SUPERSCALAR RISC-V PROCESSOR WITH SIMD VECTOR EXTENSION

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

By

Jiongrui He

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering
3B48 Engineering Building
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan S7N 5A9
Canada

Or

Dean of College of Graduate and Postdoctoral Studies
116 Thorvaldson Building
University of Saskatchewan
110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

With the increasing number of digital products in the market, the need for robust and highly configurable processors rises. The demand is convened by the stable and extensible open-sourced RISC-V instruction set architecture. RISC-V processors are becoming popular in many fields of applications and research.

This thesis presents a dual-issue superscalar RISC-V processor design with dynamic execution. The proposed design employs the global sharing scheme for branch prediction and Tomasulo algorithm for out-of-order execution. The processor is capable of speculative execution with five checkpoints. Data flow in the instruction dispatch and commit stages is optimized to achieve higher instruction throughput.

The superscalar processor is extended with a customized vector instruction set of single-instruction-multiple-data computations to specifically improve the performance on machine learning tasks. According to the definition of the proposed vector instruction set, the scratch-pad memory and element-wise arithmetic units are implemented in the vector co-processor.

Different test programs are evaluated on the fully-tested superscalar processor. Compared to the reference work, the proposed design improves 18.9% on average instruction throughput and 4.92% on average prediction hit rate, with 16.9% higher operating clock frequency synthesized on the Intel Arria 10 FPGA board.

The forward propagation of a convolution neural network model is evaluated by the standalone superscalar processor and the integration of the vector co-processor. The vector program with software-level optimizations achieves 9.53× improvement on instruction throughput and 10.18× improvement on real-time throughput. Moreover, the integration also provides 2.22× energy efficiency compared with the superscalar processor along.

Acknowledgements

First of all, I would like to express thanks to my family. Their understanding and emotional care support me to complete my study.

Moreover, I would like to show sincere appreciation to my supervisor, Dr. Seokbum Ko, for all of his support and mentorship during my master program. Whenever I face a challenge, I know that Dr. Ko is sitting in the office and willing to provide guidance. At the beginning of my master program, Dr. Ko gave me the freedom to select the research topic. I am lucky to study on this topic that I am really interested in. Dr. Ko's patience and knowledge motivate me to accomplish this work.

Especially, I would like to appreciate Dr. Dongdong Chen. We have a great period of the corporation since the beginning of my master program. His experience in both research and industry really helped me understand the computer architecture and the IC design flow.

Thanks to my committee members Dr. Anh Dinh, Dr. Dwight Makaroff, and Dr. Francis Bui for their efforts to review and improve my thesis.

I also would like to thank all colleagues in Dr. Ko's lab. Their impressive research topics and progress have broadened my perspective. In particular, I am grateful to Hao Zhang and Yi Wang for their comments and helping me solve the technical issues of the development software and tools.

Thank you for all your encouragement.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence
ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
BTB	Branch Target Buffer
CDB	Common Data Bus
CISC	Complex Instruction Set Computer
CNN	Convolution Neural Network
COM	Commit
CPU	Central Processing Unit
CSR	Control and Status Register
DLP	Data-Level-Parallelism
DP	Data Dispatch
DUT	Device Under Test
EX	Execution
FC	Fully-Connected
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GPU	Graphic Processing Unit
ID	Instruction Decode
IF	Instruction Fetch
ILP	Instruction-Level-Parallelism
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
MMV	Matrix Multiply Vector
MSB	Most Significant Bit
PC	Program Counter
PHT	Pattern History Table

PU	Processing Unit
RISC	Reduced Instruction Set Computer
RS	Reservation Station
SIMD	Single-Instruction Multiple-Data
SPM	ScratchPad Memory
VGTM	Vector Greater Than Merge
VLIW	Very-Long-Instruction-Word

Chapter 1

Introduction

1.1 Introduction

Past decades have witnessed the rapid advancement of computer hardware related to ARM and x86 processors, which have become the foundation of global semiconductor markets. Processors are pervasive from cores of supercomputers to controllers of embedded chips. The processor is a digital circuit that handles a simple step of operation on a certain instruction, which takes its origin from the Turing Machine. The combination of numerous simple instructions can reproduce different types of algorithms. There are two major categories of processors, general-purpose processors and single-purpose processors. The general-purpose processor, also known as a microprocessor, can support many different applications by only programming the software, such as the central processing units (CPU) in personal computers and mobile phones. The single-purpose processor usually has better performance in speed, power, and area, however, its utilization is limited to a certain type of application, such as the graphic processing unit (GPU) and artificial intelligence (AI) accelerators.

The functionality of microprocessors is defined by the instruction set architecture (ISA). The instruction set describes the instruction that the microprocessor can execute. According to the layers of abstraction in computers, which is presented in Figure 1.1, an ISA is a bridge between software and hardware and it is the specification of microprocessor design.

There are also two categories of ISA, the complex instruction set computer (CISC) and the reduced instruction set computer (RISC). Table 1.1 shows the major differences between CISC architecture and RISC architecture. X86 is a typical CISC ISA, which instructions are complex and capable of operating directly upon memory address. However, RISC instructions are treated as an improvement over CISC by simplifying the instruction format and the op-

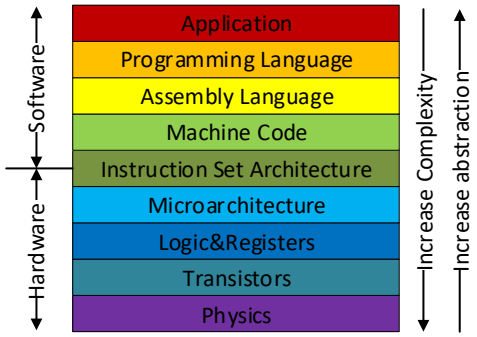

Figure 1.1: Layers of Abstraction.

eration of each instruction. RISC microprocessors often execute one instruction per machine cycle so that it is easier to pipeline the design to achieve higher clock frequency. However, the simple operations in RISC instructions bring the complexity in software compilers. The difference between CISC and RISC is explained by the basic performance equation:

$$\frac{seconds}{program} = \frac{instructions}{program} \times \frac{cycles}{instrucion} \times \frac{seconds}{cycle} \tag{1.1}$$

Table 1.1: Comparison between general CISC and RISC

ISA	CISC	RISC
Number of instructions	Extended	Reduced
Duration of an instruction	Multiple cycles	One cycle
Instruction length	Variable	Fixed
Memory access	Many instructions	Load and Store
Registers	Unique	Multiple
Complexity	In compiler	In hardware

CISC architecture emphasizes the efficiency in instructions per program while RISC architecture emphasizes the efficiency in cycles per instruction. There is a trade-off in terms of performance. However, the booming market of smartphones and embedded projects brings people's concern about power consumption. Because complex CISC instructions require more

logic and transistors to delay with more power consumption, RISC ISA is dominating the market of mobile devices nowadays.

A typical microprocessor consists of arithmetic logic units (ALU), control unit, and data storage unit. The microprocessor takes instruction from the external memory. Then the control unit, based on that instruction, reconnects datapath to feed the operands to ALU and selects the corresponding function in ALU to finish one operation. If the instruction and data are stored at two different places, it is considered as the Harvard Architecture. On the other hand, the instruction and data share single storage in the Von Neumann Architecture. Under Harvard Architecture, microprocessors can access instructions and data, from the program memory and the data memory simultaneously, to release the Von Neumann bottleneck. Those two basic architectures are presented in Figure 1.2. However, most of the modern microprocessors fetch instruction and data, from the instruction cache and data cache respectively, which is recognized as the Harvard Architecture from the core perspective. The two caches are connected to the main memory in the memory hierarchy. Instructions and data eventually are stored in hard disks, which is recognized as the Von Neumann Architecture from the system perspective. Therefore, the boundary between the two architectures is blurred [2].

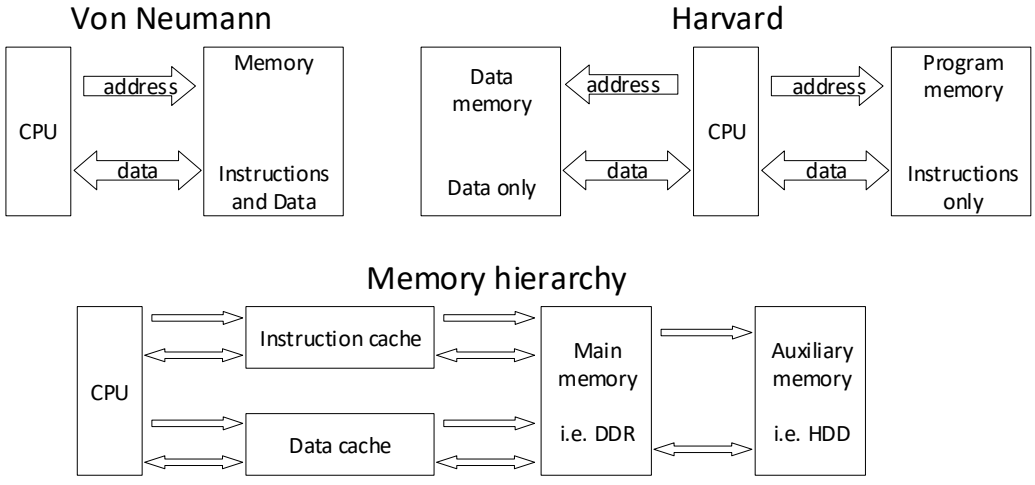

Figure 1.2: Von Neumann Architecture, Harvard Architecture and Memory Hierarchy.

1.2 Objective

Many applications require a controller to monitor the peripherals and handle the analog-to-digital conversion [3–5]. Using a general-purpose processor can significantly shorten the period of research, compared with developing a task-specific controller. Thanks to the upgrading field-programmable gate arrays (FPGA), soft cores can be programmed in an FPGA project without exceeding the onboard resource limitation. An open-source and customizable microprocessor brings flexibility to developers, which can rapidly change the design, such as, adding new task-specific instructions, extending with a co-processor, and investigating a new bus protocol. Therefore, an open-source microprocessor can accelerate the hardware design flow by providing different configurations to developers. However, both ARM and x86 ISA are proprietary leading to high research costs for hardware designers. Thankfully, RISC-V, an open-source ISA, is becoming popular in industry because of the maturity of its software ecosystem and toolchains.

Many RISC-V core designs have relatively short pipeline stages focusing on energy and area efficiency in the literature. PicoRV32 [6] is a decent compact core with a general sequencer to process every instruction so that its average instruction per cycle is only approximately 0.25. Hummingbird E203 [7] also focuses on low power and small area. It has two-stage in-order pipelines to improve the performance. Similarly, Zero-riscy [8] with a two-stage pipeline, orients to the energy efficiency for Internet-of-Thing applications and Micro-riscy [8], based on Zero-riscy, aggressively halves the register file to further reduce the area.

On the other hand, performance-oriented cores also bring interests to researchers and designers. Riscy [9] is integrated with customized instructions to support Zero Overhead Loops and packed single instruction multiple data (SIMD) computation. Ariane [10] and Rocket core [11], with the similar six-stage pipeline and single-issue architecture, are capable of out-of-order execution by the Scoreboarding method. Ridecore [12] is a dual-issue superscalar core and follows the Tomasulo Algorithm to handle dynamic execution.

The motivation for this research is to understand the modern architectures inside of CPU cores to improve the performance, and the compilation flow to connect the software and the

hardware. The goals of this thesis are summarized below:

- A general-purpose processor design is proposed following the RISC-V ISA specification. The proposed design supports the basic 32-bit RISC-V instruction set with the integer multiplication/division extension.
- Modern processors' features including branch prediction, superscalar architecture, and out-of-order execution are implemented in the proposed design to improve the performance.
- The proposed design must pass the RISC-V ISA regression test to verify its functionality. Several testcases and benchmarks are evaluated to compare the performance.
- A customized vector instruction set is proposed to efficiently support the SIMD computation. The extended instruction set is mapped to the standard 32-bit RISC-V format.
- A vector co-processor, with scratchpad memory, address sequencers, and the dot-product unit, is proposed to support the vector instructions and SIMD computation.
- The vector co-processor is coupled with the previous general-purpose processor. An inference of a convolution neural network (CNN) is evaluated to reveal the performance increase of SIMD computation.
- The proposed designs are implemented on the Intel Arria 10 FPGA board.

1.3 Organization

This thesis is organized as follows. Chapter 2 discusses the standard RISC-V ISA and extended SIMD instructions based on Cambricon [1]. Modern processors' flow and parallelism in hardware are also included in this Chapter. Chapter 3 describes the hardware implementation of the main microprocessor, while, Chapter 4 describes the hardware implementation of the SIMD vector co-processor. Chapter 5 documents the compiling flow of the RISC-V GNU toolchain for the RISC-V microprocessor and optimizations of assembly codes for the vector co-processor. Chapter 6 compares performance results and synthesis reports to summarize

what has been achieved during this research. Chapter 7 reflects the summary of the proposed design with several potential future improvements and advancements in the proposed work.

Chapter 2

Background

2.1 RISC-V ISA

RISC-V is an open-source ISA and it has brought a huge amount of momentum since the first release in 2010 by the University of California Berkeley [13]. RISC-V ISA has its origin in a computer architecture project in education. Now, RISC-V ISA brings more and more attention to not only academia but also industry, because of the maturity of the software ecosystem and toolchains. RISC-V International is managing the RISC-V specification and the community released the ratified version of the privileged specification in June 2019 [14]. The latest specification defines the solid machine-level and supervisor-level ISA, which guarantees that hardware is compatible with all RISC-V software and operating systems. Moreover, unlike the proprietary ISA, like ARM and x86, the RISC-V ISA offers the possibility to modify and customize the architecture, without requesting the permission or subscribing the license in other expensive commercials ISAs.

The latest specification of RISC-V ISA can be found on the website of RISC-V International.[1] According to the preface part in the specification, a typical standard RISC-V processor is started by defining the number of general-purpose registers and the data length of both addresses and data. The base integer ("I") ISA varies among 32-bit, 64-bit, and 128-bit. The 32-bit processor represents that its addresses and data have the data size of 32-bit. The base ISA is followed by some optional standard extensions, which further enhances the generality and flexibility in RISC-V ISA. Some of the most common and useful extensions are listed below:

[1] https://riscv.org/

- "Zifencei", instruction-fetch fence,
- "Zicsr", control and status register instructions,
- "M", standard extension for integer multiplication/division instructions,
- "A", standard extension for atomic instructions,
- "F", standard extension for single-precision floating-point instructions,
- "D", standard extension for double-precision floating-point instructions,
- "Q", standard extension for quad-precision floating-point instructions, and
- "C", standard extension for compressed instructions.

