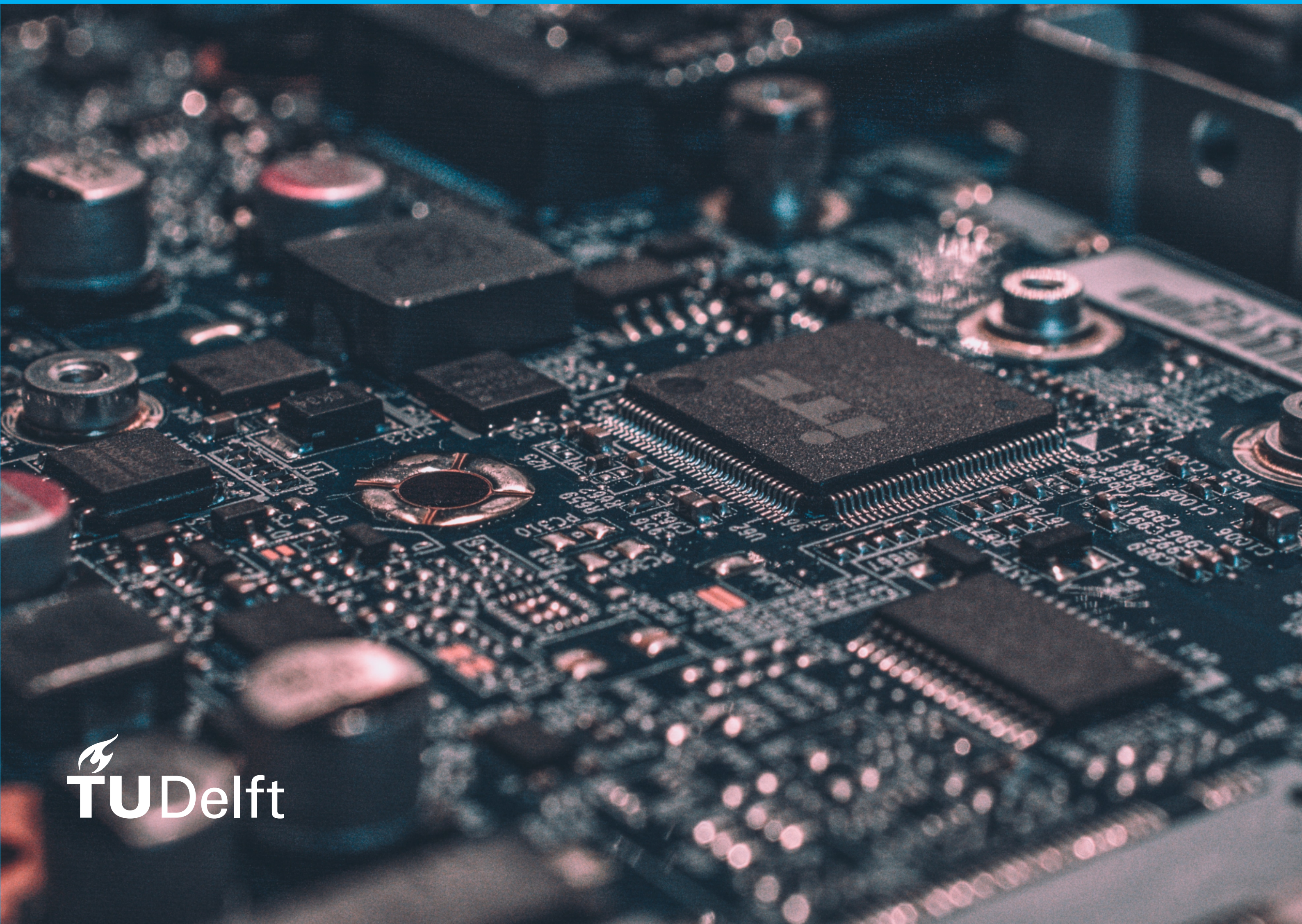
Implementation and Evaluation of Packed-SIMD Instructions for a RISC-V Processor

D.A.M. Koene



Implementation and Evaluation of Packed-SIMD Instructions for a RISC-V Processor

by

D.A.M. Koene

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 27 at 14:00.

Author:	D.A.M. Koene	
Student number:	4593561	
Project duration:	November 9, 2020 – August 27, 2021	
Thesis committee:	Dr. ir. J.S.S.M. Wong	TU Delft, supervisor
	Dr. ir. T.G.R.M. van Leuken	TU Delft
	Ir. M. Z. Zahedi	TU Delft

Cover image taken from [1].

Abstract

With the increase in the amount of data being gathered, the need for data processing is also rising. Furthermore, in addition to the proprietary ISAs that have been prevalent, the free and open RISC-V ISA has seen major interest. The modularity of the RISC-V ISA allows it to be extended with many instruction set extensions. One such extension that aids in the processing of large amounts of data is the P-extension, which introduces packed-SIMD instructions. In this thesis, the RISC-V based open-source CVA6 processor is extended to support the SIMD instructions defined by the P-extension. In order to do so, the 332 instructions of the P-extension are divided into subsets based on the type of instructions used by applications that make use of SIMD instructions and the hardware needed to implement those instructions. Due to time constraints 268, or 80.7%, of the total 332 instructions were implemented. However, this includes all the instructions that could be utilized by the used benchmarks. Therefore, the benchmark results show the full performance achievable by the P-extension. These 268 instructions make up the basic, MAC 8-bit, MAC 16-bit, and MAC 32-bit subsets. The ALU has been modified to operate in a SIMD manner on 8 8-bit, 4 16-bit, 2 32-bit, and 64-bit elements. Moreover, it has been extended to support new operations like data movement or reorganization instructions. Like the ALU, the multiplier has also been converted into a SIMD multiplier using a SIMD Baugh-Wooley scheme. Furthermore, the multiplier has been extended to also function as a MAC unit. The impact of these newly added SIMD instructions is tested in an ideal scenario of matrix multiplication as well as in a real-world machine learning application. In matrix multiplication, a speedup of up to 8.8x and 7.2x is seen for 8-bit and Q7 elements and 4.9x and 3.8x for 16-bit and Q15 elements when only using instructions from the basic subset. When also using the MAC instructions, the speedup increases to up to 12.3x and 12.6x for 8-bit and Q7 elements and 6.7x and 6.5x for 16-bit and Q15 elements. The real-world benchmark consists of an image recognition convolutional neural network based on the CIFAR-10 data set. In this benchmark, a speedup of 2.1x and 3.3x is obtained for respectively the basic subset and with MAC instructions. However, the additional hardware comes at a cost, specifically an increase of 5.0% LUT and 0.05% flip-flop usage for the basic subset or 7.2% increased LUT and 0.56% increased flip-flop usage with the basic and MAC subsets. While the additional hardware can have an impact on the maximum achievable clock frequency, the critical path remains in the FPU. The maximum achievable clock frequency is therefore not impacted and reaches 70 MHz on a Xilinx Kintex-7 FPGA.

Acknowledgements

This thesis is my last project at the TU Delft. It has not always been an easy journey but overall has been a very positive experience. This is of course helped by the professors and my fellow students. While studying and working on a thesis in these strange times has not always been ideal, I am confident that everyone at the TU Delft has done their best to make the experience as good as possible.

First of all, I would like to thank Stephan Wong and Mahdi Zahedi for their guidance during this project. The weekly meetings kept me on track and working in the right direction. Additionally, I would like to thank them along with René van Leuken for taking the time to be part of the thesis committee.

I would also like to thank my family for their support. Through the ups and downs they never stopped believing in me and no matter what problem I was facing, ensured me that everything will be all right.

D.A.M. Koene
Poeldijk, August 19, 2021

Contents

1 Introduction

The amount of data that is gathered is continuously increasing, with an expected 175 zettabytes (175 billion terabytes) of data expected to be generated in 2025 [2]. However, all this data is useless if nothing is done with it. Therefore, the need to process large amounts of data is also increasing. An efficient way to process a lot of data is by utilizing SIMD instructions. The acronym SIMD stands for single instruction, multiple data. In a basic sense this means that an operation like addition is applied to multiple pieces of data at the same time. By doing this, data parallelism can be exploited to perform multiple operations in parallel. This of course is not possible for all use cases, for example where on each piece of data a different operation needs to be performed. However, many applications can make good use of SIMD instructions like computer vision, audio and video processing, and machine learning.

In the past, SIMD was implemented by having multiple processors that would execute the same instruction in parallel as in the Connection Machine [3]. However, starting with HP in 1994, SIMD capable processors store multiple variables in a single register on which the operations are executed in parallel [4]. In this way, for example, 4 16-bit values can be stored in a 64-bit register and operations can be applied on all 4 values with a single instruction. This concept has been improved upon many times and many instruction set extensions have been proposed, for example SSE, AVX, AVX2, and AVX512 for x86 processors or NEON for ARM processors [5]. These extensions generally improve upon the previous version by providing more instructions or by providing wider registers in which more values can be stored and operated upon simultaneously.

The previously mentioned x86 and ARM instruction set architectures (ISAs) are proprietary. This means that they are not openly available and would require a license to use in a design. RISC-V on the other hand is a relatively new, open, and free ISA. With this instruction set, anyone is allowed to create and possibly sell products based on it without having to pay for a license. This openness also allows for extensibility, better security, and more competition. The RISC-V ISA has seen interest to be used for different use cases like in academia, internet of things (IoT), embedded systems, and even for use in space [6–8]. Apart from that, many large companies support the RISC-V ISA and are members of the RISC-V foundation like Google, IBM, NXP, Qualcomm, Samsung, and more [9].

Like the other instruction set architectures, RISC-V also has an extension for providing packed-SIMD instructions which is called the P-extension. This extension provides support for 8-bit, 16-bit, and 32-bit integer SIMD instructions. It is worth noting that, like many other RISC-V instruction set extensions, the extension has not yet been ratified and is at time of writing on version 0.9.5. Therefore, not many currently existing hardware implementations make use of this instruction set extension. Moreover, software support for this extension is also still in development. However, due to the importance of SIMD extensions in other ISAs and the potential speedup that SIMD instructions bring, the P-extension is worth investigating.

1.1. Problem Statement and Thesis Goals

The need for fast and efficient data processing and the recent popularity of the open RISC-V ISA lead to the research question of this thesis, which is stated as follows:

How can packed-SIMD instructions be added to a RISC-V processor and how does this affect the performance, energy usage, and area usage of the processor?

This research question gives rise to a number of goals that this thesis aims to achieve. Firstly, implementing packed-SIMD instructions on a RISC-V processor. To properly evaluate the impact of packed-SIMD instructions, they will need to be implemented in hardware. Therefore, packed-SIMD instructions will be added to an already existing RISC-V processor. Secondly, investigate and discuss the impact of adding these instructions in terms of performance, critical path, energy usage, and area usage. Adding SIMD instructions will likely result in increased performance for applications that can make use of these instructions. However, the additional hardware required to provide this speedup results in an increased area usage and could increase the critical path and thus potentially reduce the clock frequency of the processor.

1.2. Methodology

In order to be able to achieve these goals and to answer the research question, a number of steps need to be taken. These steps are shown below.

1. Research applications that can make use of SIMD instructions and which specific instructions they use.
   - This information can be used to select an appropriate benchmark to test the performance of the created design. Furthermore, by investigating what instructions are used, the importance of specific instructions can be gauged.
2. Divide the P-extension into subsets.
   - Due to the size of the P-extension and the fact that only a limited set of instructions is used by an application, it makes sense to divide the extension into subsets. This way less resources are wasted on implementing instructions that would remain unused. By using the information gained by investigating applications that use SIMD instructions, the subsets can be defined and prioritized properly.
3. Implement the P-extension to the RISC-V instruction set.
   (a) Implement a subset of the P-extension.
   (b) Implement the complete P-extension.
   - Due to the size of the P-extension it is possible that not enough time is available to implement all instructions. However, a functional design should be available at the end of this thesis to evaluate the performance. Therefore, first a subset with essential instructions should be implemented. Afterwards, the subsets should be implemented in the order determined in the previous step.
4. Verify that the implemented hardware functions correctly and thus adheres to the specification.
   - Results generated by a processor can only be used if there is confidence that the generated results are correct. Therefore, to give confidence in the created design, it should be thoroughly verified. The modified components as well as the whole design should be tested to make sure they adhere to the specification.
5. Test the performance of the improved processor.
   (a) Create a small test program to evaluate the performance under ideal circumstances.
   (b) Create or port a program suitable for a real-world application to evaluate real-world performance.
   - To show the impact of the newly introduced instructions, some benchmarks should be created and run. Usually some overhead is present when using SIMD instructions and not every part of the program can be accelerated with SIMD instructions. Therefore, a synthetic benchmark that can be accelerated almost completely with little overhead is created to show the ideal speedup that can be achieved with these instructions. As real-world programs are often less ideal, a realistic measure of the achievable speedup should also be provided by testing such a real-world application.

1.3. Thesis Synopsis

Following this, in Chapter 2, background information is provided. This includes discussions on the RISC-V ISA, SIMD instructions, SIMD instructions for the RISC-V ISA, hardware verification, as well as work related to this thesis. Subsequently, in Chapter 3 open-source applications using SIMD instructions are investigated. Furthermore, based on this, the instructions of the P-extension are divided into subsets and the subsets are prioritized. Thereafter, in Chapter 4 the hardware modifications to the core required to support the SIMD instructions are discussed. Additionally, the used compiler and the implementation of the benchmark programs are also detailed in this chapter. The results of these benchmarks along with the verification procedure of the processor and the results from synthesizing the design for an FPGA are presented in Chapter 5. Finally, the thesis is concluded in Chapter 6 by providing a summary, the main contributions of this thesis, and proposals for future work.

2 Background

In this chapter the background around this thesis is explained. First, some information is provided about the RISC-V instruction set architecture as well as the open-source processor that is used as a starting point for this project. Second, SIMD instructions will be detailed with regard to their history, their proposed implementation in RISC-V, and in what types of applications they can be used. Third, the process of testing and validating a hardware design is discussed. Finally, an overview of related work is presented.

2.1. RISC-V

RISC-V is a relatively new instruction set architecture (ISA) which has been introduced in 2010 and is created by the university of California, Berkeley [10]. As the name suggests, this ISA makes use of a reduced instruction set computer (RISC) design, which aims to have a small but optimized set of instructions. Other ISAs that use a RISC design include, for example, ARM or MIPS. This is in contrast to the widely used x86 ISA, which is a complete instruction set computer (CISC) design where the set of available instructions is larger. The RISC-V ISA has support for 32-bit and 64-bit designs as well as a draft version for 128-bit designs. Furthermore, a version of the 32-bit ISA is available as a draft meant for very small embedded processors that uses 16 instead of 32 general-purpose registers. Unlike most ISAs however, RISC-V is a free and open ISA. This means that anyone can design and potentially sell products based on this ISA without having to pay for a license.

Another major point in the design of RISC-V is its modularity. Apart from the different base instruction sets, the ISA can be extended with instruction set extensions that provide additional functionality. The base part of the ISA includes integer computational, control transfer, load and store, memory ordering, and hint instructions as well as environment calls and breakpoints. Even instructions like those for integer multiplication are not included in the base set of instructions but are instead part of the M-extension. The different base instruction sets can be seen in Table 2.1. A list of currently existing instruction set extensions can be seen in Table 2.2. It is worth noting that due to the fact that the RISC-V ISA is relatively new, some instruction set extensions have not yet been ratified and are thus still under development. Having such modularity within the ISA allows it to be used for a wide range of applications, from small power efficient embedded processors to high performance processors.

Apart from companies that make use of the RISC-V ISA in their proprietary designs, due to the free and open nature of the ISA, open-source processors have been created. Open-source processors have been designed

Table 2.1: Base instruction sets of RISC-V [10].

Base	Purpose	Version	Status
RVWMO	Weak memory ordering	2.0	Ratified
RV32I	32-bit integer instruction set	2.1	Ratified
RV64I	64-bit integer instruction set	2.1	Ratified
RV32E	32-bit reduced register integer instruction set	1.9	Draft
RV128I	128-bit integer instruction set	1.7	Draft

Table 2.2: RISC-V instruction set extensions [10].

Extension	Purpose	Version	Status
M	Integer multiplication and division	2.0	Ratified
A	Atomic instructions	2.1	Ratified
F	Single-precision floating-point	2.2	Ratified
D	Double-precision floating-point	2.2	Ratified
Q	Quad-precision floating-point	2.2	Ratified
C	Compressed instructions	2.0	Ratified
Counters	Performance counters and timers	2.0	Draft
L	Decimal floating-point	0.0	Draft
B	Bit manipulation	0.0	Draft
J	Dynamically translated languages	0.0	Draft
T	Transactional memory	0.0	Draft
P	Packed-SIMD	0.2	Draft
V	Vector operations	0.7	Draft
Zicsr	Control and status register	2.0	Ratified
Zifencei	Instruction-fetch fence	2.0	Ratified
Zam	Misaligned atomics	0.1	Draft
Ztso	Total store ordering	0.1	Frozen

not only by academia, but also by regular people and industry. This means that when one wants to make a product, they do not have to start designing a processor from scratch but can instead start from a well tested existing platform. This reduces implementation and verification time and therefore reduces the cost and complexity when designing a product using the RISC-V ISA.

2.1.1. Open-Source RISC-V CPUs

Due to the openness of the RISC-V ISA, many different designs of processors based on the ISA have been proposed. Moreover, just like the ISA, many of these designs are open-source. Since in this thesis packed-SIMD instructions will be added to a RISC-V processor to perform an evaluation, a processor needs to be chosen as a starting point. Important factors in deciding between these processors are a license permitting modification, support for the M instruction set extension, active development, good documentation, and well structured HDL code. The requirement for the M-extension is stated since the P-extension adds SIMD multiplication and multiply-accumulate (MAC) instructions. Comparing the performance of these against a core without a hardware multiplier would lead to unrealistic results. Moreover, a core that omits a hardware multiplier is likely not targeted at applications that would make use of SIMD instructions. A plus would be if the design makes use of RV64I rather than the RV32I base instruction set. Since the RV64I instruction set makes use of 64-bit registers rather than the 32-bit registers of RV32I, twice as many 8-, 16-, or 32-bit elements can fit in a single register. This allows the SIMD instructions to perform twice the number of operations with a single instruction, resulting in higher potential speedup.

As stated before, a lot of RISC-V processor designs are available, many are summarized in [11]. Due to the large amount of designs, a limited number are shown in Table 2.3. This list contains designs that originated from academia such as CVE32E40P and Rocket, from industry such as the SweRV cores, or from an individual with the PicoRV32 core. As shown in the table, most designs focus on the 32-bit RV32I base instruction set. Due to the preference for a 64-bit instruction set, the CVA6, Rocket, and BOOM cores remain. The Rocket and BOOM cores make use of the Chisel language which is based on the Scala programming language. Chisel is a relatively new hardware description language (HDL) of which the code can be converted to Verilog for synthesis for an FPGA or an ASIC. Use of the Chisel language allows circuit generation, which allows different versions of the cores to be generated. The CVA6 core on the other hand makes use of the more classical HDL SystemVerilog. As the CVA6 processor uses a more familiar HDL, has good documentation, and due to the reduced complexity by being less configurable, it was chosen as the starting point for this thesis.

Table 2.3: Overview of some of the open-source RISC-V processors, for instruction ordering the abbreviations in-order (IO) and out-of-order (OoO) are used

Name	ISA	Instruction Ordering	HDL
CVA6 [12]	RV64IMAFDC	IO issue, OoO write-back	SystemVerilog
CV32E40P [13]	RV32IMFC	IO	SystemVerilog
Ibex [14]	RV32EC or RV32IMCB	IO	SystemVerilog
Rocket [15]	RV32IMAFD or RV64IMAFD	IO	Chisel
BOOM [15]	RV64IMAFD	OoO	Chisel
PicoRV32 [16]	RV32E or RV32IMC	IO	Verilog
SweRV EH1 [17]	RV32IMC	IO	SystemVerilog
SweRV EH2 [18]	RV32IMAC	IO	SystemVerilog
SweRV EL2 [19]	RV32IMC	IO	SystemVerilog

2.1.2. CVA6

CVA6 is an open-source processor, written in SystemVerilog, based on the RV64I instruction set. The base instruction set has been expanded by the M, A, F, D and C instruction set extensions. Additionally, it has support for the RISC-V privileged extension. Originally this processor was designed and developed by ETH Zürich under the name Ariane [12], but control has been handed over to the OpenHW group and the name was changed to CVA6. The source code of the processor is available on their GitHub page and is licensed under a Solderpad hardware license [20]. This repository also includes scripts to simulate the design using QuestaSim or Verilator, to synthesize the design using Vivado, and directions to build a Linux image for the design. Furthermore, the CVA6 core has been implemented in silicon for different designs on various different fabrication processes like GlobalFoundries 22 nm, UMC 65 nm, and TSMC 65 nm [21].

A main differentiator of the CVA6 processor is that it is designed to run fully featured operating systems like Linux. In order to be able to efficiently run such an operating system, a translation lookaside buffer (TLB) and a page table walker (PTW) are needed [12]. Additionally, the privileged extension is needed to provide these operating systems with different privilege levels. These specifically include, in increasing privilege level, user/application, supervisor, and machine [22].

The structure of the CVA6 processor can be seen in Figure 2.1. Not shown in this image is the inclusion of a floating-point unit as a functional unit in the execution stage. The processor features a single issue, in-order design with 6 pipeline stages in a single core configuration. While the instructions are issued in-order by the issue stage, the scoreboard allows results of functional units to be written back out-of-order. This is beneficial when using instructions that take multiple cycles to complete like floating-point operations or load or store operations. Since an instruction issued after such a multi-cycle instruction, that uses a different functional unit such as the ALU, can complete without waiting for the multi-cycle instruction to complete. The processor includes separate configurable instruction and data caches. By default the core is configured to have a 4-way set-associative instruction cache of 16 KiB and an 8-way set-associative data cache of 32 KiB, both caches make use of a random replacement policy.

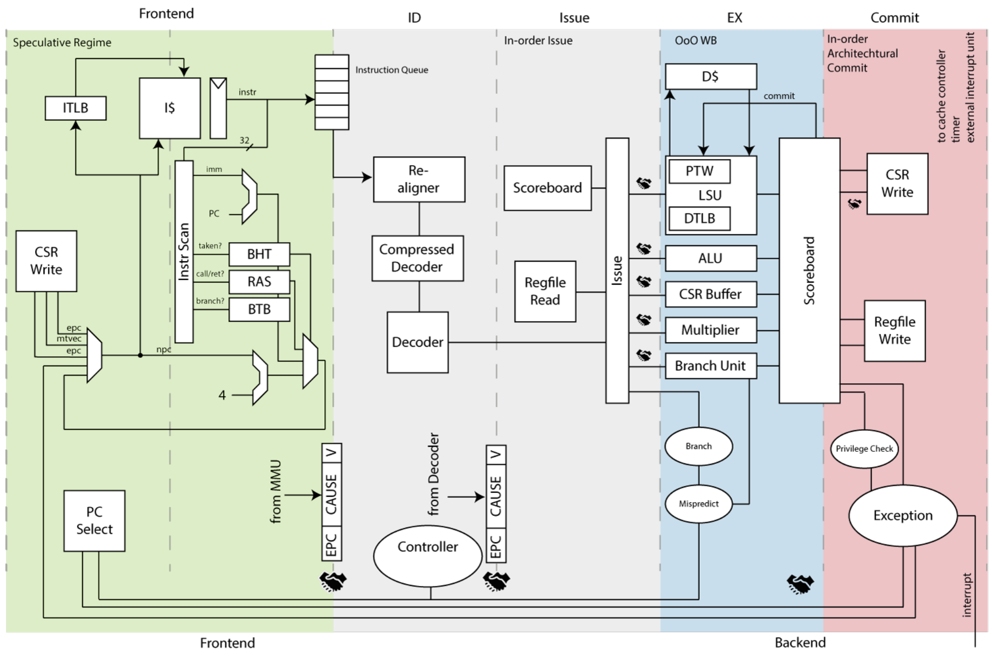


Figure 2.1: An overview of the structure of the CVA6 processor, adapted from [12].

2.2. SIMD Instructions

Flynn's taxonomy divides computing architectures into several types. Specifically, single instruction, single data (SISD), single instruction, multiple data (SIMD), multiple instruction, single data (MISD) and multiple instruction, multiple data (MIMD) [23]. As the names suggest, these classes are divided based on how many instructions on how many pieces of data are executed in parallel. SIMD specifically means that on a number of data items, the same operation is performed. An example of a SIMD add operation on 8-bit elements in a 64-bit register is illustrated in Figure 2.2. By performing operations like this, data level parallelism can be exploited [24]. Unfortunately, not all applications possess this data level parallelism and can thus not make efficient use of SIMD instructions. However, applications like audio and video processing, computer vision, and machine learning can be sped up significantly using this type of instructions.

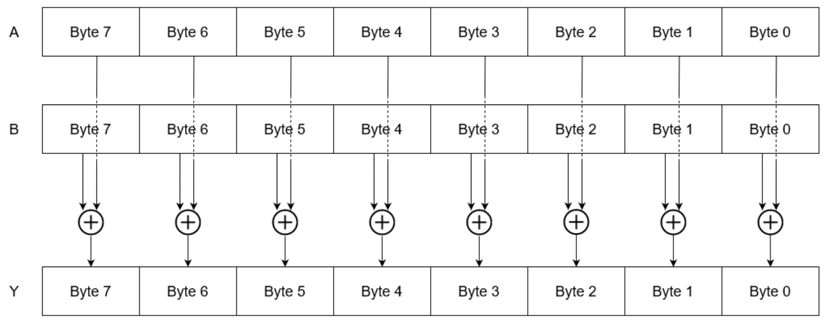


Figure 2.2: An example of a SIMD add operation on 8-bit elements in 64-bit registers.

2.2.1. History

Originally, SIMD computers consisted of a large number of processors that would perform the same instruction in parallel. The first of these was the ILLIAC IV which was designed in 1966 [25]. This system consisted of 256 processing elements which were divided in 4 arrays of 64 processing elements. All 64 processing elements in an array execute the same instruction. Moreover, each processing element consists of 4 64-bit registers and a 64-bit floating-point unit. This system was used by NASA, mainly for solving two-dimensional aerodynamic flow equations [26].

