purpose Register	Element
INS (element)	Insert Vector Element	Element
MOVI	Move Immediate	Immediate

Continued on next page

Acronym	NEON Instruction	Mode
MVNI	Move Inverted Immediate	Immediate
SCVTF	Signed Integer Convert to Floating-point	Regular
UCVTF	Unsigned Integer Convert to Floating-point	Regular
UQXTN	Unsigned Saturating Extract Narrow	Narrow
SQXTN	Signed Saturating Extract Narrow	Narrow
SQXTUN	Signed Saturating Extract Unsigned Narrow	Narrow
XTN	Extract Narrow	Narrow
EXT	Extract	Special
ZIP	Zip Vectors	Special
UZP	Unzip Vectors	Special
TRN	Transpose	Special
TBL	Table Vector Lookup	Special
REV64	Reverse Doublewords	Special
REV32	Reverse Words	Special
REV16	Reverse Halfwords	Special
AES...	Cryptography AES	—
SHA...	Cryptography SHA	—
F...	Floating-point Instructions	—
LD...	Load Instructions	—
ST...	Store Instructions	—