The RISC-V instruction set is organized by a combination of the base integer ISA and optional extensions. For example, the ISA, "RV32IM", indicates that this instruction set supports the 32-bit base integer instructions and integer multiplication/division instructions. The "Zifencei" defines the FENCE.I instruction that synchronizes the instruction and data streams. A FENCE.I instruction guarantees that the following instruction fetches on a RISC-V core will see the latest content in the memory, by stalling the processor until the previous STORE instruction is finished. The "Zicsr" defines an additional address space of 4096 control and status registers with associative instructions that modify and control those registers (CSR). The usage of CSRs is described in the privileged specification. Examples include interrupt handlers, exceptions, and memory virtualization.

The coding formats of the 32-bit RISC-V instructions are presented in Figure 2.1. These precise formats place the register fields in the same position to simplify the hardware decoding logic. Besides, the top-bit in the immediate fields is always placed in the most significant bit (MSB) of instructions, which reduces the sign extension case during the expansion of the immediate value. The operations of RISC-V instructions are primarily grouped by the opcodes. Table 2.1 shows the opcode map of the instruction set. The "opcode[1:0]=11" marks the 32-bit instruction, while, other bits combination is reserved for the "C" extension of the compressed 16-bit instructions. The gray column is reserved for instructions which lengths are greater than 32-bit. The four free opcodes, noted as *custom_[0,1,2,3]*, give fully

empty encoding space for customized instructions, which provides the basis for specialized instruction set extension and customized accelerators.

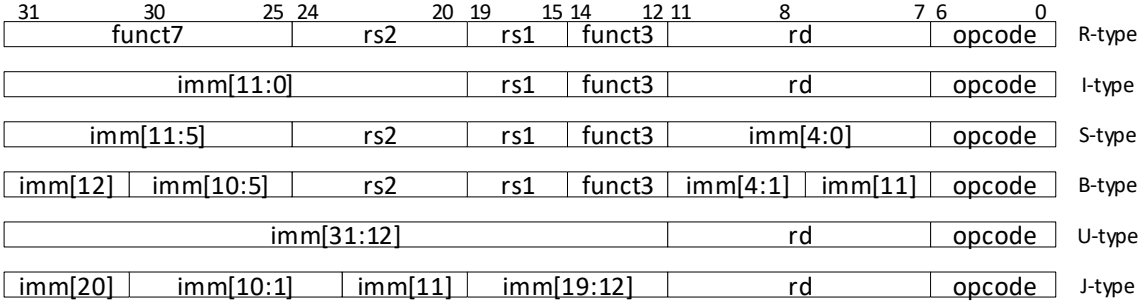

Figure 2.1: 32-bit RISC-V instruction formats. The sub-field of each immediate indicates the bit position of the produced immediate value.

Table 2.1: RISC-V base opcode map, for opcode[1:0]=11

opcode[4:2] / opcode[6:5]	000	001	010	011	100	101	110	111 >32b
00	LOAD	LOAD_FP	*custom_0*	MISC_MEM	OP_IMM	AUIPC	OP_IMM_32	48b
01	STORE	STORE_FP	*custom_1*	AMO	OP	LUI	OP_32	64b
10	MADD	MSUB	NMSUB	NMADD	OP_FP	*reserved*	*custom_2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom_3/rv128*	≥80b

2.2 Parallelism in Hardware

Pipelining the design is a powerful and straightforward technique to speedup the throughput. In the field of microprocessors, pipelining partitions each instruction into multiple stages. Pipelining usually brings data dependency hazard in microprocessor. The instruction may need the result of the previous instruction, which is ready in the ALU but not in the registers. In another word, the previous instruction is not fully completed, but the current instruction requires the latest output because of the deep pipeline stages. The data hazard can be solved by a control unit to forward the latest output to the required stage directly. Such control unit does not affect the throughput by providing the correct results, however, introduces the

hardware complexity. The instruction throughput of microprocessors is defined as:

$$\frac{instructions}{second} = \frac{cycles}{second} \times \frac{instructions}{cycle}, \tag{2.1}$$

where the first term indicates the operating frequency, or clock frequency, and the second term indicates how many instructions can handle in each stage.

However, in reality, the improvement of increasing the pipeline stages decreases as the stages go deeper and deeper. One of the biggest problems is that it is hard to exactly equally split the task. For example, a full datapath of one instruction takes 50ns, including 10ns in decoding, 20ns in operands fetch, and 20ns in computation, which can operate under the clock frequency no more than 20Mhz. To double the clock frequency, the perfect pipeline scheme is to split the datapath by two, 25ns latency in each stage. However, the datapath can only divided by two in the form of *decoding+operands fetch* (30ns) and *computation* (20ns). Just like the shortest stave in a barrel, the clock frequency is limited by the first 30ns latency so that the frequency cannot reach higher than 33.33Mhz. On the other hand, the branch miss penalty increases as the pipeline stage go deeper.

Branch instructions are very common in programs. The program counter (PC), or the address of instructions, increases sequentially by the default. The branch instruction may cause the PC to jump to another address for a new sequence of instructions. The condition, whether to jump or not, usually is ready after the computation stage. At the same time, the instructions related to the wrong PC path are already processing in previous pipeline stages.

To avoid that situation, the most simple solution is to suspend the processor until the branch instruction is finished, which decreases the second term in the equation of throughput, instructions per cycle (IPC). There are many branch prediction techniques that can detect and predict the branch condition at the early stage, which does not cause stall cycles on the successful prediction. However, as long as the rate of successful prediction is not 100%, the branch miss penalty exists and increases as the pipeline stages increase, which makes traditional pipelined microprocessors to have less than one IPC. No matter of the depth of the pipelined processor, it belongs to the single-instruction-single-data architecture in Flynn Taxonomy [15], which is presented in Figure 2.2.

Parallel computing can further improve the throughput besides pipelining. The paral-

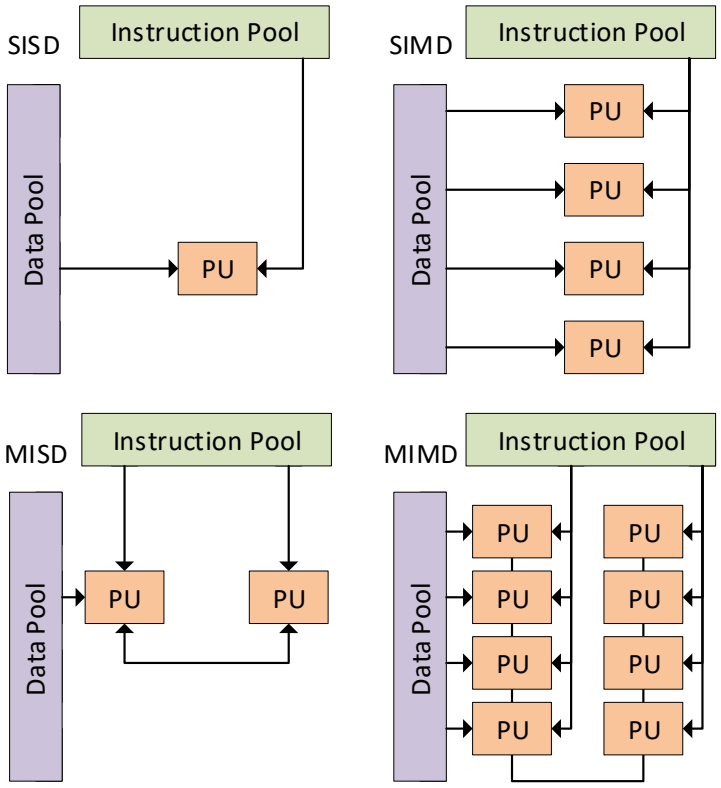

Figure 2.2: Flynn Taxonomy in computer architecture.

lelism in hardware consists of multi-instructions, multi-data, multi-cores, and multi-computers. The design is focused on instruction-level parallelism (ILP) and data-level parallelism (DLP) within the scope of a single core problem.

2.2.1 Instruction-Level Parallelism

Superscalar and very-long-instruction-word (VLIW) are two models in computer architecture to execute multiple instructions in one clock cycle within a single processor. Unlike the SISD processors, a superscalar processor can dispatch multiple instructions to their targeting processing units (PU).

The superscalar processor requires the data dependency check not only on the different stages, like pipelined processors, but also on the same clock cycle. The multiple instructions fetched and decoded at the same clock cycle may not be dispatched or finished at the same

time. The dependency check logic brings complexity in hardware, but the superscalar approach raises the roofline IPC to the number of multiple instructions from one compared to the conventional pipelined processor. In other words, regardless of how deeper the pipeline stages go, the highest IPC of the pipelined processor is limited to one. Whereas, with the multiple instructions processing in the same clock cycle, the superscalar processor increase its highest IPC to the number of the paralleled instructions.

VLIW processors also can execute multiple instructions in one clock cycle and there is no dependency check logic. VLIW approach heavily depends on the compiler side, which resolves all data dependency conflicts in machine codes. This approach is also called static scheduling. The VLIW architecture comes after the superscalar architecture and tries to retain the same throughput while reducing the hardware complexity. So far, the RISC-V toolchain does not support VLIW because the static scheduling has failed in general-purpose computing. Major drawbacks include unpredictable branches, code size explosion, and compiler complexity.

Tomasulo Algorithm

In contrast to VLIW processors, superscalar processors dynamically resolve data dependencies in hardware that brings the capability of out-of-order execution. The Tomasulo algorithm [16] was developed by Robert Tomasulo and it has become the basic structure in many modern processors. According to the algorithm, hardware register renaming, reservation stations, and a common data bus (CDB) for broadcasting are introduced to computer microarchitecture.

Hardware register renaming abstracts the physical address of destination registers to the logical address based on the order of the incoming instructions, which is essential to perform the out-of-order execution correctly. Reservation stations (RS) are the unified scheduler regarding on each processing unit. Every processing unit has its own reservation station to temporarily hold instructions. The reservation station dispatches the instruction to the targeting processing unit if all source operands are ready and the processing unit is free. The oldest instruction in the reservation station has the highest priority to be dispatched if multiple instructions are ready at the same time. When the instruction is finished in the processing unit and the result is ready, the common data bus takes the value and renamed

address to broadcast to every reservation station. The renamed addresses are also called as tags to differentiate which instruction in the reservation station needs the latest results.

Figure 2.3 shows the example of out-of-order execution to solve the read-after-write data hazard. Before four instructions allocate to the reservation station, their destination addresses are renamed to "1,2,3,4" from "c,d,e,f", in the sequence of instructions. The third AND instruction has registers "c,d" as the source operands. At the time of the third instruction entering the reservation station, registers "c,d" already set busy by the ADD and SUB instructions, and the values in registers "c,d" are no longer valid. Therefore, the AND instruction copies the renamed addresses "1,2" as the tags and waits for the latest data from the common data bus. The AND instruction depends on the latest results of ADD and SUB instructions, which is a typical read-after-write data hazard.

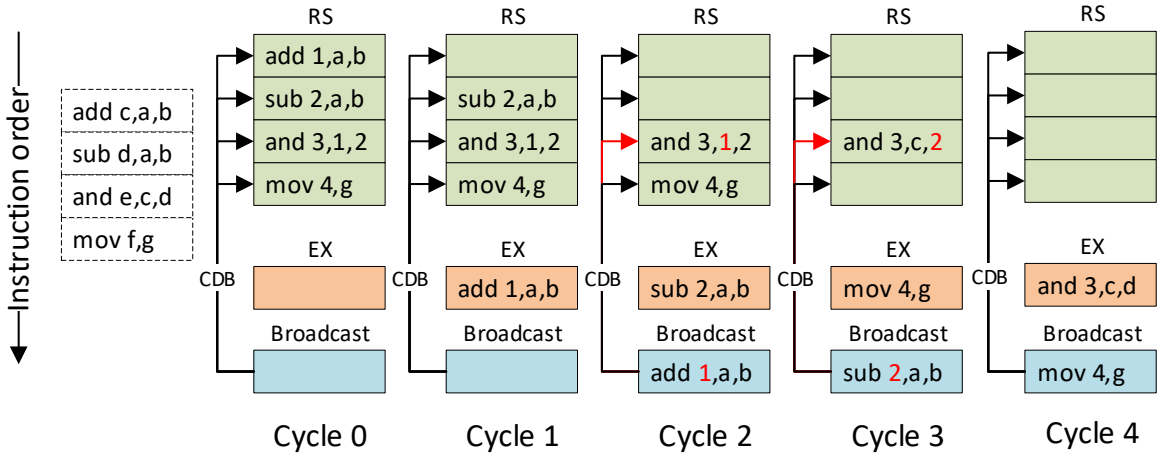

Figure 2.3: Example of dynamic scheduling following Tomasulo algorithm.

In the period of Cycle 0, the "ADD, SUB, MOV" instructions are ready to be dispatched. Because the ADD instruction is the oldest ones in the reservation, it is dispatched to the processing unit in the next clock cycle. In the period of Cycle 2, the result of the first instruction, that is required by the AND instruction as the latest value in resister c, enters the common data bus. The AND instruction saves the result as the correct operand by matching the tag 1. At the same time, the reservation station dispatches the MOV instruction to the processing unit even though the previous AND instruction is not executed, which dynamically

schedules the dispatching scheme and keeps the processing unit working to provide higher utilization.

The example shows the case of one reservation station with one processing unit. In a real implementation, there are multiple reservation stations with processing units that each of them is responsible for one specific type of function. Different processing units and reservation stations are interconnected by the common data bus. Although the bus-type connection cause hardware complexity, processing units are isolated with each other so that they can have different pipeline stages so that processing units with different length of the datapath are easy to concatenate together.

For example, integer arithmetic units (ALU) always have much shorter datapath than floating-point units (FPU). There are two choices in traditional pipelined processors to add a new FPU datapath. One is to directly insert the FPU, but lower the clock frequency. The other one is to remake the integer ALU to align with the pipelined FPU. However, in out-of-order processors, FPU with multiple stages operate correctly with integer ALU with one stage without impact on the clock frequency. In summary, by considering reservation stations as the instruction pool, processing units in superscalar processors agree with the multiple-instruction-single-data architecture.