Previously, SIMD was mainly used for supercomputers since in their applications data level parallelism could be exploited. However, in order to support multimedia like video and audio playback on an entry-level computer, SIMD instructions were added to processors for consumer desktop computers [4]. However, for this use case, SIMD was not implemented by having multiple processors performing the same instruction. Rather, the trend has been to pack multiple data items in a wide register, for example 4 16-bit integers in a single 64-bit register. The first of these instruction set extensions came from HP for their PA-RISC ISA with the MAX instruction set extension in 1994 [4]. This was followed soon after by Intel's MMX for x86 and MDMX for MIPS in 1996 and AltiVec for PowerPC in 1998 [27].

Current desktop and server processors from Intel and AMD still feature support for SIMD instructions. Instruction set extensions for the x86 platform include MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, and AVX512 [5]. The most recent of which, AVX512, was introduced in 2015. The other main ISA used currently, ARM, also has support for SIMD instructions in the form of the Neon, Helium, and SVE instruction set extensions [28]. These instruction set extensions expand on the previous generations mainly by providing new instructions as well as more and wider registers.

2.2.2. Applications

In this section, some applications that can make use of SIMD instructions are discussed. Each application is discussed in their own paragraph.

Audio Processing
Recorded audio can be noisy due to wind blowing in the microphone or due to electrical noise. Therefore, filters are often applied to audio to clean up the signal. Furthermore, audio in the form of speech is often encoded with specific algorithms before it is sent over the network to reduce the required bandwidth. Due to the limited precision of audio samples and the data parallelism present, SIMD can be used effectively for processing audio [29, 30].

Video Processing
Video content is often compressed in formats like MPEG-4, HEVC, or others to reduce file sizes or the amount of bandwidth required to view a video. These algorithms apply transforms, like the discrete cosine transform, to the pixel data which allows for parallel execution [4]. Therefore, SIMD instructions can be used to exploit this parallelism and thus to accelerate the encoding and decoding of video data.

Computer Vision
Computer vision aims to allow computers to infer information from an image or video, like detecting what objects are in an image or inferring depth information. This can then, for example, be used to allow a robot to navigate a room without bumping into objects or walls. Since these algorithms process images or video, operations are performed on pixels which are often stored as 8-bit values. This allows computer vision algorithms to also make use of SIMD instructions to speed up their execution [31].

Machine Learning
Machine learning and specifically convolutional neural networks (CNNs) depend on convolutions to, for example, perform image recognition. These convolutions can be implemented by way of matrix multiplication which in their turn can be performed using multiply-accumulate instructions provided by SIMD extensions. Machine learning is usually quite computationally expensive and is therefore mostly done on servers. However, there has been interest to move this computation to lower power devices such as smartphones to reduce latency and bandwidth requirements. In order to reduce the computational load for smartphones, quantized neural networks (QNNs) are suggested [32]. Like regular neural networks, the training process of quantized neural networks often still makes use of floating-point data [33]. However, after training the network, the model is converted to use small fixed-point data types such as 8-bit integers. Due to these small data types, SIMD instructions can be used to speed up processing of neural networks. Support for these QNNs has already been introduced for the popular PyTorch and TensorFlow libraries [33, 34].

2.3. RISC-V P-Extension

For the RISC-V instruction set there is a proposal for introducing packed-SIMD instructions as an instruction set extension. These instructions are defined in the P-extension, which at time of writing is at version 0.9.5 [35]. This extension is in a draft state and therefore may see changes before the specification is ratified. It aims to improve performance of digital signal processing (DSP) algorithms as used in audio and video processing, computer vision, robotics, and more. The extension does this by introducing SIMD instructions for integer and fixed-point data types of 8-, 16-, and 32-bit sizes. Moreover, it adds some non-SIMD instructions mainly for fixed-point arithmetic.

Unlike many other SIMD instruction set extensions, the P-extension makes use of general purpose registers to perform the SIMD operations on, rather than dedicated registers. The reasoning behind this decision is that it is believed to be more efficient for embedded applications [36]. However, a downside to this is that the achievable parallelism is limited to the amount of elements that fit in a general purpose register. Other SIMD instruction set extensions often opt for 128-bit or wider registers with which a higher level of parallelism can be achieved.

A main omission from this extension, compared to other SIMD instruction set extensions, are floating-point operations. The P-extension does however have quite extensive support for fixed-point arithmetic. Fixed-point arithmetic is a way of utilizing integers as fractional numbers. By interpreting the bits as if the decimal point is placed in a fixed position somewhere in the integer, a fractional number can be represented [37]. This can also be seen as scaling the integer by $\frac{1}{2^n}$, where n is the bit position of the decimal point. A fixed-point addition is no different from a regular addition since $Nx + Ny = N(x + y)$. However, a fixed-point multiplication using a regular multiplier leads to $Nx \cdot Ny = N^2 xy$ and therefore needs to be divided by N to return to the same scale factor. Since the scale factor is taken as a power of 2, this can easily be done with a bit shift. The amount of bits used for the integer (i) and fractional (f) parts of a signed number, excluding the sign bit, are expressed in the P-extension as Qi.f. When the integer part is taken as zero, leading to a range between -1 and $1 - 2^{-f}$, it is represented as Qf. For example, a Q31 variable requires 32-bits and can hold a value between -1 and $1 - 2^{-31}$ with a resolution of 2^{-31}. Fixed-point data can also be unsigned, in which case the format is represented by the P-extension as Ii.f, of course without the need for a sign bit. Utilizing fixed-point arithmetic instead of floating-point mitigates the need for a complex and possibly slower floating-point unit.

2.3.1. Instructions Included in the P-Extension

The P-extension includes a total of 332 instructions, all with different functionality. In this section, the types of instructions included in the P-extension and their purpose will be discussed.

Add and Subtract

A number of different types of add and subtract instructions are provided by the P-extension. These consist of regular, halving, and saturating instructions. The halving instructions perform the addition or subtraction and subsequently divide the result by 2. This can be used to calculate a truncated average of two values. The saturating instructions, on the other hand, clip the result to either the most positive value or the most negative value when overflow would occur. Overflow can, for example, result in unrealistic colors when processing video or images, as a very bright pixel overflows to a dark pixel or vice versa. By using saturating operations these unwanted effects can be avoided [38].

Furthermore, add and sub instructions are included that add the odd elements and subtract the even elements of two registers, or the other way around. Instructions are also specified that perform this operation in a crossed manner so that the odd elements of the first input are added to the even elements of the second input or vice versa. The inclusion of these instructions allows the operation to be performed with a single instruction, rather than having to use multiple instructions to place the elements in the correct order for the operation. Moreover, these add and sub instructions are provided in the regular, halving, and saturating variants.

Bit Shift

The P-extension includes the typical shift instructions like arithmetic and logical right shift, and logical left shift operations. All instructions are available in variants where the shift amount is stored either in a register or is provided as an immediate value. Like for the add and subtract instructions, a saturating version of the left shift operation is provided. Furthermore, the right shift operation is available in a rounding variant. This rounding right shift instruction adds half a least significant bit (LSB) to the result of the shift operation. Without this all results are rounded down, leading to an error compared to a regular division. Errors like this can, for example, cause the accuracy of a neural network to decrease [39]. Finally, a shift instruction is provided that performs a saturating left shift operation when the shift amount is positive and a (rounding) right shift operation when the shift amount is negative.

Comparison
Instructions to compare elements in a SIMD manner are also included in the P-extension. These consist of less than, less than or equal, and equal instructions for both signed and unsigned data. When the condition for an element is true, that element in the destination register gets set to all ones and all zeros otherwise. Furthermore, min and max instructions are also provided in signed and unsigned variants. Instead of setting the result to all ones or all zeros, the result gets set to the smallest or largest element respectively.

Multiplication
The multiplication instructions present in the P-extension produce results twice the element size of the inputs. This makes sure that the result is provided without overflow. There are instructions to perform signed and unsigned multiplication. Furthermore, fixed-point multiplication instructions are provided that include the shift operation needed to return to the original scaling factor. Therefore, unlike the other multiplication instructions, the results of these fixed-point multiplications have the same element size as the inputs. Like with the add and subtract instructions, crossed variants are also provided that multiply the odd elements of one register with the even elements of the other register and vice versa.

Data Movement
In case the elements that need to be operated upon are not placed correctly in the register, some movement or reordering of data is necessary. The P-extension includes some instructions to help with this. Swapping the odd and even elements in a register is done using the `swap` instructions. Furthermore, the `sunpkd` and `zunpkd` instructions can be used to respectively sign extend and zero extend a combination of 8-bit elements to 16-bit elements. Finally, the `pkbb`, `pkbt`, `pktb`, and `pktt` instructions are provided for 16- and 32-bit elements to combine two inputs into a single register. The bb, bt, tb, and tt in the name of the instruction signify whether the bottom/even (b) elements or the top/odd (t) elements are taken from the first and second input register respectively.

Multiply-accumulate
Many different multiply-accumulate (MAC) instructions are provided by the P-extension. MAC instructions are provided that perform multiplication on 8-, 16-, and 32-bit elements. The accumulation is performed in either two 32-bit elements or a single 64-bit element and can contain additions and subtractions. Variants are again provided that perform saturating or crossed operations. MAC instructions can for example be used in matrix multiplication, where a sum of products needs to be computed. An example of when a crossed MAC instruction could be useful, is when multiplying complex numbers. A complex number can be represented by two consecutive elements in a register, where one represents the imaginary part and the other the real part. Since the imaginary part of the result is obtained by multiplying the real and imaginary parts, a crossed MAC instruction can directly compute this result. To obtain the real part of the result, a MAC instruction with subtraction can be used, as multiplying the imaginary parts results in a negative real number.

Sum of Absolute Differences
Another type of instruction that is present in the P-extension is the sum of absolute differences (SAD). This instruction computes the absolute differences of the 8-bit elements and accumulates the results. A variant of this instruction is provided to add this result to the value already in the destination register. SAD operations are often used in video processing [40].

Fixed-point Arithmetic
As mentioned previously, the P-extension focuses on fixed-point arithmetic to deal with fractional numbers rather than floating-point. Therefore, instructions are included to perform addition, subtraction, and

multiplication with Q15 and Q31 elements in a SISD manner. Moreover, SISD addition instructions are also provided for unsigned I16 and I32 elements. As stated before, multiplications for Q7 and Q15 elements are provided that include the shift operation needed to return to the original scaling factor. However, SIMD instructions are also provided for multiplications with an input of Q15 elements that produce Q31 elements. Furthermore, SIMD MAC instructions are present that add Q31 elements to the Q31 multiplication result of two Q15 elements.

Miscellaneous
Finally, some instructions are present in the P-extension that do not fit in a specific category. These instructions will be discussed in this section. Absolute value instructions are provided to compute the absolute value of 8-, 16-, and 32-bit elements. Clip instructions are also included which saturate the input to a range specified by the immediate value. This instruction is provided for 8-, 16-, and 32-bit elements and in signed and unsigned variants. Lastly, the P-extension contains instructions to count leading zeros (CLZ), leading ones (CLO), and leading redundant sign bits (CLRS). Like the other instructions, these are provided for 8-, 16-, and 32-bit elements.

2.3.2. P-Extension and V-Extension Comparison
As stated before, the RISC-V ISA has many instruction set extensions. For RISC-V there are two extensions that provide a way to introduce SIMD operations, the P-extension and the V-extension. This leads to the fact that many instructions in these extensions perform the same operations. The difference then lies mainly in the implementation. While the P-extension focuses on adding packed-SIMD or subword SIMD instructions [35], the V-extension adds support for vector-SIMD instructions [41].

Packed-SIMD instructions are meant to run on array processors while vector-SIMD instructions are meant to run on vector processors. An array processor has multiple processing elements that perform the operation simultaneously. This can also be achieved by a partitionable adder, which can perform for example a single 64-bit addition, two 32-bit additions, or four 16-bit additions. A vector processor, on the other hand, uses the same the same processing element multiple clock cycles to execute a single instruction [42]. However, a vector processor can still make use of a partitionable processing element or even multiple processing elements [43].

Like many other SIMD instruction set extensions, the V-extension uses dedicated registers to perform the SIMD operations on. There are 32 of these registers and while the size of these registers is not fixed, the base V-extension requires a minimum size of 128-bits. The P-extension, on the other hand, reuses the general purpose registers of the RISC-V ISA. In the case of the RV64I based CVA6 processor, there are 32 64-bit registers, while for an RV32I based processor they are only 32-bits wide. This means that for an RV32I based processor only half as many elements fit in a single register. Therefore, the parallelism achievable for an RV32I based processor is also half of what is possible with an RV64I based processor. Since the P-extension uses the general purpose registers, it can reuse the datapath of the scalar operations for the SIMD operations. When using the V-extension, the datapath is different and thus requires an additional ALU and multiplier.

Another main difference between the P- and V-extensions is how elements of different sizes are handled. In the case of the P-extension, instructions are specified for a specific element size. This for example leads to a separate add instruction for 8-, 16- and 32-bit elements. The V-extension, on the other hand, specifies the element size to operate on in a register independently from the instructions. Element sizes from 8- to 64-bits in size can be selected. However, encodings are reserved for elements up to 1024-bits in size. Selecting the element size this way allows the same code written for the V-extension to run on processors with varying vector register widths [44]. Additionally, the V-extension allows grouping multiple vector registers together in order to perform an operation on more elements with one instruction. This is not possible in the P-extension and is therefore limited to the amount of elements that fit in the 32- or 64-bit registers for RV32I and RV64I respectively.

As stated before, a major missing feature from the P-extension is the support for floating-point operations. This is supported in the V-extension for 16-, 32-, and 64-bit element sizes. However, the specification requires that the scalar operation on these sizes is also supported. An alternative method to deal with fractional numbers is fixed-point arithmetic. Both the P- and V-extensions have support for SIMD fixed-point operations.

While omitting the need for a floating-point unit (FPU) saves area and reduces the complexity of a design, floating-point arithmetic can not always be replaced by fixed-point arithmetic. The V-extension is therefore more widely applicable at the cost of additional complexity.

The vector processing nature of the V-extension allows easy scaling to achieve higher performance by adding additional lanes of processing elements (ALU, multiplier, and FPU). Furthermore, the variable length vector registers, with a minimum of 128-bits, allows more elements to be operated upon in parallel. However, these benefits come at the cost of the vector register file and duplicated functional units for the scalar and vector core. Furthermore, since the P-extension does not support floating-point operations, even more area is saved. Apart from the additional area needed by the vector register file, extra overhead is introduced in the form of moving data between register files. Everything in consideration, leads to the conclusion that the P-extension is meant for smaller and lower power designs while the V-extension aims at higher performance and scalability.

2.4. Hardware Verification

While designing hardware it is easy to overlook a scenario in which the hardware does not perform the intended operation as expected. However, when using a processor, or any other piece of hardware, it is essential that there is confidence that the hardware provides the correct results. This makes it important to verify the correctness of the hardware so that no bugs remain in the design. Moreover, since large costs are connected to fabricating a chip, as many bugs as possible should be found and fixed before manufacturing. A well known example of a bug present in a processor that was sold to consumers is the floating-point divide bug in Intel's Pentium processors. Because of this bug, the processors had to be recalled. This recall cost Intel an estimated 475 million dollars in 1994 [45].

The process of making sure that the designed system functions as expected is called functional verification [46]. Functional verification usually involves multiple types of tests to give confidence in the correct functioning of the system. Specifically, manually created tests, randomly generated tests, legacy tests or commercial test suites, and complex applications [47]. While manually generated tests are good at targeting specific scenarios, something that might be overlooked in the design might also be overlooked when creating tests. This might lead to the bug remaining undetected. However, a randomly generated test could trigger this bug so that it can be found and subsequently fixed. On the other hand, a specific scenario is easily targeted by a manually created test while the chance for a randomly generated test to trigger such a scenario is small. It is therefore important that the design is tested with multiple types of tests.

While working on a new feature or when fixing some bug, it is possible that inadvertently something else in the design is broken. In order to detect these kind of bugs, a regression suite is run [46]. This is a set of tests that is run when changes are made to the design. As these tests verify the correctness of the previously created operations, it can be ensured that the change did not break the existing functionality in the process.

In a hardware implementation, there is often a hierarchy to the design. For example, the processor consists of the pipeline stages, a pipeline stage might have an ALU, a multiplier, a load/store unit, and so on. It is important to perform verification on multiple of these levels of the design [46]. Testing a functional unit on its own makes it easier to verify that the module performs its operations correctly. When subsequently verification is performed on the whole core, it can be assumed that bugs found at this stage are likely caused by the interaction between the modules rather than the modules themselves.

2.5. Related Work

In this section, work related to this thesis is discussed.

2.5.1. Ara Vector Coprocessor

Ara is a vector coprocessor based on draft version 0.5 of the RISC-V V-extension [43]. The vector coprocessor is integrated with the scalar CVA6 core. In the instruction decoder, vector instructions are partially decoded to recognize that they should be sent to Ara. The instructions are further decoded in the Ara front end, which is added to the execute stage of the pipeline of the CVA6 core. The front end then sends the instructions to Ara itself. Ara consists of a sequencer, a vector load/store unit, a slide unit, and a number of lanes. The lanes themselves contain the functional units that operate on 64-bit wide data, specifically an ALU, a multiplier, and an FPU. All functional units are capable of performing operations in a SIMD manner. The ALU and multiplier can produce a single 64-bit, 2 32-bit, 4 16-bit, or 8 8-bit results. Similarly, the FPU can produce one double precision, two single precision, or four half precision results. The amount of lanes that are used in the design is configurable.

Even though this work also presents a way of adding SIMD instructions to the CVA6 processor, the implementation is very different. In this case it is done by adding a coprocessor based on the V-extension rather than the P-extension, which would make use of the ALU and multiplier of the main processor. Furthermore, the FPU consumes about 37% of the area consumed by the lane while the vector register file takes 35%. This results in quite a large design, where a single lane takes about as much area as the whole CVA6 core.

2.5.2. AndesCore AX25MP

The AndesCore AX25MP is a 64-bit RISC-V processor designed by Andes Technology that also has support for the P-extension [48]. The initial proposal for the P-extension is based on the AndeStar V3 DSP ISA which was also developed by Andes Technology [36]. Apart from the P-extension, the AX25MP processor also supports the M, A, C, F, D, and N extensions of the RISC-V ISA. Moreover, it supports multi-core configurations of up to four cores. While this processor does have support for the P-extension, its design is proprietary. This means that the design is not openly available and requires a license to use. Unfortunately, as this is a proprietary design, not much information about it is publicly available.

2.5.3. PULP Core with DSP Extensions

Another RISC-V processor with support for SIMD instructions is presented in [49]. This processor is based on the parallel ultra low power (PULP) platform and adds support for hardware loops, fixed-point arithmetic, saturated arithmetic, and SIMD operations. Since a 32-bit, RV32IMC, processor is used and 8- and 16-bit element sizes are supported, 4 and 2 elements can respectively be operated on in parallel. This work predates the publication of the specification of the P-extension. Therefore, while there are similarities with the P-extension, this design uses its own custom instruction set extension. Furthermore, the P-extension does not add support for hardware loops and does include SIMD operations for 32-bit data types on 64-bit processors.

2.6. Conclusion

In this chapter, the background of this thesis has been explained. The RISC-V ISA has been introduced along with its extensions and the openly available processors based on this ISA. From these available processors, the CVA6 processor was chosen as a starting point for this thesis. This is a 64-bit in-order RISC-V processor with 6 pipeline stages that supports the M, A, F, D, and C instruction set extensions. SIMD instructions are explained and its history is discussed. Moreover, audio processing, video processing, computer vision, and machine learning are shown as applications that can make use of SIMD instructions. Furthermore, the P-extension is introduced as a way to realize SIMD instructions in the RISC-V ISA and the instructions included in this extension are discussed. Additionally, the P-extension is compared against the V-extension which introduces vector-SIMD instructions rather than packed-SIMD instructions. From this comparison it is concluded that the P-extension is meant for small and low power designs, while the V-extension is meant for scalability and high performance. The process of hardware verification is investigated and is found to usually consist of manually created tests, randomly generated tests, commercial test suites, and complex applications. Finally, the related work is discussed in the form of the Ara vector coprocessor, Andes Technology's AX25MP processor, and a PULP Core with DSP Extensions.

3 Subset Definition

Version 0.9.2 of the P-extension defines a total of 332 instructions. This makes the P-extension the second largest currently proposed extension for the RISC-V ISA. While ideally all instructions would be implemented, due to time constraints this was not feasible. To make sure that a functional design is present at the end of the project, the essential SIMD instructions need to be identified so that they can be implemented first. Furthermore, the implementation order of the other instructions needs to be prioritized. The P-extension does provide some subsets, however these are specifically meant for 32-bit RISC-V processors. This allows a 32-bit processor to not include the instructions that, for 32-bit processors, work on register pairs so that 64-bit inputs and outputs can be processed. Furthermore, a subset is defined for SIMD instructions that operate on 32-bit elements which is only included for 64-bit processors. It is essential that after implementation of the basic subset of the extension, it is possible to run a functional application in a SIMD manner. To identify which instructions are necessary to run such a functional application, the usage of SIMD instructions in open-source applications was studied. This is discussed first, followed by the definition of the subsets along with the implementation priority of these subsets.

3.1. Application Investigation

As the specific SIMD instructions and element sizes used differs for each application, it is important to support at least a set of essential instructions such that the application can run in a SIMD manner. Furthermore, some instructions, like MAC instructions, are not essential to run the algorithm, as they can be replaced with separate multiply and add instructions. They can however have a large impact on the performance of the algorithm. To determine the importance of specific SIMD instructions, three open-source applications in different fields were investigated. Specifically, these three fields consist of machine learning, video encoding, and computer vision. Based on this information, the P-extension can be divided into subsets which subsequently can be prioritized.

3.1.1. Machine Learning

As stated before, neural networks operating on 8-bit fixed-point data types have been added to the popular PyTorch and TensorFlow machine learning libraries. The source code of QNNPACK, the quantized neural network back end of PyTorch, was investigated to find which SSE2 SIMD instructions were used [50]. The SSE2 instruction set extension provides SIMD instructions for the x86 instruction set. In order to implement the neural network, a number of add, subtract, shift, data movement, compare, and multiply instructions are used operating on element sizes varying from 8- to 64-bits. For compactness, these instructions, along with the element size they operate upon, have been summarized in Table 3.1. Apart from the instructions shown in the table, the neural network also makes use of a MAC instruction. This instruction multiplies the 16-bit elements of two registers and adds these 32-bit multiplication results in pairs. Since SSE2 makes use of 128-bit registers, 8 multiplications and 4 additions are performed with this single instruction.

3.1.2. Video Encoding

One of the reasons to include SIMD instructions on consumer processors was to accelerate video processing [51]. While the video coding format has changed from, for example, MPEG-2 to something newer like

HEVC/H.265, SIMD instructions are still used to accelerate the algorithms. To investigate the SIMD instructions used for video encoding, the Kvazaar HEVC/H.265 video encoder was analyzed [52]. This video encoder makes use of SIMD instructions from the AVX2 instruction set extension. Again, the basic SIMD instructions used by this application are gathered in Table 3.1. Like the quantized neural network, the video encoder makes use of a MAC instruction that adds the results of a 16-bit SIMD multiplication in pairs to form 32-bit results. Similarly, a MAC instruction is used that performs 8-bit multiplication and adds the 16-bit results together in pairs. However, this is a saturating addition, which means that if overflow would occur, the result is saturated to the largest positive or negative value. Furthermore, it makes use of instructions for counting leading zeros, trailing zeros, and ones in a 32-bit value. Finally, an instruction to compute the absolute value is used as well as an instruction to compute the sum of absolute differences (SAD), where the absolute differences of two vectors with 8-bit elements are accumulated.

3.1.3. Computer Vision

The last application that was investigated is computer vision. Specifically, the SGBM and BM algorithms of OpenCV, which is an open-source computer vision library [53]. The SGBM and BM algorithms are algorithms that can be used to estimate depth from the images of a stereo camera. This could, for example, be used by a robot to navigate through a room without bumping into objects or walls. These algorithms are implemented in OpenCV using SSE SIMD instructions. As with the other applications, the basic SIMD instructions used in these algorithms are gathered in Table 3.1. Apart from these basic instructions, no other special SIMD instructions are used.

Table 3.1: Overview of the basic SIMD instructions used in the investigated applications, the numbers denote the element size the instruction operates on.

		Machine learning	Video encoding	Computer vision
Add and subtract	Add	16, 32, 64	8, 16, 32, 64	
	Saturated add	16	16	16, 32
	Horizontal add	32	16, 32	
	Sub	16, 32	8, 16, 32	
	Saturated sub		8	8, 16
	Horizontal sub		16	
Shift	Shift left logical	16, 32, 64	16, 32, 64	
	Shift right logical	16, 32, 64	16, 32, 64, 128	16
	Shift right arithmetic	32	32	
Data movement	Pack signed	32 to 16	32 to 16	16 to 8
	Pack unsigned	16 to 8	16 to 8	16 to 8
	Unpack	8, 16, 32, 64	8, 16, 32, 64	
	Shuffle single input	16, 32	8, 16, 32, 64	
	Shuffle two inputs		8, 16, 32, 64	16, 32
	Zero extend elements		8 to 16, 8 to 32	8 to 16, 16 to 32
	Sign extend elements		8 to 16, 16 to 32	16 to 32
	Insert element	16		
Comparison	Min unsigned	8	16	8
	Min signed		8	16, 32
	Max unsigned	8		8
	Max signed		8, 16	16
	Greater than	32	8, 16, 32	16, 32
	Less than			16
	Equal		8, 16, 32	16, 32
Multiplication	Multiply	16, 32	16, 32	16

3.2. Definition of Subsets

In order to bring some modularity to the specification of the P-extension, the instructions were divided into subsets. This is already done for some other instruction set extensions for the RISC-V ISA, like the B-extension. The reasoning for the decision to split the B-extension in subsets is that some instructions are not as useful in every application [54]. The 43 instructions of the B-extension are therefore split into 5 subsets. As seen from the investigated applications, like for the B-extension, the usefulness of some instructions of the P-extension also depends on the application. By implementing only the instructions of the subsets that are needed for an application saves on the area usage of a processor. Furthermore, by implementing subsets rather than just the specific instructions that are needed, a measure of compatibility is maintained between processors. This compatibility is especially useful when developing software. Additionally, the chair of the task group of the P-extension has stated to be in favor of dividing the extension into subsets after the extension has been ratified [55]. However, at time of writing, these subsets have not yet been defined and therefore a division into subsets is proposed in this thesis.