2.2.2 Data-Level Parallelism

In contrast to ILP, DLP refers to single-instruction-multiple-data (SIMD) architecture. The most common approach of SIMD is to use the packed data in registers. In vector processors, the values in each vector register are divided into multiple elements and the vector operation computes individually on each element instead of the full-width data, demonstrated in Figure 2.4. In the example, one vector instruction finish four operations compared to the common scalar instruction. The number of packed elements varies according to the application.

DLP brings tremendous speedup of applications that require massive and continuous data, including video decoding, image processing, and solving linear algebra. One graphic processing unit (GPU) in modern graphic cards is a common implementation of DLP. The graphic driver software vectorizes image processing tasks into several SIMD vector instructions that

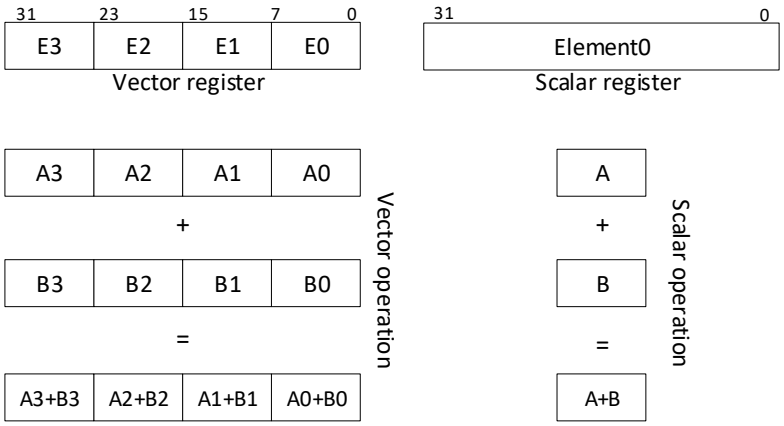

Figure 2.4: Vector operation of packed four vs scalar operation.

are executed by GPUs to deliver the output. The vector instructions in modern commercial GPUs are also proprietary.

Similar to the VLIW, challenges of DLP also involve the difficulty in general-purpose computing and compilers. However, many commercial ISAs have the vector extension instruction set to remain general-purpose computing with the basic instruction set while boosting the performance of specific tasks with the vector extension. Examples include SSE in Intel x86 and NEON extension in ARM. To fully exploit the SIMD instructions, the throughput heavily relies on software optimization and task-specific fine-tuning. Some general techniques are data alignment, loop unrolling, and prefetching. With respect to the hardware, the functionality and architecture simply follow the vector instruction set specification.

2.3 SIMD Extension

Artificial Intelligence (AI) is a trending topic today. Some important applications include image classification, object segmentation, and natural language processing, based on the structure of convolution neural networks (CNN). A typical CNN structure usually consists of convolution layers, pooling layers, activation layers, and fully-connected (FC) layers. The majority of the computations occur in the convolution layer, which uses a three-dimensional input feature map, one set of 3D parameters, to generate one channel of two-dimensional

results, aligned with the same depth of the input feature map. All channels of results are concatenated together to produce the final result in the convolution layer. The 2D convolution is a process of sliding a 3D filter matrix through the input layer. The 3D filter only shifts in two directions, width (W) and height (H) of the input feature map.

Conventionally, the size of 3D feature maps is noted as *(width,height,depth)* while the size of convolution kernels is noted as *(width,height,input depth,output depth)*. Figure 2.5 represents a 2D convolution with the *(5,5,3)* feature map and the *(3,3,3,2)* weights to generate the *(3,3,2)* output. 2D convolution requires intensive computation resources and data bandwidth in hardware realizations, therefore, it is a good benchmark to evaluate the performance of SIMD extension.

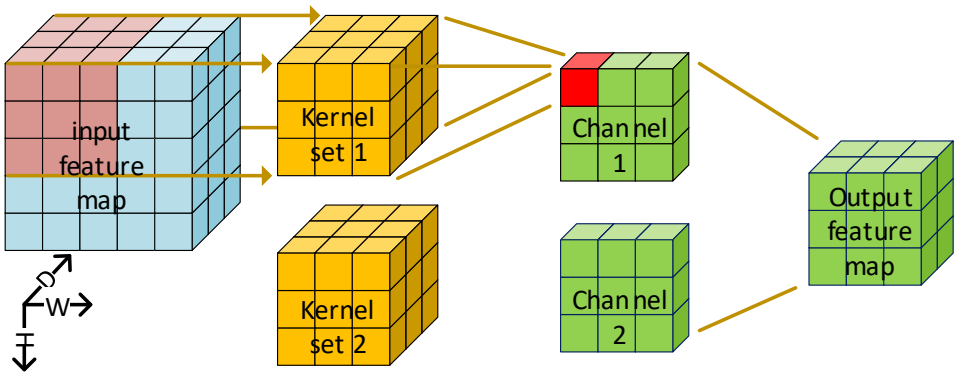

Figure 2.5: Process of 2D convolution in a convolution layer.

2.3.1 Standard V-Extension

The standard RISC-V vector extension is still a work in progress. The Ara [17], a 64-bit vector processor, is the well-known hardware implementation of the RISC-V vector extension based on the v0.5 draft. According to the latest specification, 7 new CSRs and a new set of 32 architectural registers are extended to the base RV32I instruction set. The extended vector CSRs are responsible to change the number of elements, vector length, rounding modes, which can be directly modified by Zicsr instructions during runtime. Some important CSRs

are selected from the RISC-V specification and are presented in Table 2.2. Values in the new vector register file are arranged as packed elements. The packing format varies upon the value in vector CSRs. Each 32-bit vector register can store the combination of 1 four-byte element, 2 two-byte elements, and 4 one-byte elements with the respective configuration in VTYPE. Leftover elements during the calculation are masked out by the value in vector register v0. The v0 register always supplies the byte-wise mask bits of masked vector instructions. Vector instructions are grouped into 5 categories:

Table 2.2: Definition of new vector control and status registers

Address	**Privilege**	**Name**	**Description**
0x008	URW	vstart	vector start position
0x009	URW	vxsat	fix point saturate flag
0x00A	URW	vxrm	fix point rounding mode
0x00F	URW	vcsr	vector control and status register
0xC20	URO	vl	vector length
0xC21	URO	vtype	vector data type register
0xC22	URO	vlenb	vector register length in bytes

- Vector load instructions including addresses increment with pattern unit-stride, strided and indexed,
- Vector store instructions including addresses increment with pattern unit-stride, strided and indexed,
- Vector atomic memory operations instructions to support synchronization between multi-cores,
- Vector arithmetic instructions including operations between scalar-vector, vector-vector and vector-matrix, and
- Vector configuration instructions.

The format and length of vectors change dynamically by the vector configuration-setting instructions to achieve high throughput on mixed-width operations in a single loop. The runtime configuration brings great versatility in the instruction set level.

2.3.2 Cambricon ISA

Cambricon ISA [1] is a machine learning specific instruction set and it has been proved effective among different kinds of machine learning techniques, including K-means [18], multi-layer perception [19] and convolution neural network [20]. There are three guidelines of Cambricon ISA and Cambricon-based hardware.

- Data-Level Parallelism - As mentioned before, machine learning tasks usually consist of massive data transmission and intensive computation. Thankfully, data flow trends to have a uniform and symmetric pattern. Weights in 2D convolution share the same factor with either input feature maps or output feature maps. The convolution can be vectorized by a factor of the input depth or the output depth. Either way brings the opportunity for SIMD architecture to leverage the performance while compressing the code density.

- Customized Vector/Matrix Instructions - Most of the convolution flow can be factorized as a loop of tensor operations. The 2D convolution can be unrolled into multiplying multiple vectors by matrices. Different types of layers can be abstracted to combinations of vector-matrix operations. The studied vector/matrix instructions are efficient for machine learning tasks.

- Using On-chip Scratchpad Memory - One of the biggest drawbacks of the standard vector register file is that the width of vector registers are fixed. Although multiple vector registers can be grouped together as a larger register to store longer vectors, it is more straightforward to use a block of memory as the storage. The 2D convolution often needs massive and continuous vector/matrix data with various sizes. The vector registers may not best suit the machine learning tasks due to their relatively smaller sizes, but higher costs. Moreover, because the sizes of vector/matrix are now defined

in each instruction, the limitation of fetching size caused by the fixed-width vector registers is also eliminated by using scratchpad memory.

As shown in Table 2.3, the Cambricon ISA follows the reduced instruction set computer (RISC) architecture that only has specific data transfer instructions to contact the external memory. All computing kernels initiate with vector/matrix load instruction to bring the necessary operands into the scratchpad memory. Processing units fetch operands from the scratchpad memory and also write the result back to it.

Table 2.3: Overview of Cambricon ISA (From: Table 1 Chen et al. [1])

Instruction type		**Example**	**Operands**
Control		jump, conditioanl branch	register (scalar value), immediate
Data transfer	Matrix	matrix load/store/move	register (matrix address/size, scalar value), immediate
	Vector	vector load/store/move	register (vector address/size, scalar value), immediate
	Scalar	scalar load/store/move	register (scalar value), immediate
Computational	Matrix	vector multiply matrix, matrix multiply scalar, outer product, matrix add/sub matrix	register (matrix/vector address/size, scalar value)
	Vector	vector element-wise arithmetic (add/sub, multiply/divide), vector transcendental functions (exponential logarithmic), dot product, random vector generator, max/min	register (matrix/vector address/size, scalar value)
	Scalar	scalar arithmetic/transcendental	register (scalar value)
Logical	Vector	vector compare (greater than, equal), vector logical operations(and, or, inverter), vector greater than merge	register (vector address/size, scalar)
	Scalar	scalar compare, scalar logical operations	register (scalar), immediate

The scratchpad memory is a small size of memory near the processing unit to handle quick access. Unlike the cache memory, the scratchpad memory does not belong to the memory hierarchy and only serves as a temporary storage space. The content in scratchpad memory must store to the external memory to reveal the latest data. The capacity of internal scratchpad memory is fixed to have 64KB for vector scratchpad memory and 768KB for matrix scratchpad memory.

The vector operands and matrix operands locate in the respective internal memories with different address spaces. The format of matrix multiply vector (MMV) instruction, presented in Figure 2.6, includes five source registers to provide three addresses and two sizes. Values in the Reg2 and Reg3 may be identical. However, because vector addresses and matrix

addresses are fixed by the ISA format, they refer to different memory spaces, whether matrix scratchpad memory (SPM) or vector SPM. As a result, the processing unit can only get sources and write the result back to the corresponding location.

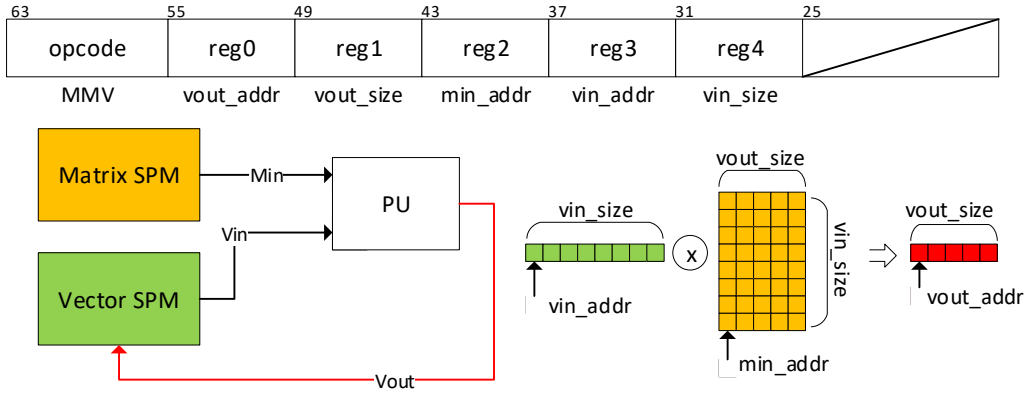

Figure 2.6: Matrix Multiply Vector instruction and its data arrangement.

Cambricon ISA assumes that the width of each element in a vector/matrix is fixed during runtime. It does not support mix-width computation to avoid data alignment issues in the scratchpad memory. Because the size of vectors is counted in terms of the number of elements, dynamically changing the element width significantly increases the logic control overhead to calculate the starting address of a vector/matrix.

Another great contribution of Cambricon ISA is that it introduces the vector greater than merge (VGTM) instruction, which is effective for max-pooling layers in CNNs. The instruction compares two vector element-wisely and keeps the larger value as one result. A max-pooling layer is commonly placed between convolution layers in CNNs. The purposes of max-pooling include to reduce the sizes of feature maps, to decrease sizes of parameters, and to prevent over-fitting by only keeping the largest value in a spatial region, as demonstrated in Figure 2.7. The max-pooling flow iterates along with the max pooling window with VGTM instructions. Each VGTM instruction recursively compares values in depth-wise with the previous results and finally keeps the largest value across the window.

This generalized instruction set covers different scenarios among machine learning techniques. Changing machine codes to realize a new algorithm are much easier than modifying

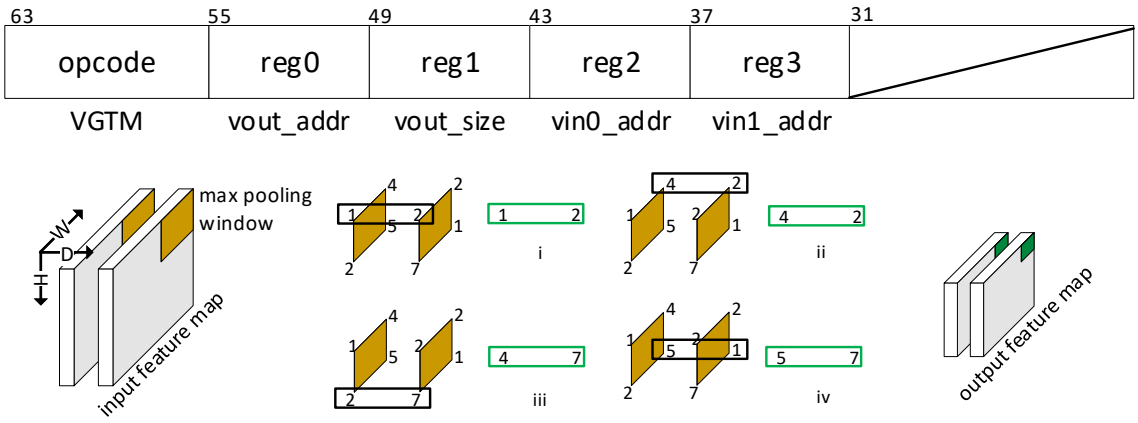

Figure 2.7: Vector Greater Than Merge instruction with the max pooling flow.

the hardware implementation. As a result, because of the rapid architecture change in the machine learning area [21–23], instruction-based accelerators [17, 24] are preferred over application-based accelerators for future-proofing.