The definition of the subsets of the P-extension was done based on the already existing categories of instructions defined by the specification. However, one exception was made in the "miscellaneous instructions" subsection. In this subsection, 32-bit SIMD clip and bit counting instructions are defined as well as two 8-bit sum of absolute differences (SAD) instructions. The SAD instructions require specific hardware changes to perform the accumulation and for reading the initial value of the accumulator an extra read port is needed on the register file. Because of this and due to the largely different functionality, the "miscellaneous instructions" subsection was split in two. The subsections that are defined in the P-extension can be seen in Table 3.2.

As seen in the previous section, instructions for SIMD addition, subtraction, multiplication, bit shift, and comparison are used in all investigated applications. These kinds of instructions, along with supporting instructions like minimum or clip instructions, are mainly provided in the "SIMD Data Processing Instructions" section of the specification for 8-bit and 16-bit elements. Therefore, the whole of section 1 is included in the basic subset. The same instructions for 32-bit elements are spread out over different sections of the specification, but are still widely used in the investigated applications. These instructions are therefore also included in the basic subset, which comprises subsections 2.6.a, 5.1 through 5.3, and 5.5. Since providing the correct placement of elements in registers is one of the major overhead costs of SIMD processing, instructions that move data around are essential to keep this overhead as low as possible. These instructions are therefore also included in the basic subset. This includes instructions to combine elements of two registers (16-bit and 32-bit packing instructions) and instructions to sign or zero extend elements of a register to a larger element size (8-bit unpacking instructions). These instructions are present in subsections 1.11, 2.1, and 5.9. Additionally, the "overflow/saturation status manipulation instructions" of subsection 4.4 are included in the basic subset. These instructions are aliases to instructions from the Zicsr extension that read or clear the `vxsat` control and status register (CSR). This register gets set when a saturating instruction detects overflow and saturates the result. In these instructions, the result is saturated to the maximum positive or maximum negative value instead of writing the overflowed result to the destination register. Summarizing, the basic subset includes instructions for SIMD add, subtract, multiply, comparison, and some data movement for 8-, 16-, and 32-bit element sizes. Specifically, subsections 1.1 through 1.11, 2.1, 2.6a, 4.4, 5.1 through 5.3, 5.5, and 5.9 are included in the basic subset, as shown in Table 3.2. Moreover, this subset contains 189 of the total 332 instructions in the P-extension which makes it 57% of the complete extension.

A major part of the instructions that have not yet been assigned to a subset consist of MAC instructions and 8-bit SAD instructions. Due to the different functionality and required hardware, the 8-bit SAD instructions in subsection 2.6.b are defined as a separate subset. The MAC instructions, on the other hand, get subdivided based on the element size of the multiplication operation. Therefore, the MAC instructions get divided into the subsets MAC 8-bit, MAC 16-bit, and MAC 32-bit. Specifically, MAC 8-bit consists of subsection 2.7, MAC 16-bit consists of subsections 2.4, 2.5, and 3.3, and MAC 32-bit consists of subsections 2.2, 2.3, 3.2, 5.6, and 5.7. Finally, the instructions that remain are a combination of instructions specifically for fixed-point data and instructions that implement some of the new types of operations, like saturating addition, in non-SIMD versions for 32- and 64-bit data. Therefore, these instructions were divided in a fixed-point and a non-SIMD 32- and 64-bit subset. The fixed-point subset is made up of subsections 4.1, 4.2, and 5.4 while the non-SIMD 32- and 64-bit subset contains subsections 3.1, 4.3, 4.5, and 5.8. An overview of the subsets that were defined and the instructions included in these subsets are represented in Table 3.2.

The implementation order of these subsets has been prioritized to make sure that the most important instructions are implemented first. Naturally, the basic subset is implemented first, as that subset contains the essential instructions to make use of SIMD. Following that, the subsets are implemented based on how often they are used in the investigated applications. As both the machine learning and video encoding applications make use of MAC instructions, the MAC subsets were decided to be implemented after the basic subset. The MAC subsets were then prioritized from 8-bit to 16-bit and finally 32-bit due to the higher parallelism that can be achieved with smaller element sizes. After that comes the SAD 8-bit subset, as SAD instructions are used by the video encoder. Finally, the fixed-point subset was prioritized over the non-SIMD subset. The reason for this is that the focus of this thesis is on SIMD instructions. This leads to the implementation order as shown in the enumeration below.

1. Basic
2. MAC 8-bit
3. MAC 16-bit
4. MAC 32-bit
5. SAD 8-bit
6. Fixed-point
7. Non-SIMD 32- and 64-bit

Table 3.2: Division of the P-extension into subsets, where #Insns denotes the number of instructions.

Section	Subsection	#Insns	Basic	MAC 8-bit	MAC 16-bit	MAC 32-bit	SAD 8-bit	Fixed-point	Non-SIMD 32- and 64-bit
1. SIMD Data Processing Instructions	1.1. 16-bit Addition & Subtraction Instructions	30	X						
	1.2. 8-bit Addition & Subtraction Instructions	10	X						
	1.3. 16-bit Shift Instructions	14	X						
	1.4. 8-bit Shift Instructions	14	X						
	1.5. 16-bit Compare Instructions	5	X						
	1.6. 8-bit Compare Instructions	5	X						
	1.7. 16-bit Multiply Instructions	6	X						
	1.8. 8-bit Multiply Instructions	6	X						
	1.9. 16-bit Misc Instructions	11	X						
	1.10. 8-bit Misc Instructions	11	X						
	1.11. 8-bit Unpacking Instructions	10	X						
2. Partial-SIMD Data Processing Instructions	2.1. 16-bit Packing Instructions	4	X						
	2.2. Most Significant Word "32x32" Multiply & Add Instructions	8				X			
	2.3. Most Significant Word "32x16" Multiply & Add Instructions	16				X			
	2.4. Signed 16-bit Multiply with 32-bit Add/Subtract Instructions	18			X				
	2.5. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions	1			X				
	2.6.a. Miscellaneous Instructions a (32-bit)	5	X						
	2.6.b. Miscellaneous Instructions b (8-bit)	2					X		
	2.7. 8-bit Multiply with 32-bit Add Instructions	3		X					
3. 64-bit Profile Instructions	3.1. 64-bit Addition & Subtraction Instructions	10							X
	3.2. 32-bit Multiply with 64-bit Add/Subtract Instructions	8				X			
	3.3. Signed 16-bit Multiply with 64-bit Add/Subtract Instructions	10			X				
4. Non-SIMD Instructions	4.1. Q15 Saturation Instructions	7						X	
	4.2. Q31 Saturation Instructions	15						X	
	4.3. 32-bit Computation Instructions	9							X
	4.4. Overflow/Saturation Status Manipulation Instructions	2	X						
	4.5. Miscellaneous Instructions	11							X
5. RV64 Only Instructions	5.1. 32-bit Addition & Subtraction Instructions	30	X						
	5.2. 32-bit Shift Instructions	14	X						
	5.3. 32-bit Miscellaneous Instructions	5	X						
	5.4. Q15 Saturating Multiply Instructions	9						X	
	5.5. 32-bit Multiply Instructions	3	X						
	5.6. 32-bit Multiply & Add Instructions	3				X			
	5.7. 32-bit Parallel Multiply & Add Instructions	12				X			
	5.8. Non-SIMD 32-bit Shift Instructions	1							X
	5.9. 32-bit Packing Instructions	4	X						
Total Instructions		332	189	3	29	47	2	31	31

3.3. Conclusion

In this chapter, the 332 instructions of the P-extension have been divided into subsets. Due to the large amount of new instructions and the limited time available, these subsets have been prioritized to make sure that the most important functionality would be available at the end of this project. By studying the code of open-source implementations of machine learning, video encoding, and computer vision applications, the occurrence of specific SIMD instructions was investigated. Using this information, the contents and the order of implementation of the subsets were defined. The subset that will be implemented first is the basic subset and contains instructions that were used by all the investigated applications. The basic subset includes SIMD instructions for addition, subtraction, comparison, bit shift, multiplication, and data movement that operate on 8-, 16-, and 32-bit element sizes, as well as a few miscellaneous instructions. Furthermore, subsets are defined for MAC instructions that operate on respectively 8-, 16-, and 32-bit elements as well as a subset for 8-bit SAD instructions. After the basic subset the MAC 8-bit subset will be implemented, followed by the MAC 16-bit, MAC 32-bit, and SAD 8-bit subsets. Finally, the remaining instructions are divided into the fixed-point and the non-SIMD 32- and 64-bit subsets which will be implemented in that order.

4 Implementation

In order to support the instructions discussed in the previous chapter, additional hardware and software is required. In this thesis, the instructions of the basic subset as well as the MAC 8-bit, MAC 16-bit, and MAC 32-bit subsets have been implemented. This comes to a total of 268 instructions or 80.7% of the total 332 instructions specified by version 0.9.2 of the P-extension. While not all instructions are implemented, all instructions that could be utilized by the used benchmarks are present in the implemented subsets. Therefore, the partial implementation of the P-extension does not limit the performance figures shown in this report with respect to a complete implementation of the P-extension. First, the hardware required to perform the operations as specified by these instructions is discussed. Second, the software implementation is detailed regarding the compiler and the construction of the benchmarks.

4.1. Hardware

The design of the CVA6 processor is split up into multiple functional units as shown in Figure 4.1. To support the instructions provided by the P-extension, multiple of these functional units need to be modified and extended. This mainly involves the ALU and multiplier for the functionality of the SIMD instructions and the decoder to recognize the encoding of the new instructions, these have been highlighted in red. Furthermore, the control and status register file (CSR), highlighted in green, was modified to include the `vxsat` saturation status register. This register gets set when a saturating instruction performs saturation on the output. Additionally, the scoreboard and the register file, also highlighted in green, were modified so that a third register can be read as needed by the MAC instructions. The modules highlighted in red are discussed in their own sections, while the modifications to the green modules are detailed in the sections of the functional unit to which the modifications are related.

4.1.1. Decoder

The modifications to the decoder are relatively simple. Since there are new instructions that need to be decoded correctly, these options need to be added. The instruction formats available in the RISC-V ISA are shown in Table 4.1. The first selection is performed based on the `opcode` field since it is present in all instruction formats. Subsequently, depending on the instruction format type, the individual instructions are selected based on the `funct7` and `funct3` fields.

Currently, all of the instructions introduced by the P-extension make use of the opcode `0b1110111`. Almost all instructions in the P-extension are R-type instructions. For these instructions, the correct functional unit gets selected and the addresses for the source and destination registers get forwarded. In the issue stage, the values of the source registers are read from the register file. The result of the operation gets written back to the register file in the commit stage. Instructions with an immediate value introduced by the P-extension, like the immediate shift instructions, use a specialized R-type format similar to the immediate shift instructions of the base integer ISA. Since for these instructions no second input register is needed, the `rs2` field is used as an immediate value instead.

The P-extension does however introduce some instructions, like absolute value, data reorganization, and bit

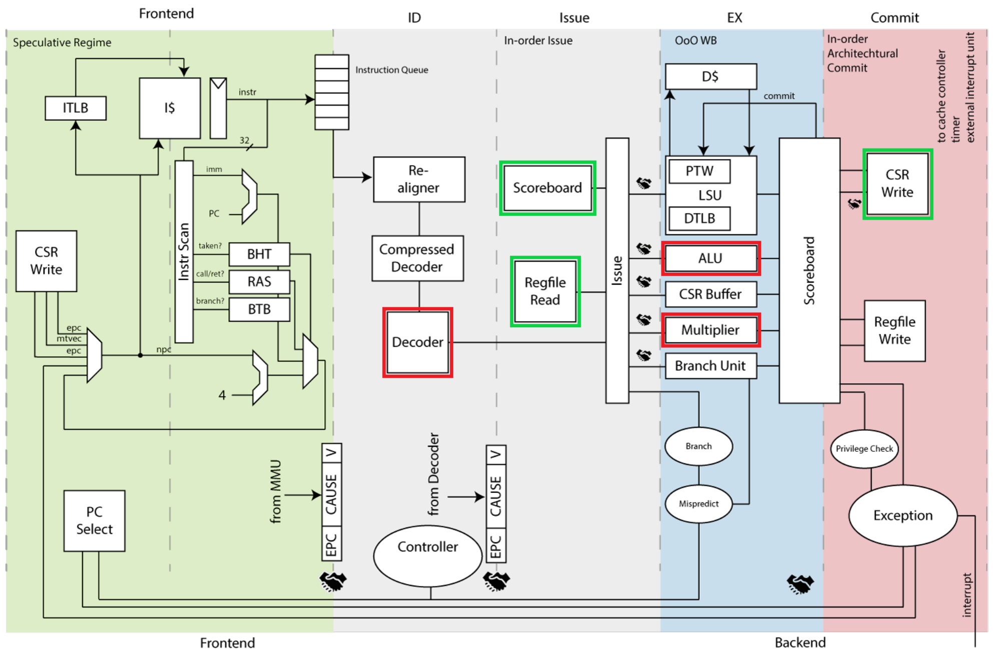


Figure 4.1: The structure of the CVA6 processor with the modified modules highlighted, adapted from [12].

counting instructions, that only utilize a single input register. These are encoded in a modified R-type format where bits 20 through 24 originally meant for `rs2`, which is not required for these instructions, are used to select between instructions. In other words, the `rs2` field is used as an additional `funct` field. By encoding these instructions in this manner, the same combination of `funct7` and `funct3` fields can be used for multiple instructions. Therefore, less of the opcode space is used, which allows the P-extension the freedom to add more instructions in the future.

Table 4.1: Instruction formats of the RISC-V ISA [10].

Type	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R-type	funct7							rs2					rs1					funct3			rd					opcode						
I-type	imm[11:0]												rs1					funct3			rd					opcode						
S-type	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
B-type	imm[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				imm[11]	opcode						
U-type	imm[31:12]																				rd					opcode						
J-type	imm[20]	imm[10:1]										imm[11]	imm[19:12]								rd					opcode						

4.1.2. ALU

The arithmetic logic unit (ALU) has multiple tasks, specifically addition, subtraction, bit shifts, comparisons, and bitwise operations. A schematic of the structure of the unmodified ALU is depicted in Figure 4.2. New elements are introduced and existing elements are modified to implement the operations specified by the P-extension. Since these elements operate more or less independently, they are discussed in their own subsections.

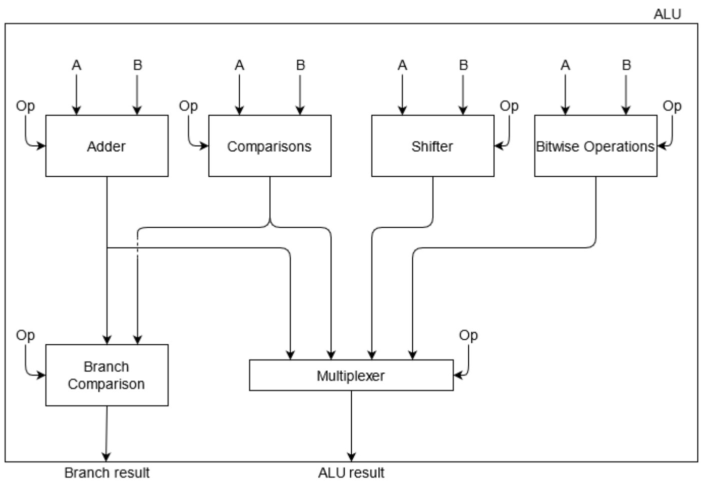


Figure 4.2: Schematic of the structure of the unmodified ALU.

Adder

The P-extension adds a total of 70 SIMD add and subtract instructions, specifically in sections 1.1, 1.2, and 5.1, as shown in Table 3.2. These consist of regular SIMD addition and subtraction for 8-, 16-, and 32-bit integers. Apart from this, there are also signed and unsigned halving addition and subtraction instructions. These instructions perform a signed or unsigned right shift of one position on the result. This results in a truncated element-wise average for the halving addition and a truncated halved difference for the halving subtraction. Another way to prevent wrap around in case of an overflow are saturating operations, where in the case of overflow the result is saturated to the largest or smallest value depending on the type of overflow. These instructions are also provided for addition and subtraction in signed and unsigned variants for 8-, 16-, and 32-bit integers. First, the implementation of regular SIMD addition and subtraction is discussed. Following this, the implementation of the halving and saturating operations is detailed. Finally, the combined add and sub as well as the crossed operations are explained.

Implementation of the regular SIMD add instructions is relatively simple, carries should not be propagated between elements of the size specified by the instruction. In order to do this, the adder was split in 8 sections of 8-bits. The carry out signals of these sections are then added to the least significant bit of the next section depending on the element size specified by the instruction. An example of a 16-bit addition is shown in Equation 4.1. For subtraction, the second operand needs to be negated before adding it to the first operand. The two's complement is calculated by inverting the bits and adding an LSB. Since the positions where the LSBs need to be inserted for subtraction do not require a carry input, the LSB can inserted instead of the carry. Depending on the instruction, signals are generated that specify whether that 8-bit element needs to be negated. These signals are then XOR-ed with the second input to invert it and inserted as carry in the appropriate positions to account for the addition of the LSB. An example of 16-bit and 8-bit subtraction is shown in Equations 4.2 and 4.3.

$$\begin{array}{rr} \text{a[15:8],} & \text{a[7:0]} \\ \text{b[15:8],} & \text{b[7:0]} \\ \text{Cout,} & 0 \\ \hline \multicolumn{2}{c}{\text{a[15:0] + b[15:0]}} \end{array} + \tag{4.1}$$

$$\begin{array}{rr} \text{a[15:8],} & \text{a[7:0]} \\ \overline{b[15:8]}\text{,} & \overline{b[7:0]} \\ \text{Cout,} & 1 \\ \hline \multicolumn{2}{c}{\text{a[15:0] - b[15:0]}} \end{array} + \tag{4.2}$$

$$\begin{array}{rr} \text{a[15:8]}, & \text{a[7:0]} \\ \overline{b[15:8]}, & \overline{b[7:0]} \\ 1, & 1 \\ \hline \text{a[15:8] - b[15:8]}, & \text{a[7:0] - b[7:0]} \end{array} + \tag{4.3}$$

Adding two 8-bit values leads to a 9-bit result, for a regular addition the lower 8-bits would be returned but in the case of a halving addition the upper 8-bits are returned. A halving addition thus computes a kind of truncated average of the two inputs. To properly handle signed values, the most significant bit (MSB) of the result of a halving operation is set to the carry out of the 8-bit adder segment XOR-ed with the MSB of both inputs. For unsigned operations, the carry out of the adder is inverted if a halving subtraction is performed. This takes care of the sign extension of the second operand.

To perform saturating addition or subtraction, overflow needs to be detected. Detecting unsigned overflow is fairly straightforward. Unsigned refers to the fact that the inputs to these instructions are interpreted as unsigned values. For addition, saturation should occur when the result is larger than $2^n - 1$, where n is the element size in bits. A result is larger than $2^n - 1$ when the carry out is equal to 1. For subtraction, saturation should occur when the subtrahend is larger than the minuend which would give a negative result. A negative result can also be detected by looking at the carry out of the result, which in the case of a negative result is equal to 0. The reason for this is that when the result of the subtraction is positive, overflow is present. When there is no overflow, the result is negative and thus has to be saturated. This leads to the equation for detecting unsigned overflow as shown in Equation 4.4. For addition, the result gets saturated to $2^n - 1$, all ones in binary, and for subtraction to 0. This means that the result can be expressed as in Equation 4.5.

$$\texttt{unsigned\_overflow} = \texttt{adder\_result}[n] \oplus \texttt{sub} \tag{4.4}$$

$$\texttt{result} = \begin{cases} \texttt{extend(!subtract)} & \text{if } \texttt{unsigned\_overflow} \\ \texttt{adder\_result}[n-1:0] & \text{otherwise} \end{cases} \tag{4.5}$$

With signed saturation, both positive and negative overflow can occur with addition and subtraction. For addition, positive overflow can occur when both operands have the same sign while for subtraction it can occur when the minuend is positive and the subtrahend is negative. Negative overflow, on the other hand, can occur for addition when both operands are negative and for subtraction when the minuend is negative and the subtrahend is positive. Whether overflow occurs can be detected by looking at the sign of the result. When positive overflow occurs, the result flows over into the sign bit, thereby making the result negative. For negative overflow the result also flows over, making the result positive. These can be combined for saturated addition and subtraction as shown in Equations 4.6 and 4.7 respectively for positive and negative overflow. Another way of detecting overflow is by looking at the carry in and carry out signals of the most significant bit of the adder segment. Positive overflow can be detected when the carry in is 0 and the carry out is 1, as this occurs when the addition of two positive numbers gives a negative result. Negative overflow, on the other hand, occurs when the carry in is 0 and the carry out is 1, since here a positive result is obtained when adding two negative numbers. This is illustrated in Table 4.2. Since dedicated carry lookahead logic is included in the FPGA, the carry output is available before the sum. This makes this method faster than the one shown in Equations 4.6 and 4.7. However, while after synthesizing the logic delay was reduced, the routing delay was increased. This can be caused by the fact that the carry signals now need to be routed out of the adder to be used in the overflow check, thereby increasing the routing complexity. Because of this, the total delay on this path was increased. Therefore, the method shown in Equations 4.6 and 4.7 was used.

$$\texttt{positive\_overflow} = (!\texttt{input\_a}[n-1]) \,\&\, (!\texttt{input\_b}[n-1] \oplus \texttt{sub}) \,\&\, \texttt{adder\_result}[n-1] \tag{4.6}$$

$$\texttt{negative\_overflow} = \texttt{input\_a}[n-1] \,\&\, (\texttt{input\_b}[n-1] \oplus \texttt{sub}) \,\&\, (!\texttt{adder\_result}[n-1]) \tag{4.7}$$

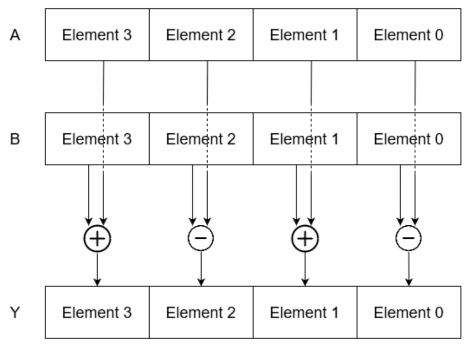
When overflow occurs, the result should be saturated to $2^{n-1} - 1$ for positive overflow and -2^{n-1} for negative overflow. The checks are performed for all 8-bit adder segments. However, when for example a 16-bit

Table 4.2: Detecting overflow by looking at the carry in and out of the most significant bit of the element.

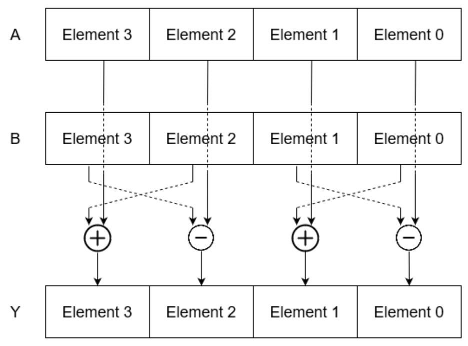
input_a[n-1]	input_b[n-1]	carry_in	carry_out	adder_result[n-1]	overflow
0	0	0	0	0	no
0	0	1	0	1	positive
0	1	0	0	1	no
0	1	1	1	0	no
1	0	0	0	1	no
1	0	1	1	0	no
1	1	0	1	0	negative
1	1	1	1	1	no

signed saturating addition is performed, only the signals corresponding to the most significant 8-bit element are checked. In this case, the even 8-bit elements get assigned `0xFF` and `0x00`, while the odd 8-bit elements get assigned `0x7F` and `0x80` for positive and negative overflow respectively. When no overflow occurs, the elements get assigned the result of their adder element.

One often stated downside of SIMD instructions is that programs have considerable overhead in placing the elements in the correct order in the register. To minimize this overhead, instructions are included that perform straight add and sub (`stas`, `stsa`), and crossed add and sub (`cras`, `crsa`) operations. For example, the `stas16` instruction adds 16-bit elements 1 and 3 and subtracts 16-bit elements 0 and 2, this is illustrated in Figure 4.3a. The `stsa16` instruction, on the other hand, subtracts 16-bit elements 1 and 3 and adds 16-bit elements 0 and 2. These instructions are defined similarly for 32-bit elements and are also provided in signed and unsigned, saturating and halving variants. To expand on this, crossed add and subtract (`cras`) and crossed subtract and add (`crsa`) instructions are also provided that perform the same operation but swap the elements of the second operand. The `cras16` instruction is illustrated in Figure 4.3b. These are again also provided in saturating and halving variants for signed and unsigned data. The implementation of these instructions is fairly straightforward. Since the adder is split into segments based on the required element size, the elements can also independently add or subtract. Only the even or odd elements are negated for the `stas` and `stsa` instructions respectively. The `cras` and `crsa` instructions are implemented similarly to the `stas` and `stsa` instructions, however the elements in the second operand are swapped. Swapping the elements before the input of the adder is easily implemented by using a multiplexer.



(a) An example of the `stas16` instruction.



(b) An example of the `cras16` instruction.

Figure 4.3: Example of the `stas16` and `cras16` instructions.

Shifter

In this section, the implementation of the shift instructions from sections 1.3, 1.4, and 5.2 of the P-extension, as shown in Table 3.2, is discussed. Since the P-extension only specifies shift instructions that shift all elements with the same amount, the original shifter of the processor can be reused. The shifter only performs right shifts and implements left shifts by reversing the input and the result of the shifter. The problem with reusing the shifter is that by shifting, bits get shifted out one element into the other. In order to get rid of these unwanted bits, a mask gets applied. This mask consists of a number of zeros equal to the shift amount while the rest is set to one, this is the inverse of the shift amount represented in thermometer coding. By performing an AND operation with the mask and the shift result, the unwanted bits get cancelled out. This works well for logical shifts where zeros get shifted in. However, for arithmetic shifts, where the sign bit gets replicated for the shifted in bits, this does not work. To reintroduce the sign bits, another mask gets applied. This mask is constructed by taking the inverse of the original mask and performing an AND operation with the sign bit of that element. This results in a mask that has the sign bit replicated as many times as the shift amount and the rest of the bits equal to zero. By applying an OR operation with this mask on the result of the previous mask application, the sign bits get correctly inserted and an arithmetic shift is realized. The SIMD shift operation for an arithmetic right shift by 4 positions with an 8-bit element size is shown in Equation 4.8.

$$\begin{array}{rrrrc} \text{a[15:12]}, & \text{a[11:8]}, & \text{a[7:4]}, & \text{a[3:0]} & \multirow{2}{*}{\text{>>4}} \\ \hline 0000, & \text{a[15:12]}, & \text{a[11:8]}, & \text{a[7:4]} & \\ 0000, & 1111, & 0000, & 1111 & \text{AND} \\ \hline 0000, & \text{a[15:12]}, & 0000, & \text{a[7:4]} & \\ \text{a[15]}, & 0000, & \text{a[7]}, & 0000 & \text{OR} \\ \hline \text{a[15]}, & \text{a[15:12]}, & \text{a[7]}, & \text{a[7:4]} & \end{array} \tag{4.8}$$

The rounding right shift instruction is a more accurate type of shift instruction. In this instruction, the input gets shifted as normal but gets half an LSB added to the result. This causes the result of the bit shift operation to be rounded instead of truncated like with a regular shift instruction. To implement this, 1 is subtracted from the shift amount to shift one place fewer. An adder is then used to add 1 and shift the result one place to the right, similar to the halving addition instruction. In the special case that the shift amount is equal to 0, no shift or rounding operation is performed and the original value is returned. An example of a SIMD rounding logical shift operation on 8-bit elements is shown in Equation 4.9. In this example, a 4 position rounding right shift is performed on the values 105 and 101. Shifting these values by 4 positions gives $\frac{105}{2^4} \approx 6.56$ and $\frac{101}{2^4} \approx 6.31$, rounding these values to an integer would result in 7 and 6 respectively. This is also the result provided by the rounding shift operation compared to the truncated result of 6 which a regular shift operation would return for both inputs.

$$\begin{array}{rrrrc} 0110, & 1001, & 0110, & 0101 & \text{>>3} \\ \hline 0000, & 1101, & 0010, & 1100 & \\ 0001, & 1111, & 0001, & 1111 & \text{AND} \\ \hline 0000, & 1101, & 0000, & 1100 & \\ 0000, & 0001, & 0000, & 0001 & + \\ \hline 0000, & 1110, & 0000, & 1101 & \text{>>1} \\ \hline 0000, & 0111, & 0000, & 0110 & \end{array} \tag{4.9}$$

As with the addition and subtraction instructions, a saturating shift instruction is also defined by the P-extension. Because the result needs to get larger for overflow to occur, only the left shift operation has a saturating variant. This instruction returns $2^{n-1} - 1$ when positive overflow occurs, -2^{n-1} when negative overflow occurs, or the normal result of the shift operation when there is no overflow. Since a left bit shift is effectively a multiplication with 2^p, where p is the shift amount, positive overflow can only occur if the input is positive and negative overflow only if the input is negative. Overflow occurs when the sign of the original value and the result are different or when a bit different from the sign bit is shifted out.

In order to collect the shifted out bits, the inverse of the original mask is taken and an AND operation is performed with the input. This value now holds the bits that are shifted out and zeros where the bits that remain in the result are. Another AND operation is performed on the original result with the sign bit of the most significant element so that it contains zeros in the positions where bits get shifted out and the sign bits on the other positions. By taking these two new masks and applying an OR operation on them, it contains the shifted out bits as well as sign bits. By checking if every bit in this value is equal to the sign bit, it can be detected when a bit unequal to the sign bit gets shifted out and thus if overflow occurs. An example of a saturating left shift operation by 2 positions is shown in Equation 4.10 and the corresponding saturation check is shown in

Equation 4.11. Performing a left shift with 2 positions on the lower 8-bit element should give $-92 \cdot 2^2 = -368$, however this does not fit in 8-bits. This can be seen in the saturation check where the shifted out zero is present and the result is thus saturated to -128. Performing the shift on the upper 8-bit element should give $43 \cdot 2^2 = 172$ which also does not fit in 8-bits. In this case, no bit unequal to the sign bit is shifted out, however the sign of the input is not the same as the sign of the result and thus the result also gets saturated.

$$\begin{array}{cccc|l}
0010, & 1011, & 1010, & 0100 & \\
\hline
0010, & 0101, & 1101, & 0100 & \text{Rev} \\
\hline
0000, & 1001, & 0111, & 0101 & >>2 \\
0011, & 1111, & 0011, & 1111 & \\
\hline
0000, & 1001, & 0011, & 0101 & \text{AND} \\
\hline
1010, & 1100, & 1001, & 0000 & \text{Rev} \\
\hline
0111, & 1111, & 1000, & 0000 & \text{Sat}
\end{array} \tag{4.10}$$

$$\begin{array}{cccc|l}
0011, & 1111, & 0011, & 1111 & \\
\hline
1100, & 0000, & 1100, & 0000 & \text{NOT} \\
0010, & 1011, & 1010, & 0100 & \\
\hline
0000, & 0000, & 1000, & 0000 & \text{AND}
\end{array}
\qquad
\begin{array}{cccc|l}
0011, & 1111, & 0011, & 1111 & \\
0000, & 0000, & 1111, & 1111 & \\
\hline
0000, & 0000, & 0011, & 1111 & \text{AND}
\end{array}$$
$$\begin{array}{cccc|l}
0000, & 0000, & 1000, & 0000 & \\
0000, & 0000, & 0011, & 1111 & \\
\hline
0000, & 0000, & 1011, & 1111 & \text{OR}
\end{array} \tag{4.11}$$

A left and right shift operation are also combined in the `kslra` and `kslra.u` instructions. For these operations a saturating left shift is performed if the shift amount is positive, when it is negative an arithmetic right shift is performed for `kslra` and a rounding arithmetic right shift is performed for `kslra.u`. For the `kslra.u` instruction the shift amount gets XOR-ed with the sign bit, the positive shift amount stays positive to perform the saturating left shift, while it logically negates the negative shift amount (effectively inverting the shift amount and subtracting one). The rounding operation is then performed like the other rounding shift instructions. For the `kslra` instruction however, the shift amount needs to be properly inverted when it is negative so the sign bit gets added to the result of the XOR operation. Based on the sign, either the left shift or right shift result is selected.

Comparisons
The instructions from sections 1.5 and 1.6 of the P-extension, as shown in Table 3.2, specify SIMD comparison instructions and their implementation is detailed in this section. The extension specifically adds equal, less than, and less than or equal comparisons in signed and unsigned variants for 8- and 16-bit elements. When the comparison is true, the specific element gets set to all ones and to zero otherwise. The adder can again be reused here, by performing a SIMD subtraction on the two inputs and checking if the result is zero, the equal condition can be checked. Which element is smaller can be determined by looking at the sign of the inputs and result. For example, when both inputs are positive and the result of the subtraction is negative, it can be determined that the subtrahend is larger than the minuend. This relation is presented in Table 4.3.

Miscellaneous
The miscellaneous category contains a number of instructions with very different functionality. These instructions are defined in sections 1.9, 1.10, 2.6.a, and 5.3 of the P-extension, as shown in Table 3.2. They can be divided into minimum and maximum, clip, absolute value, bit counting, and swap instructions. The implementation of these instructions will be discussed one by one in the following sections.

- The minimum and maximum instructions are available for signed and unsigned values for 8-, 16-, and 32-bit elements. Like the compare instructions discussed before, these also compare two values. Therefore, the comparison operation can be reused for these instructions. However, instead of setting the

Table 4.3: Truth table for checking whether the subtrahend or minuend is smaller both signed and unsigned.

MSB(a)	MSB(b)	MSB(result)	signed(a<b)	unsigned(a<b)
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

register to the maximum value or zero depending on if the condition is true, the minimum and maximum instructions return the smallest or largest of the two elements respectively.

- The `clip` instruction takes a value and saturates it to the amount of bits specified by an immediate value. This instruction is provided in signed and unsigned variants for 8-, 16-, and 32-bit element sizes. The signed clip instruction saturates the elements of a register to the range of $2^{imm}-1$ to -2^{imm}, where imm represents the immediate value. The unsigned clip instruction, on the other hand, saturates all negative values to zero, resulting in a range of $2^{imm}-1$ to 0. Positive overflow can be detected by checking if the value is positive and that a bit is set to one above the bit indicated by the immediate value. For negative overflow it is the exact opposite, the value being negative and a bit above the bit specified by the immediate value that is set to zero. This is the same check that is performed for a saturating shift instruction. However, the mask needed here can be created by reversing and inverting the mask used for a saturating shift instruction. This is because for a saturating left shift instruction with 1 position, the bit that needs to be checked is the MSB. For a `clip` instruction with an immediate value of 1, on the other hand, all bits except for the LSB need to be checked. When positive overflow is detected, the result is set equal to the inverse of the mask, as that is equal to $2^{imm}-1$. If negative overflow occurs, the result is set to zero for an unsigned clip instruction and set to the mask for a signed clip instruction as the value of the mask is equal to -2^{imm}.

- As the name suggests, the `kabs` instruction computes the absolute value and is defined in the specification for 8-, 16-, and 32-bit element sizes. The only exception here is that the absolute value of the most negative value does not fit in that element size. Therefore, this value is saturated to the maximum positive value. For example, with an 8-bit element size and an input value of -128 (`0x80`) the instruction returns 127 (`0x7F`). To realize this instruction, the adder is again reused. In order to only negate the element when it is negative, the value is subtracted from zero or added to zero depending on whether the sign bit is set. This is done by performing an XOR operation with the value and the sign bit and subsequently adding the sign bit to it. This operation results in the original value when it is positive and the negated value when it is negative. However, this does not yet handle the maximum negative value properly since the result stays negative. Luckily this situation is the same as positive overflow in a saturated addition or subtraction. Therefore the `kabs` instruction can be realized by performing a saturating addition or subtraction with zero depending on the sign bit of the element. This does however require that the single operand of the `kabs` instruction gets placed in the second operand of the adder, which can easily be done in the decoder.

- The `clz` (count leading zeros) and `clo` (count leading ones) instructions count the amount of zeros or ones respectively until respectively the first one or zero is encountered. The `clrs` (count leading redundant sign) instruction counts how many duplicates of the sign bit are present. These duplicate bits are redundant since the value represented by the bits would not change if these duplicate bits were removed. All three instructions are provided for element sizes of 8-, 16- and 32-bits. Many different ways exist to create a leading zero counter. One way is to check if the $\frac{n}{2}$ most significant bits of the input are all zeros, where n is the size of the whole input. This determines the MSB of the result. If the result of this check was true, the next bits to check are the $\frac{n}{4}$ bits that follow the previously checked bits. If the check was false, the next check should be on the most significant $\frac{n}{4}$ bits of the input. This check determines the next bit of the result. By performing this recursively, all leading zeros are counted [56]. This is done for all 8-bit elements, these results are then combined to form the 16- and 32-bit results if required. To construct the larger result it is first checked if both smaller results are all zeros, if so the result is set to the correct value. Otherwise, if only the most significant element consists of only zeros, the result can be constructed by performing an OR operation between the two results. Finally, if the most significant element does have a one, the larger result is set to the result of the most significant element. This leading zero counter can also be reused to count leading ones, by inverting the input. To make sure that the sign bit itself does not get counted in the `clrs` operation, the input is shifted left by 1 position and a 1 is shifted in. Furthermore, since the sign bit can be either 0 or 1 and the module counts zeros, the input is also XOR-ed with the sign bit.

- The final miscellaneous instruction is the `swap` instruction, provided for 8- and 16-bit element sizes. This instruction simply swaps the position of the elements with even and odd index. For example, applying the `swap8` instruction on the data `0x0807060504030201` gives the result `0x0708050603040102`.

Pack and Unpack

Instructions to ease data movement between two registers or to reorganize the elements within a register are defined in sections 1.11, 2.1, and 5.9 of the P-extension, as shown in Table 3.2. The implementation of these instructions is presented in this section. The unpack instructions specifically take four 8-bit elements and zero extend or sign extend this value to 16-bits respectively for unsigned and signed data. The pack instructions on the other hand combine the 16- or 32-bit elements of two registers in different ways into another register. Variations are provided to pick the elements with even index of both registers, the elements with even index of one and elements with odd index of the other, and the odd indexed elements of both registers.

Saturation Status Register

As seen in the previous sections, the P-extension includes many instructions that perform saturating operations. When overflow would occur, the output is instead saturated to the largest or smallest value depending on the type of overflow. For a program to be able to detect when this occurs, the `vxsat` status register is introduced in the control and status register file (CSR). This register gets set when an output is saturated and can be read out or cleared by using instructions from the Zicsr instruction set extension. Setting this register is implemented by reusing the exception cause field meant for a branch exception while keeping the valid signal low. The cause field is set to 1 when a saturating instruction saturates the output. By detecting if a saturating instruction is being committed, the `vxsat` register can be set when the cause field is high. The valid signal is kept low because this is not an actual exception and the result should still be committed when saturation occurs.

Having integrated all the hardware required for the ALU to perform the SIMD instructions of the P-extension, the structure is slightly modified. Furthermore, while reusing hardware is beneficial for resource utilization, it can make routing more difficult and thus negatively impact the critical path of the design. To optimize the design some changes were made with regard to hardware reuse, this will be discussed in Section 4.1.5. This leads to the modified structure of the ALU as seen in Figure 4.4.

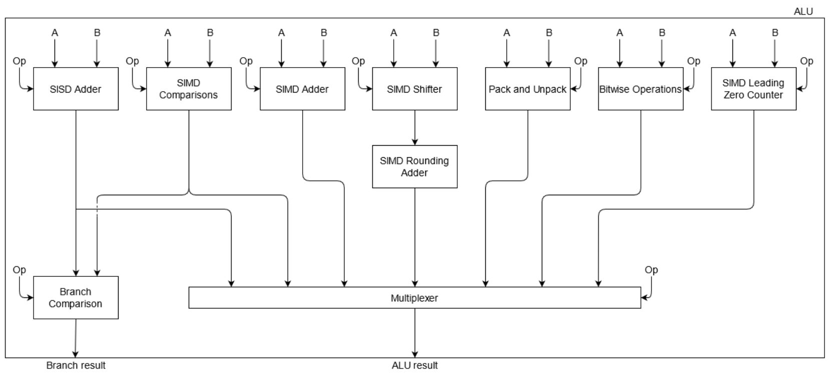


Figure 4.4: Schematic of the structure of the ALU with support for the instructions of the P-extension.

4.1.3. Multiplier

Unlike the ALU the multiplier only has one task, performing multiplications. Many different multiplication algorithms have been proposed that have their advantages and disadvantages. These include Baugh-Wooley, Booth, modified Booth, and many more. Multiplication generally consists of generating partial products and combining these partial products into a single result. An example of an unsigned multiplication is shown in Table 4.4a. For a signed multiplication however, some changes need to be made. Since in two's complement notation the MSB is weighted negatively, this needs to be taken into account when performing the multiplication. Specifically, when multiplying the MSBs of both operands together the result is positive and when multiplying the MSB of one of the operands with another bit the result is negative. A signed multiplication has been illustrated in Table 4.4b.

Table 4.4: Example of binary multiplication.

(a) Example 4-bit unsigned multiplication.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

(b) Example 4-bit signed multiplication.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				$-a_3b_0$	a_2b_0	a_1b_0	a_0b_0
			$-a_3b_1$	a_2b_1	a_1b_1	a_0b_1	
		$-a_3b_2$	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	$-a_2b_3$	$-a_1b_3$	$-a_0b_3$			
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

To realize this signed multiplication, different techniques are proposed. Baugh and Wooley noted that instead of performing a subtraction for the negatively weighted elements, they can be negated and then added like the other partial products [57]. This negation has been illustrated in Table 4.5. These two negated partial products can then again be combined as shown in Table 4.6a. When the partial products with negative sign are then replaced by this result, the partial product matrix as shown in Table 4.6b is obtained. By performing an XOR operation to selectively invert the negatively weighted partial products and only adding the ones when performing a signed multiplication, the multiplier can be used for both signed and unsigned multiplications.

Table 4.5: Two's complement of the partial products with negative sign.

(a) Two's complement of the negative partial products created by the MSB of operand a.

0	0	$-a_3b_2$	$-a_3b_1$	$-a_3b_0$	0	0	0
1	1	$\overline{a_3b_2}$	$\overline{a_3b_1}$	$\overline{a_3b_0}+1$	0	0	0

(b) Two's complement of the negative partial products created by the MSB of operand b.

0	0	$-a_2b_3$	$-a_1b_3$	$-a_0b_3$	0	0	0
1	1	$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}+1$	0	0	0

Table 4.6: Illustration of the Baugh-Wooley scheme.

(a) Summation of the negative partial products.

1	1	$\overline{a_3b_2}$	$\overline{a_3b_1}$	$\overline{a_3b_0}+1$	0	0	0
1	1	$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}+1$	0	0	0
1	0	$\overline{a_3b_2}+\overline{a_2b_3}$	$\overline{a_3b_1}+\overline{a_1b_3}+1$	$\overline{a_3b_0}+\overline{a_0b_3}$	0	0	0

(b) Signed multiplication using the Baugh-Wooley scheme.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
			1	$\overline{a_3b_0}$	a_2b_0	a_1b_0	a_0b_0
			$\overline{a_3b_1}$	a_2b_1	a_1b_1	a_0b_1	
		$\overline{a_3b_2}$	a_2b_2	a_1b_2	a_0b_2		
1	a_3b_3	$\overline{a_2b_3}$	$\overline{a_1b_3}$	$\overline{a_0b_3}$			
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

Another signed multiplication algorithm was proposed by Booth [58]. This algorithm relies on the fact that a sequence of ones can be replaced by the sequence with a one before the start of the sequence and a minus one at the end and setting the other bits in the sequence to zero. For example, $2^{n-1}+2^{n-2}+2^{n-3}+2^{n-4} = 2^n - 2^{n-4}$, or filled in with $n = 5$ gives $2^4+2^3+2^2+2^1 = 30 = 2^5 - 2^1$. This encoding is summarised in Table 4.7a where $b_{-1} = 0$. This encoding has been extended to higher radices by taking more bits into account when recoding. The modified booth recoding for radix-4 is shown in Table 4.7b. The value of the partial products therefore range between -2 and 2 times the multiplicand. A benefit of using modified Booth encoding is that the amount of partial products to add is reduced by half. This reduces the size of the reduction tree that is responsible for adding the partial products together to form the result of the multiplication. However, this comes at the cost of the added recoding logic and the logic required for creating the partial products ranging from -2 to 2 times the multiplicand.

Table 4.7: Booth recoding tables, where $b_{-1} = 0$.

(a) Booth recoding table.

b_i	b_{i-1}	c_i
0	0	0
0	1	1
1	0	-1
1	1	0

(b) Radix-4 modified Booth recoding table.

b_{i+1}	b_i	b_{i-1}	$c_{i/2}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

The previously discussed multiplication algorithms dealt with a single multiplication, however in this case SIMD multiplications need to be performed. An example of a SIMD multiplication is shown in Figure 4.5. In this section, the implementation of the SIMD multiplication instructions as defined in sections 1.7, 1.8, and 5.5 of the P-extension, as shown in Table 3.2, is discussed. Like with regular multiplication, many SIMD multiplication algorithms have been proposed. Both previously discussed schemes have been extended to support SIMD multiplications. A SIMD Baugh-Wooley multiplier is discussed in [59] and a SIMD radix-4 modified Booth multiplier is proposed in [60]. SIMD multipliers that are capable of full precision and two half precision multiplications based on the Baugh-Wooley and modified Booth schemes, are compared in [61]. The paper compares the multipliers in terms of area usage, energy usage, power usage, and delay for multiplier widths ranging from 8- to 64-bits. While the SIMD modified booth multiplier is able to achieve a slightly lower delay for some multiplier widths, it consistently has the highest area, power, and energy usage. This is stated to be caused by the complexity of converting the modified booth recoding circuit to operate in a SIMD manner. This is because the modified Booth recoding circuit processes three bits at once. On SIMD element boundaries, the b_{i-1} value must be taken as zero when that bit comes from a different element. Furthermore, since negative partial products can be generated, sign extension or another way of handling the signed value also needs to be adapted to prevent the sign bits from interfering with the result of another element.

Apart from half precision (32-bit), the multiplier for the P-extension needs to be able to also perform quarter precision (16-bit) and one eighth precision (8-bit) multiplications. Therefore, the complexity of the SIMD modified Booth multiplier would be increased and would thus likely exacerbate the higher area, power, and energy usage. Because of this, the SIMD Baugh-Wooley multiplier scheme was used. How a SIMD Baugh-Wooley multiplier functions is discussed below.

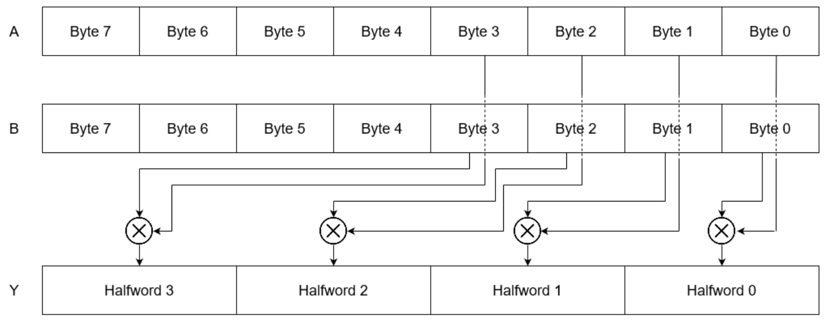


Figure 4.5: An example of a 8-bit to 16-bit widening SIMD multiply operation.

A n-bit multiplication can also be realized with an n/2-bit multiplier. In this case, four n/2-bit multiplications need to be performed. If the operands are expressed as $A = A_H 2^{n/2} + A_L$ and $B = B_H 2^{n/2} + B_L$ the product can be expressed as $Y = 2^n(A_H B_H) + 2^{n/2}(A_H B_L + A_L B_H) + A_L B_L$. These separate products can also be identified in the partial product matrix as shown in Figure 4.6. When performing a SIMD multiplication the desired result is $Y = A_H B_H, A_L B_L$ where the comma denotes a concatenation. The calculation of this result can be realized by setting the partial products in the partitions corresponding to $A_H B_L$ and $A_L B_H$ equal to zero. By masking off the partial products like this, the reduction tree responsible for adding the partial products together does not need to be modified. This strategy can be applied again on partitions $A_H B_H$ and $A_L B_L$ to compute a quarter precision SIMD multiplication (producing four n/4-bit results). By applying this strategy three times on a 64-bit multiplier, the 8-, 16-, and 32-bit SIMD multiplications specified by the P-extension can be realized.

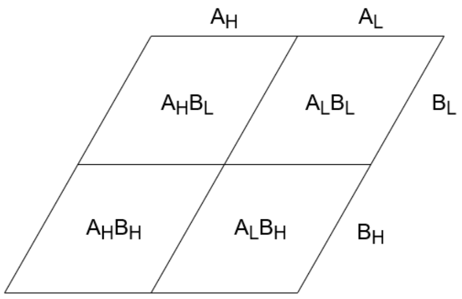


Figure 4.6: Partial product matrix showing the n/2-bit elements.

One aspect that has been overlooked in the explanation of the SIMD multiplier above, is how to handle signed multiplication. The concept proposed by Baugh and Wooley can again be applied here but requires some special handling of the different supported element sizes. For different element sizes, different bits need to be inverted and ones need to be inserted at different positions. Ones need to be added for every element in positions n and $2n-1$ where n is the element size. However, when the bit in position $2n-1$ is added, unwanted overflow can occur to the next element. Since this overflow is not desired in the result, the bit in position $2n-1$ of the result is inverted instead. Thereby only the sum is computed and the carry out bit is ignored, thus leaving the next element unaffected.

Saturation
In the basic subset of the P-extension, the only multiplication instructions that perform saturation are the Q7 and Q15 saturating multiply instructions. The Q7 and Q15 multiplications produce Q14 and Q30 results respectively, the instructions include a shift operation to return the results back to Q7 and Q15 data. Therefore, the only way to produce a result that would cause an overflow is when both inputs are the maximum negative value. The inputs can thus be checked to be equal to `0x80` or `0x8000` for Q7 and Q15 multiplications respectively in parallel to the multiplication operation. Setting the `vxsat` register is implemented in the same way as for the saturating instructions of the ALU.

4.1.4. Multiply Accumulate Unit

Since the multiply accumulate unit performs a SIMD multiplication and sums the multiplication results and a possible accumulator, it is designed as an addition to the multiplier. Again the P-extension provides a number of different MAC instructions for different combinations of accumulator and multiplication element sizes. These MAC instructions are specified in sections 2.2-2.5, 2.7, 3.2, 3.3, 5.6, and 5.7 of the P-extension, as shown in Table 3.2. The process of adding the multiplication results and the accumulator requires a number of adders. As stated in the previous section, a SIMD Baugh-Wooley multiplier is used. This multiplier uses the structure of the partial product matrix to sum the partial products of the SIMD multiplication separately. However, the structure of the partial product matrix can also be used to sum SIMD multiplication results together, as discussed in [62]. In this way, the reduction tree of the partial product matrix can be used to accumulate the results of the multiplications without the need for additional adders. The sum separate and sum together schemes have been illustrated in Figure 4.7.

The P-extension includes instructions in which some of the multiplication results get subtracted from the accumulator. An example of this is the `smalds` instruction which performs the following operation $rd = rd - A[0]B[0] + A[1]B[1] - A[2]B[2] + A[3]B[3]$, where A and B consist of 16-bit elements and `rd` is the destination register which is also used as 64-bit accumulator. Since the reduction tree only performs addition and inverting the inputs would not work when the input is the maximum negative value, not all accumulations

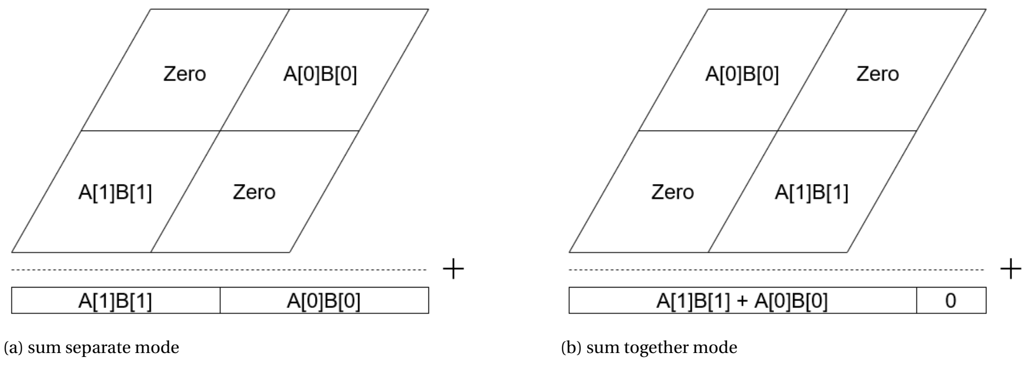


Figure 4.7: Illustration of the partial product matrix in sum separate and sum together modes.

can be performed using the reduction tree. Therefore, separate adders are required to perform this operation. Fortunately, since in these type of MAC instructions two multiplication results have the same sign, these can still be added using the sum together mode. In the given example, results A[0]B[0] and A[2]B[2], and results A[1]B[1] and A[3]B[3] for the `smalds` instruction can be added together in the partial product matrix. The subtraction can then be performed in the separate adders. Therefore, the sum together mode is used to perform accumulation on 2 times 2 16-bit multiplication results and on 2 times 4 8-bit multiplication results. These configurations have been illustrated in Figure 4.8. By performing part of the accumulation like this, only two SIMD adders, that are both capable of half precision additions, are needed to calculate the final result. One of the adders performs the final addition of the multiplication results and the other adds the multiplication results to the accumulator to obtain the final result.