2.4 Summary

This chapter introduces the background information that is related to this thesis work. The format and different extensions in RISC-V ISA are discussed firstly. It is an open-source ISA based on the RISC principles to deliver general-purpose computation. Developers can easily extend the instruction set with standard extensions and customized extensions to enhance performance. Then, the concept of parallel computing is presented. The two schemes of instruction-level parallelism, VLIW and superscalar, are compared with each other. Tomasulo algorithm, as a method to achieve the out-of-order execution, is explained with an example. Data-level parallelism is discussed as the approach to speed up data-intensive computing. In the end, the standard RISC-V vector extension and Cambricon ISA are included as two typical SIMD instruction sets. Convolution layers and max-pooling layers are shown as examples of SIMD processes.

Chapter 3

Superscalar Processor

3.1 Overview

The architecture of the processor is based on *Modern Processor Design: Fundamentals of Superscalar Processors* [25] and Ridecore [12] with several modifications to enhance the performance and support machine-level privileged instructions.

The key specification is provided in Table 3.1. The proposed design supports 32-bit RISC-V machine mode with integer multiplication and division extension, RV32IM + Zicsr + Zifencei, which is compatible with the standard GNU toolchain with "march=rv32im" flag. The 6 reservation stations (RS) match the number of execution units that include 1 load/store unit, 2 arithmetic units (ALU), 1 integer multiplication/division unit, 1 control and status registers (CSR) buffer, and 1 branch unit.

Figure 3.1 illustrates the top-level diagram of the proposed design. There are six stages of pipeline, including instruction fetch (IF), instruction decode (ID), data dispatch (DP),

Table 3.1: Key parameters of the processor

Instruction Set Architecture	RV32IM	Data Width	32-bit
Address Width	32-bit	No. of LDST Entires	4
No. of GPRs	32	No. of ALU Entires	16
No. of Commit Entries	64	No. of MUL Entires	4
No. of Speculations	5	No. of CSR Entires	4
Branch History Width	10-bit	No. of BRJ Entires	4
Ways of Superscalar	2	No. of Reservation Stations	6

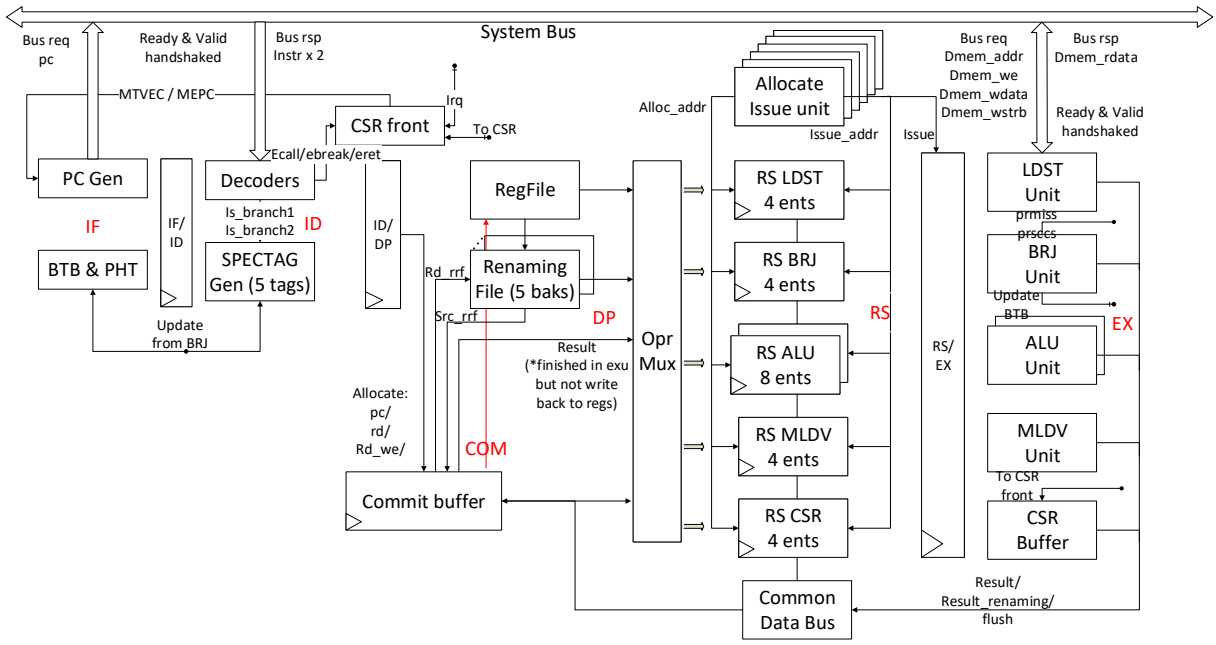

Figure 3.1: Top level diagram of the processor.

reservation stations (RS), execution (EX), and commit (COM). IF, ID, and DP can process two instructions at the same clock cycle. RS and EX can handle out-of-order execution following the Tomasulo Algorithm. The final COM re-orders the finished instructions and writes results back to registers sequentially.

3.2 Instruction Fetch

Program Counter (PC) is the address of the instruction that is currently executing. The current PC sends out to the system bus to fetch the corresponding instructions. Because the width of instructions in RV32IM is 32-bit, to match the 2-way superscalar, the instruction memory is implemented as 64-bit width so that each reading operation provides 2 instructions. As a result, the PC increases with the step of 8-byte by default. PC[31:3] is connected to the instruction memory as the true address to fetch the corresponding 2 instructions.

Due to the possible outcome of branch and jump instructions, the PC may be aligned to 4-byte. In that case, PC[31:3] still sends out as the instruction address, however, only the second instruction is valid. Based on the PC[2] bit, the invalid bit regarding the first instruction toggles high to invalidate the first instruction for the following stages.

For example, if the PC jumps to 0b1100, the 0b1 becomes the address of instruction memory. The fetched two instructions are at 0b1000 and 0b1100 respectively. The current PC[2] bit of 1 yields an invalid bit to remove the behavior of first instruction at 0b1000. The PC increases with the step of 4-byte in this case to match the 8-byte alignment. As a result, the fetch width varies between 1 instruction and 2 instructions.

3.2.1 Gshare Branch Prediction

A high-quality branch prediction algorithm improves pipeline throughput significantly. Unlike static branch prediction, such as always TAKEN for backward branch, dynamic branch prediction has a much better hit rate by visiting the past branch history to detect the correlated branches.

Gshare [26] technique is implemented in the design. The branch history register (BHR) is defined as 10-bit length generating 2^{10} numbers of 2-bit adaptive predictors. The 2-bit saturating counter has proven to have consistently good prediction performance [27].

Figure 3.2 shows the Gshare branch prediction module. A small direct mapping cache, called branch target buffer (BTB), is placed in the branch prediction logic to store pairs of the branch PC address and branch target address. The pattern history table (PHT) holds the set of predictors. The BHR indicates the record of the last 10 branch outcomes, in which the bit of 1 shows TAKEN and the bit of 0 shows UNTAKEN.

The current PC[11:2] is exclusive-or-ed with the BHR bit-wisely to generate a branch pattern. Based on the pattern, a 2-bit predictor is selected from PHT to predict the outcome of the current PC. If both the current PC hits in BTB and the predictor yields TAKEN, the matched target address in BTB becomes to the speculative PC in the next cycle. All predictors are initialized to weak TAKEN.

Both branch predictors and BTB update as soon as the branch and jump instructions finish in the execution stage. As a result, compared with updating those two parts after the commit stage, the branch prediction module can provide a better prediction hit rate based on more recent branch results. The value in the BTB can be traced back by the PC of branch instructions. The corresponding target address in BTB changes to the target address based on branch outcomes. The BHR also propagates with branch instructions through pipelines to

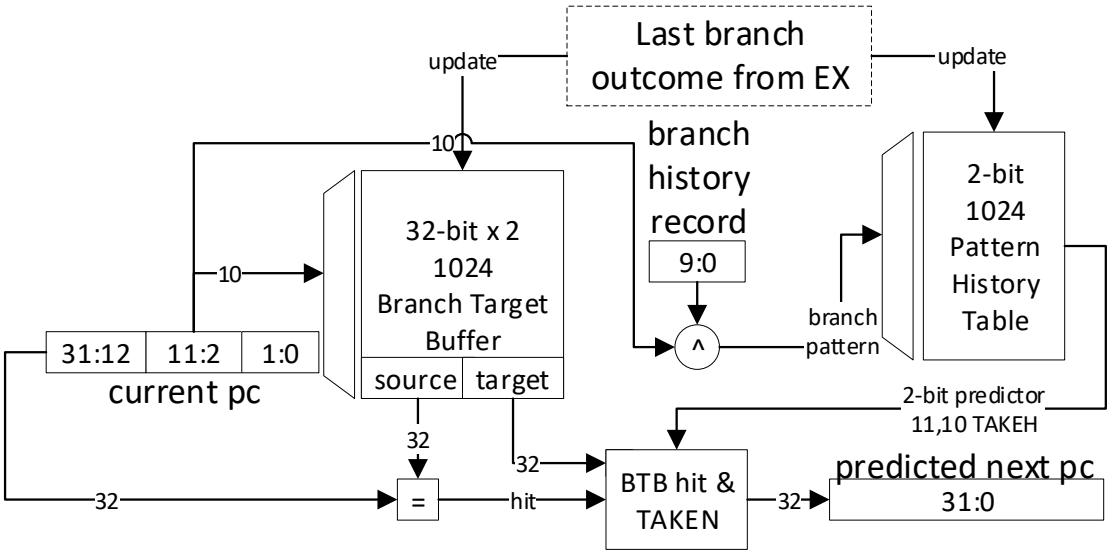

Figure 3.2: Branch prediction module.

guarantee that the same predictor can be located in the PHT, and updated based on branch outcomes, following the sequence in Figure 3.3.

3.3 Instruction Decode

There are two decoders in the ID stage to simultaneously handle two instructions coming from the instruction memory. Each decoder generates essential information related to one instruction for later stages. Some of the most important information is listed below.

- rs1, rs2, rd: They hold the register numbers of the first source operand, the second source operand, and the destination.
- imm_type: It indicates the encoding format of the immediate value.
- alu_op: It indicates the required arithmetic operation of the instruction, such as addition, shift-right, and signed-division.
- rd_we: It indicates whether the current instruction requires writing the final result back to the destination register, i.e., modifying the content in the register file. It enables the

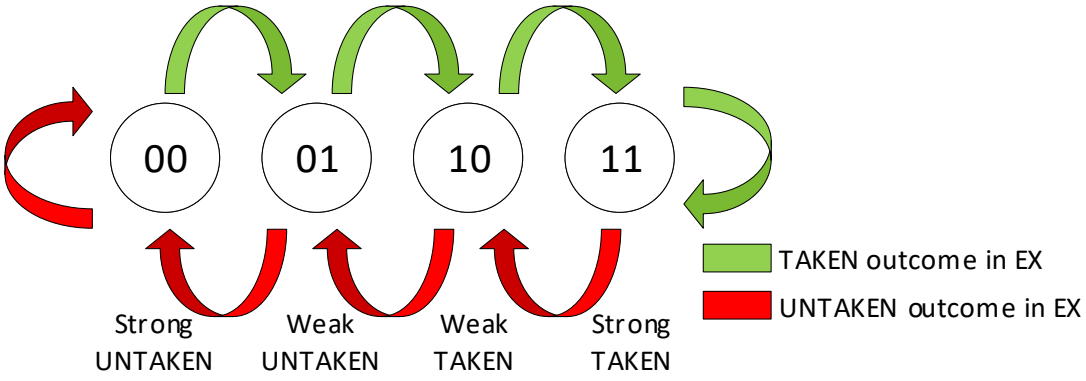

Figure 3.3: 2-bit adaptive predictor encoding scheme.

renaming process on the destination register.

- target_rs: It indicates which reservation station the current instruction is heading to.

- dmem_op: It indicates the data length of load/store instructions, including 4-byte, 2-byte, and 1-byte. It also determines the extension scheme, either sign-extend or zero-extend, regarding 2-byte and 1-byte data.

- system_op: It indicates types of system instructions, including FENCE.I, ECALL, and EBREAK. System instructions enable the CSR front module to trigger the exception handling logic.

- csr_id: It holds the CSR id that the current instruction is operating with.

- is_branch: It indicates that the current instruction is branch/jump and enables the speculation tag generator module to add a new speculative routine.

- inv: It indicates that the current instruction is not defined and supported in this processor. It disables the current instruction.

3.3.1 Speculation Tags

Two instructions in ID are padded with speculation bits and speculation tags. The speculation tag generator module is presented in Figure 3.4. Five sets of speculations and checkpoints are implemented by considering the worst-case scenario that 4 branch instructions are in the reservation station and 1 branch instruction is in the execution stage.

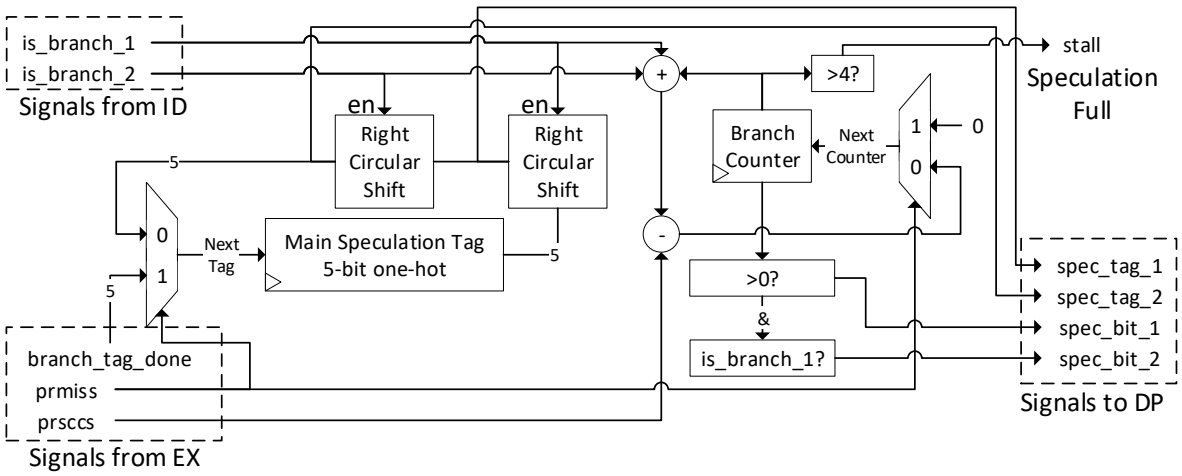

Figure 3.4: Speculation tag generator module.