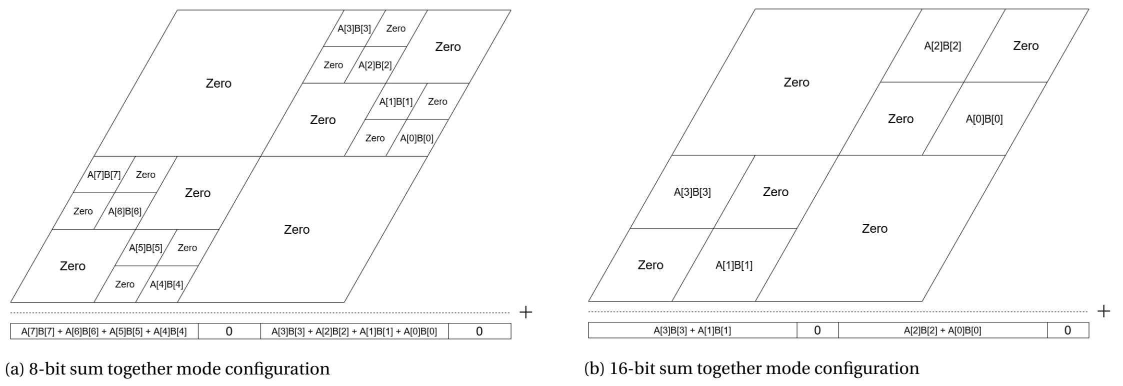


Figure 4.8: Illustration of the partial product matrix in sum together mode configuration for 8-bit and 16-bit multiplication.

Some of the instructions in the P-extension have the possibility of overflow to occur. However, specifically for the MAC instructions there is the possibility that the result of the multiplication does not fit within the element size, but gets reduced to that range after addition or subtraction of the accumulator. The specification states that the addition of the multiplication result is done with enough precision to avoid overflow from occurring on intermediate results. The P-extension provides MAC instructions that utilize either a 64-bit accumulator or an accumulator with two 32-bit elements. Therefore, like in the ALU, the adders need to be able to operate in a SIMD manner. However, for the MAC unit only 32-bit and 64-bit element sizes need to be supported. To prevent overflow from occurring on the intermediate results, 33-bit adders are needed. This is because the result of an addition on two 32-bit elements gives a 33-bit result. This results in the structure of the MAC unit as shown in Figure 4.9.

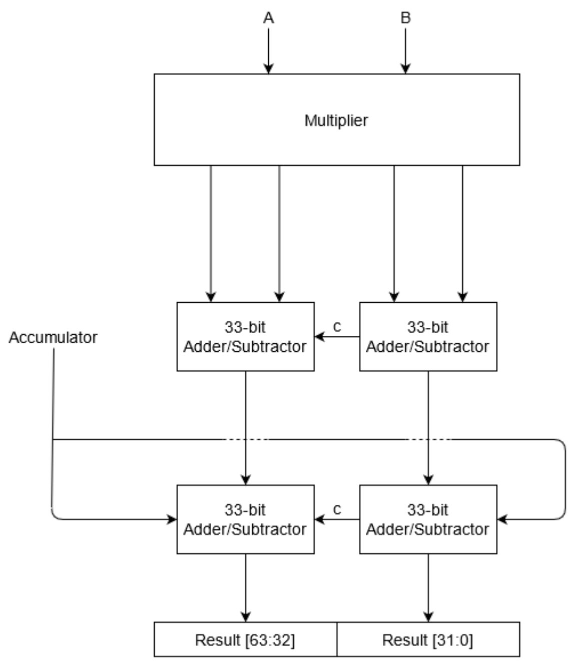


Figure 4.9: Schematic of the structure of the multiply accumulate unit.

Saturation
As stated before, the intermediate results for MAC instructions are computed with enough precision to avoid overflow. Therefore, the overflow check can be performed after the final addition or subtraction. The overflow check and setting the `vxsat` register is performed in the same way as for the adder in the ALU. One exception to this are the `kmmawb2`, `kmmawb2.u`, `kmmawt2`, and `kmmawt2.u` instructions. These instructions multiply 32-bits from the first input with 16-bits from the second input, double the result and add the most significant 32-bits of this result to a 32-bit accumulator. Here both the multiplication result and the accumulation results are checked for overflow and are saturated if necessary. The reason for the diverging definition of these instructions is to comply with the definition of the basic operations for the Enhanced Voice Services speech coding standard [63].

Reading the Third Operand
The MAC instructions in the P-extension accumulate the results to the destination register. This means that the value stored in the destination register, prior to executing the MAC instruction, needs to be available to the MAC unit. All integer instructions in the base instruction set and the instruction set extensions previously supported by the CVA6 processor operate on at most two values from the integer register file. Therefore, modifications need to be made to the core to provide the MAC unit with the three required values. Unlike the integer instructions, the floating-point extensions do include instructions that operate on three input values, specifically the fused multiply-add instructions [10]. To get the third operand to the floating-point unit, the decoder of the CVA6 processor stores the address of this register in the bits otherwise used for an immediate value. In the issue stage, depending on if the current instruction uses three operands, the specified register in the floating-point register file is read and the corresponding value is placed back into the immediate field. The third operand is then read from the immediate field in the floating-point unit in the execution stage. This however does require that the floating-point register file has three read ports. To provide the MAC unit with the value of the destination register the same method is used. Therefore, the integer register file is expanded to also feature three read ports. Furthermore, the checks responsible for forwarding results from the output of the functional units were expanded to include this third operand.

4.1.5. Timing Optimization
After the initial design was completed, it was synthesized for an FPGA using Vivado. While creating the first design, the focus was on reusing the same hardware as much as possible. This means that, for example, the rounding operation of a rounding shift instruction is performed using the SIMD adder. Although this does

result in a smaller design in terms of resource utilization, it might not always be the design with the lowest delay. When synthesizing the initial design with support for the basic subset and after adding the MAC instructions, it did not meet the timing requirements for the default 50 MHz clock frequency. Using the timing reports generated by Vivado and by inspecting the paths with the longest delay, the culprits could be identified and subsequently fixed.

For the rounding shift instructions, both a shift operation and a carry propagation is required to compute the rounded value. This addition of half an LSB could be performed by the same adder that handles the SIMD add instructions. However, routing the output of the shifter back to the adder makes routing difficult, leading to longer delays. Implementing it like this lead to a large timing violation on the rounding shift operations. Therefore, a separate adder was used for the rounding operation. While the clock speed could be reduced to accommodate for this increased critical path, this would negatively impact the performance of the processor.

One of the paths that has a limited amount of slack in the original design of the CVA6 processor is for the conditional branch instructions. For branch if equal and branch if not equal instructions, the adder is used to check if the operands are equal. This is done by performing a subtraction and then checking if the result is zero. Depending on the outcome of this comparison, the program counter is set to either the target of the branch or the consecutive value of the program counter. Due to the additions to the adder and the ALU in general, the delay of this path is increased. Furthermore, since it was already a path with a limited amount of slack, this caused the design to violate the timing requirements. To mitigate this, a separate non-SIMD adder was used for these instructions.

The addition of the MAC unit moved the critical path from the FPU to the MAC unit. Therefore, a different version of the design was made where the MAC unit has 3 cycles to compute its result instead of 2. In the original design of the CVA6 processor, the multiplier is pipelined by placing a register at the output of the multiplier. The design then relies on the synthesizer to move this register to the ideal location using retiming. To ease the timing constraint on the MAC unit, a register was placed after the output of the multiplier and one at the output of the adders performing the accumulation. However, this modification required another change, as the ALU and multiplier use the same bus to write back the result of an instruction. Because of this, in the original design, an ALU instruction can not directly follow a multiplication instruction, as both the ALU and multiplier would try to send the result over the bus in the same cycle. Since it now takes 3 cycles for the MAC unit to produce a result, an ALU instruction can not be issued 2 cycles after a MAC or multiplication instruction. While this change does improve the timing constraint, it also has an impact on the performance of the design since any multiplication or MAC instruction takes an additional cycle.

4.2. Software

Of course, a hardware implementation without any software support would not be very useful. A compiler that generates the instructions introduced by the new hardware is essential when creating a program that runs on this hardware. Furthermore, some software that can be used to show the impact of the new hardware is crucial to be able to evaluate the design. To do this, two types of benchmark are created, a synthetic and a real-world benchmark. The synthetic benchmark can show the somewhat idealized impact of the SIMD instructions, while the real-world benchmark shows the realistic improvement achievable from using the instructions of the P-extension. Both benchmarks have been written in the C programming language. The compiler and the benchmark software will be discussed in the following sections.

4.2.1. Compiler

While the performance of a well written assembly program can be better than when using a higher level programming language like C or C++, programming in a higher level language is often more convenient. In order to support the instructions added by the P-extension, the compiler and assembler need to be updated. Due to the fact that the P-extension has not been around for a long time and that it is still under development, it is not yet supported by the official RISC-V toolchain [64].

Andes Technology, the company from which the P-extension originated, has open sourced their implementation of GCC and binutils that has support for the instructions of the P-extension [65, 66]. This compiler and assembler mostly stick to the encoding specified by version 0.5.2, however, some inconsistencies were encountered. For example, the `smbb32` instruction should be an alias of the `mulsr64` instruction. However, the compiler uses a different encoding. It is possible that the encoding of the compiler corresponds to an earlier version of the P-extension. However, in the GitHub repository the specification is only available down to version 0.5.0, which also does not use the encoding generated by the compiler. Furthermore, some problems were encountered with programs compiled with this compiler. For C programs, the compiler generates an instruction that attempts to read a privileged register in user mode. Because this is an illegal operation, the program stops running when this instruction is executed. This compiler was therefore not used.

In order to still make use of benchmarks written in C, the regular RISC-V compiler can be used with the `.insn` assembler directive. This directive allows the fields of an instruction to be filled manually and is specifically meant to be used for custom instructions. For example, for an r-type instruction, like the `add` instruction, the instruction can be constructed as follows: `.insn r opcode, funct3, funct7, rd, rs1, rs2`. Initially, the design was implemented to comply with the encoding of version 0.5.2 of the P-extension to make use of the compiler by Andes Technology. Unfortunately, the opcode used by version 0.5.2 of the P-extension (`0x7F`) causes the assembler directive to misbehave and populate the fields of the instruction incorrectly. Therefore, the design was updated to use the encoding of version 0.9.2, which replaces the placeholder opcode of `0x7F` with `0x77` as well as modifies the encoding of some instructions. Using the `.insn` directive with the newer opcode does generate correctly encoded instructions.

In order to use the `.insn` assembler directive in a C program, inline assembly is used. The compiler allows variables from the C code to be used as inputs and outputs for the assembly instruction. By wrapping such an assembler directive in a function, the instructions from the P-extension can be called similarly to the intrinsic functions provided by a compiler with appropriate support. Intrinsics are functions that are built-in to the compiler and are often used to access SIMD instructions from a language like C or C++. Constructing the instructions with an `.insn` directive rather than a compiler with the proper support does have the downside that no automatic vectorization can be performed by the compiler. However, the compiler by Andes Technology that supports the P-extension only has very limited support for automatic vectorization. Therefore, manual utilization of the intrinsic functions would still be necessary. Furthermore, since the compiler has no knowledge of what operation is performed with the `.insn` directive it can not perform any optimizations. This would be possible when using a compiler with appropriate support. Apart from that, programming with `.insn` directives wrapped in a function works the same as with compiler intrinsics.

While working on this thesis, development for a compiler and assembler that have support for the P-extension has been going on. At time of writing, there are pull requests on the GitHub repositories for the official RISC-V versions of the compiler (GCC) and the assembler (Binutils-GDB) that support version 0.9.4 of the P-extension. However, as this is still under development and was made public after work on the benchmark programs already begun, this software was not used for this thesis.

4.2.2. Synthetic Benchmark

In order to investigate the ideal performance benefits of using the SIMD instructions of the P-extension, a benchmark with little overhead was constructed. One application that can make good use of SIMD instructions is matrix multiplication. This is because matrix multiplication is a computationally expensive operation that operates on regular data.

Making use of SIMD instructions can involve some overhead in making sure that the data is properly structured in the registers for the SIMD operations. However, due to the computational complexity of the applications, this overhead is compensated when performing the actual calculation by the parallelism afforded by the SIMD instructions. To show this in the benchmark, different matrix sizes are used. When multiplying smaller matrices, this overhead takes a more significant part of the execution time than when multiplying large matrices. Therefore, matrices with sizes from 4x4 to 256x256 were used in the benchmark.

Performing a matrix multiplication involves multiplying the elements in the rows of the first matrix with the elements in the columns of the second matrix and accumulating the results. A matrix multiplication of two 2x2 matrices has been illustrated in Equation 4.12. To effectively perform SIMD matrix multiplication, the data needs to be structured in such a way that optimal use can be made of the SIMD instructions without introducing too much overhead in the form of data movement. One way to do this is by transposing the second matrix before computation. By performing this transformation, the matrix multiplication can be performed by multiplying and accumulating the rows of both matrices, which are placed contiguously in memory. This SIMD multiplication is shown in Equation 4.13, by accumulating these multiplication results the top left element of the resulting matrix is obtained.

However, when multiplying small matrices the overhead of transposing the matrix might be larger than the speedup obtained by doing a SIMD computation. Therefore, it could be beneficial to use a different algorithm. Another way to perform the multiplication is to replicate the value from the first matrix and perform a SIMD multiplication with the elements in the rows of the second matrix. This effectively computes the result of multiple elements in the row of the resulting matrix in parallel. This SIMD multiplication is shown in Equation 4.14 where a partial result of the top row of the resulting matrix is computed.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a\cdot e + b\cdot g & a\cdot f + b\cdot h \\ c\cdot e + d\cdot g & c\cdot f + d\cdot h \end{bmatrix} \tag{4.12}$$

$$\begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} e & g \end{pmatrix} = \begin{pmatrix} a\cdot e & b\cdot g \end{pmatrix} \tag{4.13}$$

$$\begin{pmatrix} a & a \end{pmatrix} \begin{pmatrix} e & f \end{pmatrix} = \begin{pmatrix} a\cdot e & a\cdot f \end{pmatrix} \tag{4.14}$$

With SIMD computations, the amount of parallel operations that can be performed depends on the amount of bits that are used to represent each value. For example, the P-extension allows 4 16-bit additions or 8 8-bit additions to be performed in parallel. Multiplication is handled a little differently, where only 2 16-bit or 4 8-bit multiplications can performed, because the result of a multiplication uses twice the number of bits. Therefore, the amount of speedup that can be achieved depends on the size of the input data and the accumulator.

Since the P-extension focuses on fixed-point over floating-point arithmetic, fixed-point matrix multiplication is also benchmarked. These algorithms are implemented similar to functions from the CMSIS (Cortex Microcontroller Software Interface Standard) DSP library from ARM [67]. This library provides implementations for functions that are often used in DSP applications like filtering, matrix multiplication, transform functions, and more, which make use of ARM SIMD instructions. For Q15 matrix multiplication the library multiplies the Q15 inputs to a 2.30 result which are accumulated in a 64-bit accumulator. After all partial results are accumulated, this 34.30 value is then shifted right by 15 positions resulting in a 34.15 value, which is then saturated to a 1.15 result. The Q7 algorithm functions similarly but then with Q7 inputs, a 2.14 multiplication result, and a 32-bit accumulator. Since these algorithms perform a widening multiplication, the specific fixed-point multiplication instructions (`khm`) from the P-extension can not be used. Therefore, instead of 8 and 4 multiplications only 4 and 2 multiplications can be performed in parallel for respectively Q7 and Q15 data without using MAC instructions.

Matrix multiplication is especially suited to the use of MAC instructions, as it essentially performs a MAC operation for each element in the output matrix. To accelerate the matrix multiplication operation, the `smaqa` and `smalda` instructions are used for 8-bit and 16-bit matrix multiplication respectively. The `smaqa` instruction performs 4 8-bit multiplications and accumulates those results with a 32-bit accumulator. Moreover, since a 64-bit design is used, the `smaqa` instruction performs two of these operations, producing a result with 2 32-bit elements. This way, 8 8-bit elements can be processed with a single instruction, only requiring an addition of the two 32-bit elements after all elements are multiplied to obtain the final result. The `smalda` instruction, on the other hand, performs 4 16-bit multiplications and accumulates the results with a 64-bit accumulator. This therefore allows processing of 4 elements with a single instruction. These instructions can also be used to accelerate fixed-point matrix multiplication. The algorithms using MAC instructions all first compute the transpose of the second input matrix so that the MAC instructions can be applied without further overhead. The details for the benchmarked algorithms can be seen in Table 4.8.

4.2.3. Real-world Benchmark

While a synthetic benchmark shows an idealized scenario, the real world is often less ideal. This means that often overhead is present or that not all parts of a program can be accelerated by the SIMD instructions. Pro-

Table 4.8: Overview of the algorithms used in the synthetic benchmark.

Algorithm	Input precision	Accumulator precision	Notes
SISD	8, 16-bit	32, 64-bit	Accumulator precision does not affect parallelism
SIMD	8, 16-bit	16, 32-bit	Compute multiple results in parallel
SIMD Transpose	8, 16-bit	16, 32-bit	Compute transpose first
SIMD Transpose Widening	8, 16-bit	32, 64-bit	Compute transpose first
MAC SIMD Transpose Widening	8, 16-bit	32, 64-bit	Compute transpose first, use MAC instructions
Fixed-point SISD	Q7, Q15	32, 64-bit	Accumulator precision does not affect parallelism
Fixed-point SIMD	Q7, Q15	32, 64-bit	Compute transpose first
Fixed-point SIMD MAC	Q7, Q15	32, 64-bit	Compute transpose first, use MAC instructions

viding a benchmark that showcases a real-world application is therefore essential to show the impact that can realistically be expected by the newly added instructions, rather than the idealized performance of a synthetic benchmark.

Recently, a typical application for SIMD instructions has been machine learning. A popular way to perform machine learning is with a convolutional neural network (CNN) which is often used for image recognition. A CNN usually consists of multiple layers, the sequence of a convolutional layer followed by an activation function and a pooling layer is repeated multiple times and is ended by a fully connected layer [68]. The layers of a CNN are illustrated in Figure 4.10. The convolutional layers and activation function are there to extract features from the image. The pooling layer, on the other hand, reduces the dimensions of the data and thus reduces the amount of computation required. The final layer is the fully connected layer, which computes the final result. The convolution operation can be implemented by using matrix multiplications. Due to the computational complexity and the data parallelism of matrix multiplication, SIMD instructions can be applied efficiently. Furthermore, while CNNs previously mostly operated on 32-bit floating-point data, there has been a trend to use 8-bit fixed-point data when running on lower performance devices [32]. This makes that SIMD instructions, like those from the P-extension, can be used to speed up the computation for neural networks.

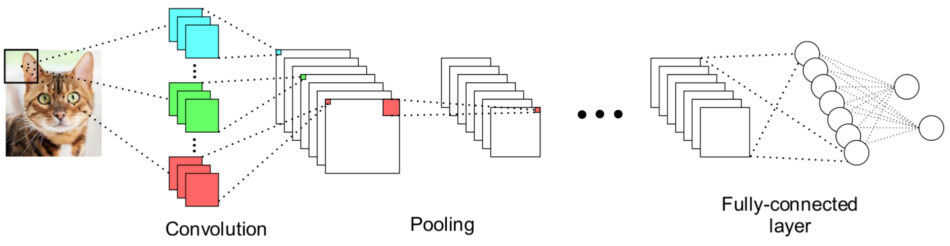


Figure 4.10: Overview of the layers of a CNN, taken from [68].

ARM's open-source library CMSIS includes functions to efficiently use their SIMD instructions for neural networks [67]. Furthermore, the library has been ported to RISC-V by making use of instructions from the P-extension and is renamed to NMSIS [69]. This library includes a convolutional neural network for image recognition using the CIFAR-10 data set. This data set contains 60,000 images with a resolution of 32 by 32 pixels that are assigned to one of ten classes [70]. Specifically, the ten classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The network provides a classification as well as the corresponding confidence level for a given input image. Moreover, this network operates on 8-bit fixed-point data. Therefore, 8 8-bit elements fit in a single register and thus allows 8 elements to be operated upon by a single instruction. The order in which the operations are applied to the data is listed in the enumeration below.

1. Preprocessing
2. Convolve RGB
3. ReLU

4. Maxpool
5. Convolve fast
6. ReLU
7. Maxpool
8. Convolve fast
9. ReLU
10. Maxpool
11. Fully connected layer
12. Softmax

The NMSIS library has been developed assuming that all instructions included in the P-extension are implemented. In Table 4.9 the SIMD instructions used by each operation is shown. All of these instructions are included in the basic subset of the P-extension with the exception of the `smaqa` instruction, which is an 8-bit MAC instruction. Even though the MAC instructions have been implemented, it is interesting to also investigate the achievable speedup when only the instructions from the basic subset are used. To be able to run the neural network with the basic subset, the operation performed by the `smaqa` instruction needs to be emulated using other instructions. As explained before, the `smaqa` instruction performs 8 8-bit multiplications which are accumulated in two 32-bit elements. These multiplication results then get accumulated with the 32-bit elements of the destination register. This operation can be split into a couple steps: 8-bit multiplication, sign extending the 16-bit multiplication results to 32-bit elements, and accumulation. The multiplication can be performed by two `smul8` instructions as those perform 4 8-bit multiplications, producing 4 16-bit results each. While 16-bit results are produced, the accumulation needs to be done with 32-bit precision. Since the P-extension does not provide an instruction that sign extends 16-bit elements to 32-bit elements, the sign extension needs to be done differently. By performing an arithmetic right shift meant for 32-bit element sizes by 16 positions, the odd elements get sign extended to 32-bits. To also sign extend the even elements, a `swap16` instruction is performed to swap the odd and even elements before applying another arithmetic right shift. Finally, these results can be accumulated using `add32` instructions. In total this takes 12 instructions rather than the 16 instructions (8 multiplications and 8 additions) needed without SIMD instructions. Moreover, since multiplication instructions take more than 1 cycle to complete, the reduction of 8 multiplications to 2 also benefits the SIMD implementation.

Table 4.9: SIMD instructions used by the operations.

Operation	SIMD instructions
Preprocessing	sclip32
Convolve RGB	sclip32, smaqa
ReLU	smax8
Maxpool	smax8
Convolve fast	sclip32, smaqa
Fully connected layer	sclip32, smaqa, zunpkd820, zunpkd831
Softmax	sclip32, uclip32

In the fully connected layer, a combination of AND, OR, and shift operations are performed to place elements in the proper order for processing. Specifically, the odd elements of two inputs are combined into one register and the even elements into another register. As the P-extension includes instructions for data movement, this operation can be performed more efficiently. The odd and even elements of a register can be extracted using the `zunpkd831` and `zunpkd820` instructions respectively. These instructions zero extend respectively the odd and even 8-bit elements to 16-bit elements. After extracting the desired elements, the odd or even elements of both inputs can be combined by shifting one of the results left by 8 positions and performing an OR operation. Optimizing the operation like this reduces the amount of instructions needed for this operation from

10 to 4.

The "convolve RGB" step performs matrix multiplication on matrices with 75 columns. Since this is not a multiple of 8, the columns are not fully aligned in memory. Therefore, when accessing, for example, the first 64-bits of data of the second column, a misaligned memory access is performed. This results in an exception that takes a significant amount of time to handle. In the original implementation the data is directly accessed via a 64-bit pointer. To prevent the misaligned memory accesses from occurring, the code was changed. Specifically a `memcpy` operation is performed. While this does introduce some overhead, it is significantly less than the overhead of handling the misaligned memory access exceptions.

In the convolutional neural network two different convolution steps are defined: "convolve RGB" and "convolve fast". The difference between the two variants is that the "convolve RGB" step is specifically optimized for 3 input channels. Since this is the first operation, after the scaling of the "preprocessing" step, the inputs are the RGB values of the image where each color is a channel. On the other hand, the "convolve fast" step is optimized for inputs with a multiple of 4 channels.

4.3. Conclusion

In this chapter, the implementations of both the hardware and software required to support the P-extension are discussed. The 268 instructions from the basic, MAC 8-bit, MAC 16-bit, and MAC 32-bit subsets have been implemented, this is 80.7% of the total 332 instructions included in the P-extension. The ALU is modified to support addition, subtraction, bit shifts, and comparisons in a SIMD manner. Furthermore, the ALU is extended to also support SIMD min and max, pack and unpack, and bit counting operations. Similarly, the multiplier has been modified to be able to perform signed and unsigned SIMD multiplication for 8-, 16-, and 32-bit element sizes. This has been realized by using a SIMD version of a Baugh-Wooley multiplier. Moreover, the multiplier has been extended to also support the MAC instructions included in the P-extension. Due to the structure of the SIMD Baugh-Wooley multiplier, sum together mode could be used to reduce the amount of required adders by performing part of the accumulation in the reduction tree of the multiplier. For further accumulation, SIMD adders were used to support generating a single 64-bit result or two 32-bit results. Furthermore, the modifications required to other components of the CVA6 processor like the decoder were also discussed. Slight modifications were made to the initial hardware design to optimize the critical path of the processor. This mainly involved reducing the amount of hardware reuse and increasing the latency of the MAC unit by one cycle. To make use of the hardware and to evaluate its performance, some software is also required. While a compiler with support for the P-extension is available, it did not work as expected. Therefore, the official RISC-V version of GCC was used with the `.insn` assembler directive. This directive allows manual construction of the fields of an instruction and is specifically meant for use with custom instructions. For evaluating the performance of the new designs, a synthetic and a real-world benchmark were constructed. The synthetic benchmark involves matrix multiplication and is meant to show the ideal performance gains of the SIMD instructions. Various different algorithms are implemented to investigate the optimal implementation. Since often not the whole application can be accelerated with SIMD instructions and overhead can be present, a more realistic real-world benchmark is also provided. For this, an image recognition neural network based on the CIFAR-10 data set is used. This benchmark was optimized and modified to also operate on a processor with support for only the basic subset of the P-extension. Although not the complete P-extension has been implemented, all instructions that could be used by the benchmarks are present in the implemented subsets.