The branch counter increases based on the incoming new branch instructions in ID and decreases based on the finished branch instruction from EX. The non-zero value represents that there are branch instructions in the pipeline but not executed yet. On the other hand, the incoming instructions are not speculative if the branch counter is equal to 0. If the branch counter is larger than 4, a stall signal is sent out to pause the IF stage to wait for the completion of previous branch instructions.

The speculation tag is encoded as 5-bit one-hot format. Those tags circularly shift right when new branch instructions are in this stage. The main speculation tag constantly shifts enabled by every new branch instruction. It rolls back to the speculation tag of the finished branch instruction, which restores its value before the prediction is performed.

The current design does not support out-or-order branch execution to simplify the recovery logic. When branch miss prediction happens, all speculative instructions, with high

speculation bit, are removed in every stage. When branch prediction hits, all the speculation bits toggle down asynchronously by matching the correct speculation tag (branch_tag_done) from the branch execution unit to remove the original speculative status.

3.4 Data Dispatch

Data dispatch and commit buffer are combined to achieve out-of-order execution. General RISC-V instructions consist of two source registers, rs1 rs2, and one destination register, rd. To handle the two-way issue and write back, the register file is programmed as 4-read-2-write memory. Figure 3.5 demonstrates the data structure of the register file, renaming files, and commit buffer.

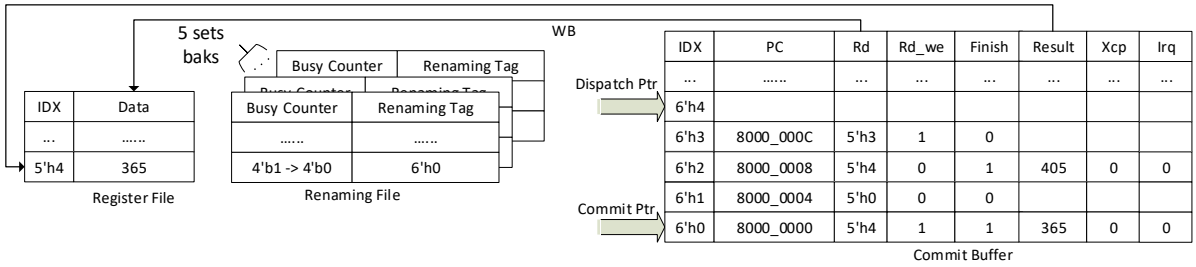

Figure 3.5: Data structure of register file, renaming files and commit buffer.

The register file consists of 32 general-purpose registers; while each renaming file consists of 32 entries of 4-bit busy counters and renaming destination to match with the register file. The 5 backups of renaming files match with each speculation tag.

The common busy vectors are replaced by the busy counters in this design, which represents that data is invalid if the respective counter is not equal to zero. For each instruction after ID, the counter increases by one, and the renaming destination (dispatch_ptr) is copied from commit buffer matching with the destination register. The counter decreases by one when the final result is written back to the register file from the commit buffer.

Each renaming file only contains busy counters and renaming tags. The temporary results in the traditional renaming register file are merged into the result column in the commit buffer to simplify the hardware. The finish bit represents the instruction is done and the result is

ready. There are three cases for source registers to take the right value in the DP stage:

1. Data is valid in the register file. Allocate the data to the target reservation station.

2. Data is invalid in both the register file and the commit buffer, i.e., the previous instruction is in the reservation station and wait to be issued. Allocate the renaming destination to the target reservation station and catch the result from the common data bus by matching the renaming destination.

3. Data is invalid in register file but ready in commit buffer, i.e., the result of previous instruction is ready in commit buffer, but it is not written back to register file yet. Take renaming destination as the address to get the result in the commit buffer and allocate the data to the target reservation station.

The dispatch pointer in the commit buffer is related to the DP stage instead of the COM stage. The instruction in DP takes dispatch pointer as its renaming destination, and allocates its PC and destination register to commit buffer, where is the beginning of the dynamic execution.

3.4.1 Renaming File Backups

Figure 3.6 shows the control of all backups. The main renaming file is always the latest renaming file that contains every dispatched instruction. By default, five backups are modified together with the main renaming file, and their content is the same as the main renaming file until branch instructions arrive at the DP stage.

The speculation mask indicates which speculation tags are activated in the processor. Once a branch instruction is issued from the dispatch stage, the corresponding bit in the speculation mask toggles up to freeze one backup renaming file as a checkpoint of this branch instruction. The corresponding bit in the speculation masks toggles down when that branch instruction is finished.

For example, the speculation mask of 5'b00110 indicates that two branch instructions, with tags of 5'b00100 and 5'b00010, are executing after the DP stage. The two frozen backups of 00100 and 00010 bypass the write logic of the following speculative instructions, but keep clearing the busy status, because all committed instructions are not speculative.

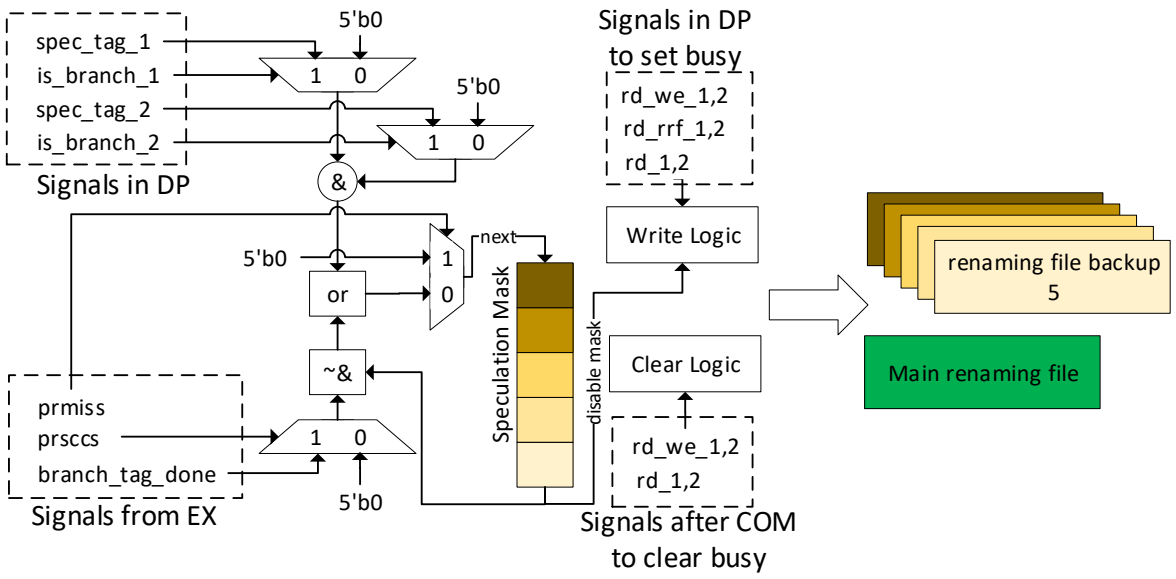

Figure 3.6: Control of renaming file backups according to speculation status.

Once the branch instruction is finished, two outcomes bring two different flows to treat the renaming files. In Figure 3.7, the left side shows the behavior of miss prediction, while, the right side shows the behavior of hit prediction.

In the prediction miss cycle, the corresponding backup that matches the completed speculation tag (branch_tag_done) is selected as the source sheet. This selected sheet overwrites back to the main renaming file and the rest of the backups to restore them before the branch instruction is dispatched. Because the processor handles branch instructions in-order, the oldest miss prediction causes the following predictions incorrect. As a result, the restoration happens not only in the main renaming file but also in every backup file.

On the other hand in the prediction hit cycle, the corresponding backup that matches the speculation tag is released. The main renaming file, with the latest busy counters and renaming destinations, is selected as the source sheet. The main renaming file overwrites the corresponding backup file.

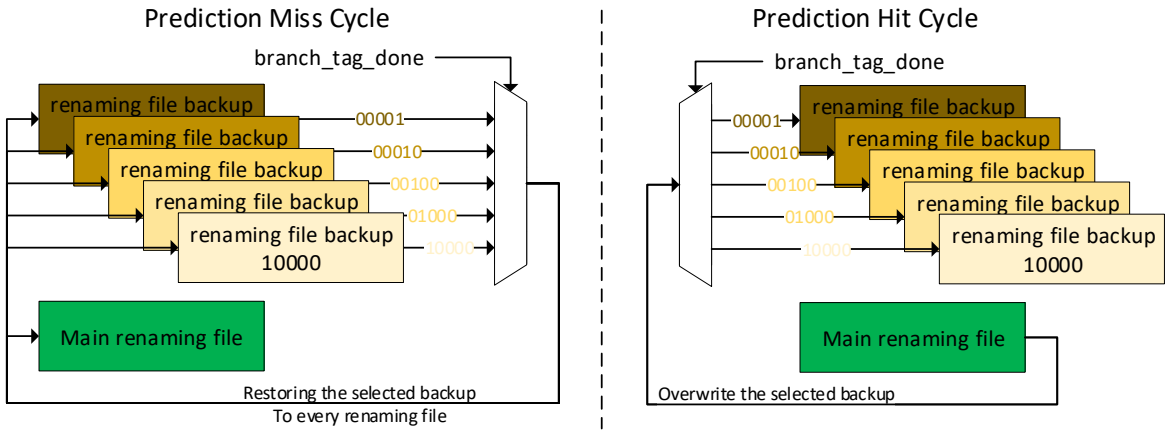

Figure 3.7: Renaming file handling in prediction miss cycle and prediction hit cycle.

3.5 Reservation Stations

There are two control logic blocks in each reservation station, instruction allocation, and issue. Instructions from the DP stage are allocated to entries of reservation stations according to their target_rs. Instructions in reservation stations with all operands ready are issued to their target processing unit in the EX stage. Figure 3.8 shows this process of the pair of one reservation station and one allocate/issue unit. For the sake of simplicity, the figure only shows the allocation of one instruction. However, dual dispatched instructions may lead to the same target reservation station so that the allocation width is doubled in the real implementation.

Instructions coming from DP are fully connected to every reservation station as the write data. However, only the write enable bit (we) of the matched reservation station toggles up. The identifiers are mapped as

- "0" for ALU: integer arithmetic instructions, such as ADDI, SLLI, AUIPC, ...
- "1" for BRJ: branch/jump instructions, such as BEQ, JALR, ...
- "2" for MLDV: integer multiplication/division instructions, such as MUL, DIVU, ...
- "3" for LDST: load/store instructions, such as LBU, SB, ...

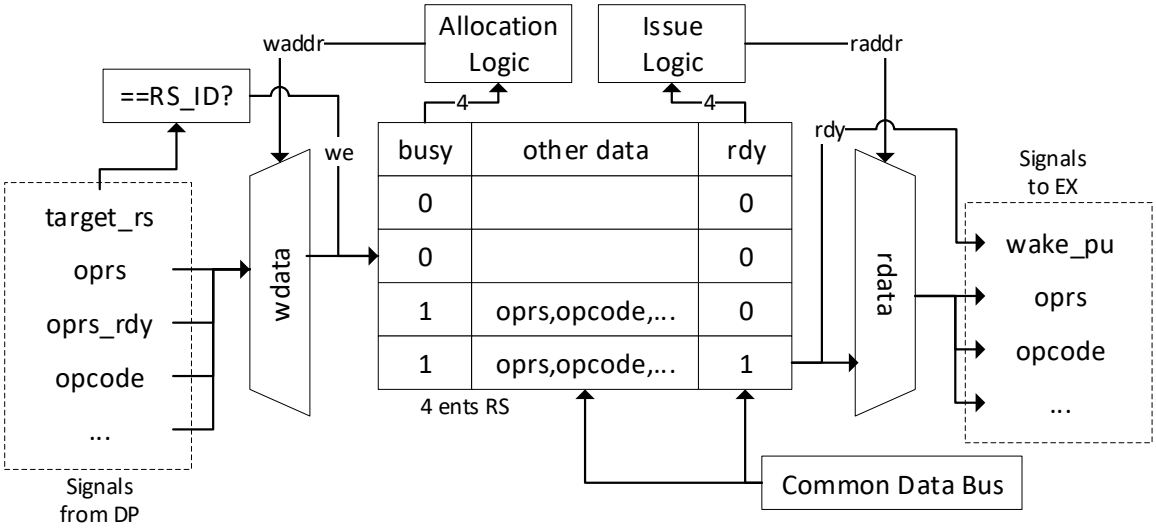

Figure 3.8: Allocation and issue process of a reservation station with 4 entries.

- "4" for CSR: CSR instructions, such as CSRRW, CSRRCI, ...

According to the busy vector, the allocation logic assigns a write address (waddr) to determine which entry to go. Similarly, according to the ready vector, the issue logic assigns a read address (raddr) to determine which instruction to be executed. The entry clears as soon as the instruction is issued to the processing unit. Only instructions with all operands ready can leave the RS stage. Every unready instruction, whose operands are renaming values instead of actual values, constantly monitors the result in the common data bus to find its operand.

There are two allocation and issue schemes in the RS stage to handle different kinds of instructions.

1. In-order scheme. The BRJ, LDST, and CSR reservation stations are programmed to issue the instruction in-order. The reason is that those types of instructions modify another space of memory, such as the external memory and the CSR buffer. Out-of-order computing causes potential synchronization issues in external memory. The in-order scheme turns each reservation station to a first-in, first-out buffer. The allocation address increases one dispatched instruction at a time, and the issue address increases one issued instruction a time. The issue address therefore always points to the entry that has the oldest instruction. As long as the oldest instruction is not ready, the

in-order reservation station does not issue any instruction to the processing unit, even though there are multiple ready instructions in the reservation station. As a result, from the perspective of the load/store unit, the load/store instructions come in as the order in the assembly code.

2. Out-of-order scheme. The ALU and MLDV reservation stations can handle out-of-order distribution. Because those instructions only modify the register file and the synchronization issue of the register file is already solved by register renaming, the out-of-order reservation station can always issue the ready instruction to increase the throughput. If there are multiple ready instructions available, the oldest instruction has the highest priority by comparing the renaming destination, because the later instruction always has a larger renaming destination.

Both schemes are limited in scope to their specific reservation stations. From the view of the whole processor, even though several load instructions are executed sequentially in their order, the rest of ALU instructions between those load instructions are executed potentially out-of-order in parallel. The commit buffer still must re-order those instructions and complete them in the original order.

Figure 3.9 shows the flush and clear logic in each reservation station based on the prediction outcome. As it is mentioned in the previous section, the processor does not support out-of-order branch. Therefore, as soon as the miss prediction happens, speculative instructions are flushed away no matter their speculation tags. RS stage suffers one clock cycle to remove those instructions and restore allocation/issue addresses. If the prediction yields success, reservation stations asynchronously clear the speculation bit with the matched speculation tag to remove the speculative status of the corresponding instructions.

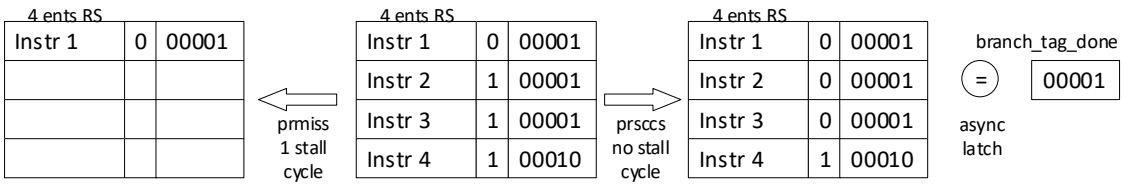

Figure 3.9: Behavior of reservation stations according to the prediction outcome.

3.6 Execution Units

There are six processing units aligned with each reservation station, except the ALU unit. Commonly, most of the instructions in a program are arithmetic instructions, as a result, two ALU units are attached in the EX stage that can execute two arithmetic instructions in parallel. Dependencies and hazards are solved in previous stages, therefore, most of the processing unit is the simple one-cycle combinational logic.

Every processing unit takes the operands and operation codes from the RS stage. The computation is activated by the wake_pu signal to show that all operands are ready. Upon the calculation completion, every result is loaded to the common data bus with its renaming destination for broadcasting. At the same time, the finished instruction also writes its result and sets the finish bit in the commit buffer, which uses the renaming destination as the address to find the corresponding entry, demonstrated in Figure 3.5. The Xcp bit is toggled high if a hardware exception happens inside the processing unit, such as load address misaligned in load/store unit and instruction address misaligned in branch/jump unit.

The one-cycle combinational integer divider usually requires excessive logic resources and has much longer latency [28]. A sequential restoring binary divider is implemented in the MLDV unit to execute division operations. The division of two 32-bit operands requires 32 clock cycles to get the quotient and remainder done. However, the overall clock frequency does not decrease with the division datapath. Moreover, the 32-cycle division is still faster than the software emulated division, and the capability of division instructions reduces the code size by removing the division subroutine in assembly codes.

3.7 Instruction Completion

The commit buffer allows instructions to complete in-order. It is implemented with the first-in, first-out method, which takes up to 2 instructions from the DP stage (dispatch_ptr) as the input, and pushes out up to 2 instructions after the COM stage (commit_ptr) as the output. The detailed structure is provided in Figure 3.10.

Upon the completion of an instruction, the commit buffer writes the final result to the

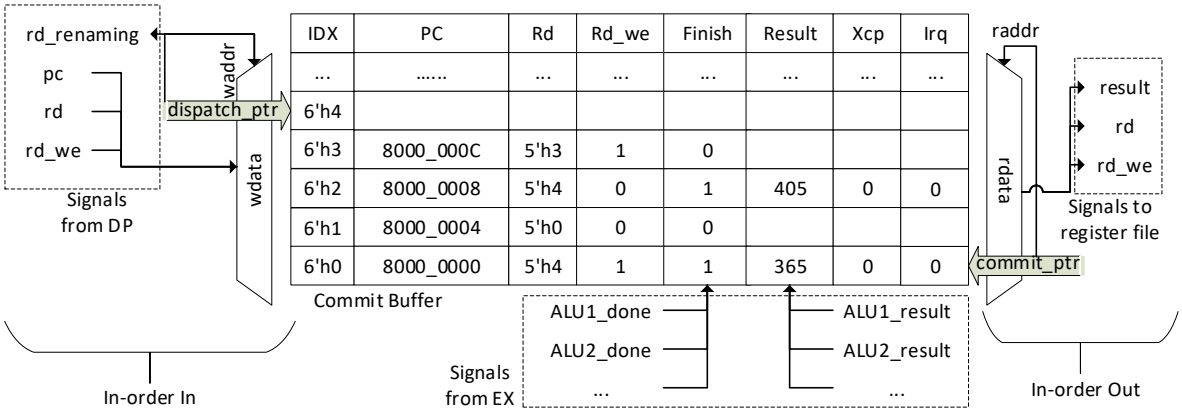

Figure 3.10: Structure and flow of the commit buffer.

destination register in the register file and decreases the busy counter in the respective entry of the renaming file to show that the destination register is up-to-date and no instruction is modifying its content. At this point, a full pipeline cycle of one instruction is ended.

As previously mentioned, the proposed design issues branch instructions to the execution unit in sequence so that the latest branch result can immediately write back to the branch prediction module instead of reordering results in commit buffer. Therefore, the commit buffer does not need to cover the scenario that can potentially cause structure hazards, e.g., commit two branch instruction to the branch target buffer that only has one write port. The commit buffer can always commit two instructions to achieve higher commit throughput.

On the other hand, busy counters avoid write-after-write hazards. Conventional 1-bit busy has to hold the instruction with the destination register already in busy. Alternatively with busy counters, the core can constantly dispatch multiple instructions with the same destination register until the corresponding counter reaches its maximum. The rearrangement of the data flow between dispatch and commit exploits the 4R2W register file to reach higher overall throughput.

3.8 Summary

This chapter introduces the detailed hardware implementation of the dual-issue out-of-order RISC-V processor compatible with RV32IM instruction set.

The IF stage sends the current PC as the address of instruction memory to fetch at most two instructions in the next clock cycle. The PC of the next cycle is predicted by Gshare branch prediction with the combination of the current PC and the last 10 branch history record.

The ID stage generates data and operation codes for the rest of the stages. Instructions coming from the instruction memory are padded with speculative tags in the case of prediction miss. The speculative tag is updated if a branch or jump instruction is detected in this stage.

The DP stage fetches the required source operands. The operands may locate in the register file, commit buffer, or common data bus, according to the renaming status of registers. Destination registers are renamed to the corresponding address in the commit buffer. Finally, all information that is required for executing the instruction is stored in the targeting entry in reservation stations.

The RS stage monitors the common data bus for the unready operands and dispatches instructions to the execution stage if their operands are all ready.

Instructions are executed in the EX stage. Each result is stored in the assigned entry in the commit buffer and is broadcasted through the common data bus to acquire the latest results for previous stages. If the branch unit detects a missed prediction, all instructions under speculation are flushed out.

The COM stage sequentially retires up to 2 instructions and frees those entries. The results of the completing instructions are moved to destination registers in the register file.

Chapter 4

Vector Co-processor

4.1 Instruction Set

The proposed extension vector instruction set follows the same idea of Cambricon [1] ISA, which introduces data-level parallelism, vector/matrix operations, and scratchpad memory. The dataflow of the vector computation follows the reduced instruction set computer style. Several Cambricon instructions are selected to perform forward inference of a typical convolution neural network.

However, unlike the Cambricon ISA that has two different internal addresses to separate vector and matrix operands, the proposed ISA uses a unified internal address to indicate operands. In this design, multiple scratchpad memory banks are implemented to store both vector and matrix operands. The 4-most-significant-bits (MSB) in the internal address become a tag to indicate which bank is being referenced. For example, the address 0x0000_0010 refers to 0x10 in bank 0 while address 0x1000_0008 refers to 0x8 in bank 1.

The tag acts like a "pseudo" vector register in the standard RISC-V vector extension to locate vector operands. Vector registers only contain the packed data that has a fixed length. In general CNN architecture, the size of each parameter varies from layers to layers, and the massive data is usually continuous. Therefore, the scratchpad memory is preferred in CNN applications instead of vector registers.

However, different instructions access to the same memory block requires additional logic, including load-link and store-conditional, to solve synchronization issues. The divided address space with tags, on the other hand, directly synchronizes the memory access orders. When an instruction is modifying bank 0, following instructions that need the content in bank 0 are idled, whereas, following instructions that do not need the content in bank 0 can

execute independently. This arrangement of addresses causes instruction-level parallelism in hardware design to improve the performance.

Figure 4.1 shows the encoding format of the vector transfer instructions. The vector load instruction (VLOAD) loads a block of data from the external_addr in external memory, and stores the data to the internal_addr in internal memory. The vector store instruction (VSTORE) has the same flow but reverses the source and destination. The V_size provided in Reg1 is the number of each element. For instance, if each element is byte-data and the address is byte-aligned, those two instruction moves a block of data from *source_address* up to *source_address+V_size-1* to *destination_address* up to *destination_address+V_size-1*. The vector copy instruction (VCOPY) does not access the external memory. It moves a block of data between internal memory banks only. The vector-scalar copy instruction (VSCOPY), on the other hand, duplicates the scalar value in Reg2 to the size of V_size and stores them to the internal destination address.

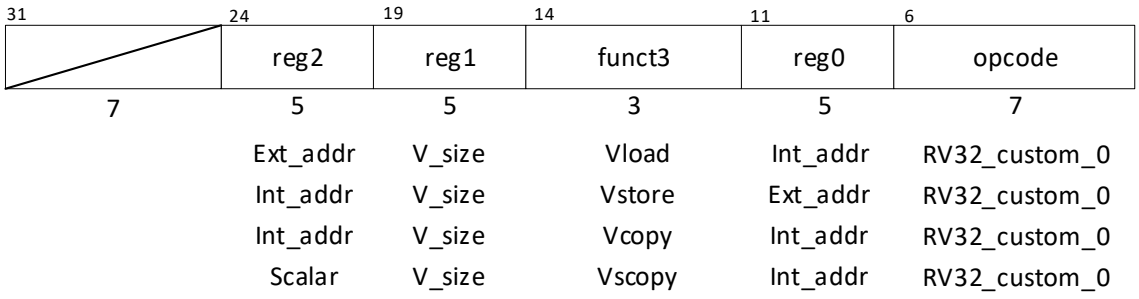

Figure 4.1: Encoding of vector transfer instructions.

Figure 4.2 shows the encoding format of the vector arithmetic instructions. Vector arithmetic instructions perform element-wise operations on two vectors. V_size represents the size of both input and output vectors. Vector addition and vector multiplication instructions (VADD and VMUL) take the operands from two source addresses and write results to destination address with the size of V_size. Vector-greater-than-merge instruction (VGTM) is efficient in the max-pooling layer. Each element in a vector is compared with each element in the other vector. The vector result contains the larger corresponding elements. The vector-scalar multiplication instruction (VSMUL) differs from the VMUL in Reg3, which is changed

to the address of a scalar source. The VSMUL multiplies every element in the source vector with the same scalar value and stores the results back to the destination address with the same size as the input vector.

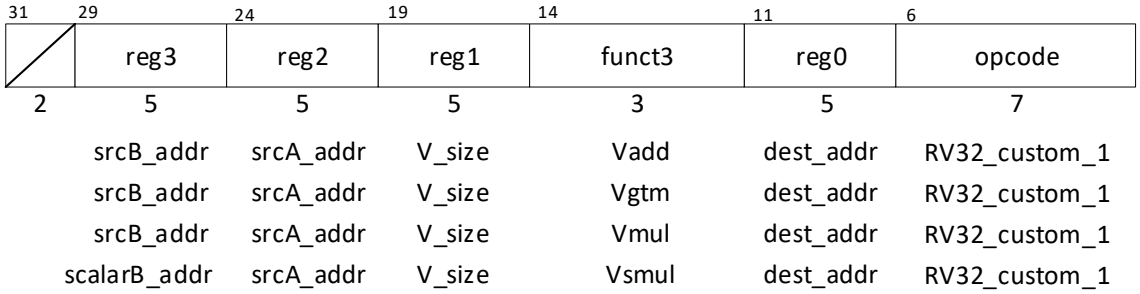

Figure 4.2: Encoding of vector arithmetic instructions.

Figure 4.3 shows the encoding format of the vector-multiply-matrix (VMM) instruction. VMM instruction calculates the matrix product of a 1-D vector with a 2-D matrix. The input vector starts at srcA_address with the size 1 $\times$ input_size. The input matrix starts at srcB_address with the size input_size $\times$ output_size. To make this instruction general in different layers, five registers have to be presented in it. There is no rest of the encoding space for function select in 32-bit instruction with five registers. However, the combination of VMM and VADD can accomplish the inference of convolution and fully-connected layers adeptly.

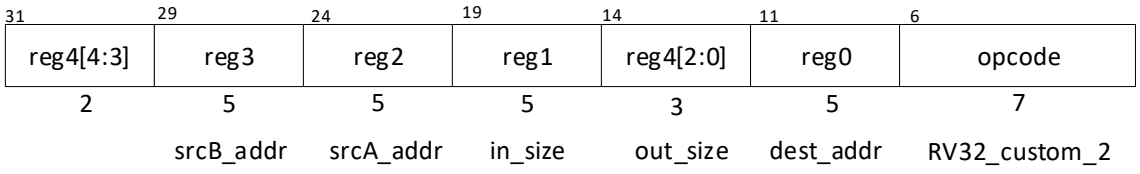

Figure 4.3: Encoding of vector multiply matrix instruction.

The extension vector instruction set is coupled with standard RV32-IM. The RISC-V instructions handle scalar calculation and jump/branch flow in a program. Table 4.1 shows

a summary of all defined instructions. Despite the RV32-IM instruction set, the proposed vector instruction set is divided into three groups. Since RISC-V ISA has the free space for custom instructions in the opcode, it is straightforward to map each group into custom_0, custom_1, and custom_2.

Table 4.1: ISA summary

	Instruction	Opcode
Scalar and control	addi, beq, xor...	General opcodes
Vector transfer	vload, vstore, vcopy,vscopy	Rv32 custom_0
Vector arithmetics	vadd, vgtm, vmul, vsmul	Rv32 custom_1
Vector multiply matrix	vmm	Rv32 custom_2

4.2 Integration

The vector instructions require additional scratchpad memories vector arithmetic units alongside a standard RISC-V processor, which creates a vector block as a co-processor. In this design, the co-processor is coupled to a dual-issue, out-of-order, 32-bit processor with RV32-IM implementation. The co-processor shares 32 general-purpose registers and an external memory port with the processor. Figure 4.4 is the full picture of the hardware. To drive the vector block, the standard RISC-V processor requires four adjustments.