5 Results

After the implementation has been completed, it is of course essential that the design is evaluated. First of all, the procedure for testing the design will be discussed. Subsequently, the verification that is performed on the design is described. Following that, the costs associated with implementing the required hardware is detailed in terms of the achievable clock speed, resource utilization, and energy usage of the design. Finally, benchmark results are presented of both the synthetic and the real-world benchmarks.

5.1. Test Procedure

Programs were tested both in simulation as well as on an FPGA. Simulation was performed with the Spike RISC-V ISA simulator version 1.0.1 which has support for version 0.9.2 of the P-extension. Furthermore, the HDL code of the design was simulated using Verilator version 4.028. The FPGA board used for testing the design is the Digilent Genesys 2. This board contains a Xilinx Kintex-7 FPGA, specifically with part number XC7K325T-2FFG900C, and 1 GB of DDR3 memory [71]. For synthesizing the design, Xilinx Vivado version 2020.2 was used and synthesis was performed with the "explore" directive. While using an operating system introduces some overhead and some variance in execution times, on the FPGA the benchmarks were run under Linux. This was done because being able to run an operating system like Linux is one of the defining features of the CVA6 processor. Therefore, it is likely that when using this processor, programs will be run under Linux to ease development over running an application without an operating system. On startup, the bootloader and the Linux kernel are loaded into memory from an SD card. The tests and benchmark programs were placed on the second partition of the SD card that can be mounted after the operating system has finished booting. Since the CVA6 processor does not provide a display output, communication was done using a serial connection over USB to provide a command-line interface. To measure execution times on the FPGA, the `cycle` hardware counter was used. This counter counts the amount of clock cycles that have been executed. Furthermore, since there is some variance to the execution times when running on the FPGA, benchmarks were ran 10 times and the execution times were averaged.

5.2. Verification

As discussed in Section 2.4, it is essential to make sure that the implemented design adheres to the specification. Due to the complexity of the system, it is unfeasible to test all possible states that the system can assume. To still make sure that everything works as intended, multiple types of tests are performed. These commonly consist of manually created tests, legacy tests or a commercial test suite, a complex application, and randomly generated tests [47]. Due to the fact that the RISC-V instruction set and especially the P-extension are relatively new, no legacy or commercial test suites exist. With the exception of the commercial test suite, all other methods were used to verify the functionality of the design. The manually and randomly generated tests will be discussed in the next two paragraphs. Moreover, the manually created tests were used as a regression suite to make sure that no existing instruction was broken while working on new instructions. The applications, in the form of benchmarks, that are run have been discussed in Sections 4.2.2 and 4.2.3.

5.2.1. Manually Generated Tests

Testing of the design was done using Verilator to perform the simulation and for generating VCD files [72]. These VCD (value change dump) files were than opened in GTKWave to view the waveforms for debugging [73]. Verilator converts and optimizes the SystemVerilog source code into C++. This C++ model can then be used in another C++ program to apply signals to the design and to record the outputs that the design generates. The modified modules in the design, the ALU and multiplier, were first tested independently and afterwards tested while integrated in the whole processor.

A testbench was created for applying signals directly to the ALU and multiplier. To make it easy to create and check tests, a function was created that applies the specified inputs and compares the generated output with the provided expected output. For instructions that perform saturation, another function was made that also checks whether the `vxsat` register gets set when it should. When the actual output and the expected output differ, an error is thrown so that the point of failure is made explicit and can be investigated. The benefit of testing the modules separately is that simulation of one module is much faster than when simulating the whole processor. Additionally, any bugs that are found can be attributed to the module itself rather than the interaction with the different components of the whole CPU.

The CVA6 repository includes a Verilator model that is able to run a RISC-V ELF file, the standard executable file for Unix systems. This can be used in conjunction with the ISA unit tests from the `riscv-tests` repository to check its functionality [74]. The tests from this repository also work by providing the inputs and the expected output for an instruction. When the actual output is not the same as the provided expected output, the program is terminated and the returned error code corresponds to the test that has failed. Special cases are also tested like using the zero register (`x0`) as one of the inputs, both of the inputs, or as the destination register. Furthermore, there are test cases where the same register is used for the first operand and the result, the second operand and the result, or for both operands and the result as well as test cases where no-ops are introduced in various positions in the test case. The `riscv-tests` repository does not yet contain tests for the instructions from the P-extension and these have therefore been created manually. Additionally, test cases were introduced for the saturating instructions to check that the saturation status register gets set when appropriate.

To make sure that the functionality during simulation and after synthesis is the same, the tests have also been run on the FPGA. However, `riscv-tests` relies on the RISC-V frontend server (fesvr) to handle system calls, while they are handled by the Linux operating system for the processor running on the FPGA. Therefore, to make the tests run on the FPGA, these system calls needed to be modified. When a test case fails, it jumps to the label `fail` where it should exit the program and return the number of the failing test as the return code. This has been ported by loading the test number in register `a0` and the value 93 in register `a7`, followed by an `ecall` instruction to make a service request to the operating system. The value 93 tells Linux to terminate the program with the return code specified in register `a0`. When no failing test is encountered, the value 0 is loaded in register `a0` to signal a successful completion, for a failing test the test number is loaded in register `a0` instead. Due to the large number of tests that need to be run, one for each new instruction, a small shell script was written to conveniently run all tests. This script runs all tests and checks the return value to make sure that every test is completed successfully.

5.2.2. Randomly Generated Tests

In order to perform randomized testing, first random programs need to be generated. For this, the CVA6 project uses the `riscv-torture` project, which is written in the Scala programming language [75]. This program works by first registering valid test sequences that operate on unspecified registers and data. These test sequences are assigned to a specific category like an ALU instruction, a memory instruction, a branch instruction, and so on. Based on a distribution that can be configured, random sequences are chosen proportionally from the different categories to roughly satisfy the requested distribution. Additionally, registers and immediate values get assigned randomly. At the end of the test program, the contents of the registers are written to a specific memory location. The RISC-V frontend server, which is integrated in the RISC-V ISA simulator (Spike) and the testbench of the CVA6 processor, accepts an argument (`+signature`) so that the data in this memory location gets written to a specified file. While the testbench of CVA6 does support the `+signature` argument, a small bug had to be fixed with regard to the handling of the command-line arguments passed to the testbench. By then comparing the signatures from running the test program on the ISA simulator and the simulation of the CVA6 HDL code, the created design can be verified. It is important to note that this process assumes that the instruction set simulator correctly implements the instructions of the specification and does not contain any errors. Since the generation of the signatures relies on a feature of the simulators, these tests have not been run on the FPGA.

The random test program generator was extended with instructions from the P-extension so that the functionality could be verified. These instructions were added in the form of three categories separate from the already existing categories, ALU instructions, multiplication instructions, and MAC instructions. This consequently allowed the generation of test programs that contain a significant amount of instructions from the P-extension.

As stated in Section 4.2.1, the compiler and assembler that were used have no support for the instructions of the P-extension. Therefore, the `.insn` assembler directive was wrapped in a function to create functions similar to intrinsics that implement the instructions of the P-extension. The generator creates assembly programs and therefore allows direct use of the `.insn` directive. While it likely would have been possible to modify the generator to generate programs using the `.insn` directive, it was implemented such that it generates the regular assembly instructions as defined by the specification. This was done because eventually the compiler and assembler will be updated, which would then allow the generator to be used without modifications. Instead, a small Python script was used to convert the assembly instructions from the P-extension with `.insn` directives using the instruction encodings specified in the `riscv-opcodes` repository [76].

5.3. Synthesis Results

Naturally, the newly introduced instructions by the P-extension come at a cost. The additional hardware that is required to perform these operations not only has an impact on resource utilization but possibly also on the critical path, power usage, and energy usage of the design. These effects will be discussed in the following sections.

5.3.1. Critical Path

All this additional hardware increases the propagation time between registers. The modifications made to the design have mainly been constrained to the ALU and the multiplier. Since the original critical path is found in the floating-point unit rather than the ALU or multiplier, there is some headroom before the additional hardware would impact the maximum achievable clock frequency. The original design of the CVA6 processor runs at a clock frequency of 50 MHz or a clock period of 20 ns on the Genesys 2 FPGA board by default. The new hardware used to implement the functionality of the P-extension initially did not meet the timing requirements. As stated in Section 4.1.5, changes were made to make the design meet the timing requirements for a clock period of 20 ns. Although the multiplier is pipelined such that it has two cycles to perform its operation, the addition of the MAC unit shifted the critical path from the FPU to the MAC unit. To mitigate this critical path, a modified design was created to give the MAC unit 3 cycles of latency instead of 2. As the MAC unit now has more time to complete its operation, the critical path was moved from the MAC unit back to the FPU, like the other two designs. While this increased latency results in an increased execution time in terms of cycles, the execution time in seconds should be decreased as a higher clock frequency is achievable.

As the designs still had some slack on the timing constraint at 50 MHz, the designs were pushed to higher clock frequencies. The design with support for the basic subset and a MAC unit with 2-cycle latency was able to hit a clock frequency of 55 MHz. Furthermore, for the clean CVA6 processor, the design with the basic subset, and the design with the basic subset and a MAC unit with 3-cycle latency, pushing the clock speed lead to a frequency of 70 MHz. For these designs the clock speed is limited by the delay of the FPU. Since many processors of this class do not have an FPU and because the P-extension emphasizes fixed-point over floating-point arithmetic, the designs were also synthesized without the FPU. With this optimization, the clock speed increased to 75 MHz for the design with the basic subset. The clean design, on the other hand, does not reach 75 MHz and fails with a negative slack of 0.002 ns. However, by preventing Vivado to infer DSP blocks for the multiplier, the design is able to pass at 75 MHz. For the design with support for the MAC instructions, the critical path was moved back to the MAC unit. Unfortunately, the delay of the MAC unit is too large for the design to hit a clock speed of 75 MHz, even with a 3-cycle latency.

5.3.2. Area Usage

Since for this thesis the design has been synthesized for an FPGA, the resource utilization is represented in terms of the hardware blocks present on a Xilinx FPGA like lookup tables (LUTs), flip-flops (FFs) etc. The lookup table blocks can then be further specified in lookup tables used to implement logic (logic LUTs) or lookup tables used as synchronous RAM (LUTRAMs). The resource utilization for the CVA6 processor, the CVA6 processor with support for the basic subset of the P-extension, and the CVA6 processor with support for the basic subset and the MAC subsets both with 2- and 3-cycle latency are summarized in Table 5.1. This shows that to implement the basic subset of the P-extension 11.2% extra LUTs and 0.07% extra flip-flops are used. Implementing the MAC instructions with 2-cycle latency causes 2195 additional LUTs and 90 additional flip-flops to be used over the basic subset. This corresponds to an increase of 3.0% in the number of LUTs and 0.2% extra flip-flops. When instead comparing to the design without SIMD instructions, 14.5% additional LUTs and 0.25% additional flip-flops are used. By changing the design to give the MAC unit a 3-cycle latency, 632 less LUTs and 166 additional flip-flops are used compared to the version with a 2-cycle latency. Comparing this design to the clean design gives a 13.5% and 0.58% increased LUT and flip-flop usage respectively.

Table 5.1: Resource utilization of the designs on the Digilent Genesys 2 FPGA board.

	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP Blocks
CVA6	66510	64950	1192	368	50342	49	2	27
CVA6 Multiplier Without DSP	70429	68869	1192	368	50349	49	2	11
CVA6 + Basic P-ext	73948	72388	1192	368	50376	49	2	11
CVA6 + Basic P-ext + 2-cycle MAC	76143	74583	1192	368	50466	49	2	11
CVA6 + Basic P-ext + 3-cycle MAC	75511	73951	1192	368	50632	49	2	11

From the table, it can be observed that the implementations that include the P-extension use 16 less DSP blocks than the unmodified CVA6 processor. In the unmodified design, the multiplication operation gets mapped to the DSP blocks by the synthesizer, which contain fixed function multipliers. Since the designs with instructions from the P-extension implement a SIMD multiplier, it can not be mapped to the DSP blocks and uses LUTs instead. While the comparison is valid when using an FPGA, when producing the design in silicon no pre-existing DSP blocks are available. Therefore, the clean design was also synthesized with the constraint that no DSP blocks would be inferred for the multiplier. This lead to a design with 70429 LUTs, 50349 flip-flops, and 11 DSP blocks. Comparing the designs with the P-extension to this, results in a 5.0% and 7.2% increase in LUTs and a 0.05% and 0.56% increase in flip-flops for respectively the design with the basic subset and the design with the basic and MAC subsets with a 3-cycle latency.

To achieve a higher clock frequency the designs were tested with the FPU disabled, which naturally means less resources will be used. A reduction in LUT and flip-flop usage of respectively 12687 and 4353 was seen for the clean design. For the design with support for the basic subset of the P-extension, 11754 less LUTs and 4350 less flip-flops are used. Furthermore, for both designs no DSP blocks are used anymore. This shows that a floating-point unit consumes a significant amount of area, justifying the omission of an FPU on many processors. The resource utilization for designs that were able to hit 75 MHz without the FPU are summarized in Table 5.2.

Table 5.2: Resource utilization of the designs with the FPU disabled on the Digilent Genesys 2 FPGA board.

	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP Blocks
CVA6 Multiplier Without DSP	57742	56182	1192	368	45996	49	2	0
CVA6 + Basic P-ext	62194	60634	1192	368	46026	49	2	0

5.3.3. Power and Energy Consumption

As new hardware is added to the design, it is expected that the maximum power consumption is increased. However, this additional hardware is used to implement instructions that reduce the total execution time. Therefore, the energy consumption to run a task that benefits from SIMD instructions is likely reduced. Unfortunately, due to time and resource constraints, power and energy consumption have not been measured.

While Vivado can estimate the power consumption of the design after synthesis, the results were unrealistic. Specifically, the estimated power consumption goes down when more hardware is added. The power estimate of Vivado can be improved by providing a SAIF file which specifies the switching activity of the internal signals. The used simulator, Verilator, does not support generating SAIF files. While QuestaSim could be used to generate a SAIF file, running a benchmark program on the simulator would take a very long time. The Genesys 2 FPGA board does include power monitoring ICs that are interfaced through I^2C. However, on the Genesys 2 board no header is soldered to the I^2C bus and because of the current situation no equipment was available to solder on this header. Therefore, these ICs could also not be used to measure the power and energy consumption of the design.

5.4. Benchmark Results

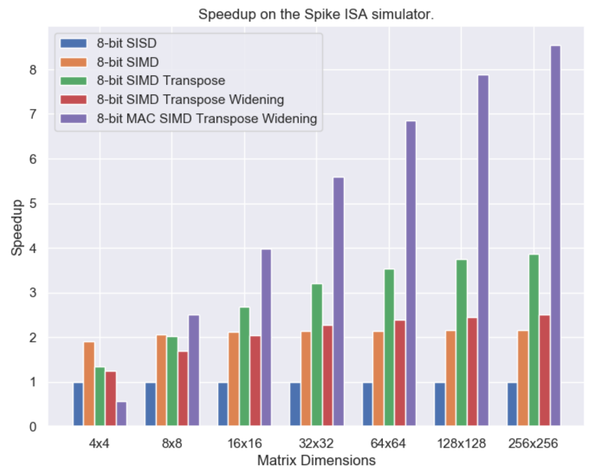
As stated before, the benchmarks as presented in Sections 4.2.2 and 4.2.3 are run both on Spike, the RISC-V instruction set simulator, and on the FPGA under Linux. First, the results of the synthetic benchmark, involving matrix multiplication, will be presented. This is followed by the results of the real-world benchmark, which is a convolutional neural network for image recognition. The benchmarks that require MAC instructions have been run on the design with a 3-cycle MAC unit. Due to the increased latency of the MAC unit, on average, 5.0% and 0.9% extra cycles are needed for respectively the synthetic and real-world benchmarks. The comparatively lower impact on the real-world benchmark can be attributed to the fact that it performs multiple multiplication or MAC instructions consecutively. Since the multiplier is fully pipelined, the instructions can be executed right after each other and thus the impact of the added latency is limited. The synthetic benchmark, on the other hand, performs an addition on the multiplication result after the multiplication. Therefore, an additional cycle is required before this addition can be executed. However, as a clock speed of 70 MHz can be obtained for the 3-cycle design rather than 55 MHz with a 2-cycle latency, a speedup in execution time is seen of, on average, 1.21x and 1.26x when using the 3-cycle design for respectively the synthetic and real-world benchmarks. As the design with support for just the basic subset does reach 70 MHz with a 2-cycle multiplier, that design is used to run the benchmarks that do not use MAC instructions. This is done to show the optimal performance of both the design with the basic subset of the P-extension as well as the design with the basic subset extended with the instructions of the MAC subsets.

5.4.1. Synthetic Benchmark

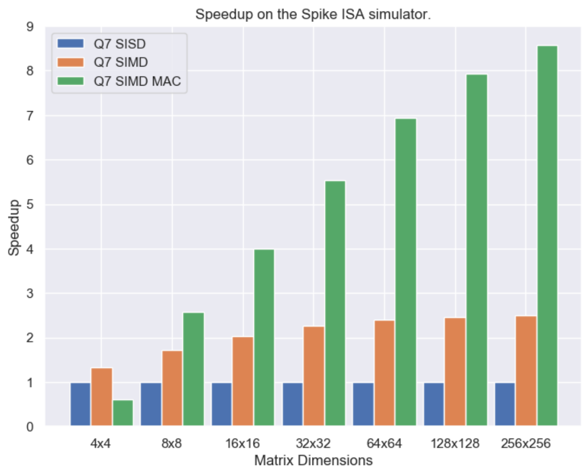
Matrices can take any number of sizes. To give insight in how the different algorithms perform, they are benchmarked operating on matrices of differing sizes. For the benchmarks, matrix dimensions ranging from 4x4 up to 256x256 are tested. The limit was set at 256x256 because execution times on the FPGA saw a large increase at this size and this effect would be exacerbated for 512x512 size matrices. Since a different amount of parallelism is obtained for different element sizes, benchmarks were performed with matrices with 8-bit and 16-bit elements. No benchmarks were run with 32-bit elements since only one 32-bit multiplication can be performed in a single cycle, as this operation produces a 64-bit result of which only one fits in a register. Therefore, only very little possible performance gain is achievable for 32-bit data using the basic subset. Although, 32-bit MAC instructions could be used to perform two multiplications and an accumulation with a single instruction. Due to the widening nature of the multiplication operation, for 8-bit and 16-bit multiplication, 4 and 2 operations can respectively be performed in parallel. This potential speedup is reduced somewhat by the overhead introduced by structuring the data to make use of the SIMD instructions. However, some more parallelism can be obtained, depending on the accumulator accuracy, by also utilizing SIMD instructions for accumulation. Furthermore, using MAC instructions a larger speedup can also be obtained.

The results are shown for each algorithm separately, details on the implementation of the various algorithms can be found in Section 4.2.2. In Figures 5.1 and 5.2 the benchmark results from Spike, the instruction set simulator, are shown respectively for 8-bit and 16-bit element sizes. With the basic subset of the P-extension, speedups of up to 3.87x and 2.50x are seen respectively for 8-bit and Q7 elements and up to 1.98x and 1.39x for respectively 16-bit and Q15 elements. Moreover, with the inclusion of MAC instructions a speedup of up to 8.54x for 8-bit elements and 8.57x for Q7 elements is achieved, for 16-bit elements and Q15 elements the speedups are 4.55x and 4.53x respectively.

For all algorithms, it can be seen that the speedup increases with the dimensions of the input matrices. This is because the overhead present in using SIMD instructions gets less dominant in the overall execution time for larger matrices. However, this effect is seen especially strongly for the algorithms that first compute a transpose of one of the input matrices. For small matrices the transpose operation results in a larger overhead, while for the larger matrices the overhead is smaller than for other algorithms. This is because after performing the transpose, no more data movement is necessary to perform the computation in a SIMD manner. Therefore, a larger speedup can be seen for algorithms with transpose over other SIMD algorithms when operating on larger matrices. Another observation that can be made is that the widening algorithms are slower than the regular ones. This is caused by the fact that in the widening algorithms the accumulator is four times the size of the input size rather than two times. Because of this, the accumulation can not be performed with the same level of parallelism and thus results in a slower but more precise algorithm. Moreover, the speedup of the transpose widening algorithm is nearly identical to the speedup of the SIMD fixed-point algorithm. This is expected because both algorithms use accumulators with the same precision and therefore have the same level of parallelism. Finally, it can be observed that the algorithm using MAC instructions for 8-bit and Q7 elements is slower than the SISD algorithm for 4x4 matrices. This is caused by the fact that the used `smaqa` instruction processes 8 elements at a time. Since for a 4x4 matrix only 4 multiplications are performed for each element in the output matrix, the `smaqa` instruction can not be applied. Therefore, the computation is done in a SISD manner. However, the transpose operation and the check if 8 elements still need to be processed result in the algorithm being slower than the normal SISD algorithm. The 16-bit and Q15 algorithms using MAC instructions are unaffected by this issue because the used `smalda` instruction processes 4 elements at a time, which is exactly the amount of elements that need to be processed for a 4x4 matrix.



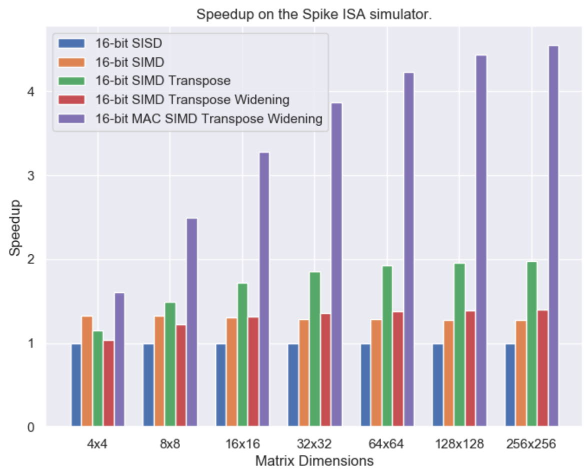
(a) 8-bit integer results



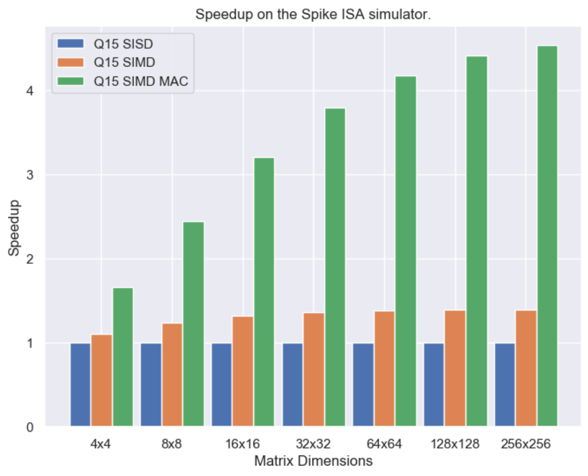
(b) Q7 fixed-point results

Figure 5.1: Results of running the synthetic benchmark on Spike, the RISC-V ISA simulator, with an 8-bit element size.

The synthetic benchmark is also run on the FPGA under Linux. In Figures 5.3 and 5.4 the results for respectively 8-bit and 16-bit element sizes are shown. This shows a speedup of up to 8.8x for 8-bit elements, 7.2x for Q7 elements, 4.9x for 16-bit elements, and 3.8x for Q15 elements when using the basic subset of the P-extension. Utilizing the MAC instructions increases this speedup to 12.3x for 8-bit elements, 12.6x for Q7 elements, 6.7x for 16-bit elements, and 6.5x for Q15 elements. From the results it can be seen that for matrix dimensions up to 128x128 the speedup when running on the FPGA is slightly worse than the simulator suggests. However, this behaviour is expected as the instruction set simulator performs all operations in a single cycle. On the FPGA however, not everything can be done in a single cycle. For example, multiplication instructions always take multiple cycles, accessing the data cache takes three cycles, and access to the DRAM takes approximately 50-100 cycles. Furthermore, some deviations in the overall trend in the speedup can be seen. For example, the speedup for Q15 4x4 matrices is higher than than expected both with the basic subset and with MAC instructions. These outliers are likely caused by the operating system performing a background task. Since a single core design is used, the benchmark needs to be paused to perform this task. However, since the cycle counter continues counting, an increased execution time is observed. Even though these results are averaged over 10 runs, some outliers can still skew the results as shown in these specific measurements. This is especially the case for benchmarks with a short execution time, since the execution time for the background task is large relative to the execution time of the benchmark.



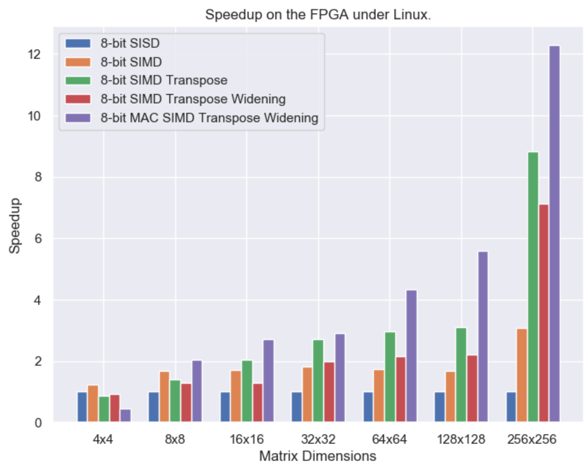
(a) 16-bit integer results



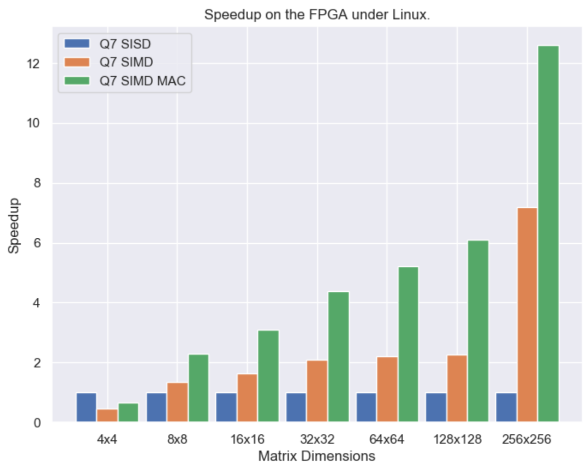
(b) Q15 fixed-point results

Figure 5.2: Results of running the synthetic benchmark on Spike, the RISC-V ISA simulator, with an 16-bit element size.