1. In the decoder of the scalar processor, extra logic is required to decode part of the vector instructions. There are two source registers and one destination register in standard RISC-V instructions. However, the proposed instructions have up to five source registers and no destination register. The decoder expands to five source ports for fetching the data from registers if a vector instruction goes to the decoder. Based on the combination of opcodes and function select bits, the decoder also generates a vector opcode that targets the vector decoder and computation block in the co-processor.

2. The general purpose registers in the original dual-issue processor are implemented as a 4-read-2-write memory file. Because two vector instructions require up to ten registers

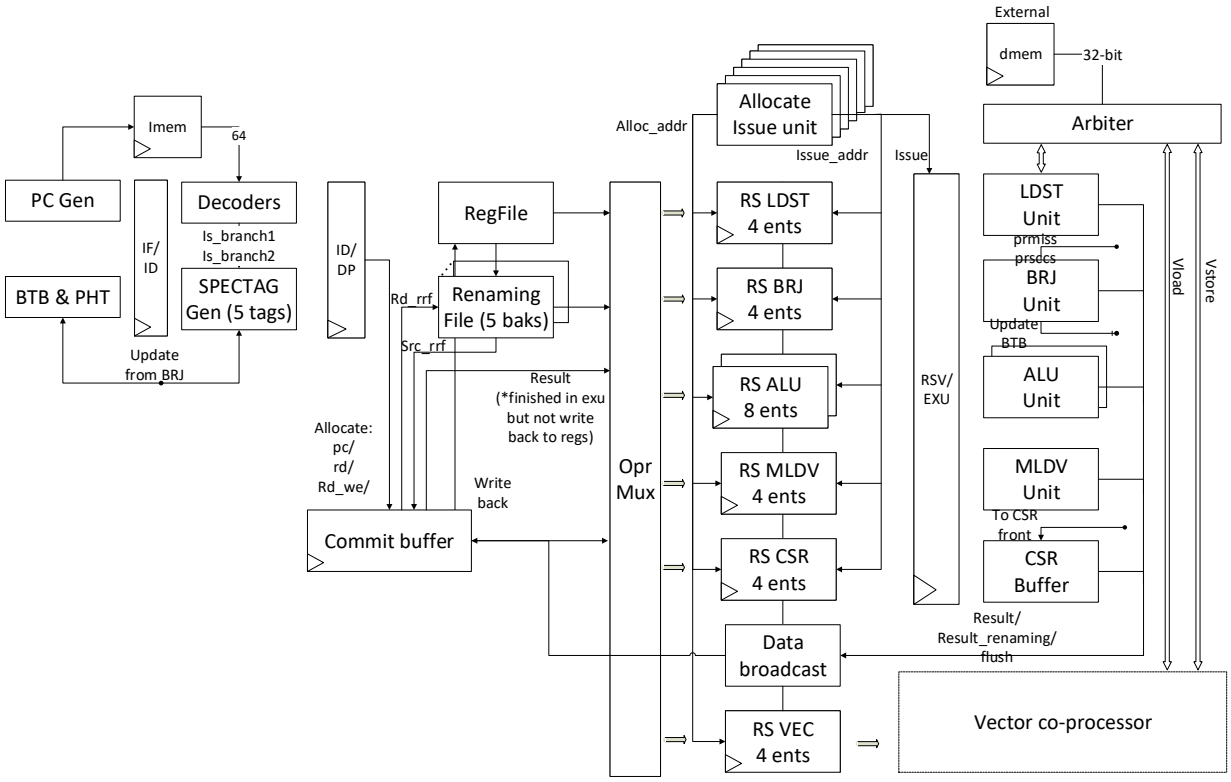

Figure 4.4: Overview of the hardware implementation.

read in each clock cycle, the register file is replaced to 32-bit registers completely from the previous 4-read-2-write memory file. The renaming register files and temporary results in the reorder buffer also expands to ten ports data read if two vector instructions happen in the same clock cycle.

3. An additional reservation station is added after the dispatch stage. All vector instructions go to the vector reservation station, which is the front-end of the vector co-processor. The vector instruction transfers to the vector block if all source registers are ready and it is not under branch speculation. As a result, the vector block can eliminate the logic for missing branches and data hazards in source registers. This reservation station in-order dispatches vector instructions to the co-processor if operands in all registers are ready.

4. To share one same external memory port, a priority arbiter is placed between the load-store-unit in the scalar processor and vector-load-store block in the vector co-

processor. Since most of the temporary results in vector calculations switch between different internal memory banks, the priority arbiter operates adequately to access the external memory. The scalar load-store-unit has the highest priority while the vector-load unit has the lowest priority. The scalar processor frequently accesses external memory relative to the stack pointer. However, if the scalar processor tends to access the external address that the vector block also tends to do so, it is necessary to add one FENCE instruction before the scalar load/store to avoid data hazards in the external memory. The FENCE operation guarantees that the scalar load operation cannot be executed until the previous vector store operation is finished.

Figure 4.5 presents the top-level diagram of the vector co-processor which consists of two pipeline stages. The implementation of memory dirty bits is referred to register renaming file, and the implementation of the entries is referred to as the combination of reservation stations and commit buffer. The instruction board stage receives vector instructions that are dispatched from the front-end in the main processor, the vector reservation station. In this stage, each entry holds sources/addresses of a vector instruction and traces its status. The instruction without memory or processing unit conflict is issued to the corresponding sequencer as its processing unit.

There are four sequencers with six master ports to drive the four scratchpad memory banks. Each sequencer is isolated from others and only accesses to the banks that the instruction board is assigned. The sequencer of the completed instruction updates the memory dirty bits and the instruction status in the corresponding entry. The retired entry sends a clear signal to the commit buffer in the main processor.

4.3 Wrapped Memory

Most of the computation in the inference of a convolution neural network needs spatial accesses to parameters and feature maps with variable data sizes, which means that if a particular memory location is referenced, its nearby memory locations will be referenced soon. Additionally, the capability of tightly storing different blocks of data increases hardware utilization. Therefore, the true-dual port random access memory (RAM) is selected to achieve

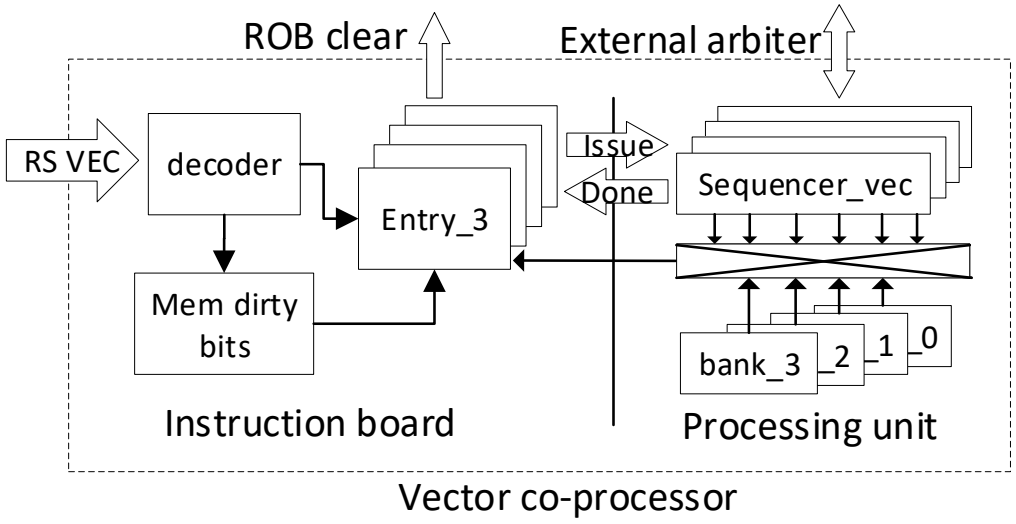

Figure 4.5: Top-level diagram of the vector co-processor.

those goals. As demonstrated in Figure 4.6, the RAM block is wrapped by additional logic to handle cross-alignment access. Each memory bank has a bandwidth of 64-bit and depth of 9-bit. Four banks yield 16KB in total.

Take reading 8-byte data from **0x3** as an example.The address **0b0011** from sequencers (byte-aligned) splits into a floored 9-bit addr_a **0b0** and a ceiling 9-bit addr_b **0b1**, which connect to two ports of the RAM respectively. The least significant 3-bit in the address **0b011** becomes the left-shift-amount value for the 128-bit combination of rdata_a and rdata_b. The least significant 64-bits are the requisite 8-byte data from the address **0x3**. The writing process is similar to the reading process. The write-strobe wires, wstrb, split to byte_ena_a and byte_ena_b after the shifting, which prevents overwriting the existing data. In short, the true-dual-port RAM simultaneously processes on single memory access to perform misaligned memory access in hardware.

4.4 Vector Instruction Board

Figure 4.7 presents the block diagrams within the instruction board stage in the proposed co-processor. There are four entries that hold the information and status of each dispatched

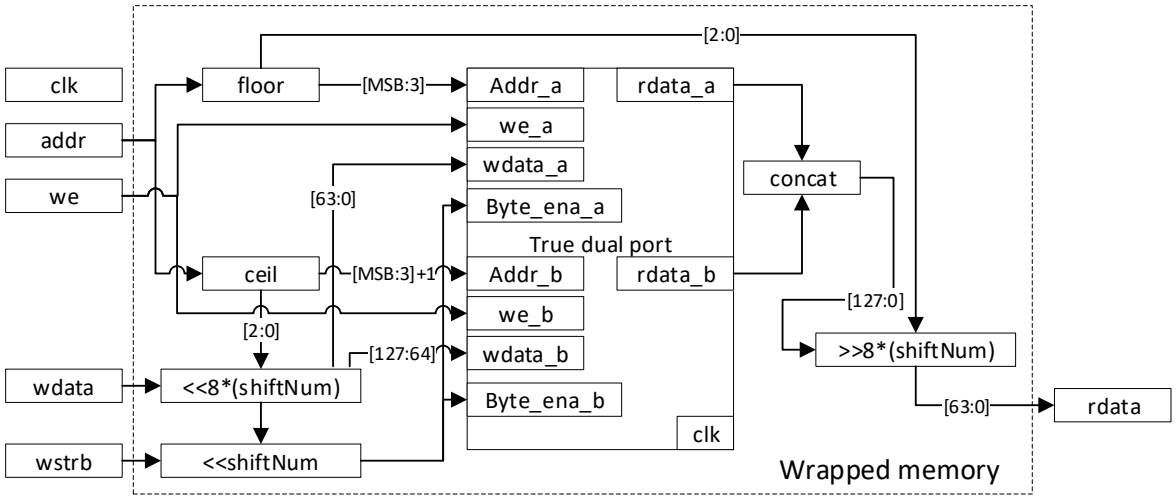

Figure 4.6: Structure of each memory bank.

instruction. The allocation address is determined by the result of a leading-zero-counter on the 4-bit busy vector. If all the entries are occupied, the scalar processor stalls to wait for the co-processor. At the same time, the vector decoder determines which memory banks are related to the vector instruction. The top hex digit in the internal address represents which memory bank is referred to. Since the wrapped scratchpad memory has one port to perform either read or write, multiple accesses to one memory bank can cause a structural hazard. To solve this problem, a 4-bit memory access mask is generated by the vector decoder based on the types of instructions. VLOAD, VSCOPY, and VSTORE access one internal memory bank as the source or the destination. VCOPY accesses up to two internal memory banks as the source and the destination which may refer to the same bank. Similarly, the rest of the instructions access up to three internal memory banks.

However, to simplify the hardware, those instructions are forced to use only two banks such one of the sources has to be the same as the destination in the assembly codes. For example, a VMM instruction multiplies a vector starting at **0x1000_0000** by a matrix starting at **0x2000_0000**. The destination bank is limited to either bank_1 or bank_2. It is detected as an exception that the other memory bank is referenced as the destination bank in the vector decoder module. At the same time, the vector decoder provides 0b0110 as the memory mask. Moreover, the vector decoder generates a 4-bit one-hot function mask to select the

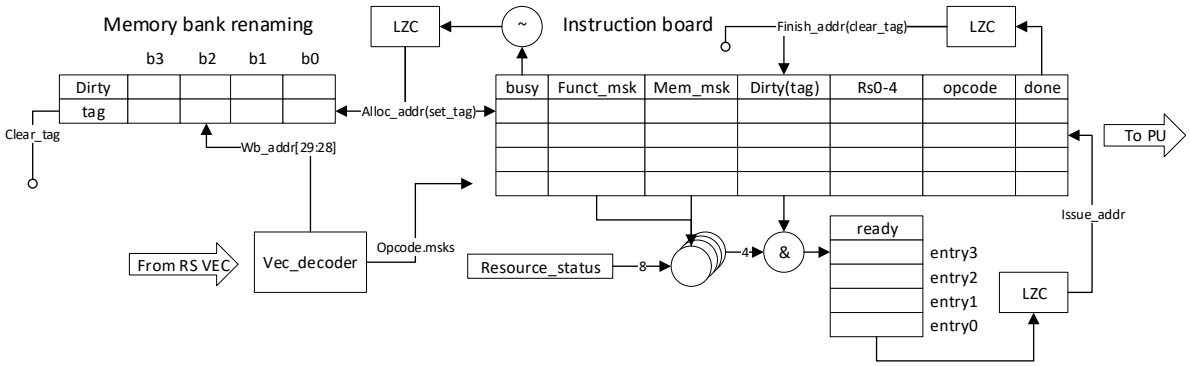

Figure 4.7: Structure of vector instruction board.

target function unit, which represents load unit, store unit, multiplication unit, and vector unit from the most-significant-bit (MSB) to the least-significant-bit (LSB).