These results show that when MAC instructions are available, using them provides the largest speedup for almost every matrix size. When only the basic subset of the P-extension is available, the conclusion is a little different. For small matrices it can give a better result to use an algorithm without transpose, as the overhead of performing a transpose can be larger than the speedup gained with the SIMD instructions. For example, for 8-bit matrix multiplication for matrices up to 8x8 in size, the algorithm without transpose gives a larger speedup than the algorithm that includes a transpose operation. However, for matrices larger than 8x8 in size, the algorithm with transpose is the faster option.



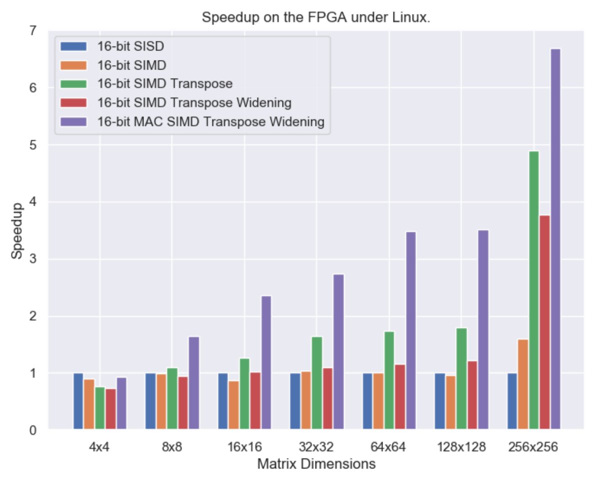
(a) 8-bit integer results



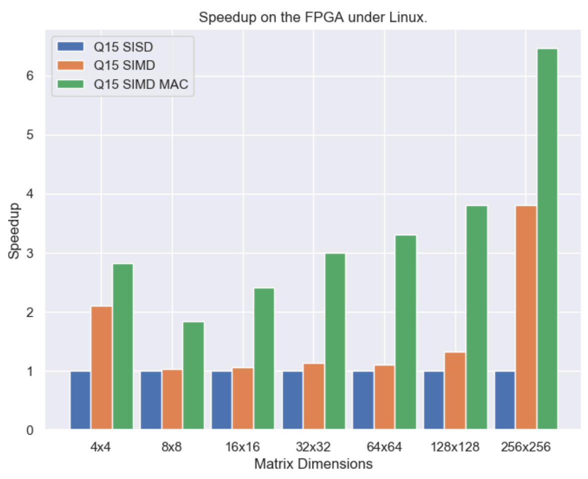
(b) Q7 fixed-point results

Figure 5.3: Results of running the synthetic benchmark on the FPGA under Linux with an 8-bit element size.

Another deviating result is seen for 256x256 matrices when running on the FPGA. The speedup of the SIMD algorithms over the SISD algorithm increases quite significantly when going from 128x128 to 256x256 size matrices. Moreover, the expected increase in execution time when doubling the dimensions of the matrices should be a factor of 8 due to the $O(n^3)$ computational complexity of matrix multiplication. However, an increase in execution time with a factor of around 37, 20, 12, and 17 is seen respectively for the SISD, SIMD, transpose SIMD, and SIMD MAC algorithms. The reason for this large increase in execution time is the size of the data cache. The CVA6 processor makes use of an 8-way set associative data cache with a size of 32 KiB and a random replacement policy. Since a 256x256 matrix with 8-bit elements takes $\frac{256*256*8}{8} = 64\ KiB$, the



(a) 16-bit integer results



(b) Q15 fixed-point results

Figure 5.4: Results of running the synthetic benchmark on the FPGA under Linux with an 16-bit element size.

data to work on becomes larger than the cache. Therefore, the data has to be fetched from the memory rather than the cache, resulting in more cycles taken to perform a load operation.

By using the hardware performance monitor counters (`hpmcounter`) present in the processor, specific hardware events can be tracked. The RISC-V privileged specification calls for 32 of these counters of which the events they count is only specified for the first two [22]. What event is tracked by the other counters can be freely chosen by the design. These first two counters count the number of cycles that have been executed (`mcycle`) and the number of instructions that have been retired (`minstret`). The letter m as a prefix to the names of the counters specifies that they are only accessible from the machine privilege mode. Since the benchmark runs in user mode, the program is not allowed to access these counters. However, read-only shadows are provided for these counters that are accessible from user mode. Unfortunately, these read-only shadows were not completely implemented in the CVA6 processor. However, this was easily fixed by adding the addresses of the user mode shadows to the read operation of the CSR register file. As stated before, the `cycle` counter is used to measure the execution time of the benchmarks. On the CVA6 processor, some of the remaining counters have been set to count the instruction cache misses, data cache misses, load operations, and store operations using counters 3, 4, 7, and 8 respectively.

Using these counters, the cache miss rate was measured for the algorithms at the different matrix dimensions. The miss rate was computed as follows $miss\ rate = \frac{icache\ miss + dcache\ miss}{total\ loads + total\ stores}$. In Figures 5.5 and 5.6 the miss rates can be seen for each algorithm at the different matrix dimensions. The miss rates for 4x4 matrices are significantly higher than, for example, for 8x8 matrices. For these small matrices the number of misses is dominated by the instruction cache misses, while these are negligible for larger matrices. Combined with a small amount of load and store operations, this leads to a relatively large miss rate for these small matrices. As expected, the figures show a large increase in the miss rate when operating on matrices with a size of 256x256. Specifically, for 256x256 matrices the miss rate of the scalar algorithm is around 40% rather than around 11% for the SIMD algorithms operating on 8-bit elements, for 16-bit elements this is respectively around 48% and 14%. To explain this difference, a further look at the used algorithms is required.

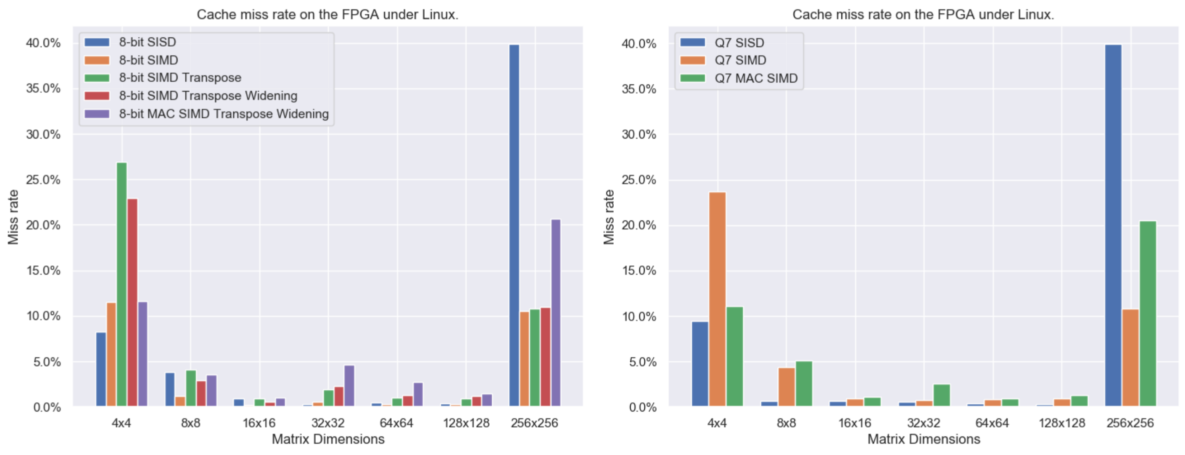
The algorithms without transpose compute multiple elements in the resulting matrix in parallel. To do this, the data from a single load instruction is reused. Since this data would not remain in the data cache when the computation comes around for a SISD algorithm, this leads to a reduction in misses. Specifically, 4 times less for 8-bit algorithms and 2 times less for 16-bit algorithms, as that is the amount of elements that are operated upon in parallel. Although the reduction in the amount of misses is different, this still leads to a similar miss rate for 8- and 16-bit algorithms since the amount of loads reduces more for 8-bit algorithms than for 16-bit algorithms.

For the algorithms that do use a transpose, all elements that need to be operated upon are placed contiguously in memory. Furthermore, as the cache lines in the CVA6 processor are 128-bits wide, a single miss loads 128-bits of data in the cache. This corresponds to 16 8-bit elements or 8 16-bit elements. This explains the reduction in cache misses of around 16 and 8 for respectively the 8- and 16-bit algorithms. A similar miss rate for 8- and 16-bit algorithms is seen, again due to the larger reduction of loads when operating on 8-bit elements rather than 16-bit elements.

While the algorithms without transpose have a lower reduction in misses than the algorithms with transpose, the observed miss rate is approximately the same. This is caused by the increased amount of store instructions used by the algorithms without transpose. These instructions are used to organize the elements correctly in a register. Moreover, this difference explains the fact that the execution time scales with a factor 20 for algorithms without transpose rather than a factor 12 for algorithms with transpose when going from 128x128 to 256x256 size matrices.

Furthermore, the algorithms using MAC instructions have a miss rate that is roughly double that of the SIMD algorithms with transpose. Since the MAC algorithms also perform a transpose before starting the computation, the same reduction in cache misses is seen. However, because the MAC instructions perform 8 8-bit or 4 16-bit multiplications instead of 4 8-bit or 2 16-bit multiplications for the SIMD multiply instructions, 64-bits of data needs to be loaded at a time rather than 32-bits. Therefore, the total amount of loads that are performed is halved. With the same amount of misses and a halved amount of loads, the miss rate is roughly doubled.

This therefore explains the increase in execution time when moving to 256x256 matrices and why this increase is less significant for the SIMD algorithms. While the increase in execution time is not seen in Spike, when the cache is configured correctly, similar miss rates are seen. However, this does not affect the execution time because Spike assumes that every instruction completes in one cycle, even a load instruction with a cache miss.



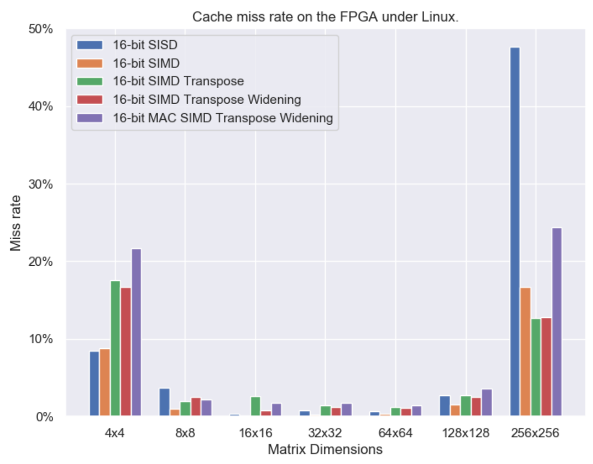
(a) 8-bit integer

(b) Q7 fixed-point

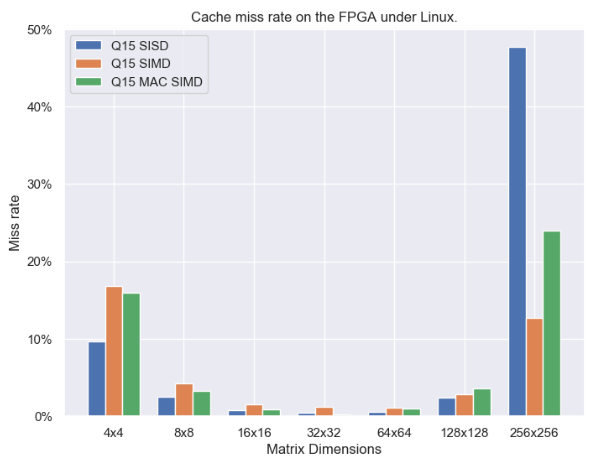
Figure 5.5: Cache miss rate operating on 8-bit elements, measured using the hardware performance monitoring counters on the FPGA under Linux.

5.4.2. Real-world Benchmark

While the results shown in the previous section show the ideal performance that can be obtained by using the instructions of the P-extension, the real-world is often less ideal. Therefore, a real-world benchmark is also used. As previously mentioned, the real-world benchmark consists of an image recognition neural network. For this benchmark, multiple different images are tested. The results of running the benchmark on the instruction set simulator can be seen in Figure 5.7. A speedup of around 2.4 times is seen when utilizing the basic subset of the P-extension, while the addition of MAC instructions results in a speedup of around 3.7 times over the implementation without SIMD instructions. The addition of the MAC subsets thus provides an additional speedup of around 1.6 times compared to just the basic subset. Moreover, these benchmark



(a) 16-bit integer



(b) Q15 fixed-point

Figure 5.6: Cache miss rate operating on 16-bit elements, measured using the hardware performance monitoring counters on the FPGA under Linux.

results show that there is very little difference in speedup between the different test images. Therefore, the "horse 1" test image was used for the following benchmark results.

The level of speedup that is attainable is different for each of the steps of the neural network since they perform different operations, this is illustrated in Figure 5.8. The figure shows that the operations that benefit from the introduction of the MAC instructions are the convolution steps and the fully connected layer. While this makes up only a third of the steps of the whole computation, these steps comprise around 96% of the total execution time when not using the P-extension, as shown in Table 5.3. Combined with the large speedup that is achieved with the MAC instructions in these steps, the execution time is reduced significantly. Something else that can be noticed from Figure 5.8 is that when using the P-extension, the preprocess step is slower than when it is performed in a SISD manner. In this step, the image data is scaled and saturated to the appropriate fixed-point format. Due to the scaling that is applied, the result can not overflow and thus saturation never occurs. In the SISD case, the compiler detects this and optimizes the saturation operation away. When using the P-extension, the saturation operation is implemented using a `clip` instruction. As stated before, due to lacking compiler support, the instructions from the P-extension are inserted as a manually constructed assembly instruction using a `.insn` directive. Therefore, the compiler is unaware of what operation is performed by these instructions and can thus not perform the same optimization. However, as the effect of this optimization is fairly small and since the preprocess step is only a very small part of the overall execution time, less than 0.1%, the impact is negligible. As the instructions of the basic subset process at most 8 elements at a time, the relu and maxpool steps are performed almost ideally in parallel with a speedup of a little less than 8x. This is because both steps rely on the `smax8` instruction which processes 8 elements at a time. However, due to their small part in the overall execution time, the effect of this speedup is limited.

Another noteworthy observation that can be made is that the speedup seen for the "convolve fast" steps reaches up to around 12 times with MAC instructions, while the "convolve RGB" step only sees a speedup of up to 2.4 times. This is caused by the dimensions of the matrices that are multiplied in these steps. The "convolve fast" steps perform matrix multiplication on a matrix with 800 and 400 columns for the first and second pass of "convolve fast" respectively. On the other hand, the "convolve RGB" step performs matrix multiplication on a matrix with 75 columns. As the `smaqa` instruction processes 8 inputs at a time, the matrix multiplication of the "convolve fast" steps can be completely performed in a SIMD manner. However, since the amount of columns for the "convolve RGB" step can not be cleanly divided by 8, the remainder needs to be computed in a SISD manner. Consequently, the amount of speedup that can be achieved by SIMD instructions for the "convolve RGB" step is limited.

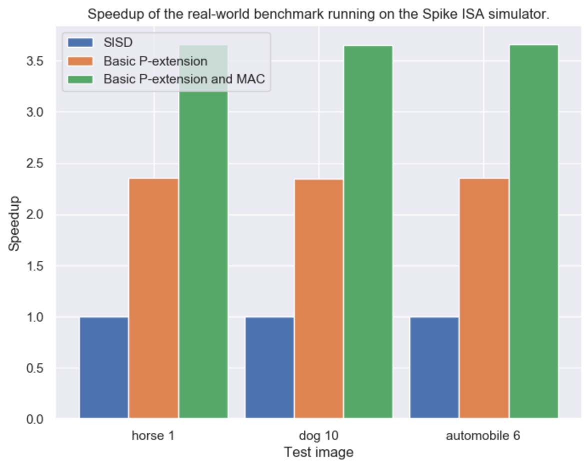


Figure 5.7: Speedup when using the SIMD instructions of the P-extension for the different images in the Cifar-10 benchmark running on Spike.

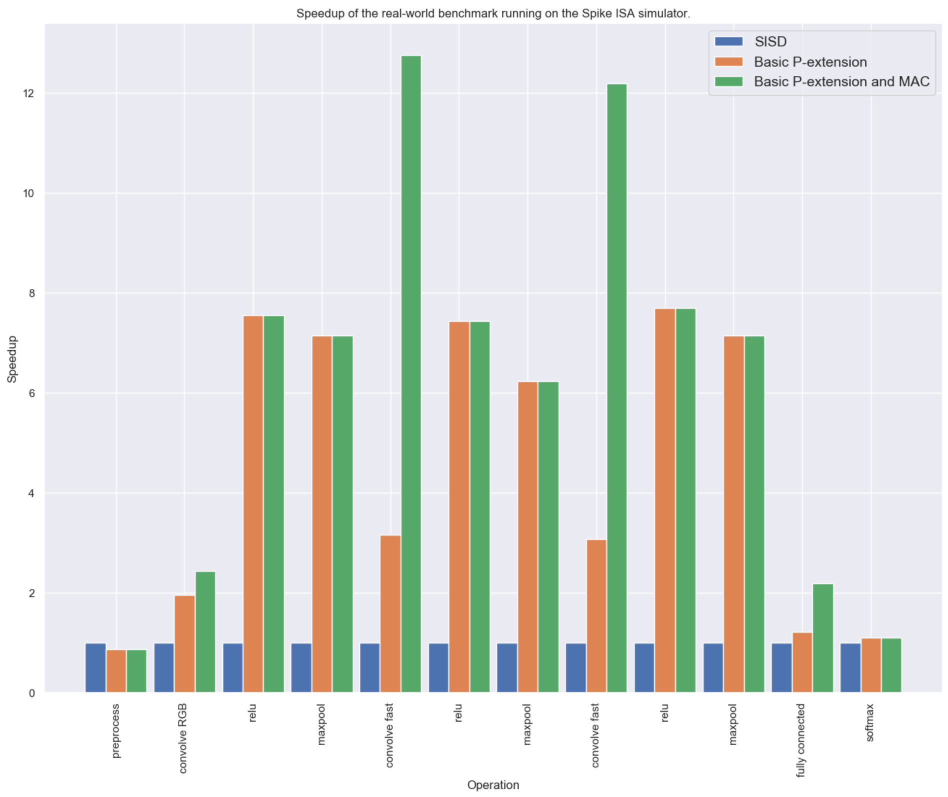


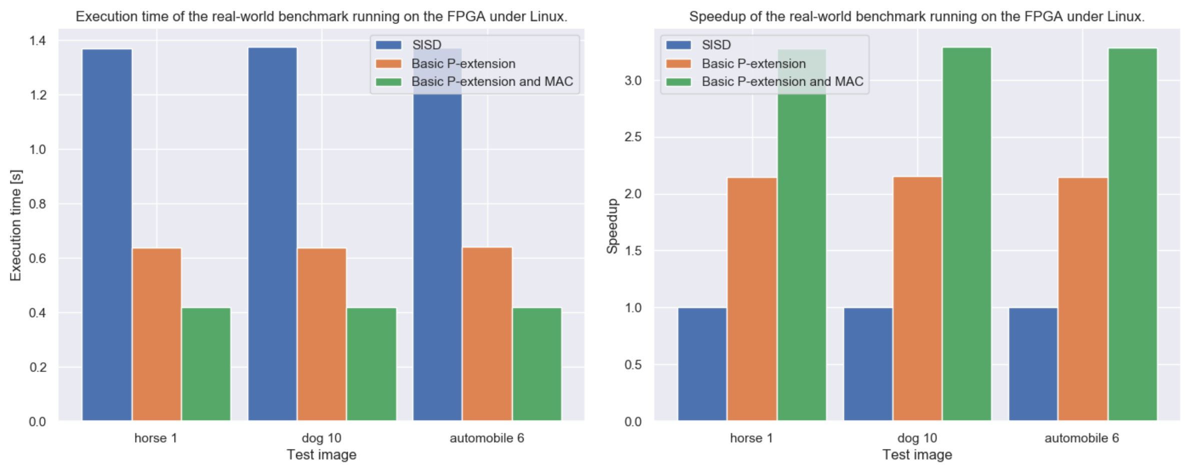
Figure 5.8: Speedup of the separate operations when using the SIMD instructions of the P-extension in the Cifar-10 benchmark running on Spike.

In Figure 5.9, the execution time and speedup achieved in the real-world benchmark is shown when running on the FPGA. The execution time ranges from 1.37 seconds without the P-extension, to 0.64 seconds with the basic subset, and to 0.42 seconds with the basic and MAC subsets. The speedup when running the real-world benchmark on the FPGA is around 2.1 times when using the basic subset of the P-extension and around 3.3 times when also using MAC instructions. Like in the synthetic benchmark, there is little difference in execution time and speedup between the different test images. Therefore, in the following benchmarks only the "horse 1" test image will be used, like the results shown from the instruction set simulator. Furthermore, as also seen with the synthetic benchmark, the achieved speedup when running on the FPGA is slightly lower than when running on Spike. This effect is also seen in the separate steps of the neural network which are

Table 5.3: Amount of cycles and percentage of the complete execution time taken for each step of the Cifar-10 benchmark running on Spike as well as the percentage reduction when going from no SIMD instructions to the basic subset and from the basic subset to the basic and MAC subsets.

	SISD		Basic P-extension		Basic P-extension and MAC	
Step	Cycles	Percentage	Cycles	Percentage	Cycles	Percentage
Preprocess	21515	0.03%	24587	0.08%	24587	0.13%
Convolve RGB	39433088	56.48%	20158315	67.90%	16199022	84.90%
Relu	154794	0.22%	20492	0.07%	20492	0.11%
Maxpool	1721676	2.47%	240970	0.81%	240970	1.26%
Convolve Fast	22352288	32.01%	7078698	23.84%	1751838	9.18%
Relu	19119	0.03%	2572	0.01%	2572	0.01%
Maxpool	214466	0.31%	34403	0.12%	34403	0.18%
Convolve Fast	5435360	7.79%	1769890	5.96%	445601	2.34%
Relu	9950	0.01%	1293	0.00%	1293	0.01%
Maxpool	105688	0.15%	14783	0.05%	14783	0.08%
Fully Connected	21522	0.03%	17636	0.06%	9828	0.05%
Softmax	348	0.00%	318	0.00%	318	0.00%
Total	69818350		29687342	57.5%	19079150	35.7%

displayed in Figure 5.10. In this figure, the effects of the outliers as discussed for the synthetic benchmark are also recognizable. This can be seen in, for example, the "maxpool" step. The speedup for this operation when running on the instruction set simulator is the same for the design with the basic subset and the design with the basic and MAC subsets, as it uses no MAC instructions. However, in the first two "maxpool" operations it is faster for the design without MAC instructions, while the third time that the operation is applied it is faster with MAC instructions. Since this operation does not make use of MAC instructions, this can be attributed to outliers. This effect is more pronounced for operations with a short execution time, as a background task consumes relatively more execution time. Although the speedup seen on the FPGA is slightly lower than suggested by the simulator, the percentage of time spent in each of the steps is roughly the same as seen in Table 5.4.



(a) Execution time

(b) Speedup

Figure 5.9: Execution time and speedup when using the SIMD instructions of the P-extension for the different images in the Cifar-10 benchmark running on the FPGA.

Table 5.4: Amount of cycles and percentage of the complete execution time taken for each step of the Cifar-10 benchmark running on the FPGA as well as the percentage reduction when going from no SIMD instructions to the basic subset and from the basic subset to the basic and MAC subsets.

	SISD		Basic P-extension		Basic P-extension and MAC	
Step	Cycles	Percentage	Cycles	Percentage	Cycles	Percentage
Preprocess	50295	0.05%	55407	0.12%	55777	0.19%
Convolve RGB	56405908	58.84%	29818187	66.66%	23798736	81.39%
Relu	453969	0.47%	110728	0.25%	105833	0.36%
Maxpool	2907506	3.03%	478923	1.07%	486700	1.66%
Convolve Fast	27538985	28.73%	10744950	24.02%	3005790	10.28%
Relu	55596	0.06%	12771	0.03%	12578	0.04%
Maxpool	346192	0.36%	58285	0.13%	58573	0.20%
Convolve Fast	7004478	7.31%	2585629	5.78%	816298	2.79%
Relu	22663	0.02%	5282	0.01%	5063	0.02%
Maxpool	175309	0.18%	26203	0.06%	23336	0.08%
Fully Connected	42608	0.04%	36483	0.08%	32345	0.11%
Softmax	1342	0.00%	1026	0.00%	1108	0.00%
Total	95862167		44728839	53.3%	29239531	34.6%

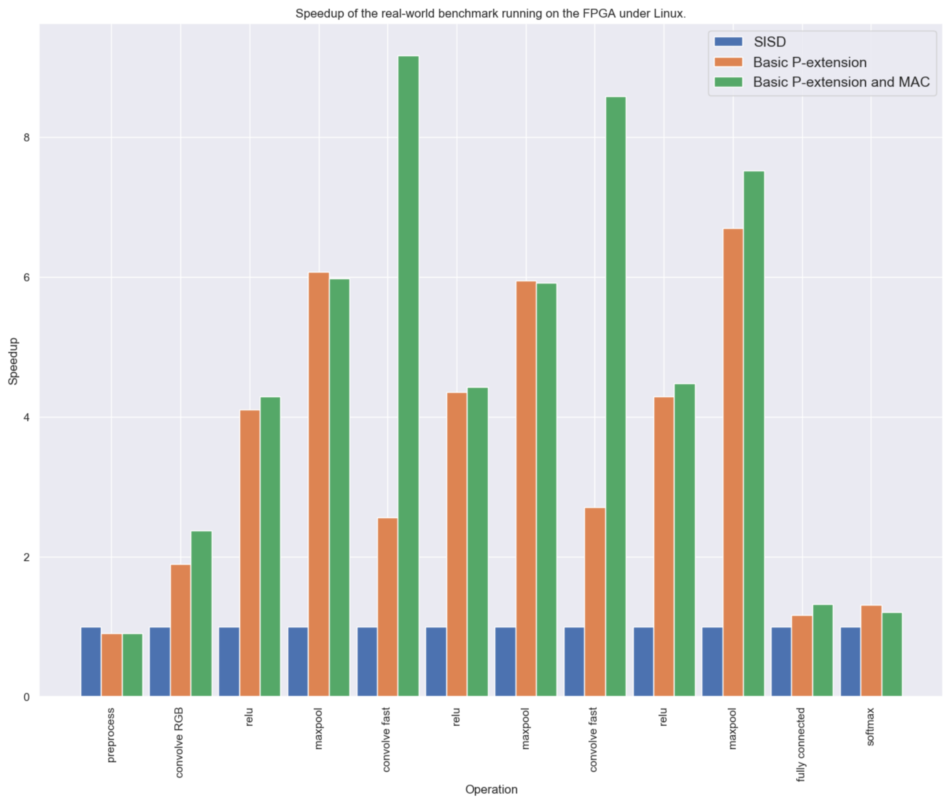


Figure 5.10: Speedup of the separate operations when using the SIMD instructions of the P-extension in the Cifar-10 benchmark running on the FPGA.

5.5. Conclusion

In this chapter, the verification process of the newly introduced instructions, the impact of the design on the critical path and area usage, and the benchmark results have been detailed. Verification of the design has been performed by running manually created tests, randomly generated tests, and running applications in the form of the benchmarks. After the design was verified, the synthetic and real-world benchmarks were run. The synthetic benchmark saw a speedup of up to 8.8x and 7.2x for 8-bit and Q7 elements and a speedup of 4.9x and 3.8x for 16-bit and Q15 elements while using the basic subset of the P-extension running on the FPGA. When additionally utilizing the MAC instructions, this speedup increases to up to 12.3x and 12.6x for 8-bit and Q7 elements and a speedup of 6.7x and 6.5x for 16-bit and Q15 elements when running on the FPGA. The speedup achieved when running on the instruction set simulator is slightly higher since, in contrast to the actual processor, all instructions are completed in a single cycle. Furthermore, when operating on 256x256 matrices, a large increase in execution time as well as an increased speedup for the SIMD algorithms was seen. This was shown to be caused by the size of the data cache. The cache miss rate is shown to be higher for the SISD algorithm than for the SIMD algorithms when the matrices do not fit in the data cache of the processor. The real-world benchmark, on the other hand, exhibits a speedup of 2.1x and 3.3x on the FPGA respectively when using just the basic subset and when also using the instructions of the MAC subsets. Unfortunately, these speedups come at a cost. An increase in LUT usage of 5.0% and an increase of 0.05% in flip-flop usage is seen when the basic subset is implemented. Also implementing the instructions of the MAC subsets uses 7.2% and 0.56% more LUTs and flip-flops over the design without SIMD instructions. Apart from the increased area usage, the additional hardware can also affect the critical path of the design. While the original design runs at 50 MHz, it, along with the implementation of the basic subset, and the design with support for the basic subset and a MAC unit with 3-cycle latency can be pushed to 70 MHz. This is possible since the critical path is present in the FPU even after extending the ALU and the multiplier. Although increasing the latency of the MAC unit by one cycle leads to an increase in the number of cycles taken for each benchmark, the increase in clock frequency compensates for this and results in lower execution times.

6 Conclusion

In this thesis, the implementation of SIMD instructions for a RISC-V processor is discussed along with the effects of this implementation on the obtained speedup, area usage, and critical path. First, a summary of this thesis is given. Subsequently, the main contributions of this thesis are presented. Finally, suggestions for future work based on this thesis are made.

6.1. Summary

In Chapter 2, the background of this thesis has been explained. The RISC-V ISA has been introduced along with its extensions and the openly available processors based on this ISA. From these available processors, the CVA6 processor was chosen as a starting point for this thesis. This is a 64-bit in-order RISC-V processor with 6 pipeline stages that supports the M, A, F, D, and C instruction set extensions. SIMD instructions are explained and its history is discussed. Moreover, audio processing, video processing, computer vision, and machine learning are shown as applications that can make use of SIMD instructions. Furthermore, the P-extension is introduced as a way to realize SIMD instructions in the RISC-V ISA and the instructions included in this extension are discussed. Additionally, the P-extension is compared against the V-extension which introduces vector-SIMD instructions rather than packed-SIMD instructions. From this comparison it is concluded that the P-extension is meant for small and low power designs, while the V-extension is meant for scalability and high performance. The process of hardware verification is investigated and is found to usually consist of manually created tests, randomly generated tests, commercial test suites, and complex applications. Finally, the related work is discussed in the form of the Ara vector coprocessor, Andes Technology's AX25MP processor, and a PULP Core with DSP Extensions.

In Chapter 3, the 332 instructions of the P-extension have been divided into subsets. Due to the large amount of new instructions and the limited time available, these subsets have been prioritized to make sure that the most important functionality would be available at the end of this project. By studying the code of open-source implementations of machine learning, video encoding, and computer vision applications, the occurrence of specific SIMD instructions was investigated. Using this information, the contents and the order of implementation of the subsets were defined. The subset that will be implemented first is the basic subset and contains instructions that were used by all the investigated applications. The basic subset includes SIMD instructions for addition, subtraction, comparison, bit shift, multiplication, and data movement that operate on 8-, 16-, and 32-bit element sizes, as well as a few miscellaneous instructions. Furthermore, subsets are defined for MAC instructions that operate on respectively 8-, 16-, and 32-bit elements as well as a subset for 8-bit SAD instructions. After the basic subset the MAC 8-bit subset will be implemented, followed by the MAC 16-bit, MAC 32-bit, and SAD 8-bit subsets. Finally, the remaining instructions are divided into the fixed-point and the non-SIMD 32- and 64-bit subsets which will be implemented in that order.

In Chapter 4, the implementations of both the hardware and software required to support the P-extension are discussed. The 268 instructions from the basic, MAC 8-bit, MAC 16-bit, and MAC 32-bit subsets have been implemented, this is 80.7% of the total 332 instructions included in the P-extension. The ALU is modified to support addition, subtraction, bit shifts, and comparisons in a SIMD manner. Furthermore, the ALU is extended to also support SIMD min and max, pack and unpack, and bit counting operations. Similarly, the multiplier has been modified to be able to perform signed and unsigned SIMD multiplication for 8-, 16-, and 32-bit element sizes. This has been realized by using a SIMD version of a Baugh-Wooley multiplier. Moreover, the multiplier has been extended to also support the MAC instructions included in the P-extension. Due to the structure of the SIMD Baugh-Wooley multiplier, sum together mode could be used to reduce the amount of required adders by performing part of the accumulation in the reduction tree of the multiplier. For further accumulation, SIMD adders were used to support generating a single 64-bit result or two 32-bit results. Furthermore, the modifications required to other components of the CVA6 processor like the decoder were also discussed. Slight modifications were made to the initial hardware design to optimize the critical path of the processor. This mainly involved reducing the amount of hardware reuse and increasing the latency of the MAC unit by one cycle. To make use of the hardware and to evaluate its performance, some software is also required. While a compiler with support for the P-extension is available, it did not work as expected. Therefore, the official RISC-V version of GCC was used with the `.insn` assembler directive. This directive allows manual construction of the fields of an instruction and is specifically meant for use with custom instructions. For evaluating the performance of the new designs, a synthetic and a real-world benchmark were constructed. The synthetic benchmark involves matrix multiplication and is meant to show the ideal performance gains of the SIMD instructions. Various different algorithms are implemented to investigate the optimal implementation. Since often not the whole application can be accelerated with SIMD instructions and overhead can be present, a more realistic real-world benchmark is also provided. For this, an image recognition neural network based on the CIFAR-10 data set is used. This benchmark was optimized and modified to also operate on a processor with support for only the basic subset of the P-extension. Although not the complete P-extension has been implemented, all instructions that could be used by the benchmarks are present in the implemented subsets.

In Chapter 5, the verification process of the newly introduced instructions, the impact of the design on the critical path and area usage, and the benchmark results have been detailed. Verification of the design has been performed by running manually created tests, randomly generated tests, and running applications in the form of the benchmarks. After the design was verified, the synthetic and real-world benchmarks were run. The synthetic benchmark saw a speedup of up to 8.8x and 7.2x for 8-bit and Q7 elements and a speedup of 4.9x and 3.8x for 16-bit and Q15 elements while using the basic subset of the P-extension running on the FPGA. When additionally utilizing the MAC instructions, this speedup increases to up to 12.3x and 12.6x for 8-bit and Q7 elements and a speedup of 6.7x and 6.5x for 16-bit and Q15 elements when running on the FPGA. The speedup achieved when running on the instruction set simulator is slightly higher since, in contrast to the actual processor, all instructions are completed in a single cycle. Furthermore, when operating on 256x256 matrices, a large increase in execution time as well as an increased speedup for the SIMD algorithms was seen. This was shown to be caused by the size of the data cache. The cache miss rate is shown to be higher for the SISD algorithm than for the SIMD algorithms when the matrices do not fit in the data cache of the processor. The real-world benchmark, on the other hand, exhibits a speedup of 2.1x and 3.3x on the FPGA respectively when using just the basic subset and when also using the instructions of the MAC subsets. Unfortunately, these speedups come at a cost. An increase in LUT usage of 5.0% and an increase of 0.05% in flip-flop usage is seen when the basic subset is implemented. Also implementing the instructions of the MAC subsets uses 7.2% and 0.56% more LUTs and flip-flops over the design without SIMD instructions. Apart from the increased area usage, the additional hardware can also affect the critical path of the design. While the original design runs at 50 MHz, it, along with the implementation of the basic subset, and the design with support for the basic subset and a MAC unit with 3-cycle latency can be pushed to 70 MHz. This is possible since the critical path is present in the FPU even after extending the ALU and the multiplier. Although increasing the latency of the MAC unit by one cycle leads to an increase in the number of cycles taken for each benchmark, the increase in clock frequency compensates for this and results in lower execution times.

6.2. Main Contributions

The research question as stated at the beginning of the thesis can now be answered. For convenience, the research question is repeated below.

> *How can packed-SIMD instructions be added to a RISC-V processor and how does this affect the performance, energy usage, and area usage of the processor?*

Packed-SIMD instructions can be added to a RISC-V processor by implementing the instructions of the P-extension. The P-extension is a proposed instruction set extension to the RISC-V ISA. The specification is still in a draft state and is therefore not officially ratified yet. To implement the instructions specified by the P-extension, changes were required to the ALU, multiplier, and decoder as well as in a lesser manner to the scoreboard, register file, and the control and status register file. The introduction of SIMD instructions provided a speedup in matrix multiplication of up to 12.3x and 12.6x respectively for 8-bit and Q7 elements and of up to 6.7x and 6.5x respectively for 16-bit and Q15 elements. Moreover, in an image recognition neural network a speedup of 2.1x and 3.3x was seen respectively for a design with just the basic subset and a design with the basic and MAC subsets. This large speedup comes at the relatively small cost of an increase of 5.0% LUT and 0.05% flip-flop usage for the basic subset or an increase of 7.2% LUT and 0.56% flip-flop usage when also implementing the instructions of the MAC subsets with 3-cycle latency. Since the critical path of the design is present in the FPU, the addition of the SIMD instructions does not impact the maximum achievable clock frequency of 70 MHz. Unfortunately, due to resource constraints the impact on energy usage could not be measured. To summarize, implementing the basic subset on its own provides a significant speedup of 2.1x in the real-world benchmark for a roughly 5% area increase, the addition of MAC instructions boosts this speedup substantially to 3.3x for just around 8% more area usage.

The main contributions of this thesis are summarized below.

- A possible division of the P-extension into subsets has been proposed.
  - With 332 additional instructions, the P-extension is the second largest currently proposed extension for the RISC-V ISA. Because of this and the fact that some of the added instructions are very specialized, not all instructions might be useful for a specific use case. By dividing the extension into subsets, the required instructions for a target application can be implemented in a modular manner. This prevents unnecessary impact on area usage, critical path, and energy usage when full functionality of the extension is not required. Furthermore, a measure of compatibility is maintained by implementing subsets of a specification compared to implementing only specific instructions. The instructions have been divided in a basic subset along with MAC 8-bit, MAC 16-bit, MAC 32-bit, SAD 8-bit, fixed-point, and non-SIMD 32- and 64-bit subsets. The division of the extension into subsets and the prioritization of the subsets was done by investigating open-source applications that make use of SIMD instructions.
- In this thesis the first open implementation of (a subset of) the P-extension to the RISC-V ISA is presented.
  - The implementation has also been evaluated in terms of the gained performance as well as the cost in terms of area usage and critical path. While proprietary designs exist that have support for the P-extension, no open implementation is available. Therefore, the cost of implementing the P-extension in terms of area and critical path is not openly available. Having this information publicly available helps when evaluating whether it is beneficial to add support for the P-extension to a design or not. A simulator like Spike can give an indication of the performance impact of using these instructions. However, as shown in this thesis, the real-world impact is often different. While ideally all of the 332 instructions of the P-extension would have been implemented, due to time constraints only 268 were implemented. Specifically, these are the instructions of the basic, MAC 8-bit, MAC 16-bit, and MAC 32-bit subsets. Moreover, these 268 instructions make up 80.7% of the complete P-extension. However, all instructions required to run the created benchmarks optimally were implemented. Therefore, the performance seen from this incomplete implementation in these benchmarks is the same as would be seen from a complete implementation of the P-extension. Furthermore, most of the instructions of the subsets that were not yet implemented will mostly make use of the same hardware. Therefore, the impact of implementing these instructions on, for example, area usage and critical path are likely limited.
- Software for verifying the correctness of hardware implementations of the P-extension has been constructed.
  - Using manually created tests, the modified ALU and multiplier were tested individually as well as integrated in the whole processor. Moreover, these tests were run in simulation as well as on the FPGA. Furthermore, a random test generator was extended to generate programs that include instructions from the P-extension. For validation, simulation results of these programs are

compared against the official RISC-V instruction set simulator. Apart from this, the benchmark programs were also used to verify the correctness of the added hardware. All this testing gives confidence that the processor adheres to the specification and functions as intended.

- To test the performance of designs with the basic subset and with the basic and MAC subsets of the P-extension, a synthetic benchmark has been created and a real-world benchmark has been modified and optimized.
  - A synthetic benchmark in the form of matrix multiplication was developed to evaluate the ideal performance of the P-extension. Furthermore, an image recognition neural network was optimized and modified to function as a real-world benchmark. Both these benchmarks were constructed to show the performance of the P-extension when just the basic subset is available and when the instructions from the basic and MAC subsets are available. Using these benchmarks, execution times and speedup when using the P-extension were measured when running on the instruction set simulator as well as on the actual design running on an FPGA. The synthetic benchmark achieves a speedup of around 8x and 4x for 8- and 16-bit data while adding MAC instructions increases the speedup to around 12x and 6.5x for 8- and 16-bit data. The real-world benchmark, on the other hand, sees a 2.1x speedup when using the basic subset and 3.3x speedup when also using the MAC subsets.

6.3. Future Work

Because of the limited time available and new ideas found while working on the project, the topic of adding SIMD instructions to RISC-V processors has not been fully explored. Below, ideas for future work based on this thesis are proposed.

- Unfortunately, due to time constraints only 268 of the 332 instructions of the P-extension, around 81%, were implemented. To be able to evaluate the P-extension in its entirety, the remaining instructions should also be implemented. Even though the benchmarks presented in this thesis would not benefit from these additional instructions, other applications could. Furthermore, by doing so the complete impact of the extension on area usage and critical path could be evaluated. As most of the remaining instructions are various types of operations that are performed by either the ALU or MAC unit, likely much of the required hardware is already present. With the exception of, for example, the SAD instructions which would require accumulation in the ALU. However, reading the third operand for this operation can be implemented in a similar way as for the MAC unit.

- As stated before, there are many applications in which SIMD instructions are useful. However, only a machine learning application was used as a real-world benchmark. To provide a more complete overview of how beneficial inclusion of the P-extension is, different types of applications should also be benchmarked. Furthermore, these benchmarks could also make use of the instructions from the subsets that have not been implemented in this thesis. Evaluating the instructions used in these benchmarks could then also prove whether the proposed division of the P-extension in subsets has been done properly.

- It would be interesting to investigate the impact of the introduced packed-SIMD instructions on the power and energy usage of the processor. As additional hardware has been implemented, the maximum power usage of the design has most likely been increased. However, as the execution time of applications that can make use of SIMD instructions has been decreased, the energy usage for those applications is likely decreased. This is of great importance since a processor like this is often used in an embedded or internet of things application, where low energy usage is essential. On the other hand, energy usage for applications that can not make use of SIMD instructions is likely increased due to the extra unused hardware. As a device often has multiple tasks, of which likely not all can utilize SIMD instructions, the overall energy usage should also be investigated.

- A downside of making use of SIMD hardware is the additional development time required to convert an algorithm to use SIMD instructions. To overcome this, many compilers have support for automatically vectorizing code to make use of SIMD instructions. Unfortunately, as the P-extension is still in development, compiler support is limited. It would be interesting to investigate the effect of automatic vectorization on the P-extension. Specifically, comparing the speedup gained by automatic vectorization against hand optimized SIMD code. Automatic vectorization would also make benchmarking multiple types of applications using the P-extension easier, as the amount of work required to introduce SIMD instructions in these applications is reduced.

- The V-extension adds functionality similar to the P-extension while taking a very different approach to the implementation. Instead of reusing the integer or floating-point register file, the V-extension uses a dedicated register file. Furthermore, as the width of these registers can be chosen by the implementation, the amount of parallelism that can be achieved by the V-extension is also variable. It might be interesting to implement the V-extension for the CVA6 processor with a similar performance level. The cost of implementing the P- and V-extensions can then directly be compared. Another option would be to implement the V-extension with a similar area usage to the P-extension. This would instead allow the performance of both extensions to be compared. Moreover, by implementing the same benchmarks for both extensions, the difficulty and time required to effectively make use of these extensions can be compared.

Bibliography

[1] A. Debiève. [Online]. Available: https://unsplash.com/photos/FO7JIlwjOtU

[2] D. Reinsel, J. Gantz, and J. Rydning, “The digitization of the world from edge to core,” *Framingham: International Data Corporation*, 2018.

[3] W. D. Hillis, *The connection machine*. MIT press, 1989.

[4] R. B. Lee, “Accelerating multimedia with enhanced microprocessors,” *iEEE Micro*, vol. 15, no. 2, pp. 22–32, 1995.

[5] C. C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink, and T. Schierl, “Simd acceleration for hevc decoding,” *IEEE Transactions on circuits and systems for video technology*, vol. 25, no. 5, pp. 841–855, 2014.

[6] V. Herdt, D. Große, H. M. Le, and R. Drechsler, “Extensible and configurable risc-v based virtual prototype,” in *2018 Forum on Specification & Design Languages (FDL)*. IEEE, 2018, pp. 5–16.

[7] V. Herdt, D. Große, and R. Drechsler, “Fast and accurate performance evaluation for risc-v using virtual prototypes,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 618–621.

[8] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, “Leveraging the openness and modularity of risc-v in space,” *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 454–472, 2019.

[9] “Risc-v international members.” [Online]. Available: https://riscv.org/members/

[10] A. Waterman and K. Asanovic, “The risc-v instruction set manual volume i: Unprivileged isa,” December 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[11] “Risc-v cores, soc platforms and socs.” [Online]. Available: https://github.com/riscv/riscv-cores-list

[12] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[13] OpenHW Group, “Openhw group core-v cv32e40p risc-v ip.” [Online]. Available: https://github.com/openhwgroup/cv32e40p

[14] lowRISC, “Ibex risc-v core.” [Online]. Available: https://github.com/lowRISC/ibex

[15] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[16] C. Wolf, “Picorv32 - a size-optimized risc-v cpu.” [Online]. Available: https://github.com/cliffordwolf/picorv32

[17] CHIPS Alliance, “Swerv eh1 core.” [Online]. Available: https://github.com/chipsalliance/Cores-SweRV

[18] ——, “Swerv eh2 core.” [Online]. Available: https://github.com/chipsalliance/Cores-SweRV-EH2

[19] ——, “Swerv el2 core.” [Online]. Available: https://github.com/chipsalliance/Cores-SweRV-EL2

[20] OpenHW Group, “Cva6 risc-v cpu.” [Online]. Available: https://github.com/openhwgroup/cva6

[21] ETH Zürich, “The iis chip gallery.” [Online]. Available: http://asic.ethz.ch/

[22] A. Waterman and K. Asanovic, “The risc-v instruction set manual volume ii: Privileged architecture,” June 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf

[23] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[24] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

[25] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, “The illiac iv computer,” *IEEE Transactions on computers*, vol. 100, no. 8, pp. 746–757, 1968.

[26] H. Falk, “What went wrong v: Reaching for a gigaflop: The fate of the famed illiac iv was shaped by both research brilliance and real-world disasters,” *IEEE spectrum*, vol. 13, no. 10, pp. 65–70, 1976.

[27] N. T. Slingerland and A. J. Smith, “Multimedia extensions for general purpose microprocessors: A survey,” *Microprocessors and Microsystems*, vol. 29, no. 5, pp. 225–246, 2005.

[28] “Simd isas - arm developer.” [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas

[29] D. Talla, L. K. John, V. Lapinskii, and B. L. Evans, “Evaluating signal processing and multimedia applications on simd, vliw and superscalar architectures,” in *Proceedings 2000 International Conference on Computer Design*. IEEE, 2000, pp. 163–172.

[30] J. A. Belloch, F. J. Alventosa, P. Alonso, E. S. Quintana-Ortí, and A. M. Vidal, “Accelerating multi-channel filtering of audio signal on arm processors,” *The Journal of Supercomputing*, vol. 73, no. 1, pp. 203–214, 2017.

[31] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, “Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms,” in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1107–1116.

[32] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, “Machine learning at facebook: Understanding inference at the edge,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[33] “Quantization - pytorch 1.9.0 documentation.” [Online]. Available: https://pytorch.org/docs/stable/quantization.html

[34] “Post-training integer quantization | tensorflow lite.” [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_integer_quant

[35] C. Chang, “Risc-v p extension specification.” [Online]. Available: https://github.com/riscv/riscv-p-spec

[36] ——, “Status update of risc-v p extension task group,” June 2019. [Online]. Available: https://riscv.org//wp-content/uploads/2019/06/17.20-P-ext-RVW-Zurich-20190611.pdf

[37] R. Yates, “Fixed-point arithmetic: An introduction,” *Digital Signal Labs*, vol. 81, no. 83, p. 198, 2009.

[38] R. B. Lee, “Multimedia extensions for general-purpose processors,” in *1997 IEEE Workshop on Signal Processing Systems. SiPS 97 Design and Implementation formerly VLSI Signal Processing*. IEEE, 1997, pp. 9–23.

[39] M. Xu, M. Zhao, S. Yu, X. Zheng, N. Wu, and L. Liu, “Rounding shift channel post-training quantization using layer search,” in *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)*. IEEE, 2021, pp. 545–549.

[40] B. Silveira, G. Paim, B. Abreu, M. Grellert, C. M. Diniz, E. A. C. da Costa, and S. Bampi, “Power-efficient sum of absolute differences hardware architecture using adder compressors for integer motion estimation design,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 12, pp. 3126–3137, 2017.

[41] K. Asanovic and R. Espasa, "Risc-v v extension specification." [Online]. Available: https://github.com/riscv/riscv-v-spec

[42] D. Dabbelt, C. Schmidt, E. Love, H. Mao, S. Karandikar, and K. Asanovic, "Vector processors for energy-efficient embedded systems," in *Proceedings of the Third ACM International Workshop on Many-core Embedded Systems*, 2016, pp. 10–16.

[43] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[44] D. Patterson and A. Waterman, "Simd instructions considered harmful," September 2017. [Online]. Available: https://www.sigarch.org/simd-instructions-considered-harmful/

[45] V. Pratt, "Anatomy of the pentium bug," in *Colloquium on Trees in Algebra and Programming*. Springer, 1995, pp. 97–107.

[46] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[47] L. Fournier, Y. Arbetman, and M. Levinger, "Functional verification methodology for microprocessors using the genesys test-program generator. application to the x86 microprocessors family," in *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*. IEEE, 1999, pp. 434–441.

[48] "Andescore ax25mp multicore." [Online]. Available: http://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax25mp/

[49] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[50] PyTorch, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration." [Online]. Available: https://github.com/pytorch/pytorch

[51] G. Conte, S. Tommesani, and F. Zanichelli, "The long and winding road to high-performance image processing with mmx/sse," in *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*. IEEE, 2000, pp. 302–310.

[52] M. Viitanen, A. Koivula, A. Lemmetti, A. Ylä-Outinen, J. Vanne, and T. D. Hämäläinen, "Kvazaar: open-source hevc/h. 265 encoder," in *Proceedings of the 24th ACM international conference on Multimedia*, 2016, pp. 1179–1182.

[53] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[54] "Risc-v bit-manipulation isa-extensions," June 2021. [Online]. Available: https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf

[55] C. Chang. [Online]. Available: https://lists.riscv.org/g/tech-p-ext/message/135

[56] T. Lang and J. D. Bruguera, "Floating-point multiply-add-fused with reduced latency," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 988–1003, 2004.

[57] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on computers*, vol. 100, no. 12, pp. 1045–1047, 1973.

[58] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

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

[60] 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.

[61] 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, 2009.

[62] L. Mei, M. Dandekar, D. Rodopoulos, J. Constantin, P. Debacker, R. Lauwereins, and M. Verhelst, "Subword parallel precision-scalable mac engines for efficient embedded dnn inference," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2019, pp. 6–10.

[63] "Questions about kmmawb2/kmmawb2.u/kmmawt2/kmmawt2.u." [Online]. Available: https://github.com/riscv/riscv-p-spec/issues/11

[64] "riscv-gnu-toolchain." [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain

[65] Andes Technology, "gcc." [Online]. Available: https://github.com/andestech/gcc

[66] ——, "binutils." [Online]. Available: https://github.com/andestech/binutils

[67] ARM-software, "Cmsis version 5." [Online]. Available: https://github.com/ARM-software/CMSIS_5

[68] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[69] Nuclei-Software, "Nuclei microcontroller software interface standard." [Online]. Available: https://github.com/Nuclei-Software/NMSIS

[70] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[71] Digilent, "Genesys 2 fpga board reference manual," August 2017. [Online]. Available: https://reference.digilentinc.com/_media/reference/programmable-logic/genesys-2/genesys2_rm.pdf

[72] "Verilator." [Online]. Available: https://www.veripool.org/wiki/verilator

[73] "Gtkwave." [Online]. Available: http://gtkwave.sourceforge.net/

[74] "riscv-tests." [Online]. Available: https://github.com/riscv/riscv-tests

[75] "riscv-torture." [Online]. Available: https://github.com/ucb-bar/riscv-torture

[76] "riscv-opcodes." [Online]. Available: https://github.com/riscv/riscv-opcodes