Meanwhile, the vector renaming module controls the write-back order of the destination bank. For each vector instruction that requires memory write-back, the renaming module sets the dirty bit regarding the destination memory bank and leaves the entry address as the renaming tag for the following instructions. When this instruction is ended, it broadcasts the entry address to other entries to release the dirty bit by matching their tags. This process treats each memory bank as a pseudo-register and each vector entry as the renaming destination, which is similar to the register renaming technique in the Tomasulo algorithm. From the sequencers and the memory multiplexer in the next stage, the resource status that is a concatenation of the busy status of function units and memory banks, is connected to each vector entry. The resource status bit-wise **AND** with the concatenation of function mask and memory mask identifies whether the required computation resources are idle. The instruction can be dispatched to the corresponding functional unit with the memory dirty bit low. The dispatch address is also determined by a leading-zero-counter (LZC) on the 4-bit ready vector. At this point, the potential data dependency is resolved by the dirty bit generated in the renaming module; therefore, multiple independent vector instructions can be computed simultaneously in the functional units to provide the potential instruction-level parallelism.

4.5 Processing Units

Figure 4.8 shows the block diagrams with the second stage in the proposed co-processor. There are four sequencers based on finite-state machines (FSM) to generate the address in memory banks, feed operands to function units, and write results back to memory banks. In this prototype, each function unit can handle 8 elements in parallel, dependent on the bandwidth of the scratchpad memory. As previously mentioned, vector arithmetic instructions are designed to access up to 2 memory banks, therefore, there are 6 master ports to initiate the internal memory access. The store and load units have one master port in each, while vector and multiplication units have two master ports in each. The internal bus multiplexer updates the memory banks' status regarding the memory mask and the master port id. The multiplexer changes the connection to 4 slave ports representing memory banks. Once an instruction is finished in a sequencer, the corresponding bits in memory banks' status turn low to release the access of internal memory. The end-stage in each sequencer also sets the finish bit in the corresponding entry, which further releases the entry for subsequent instructions.

The load and store sequencers share the same external memory bandwidth, which must hold their current status if the external bandwidth is not granted. The internal port of the load sequencer initiates the WRITE operation with the increment of 4-byte address due to the 32-bit bandwidth of the external memory. The store sequencer performs the same address pattern. No arithmetic unit is required in load and store sequences.

The FSM in the vector sequencer has three different pattern, *read-write*, *read_a-read_b-write* and *copy*. The *read-write* pattern works for arithmetic instructions with the access of two memory banks. Two master ports initiate **READ** operations at the same time and write the results back through the corresponding port. However, two source addresses may refer to the same memory bank. In this case, the read stage splits into 2 cycles with the pattern of *read_a-read_b-write*. The *copy* pattern is in charge of data transfer instructions, which does not require an arithmetic operation. One port performs **READ** and the other one performs **WRITE** to move the data between two banks. Eight 8-bit adders and greater-than-merge logic are embedded in the vector sequencer to support VADD and VGTM respectively.

A two-stage dot-product unit in Figure 4.9 is implemented in the multiplication sequencer.

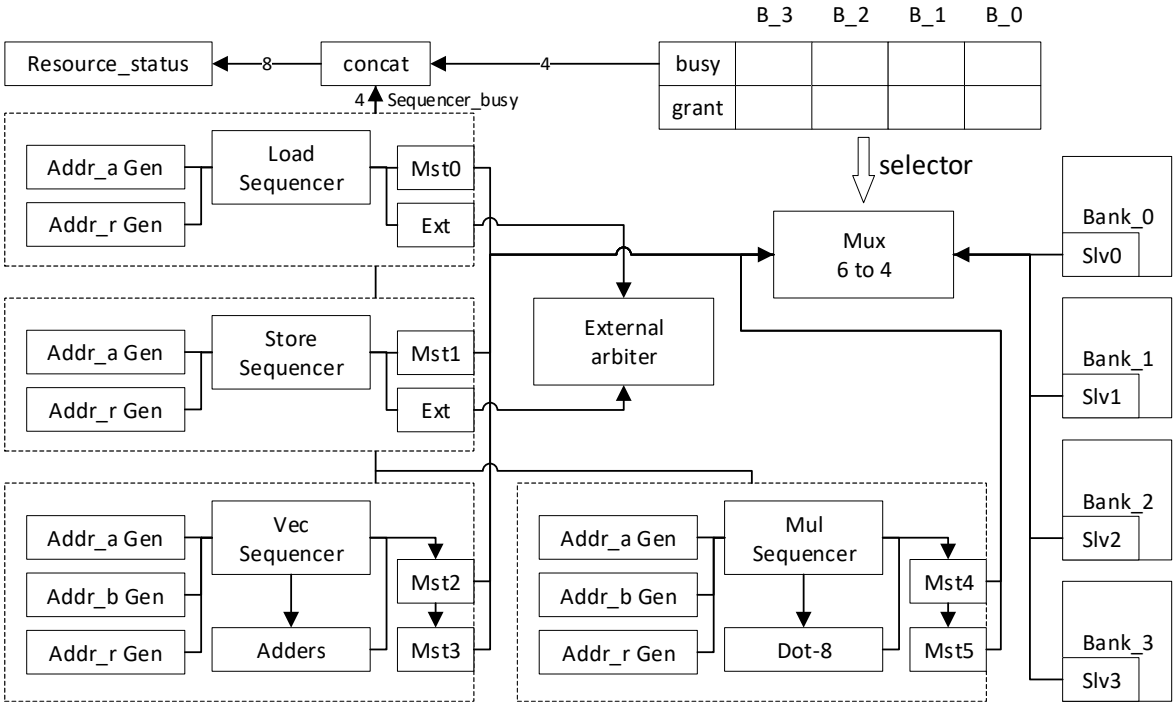

Figure 4.8: Structure of processing units.

The first multiplication stage calculates the product of each pair of elements and the second accumulation stage sums up the previous products. For VMUL instructions, only the first stage is activated and the FSM pattern is the same as the pattern in the vector sequencer. The VMM instructions execute differently not only with the accumulation stage enabled but also with the address generators. For example, a VMM instruction has the output_size of 2, the input_size of 17, the vector address of **0x1000_0000** and the matrix address of **0x0000_0000**, which multiplies a 1×17 vector with a 17×2 matrix to generate a vector of length 2. Because of the 8-element bandwidth in memory banks, 3 read cycles and 1 write cycle is required to store the first result element. In the third read cycle, only one element is feed to the dot-product unit while the other 7 elements are dropped out by the control of the mask bits. Once the calculation of the first result element is finished, the vector address rolls back to **0x1000_0000** from **0x1000_0010** and the matrix address increases to **0x0000_0011** from **0x0000_0010**. At the same time, the accumulator in the dot-product unit resets to zero to start the new iteration for the next result element.

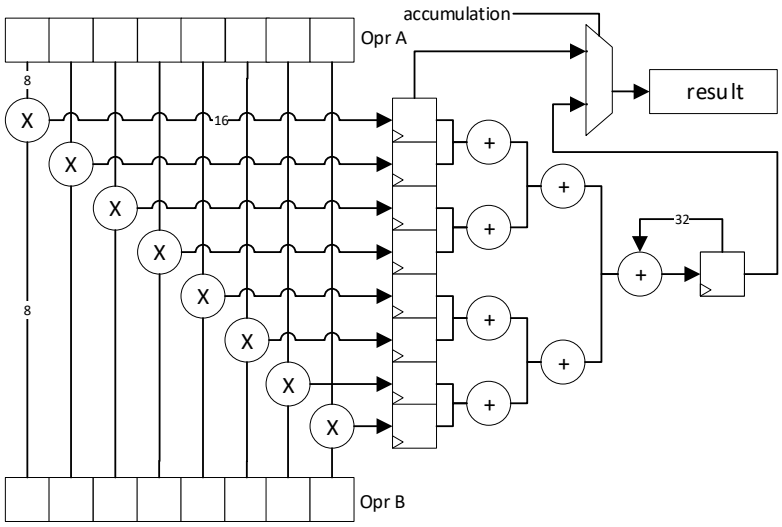

Figure 4.9: Structure of dot-product unit.

The proposed design mainly focuses on the evaluation and optimization of the LeNet-5 benchmark in hardware. Although the LeNet-5 model is relatively simple among other CNN models, such as the VGG [29] and the MobileNet [21], it contains the basic layers and common structures of a typical CNN model. Since the proposed CNN processor is instruction-based, other models can be implemented with the similar optimizations and software flows. For the model with a large scale of parameters and intermediate data, like the VGG, the vector program can separate the input feature map and weights at the same time. Based on the size of the defined internal memory, the excessive temporary data must be stored to the external memory and loaded back upon the computation request. Such memory scheduling heavily depends on the control program of the RISC-V core. For the novel types of convolution layers, like the depth-wise convolution in the MobileNet, the vector program can deliver specific modifications based on the generalized assembly kernel. To perform the depth-wise convolution, the VMM instruction is replaced by the VMUL instruction in the assembly kernel, which only provides element-wise multiplication without the summation. After the depth-wise convolution is finished, the point-wise convolution sums up the previous results, which can be treated as a normal convolution with the kernel size of 1×1.

4.6 Summary

This chapter presents the detailed extension vector instruction set for CNN tasks. A vector co-processor is attached to the previous superscalar RISC-V core to specifically execute the vector instructions. Detailed modules are explained in this chapter with examples.

The proposed vector instruction set following the Cambricon ISA changes the original vector/matrix address space to one unified internal address space. The vector instructions are mapped to 32-bit RISC-V format as the custom instructions.

Four adjustments in the main processor are listed in this chapter. The adjustments take place in decoders, general-purpose registers, renaming files, and commit buffer of the main core. A new vector reservation station and memory arbiter are also added to the previous core.

The detailed components inside the vector co-processor are presented. The wrapped memory achieves misaligned memory access to leverage memory utilization. The instruction board stage resolves the data dependency in vector instructions. The independent instructions are executed in the corresponding processing unit in parallel to enhance the performance.

Chapter 5

Software Work-flow

5.1 Test Environment

The whole prototype, labeled as DUT (device under test), is placed in the test environment that is illustrated in Figure 5.1. Two memory blocks are attached to the design as the simple Harvard architecture. A 64KB memory is mapped to the address starting with 0x0 as the instruction memory. A 1MB memory is mapped to the address starting with 0x1 as the data memory. When the processor accesses the address starting with 0x2, a host function is selected according to the access address, including trigger a breakpoint, continue from the breakpoint, display character, send termination signal, and read data from the host.

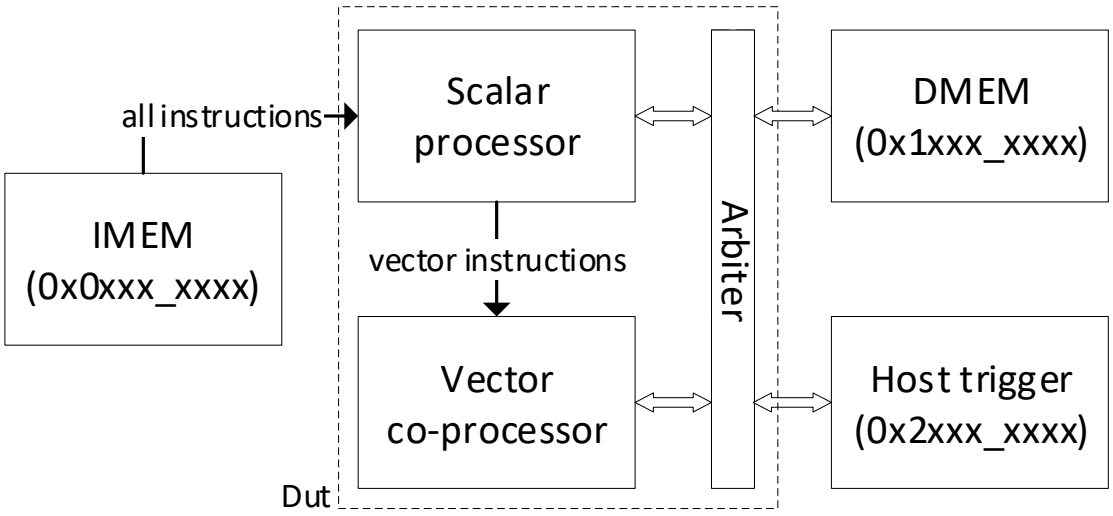

Figure 5.1: Hardware arrangement in test-bench.

On the host side, the test-bench monitors and counts important signals to record as a log file during each run-time.

- prediction miss number: how many prediction yields miss during each program. The counter increases one if the prmiss signal is high.
- prediction hit number: how many prediction yield hit during each program. The counter increases one if the prsccs signal if is high.
- branch/jump unit issue time: how many instructions are issued to the branch/jump unit from the branch reservation station. As the branch reservation station handles issue logic in order, all instructions leaving the reservation station are guaranteed to complete without speculation. The fraction of the missing number and the total predictions show the prediction miss rate.
- committed instruction number: the commit_1 and commit_2 signals in the commit buffer. The total completed instructions in the very last COM stage show the total executed instructions in a program. As there are always numerous loops in every program, the code size, or the number of instruction in the instruction memory, does not reflect the computation complexity. The same portion of code may proceed repeatedly. Therefore, the number of committed instructions is traced in the test-bench to clarify the computation complexity in each program.
- total machine time: indicates the CPU time between the reset signal and the termination signal. The machine cycles multiply with clock frequency to provide real-time results. The average instruction per cycle (IPC) is calculated by the division of the total committed instruction number and total machine cycle.

The RISC-V GCC compiler is built from the source code on the website.[1] The software design focuses on pure single-core execution, without any kernel, firmware, or software layer, therefore, the GCC tool with the prefix riscv-none-embed is selected to compile the software.

The standard RISC-V GCC compiler generates 64-bit instructions within the extension set of IMAFD by default. To make the machine code executable by the design, the flag

[1] https://github.com/riscv/riscv-gnu-toolchain

"-march=rv32im" is passed to the GCC to control the toolchain only using basic integer 32-bit instruction set with the extension of integer multiplication and division. The flag "-mabi=ilp32" is also presented as the complication option to force "int", "long" and pointers to be 32-bits, "char" 8-bits, and "short" 16-bits in size, respectively.

5.1.1 ISA tests

SiFive provides multiple self-check testcase [30], which verifies whether the functionality of a processor meets the RISC-V ISA specification. Those testcases are written in assembly language.

Each ISA testcase consists of multiple checkpoints to test one specific instruction. For example, the testcase of ADD instruction has arithmetic tests, source/destination tests, and bypassing tests. In arithmetic tests, ADD instructions take 15 combinations of given source operands to generate corresponding results, which are then compared with expecting values to reveal the correctness in arithmetic. 3 combinations are presented in source/dest tests, including rs_1 register equal to rd, rs_2 register equal to rd, rs_1,2 equal to rd. The source/destination tests reveal the correctness in register addressing. Bypassing tests introduce 19 combinations of potential read-after-write hazards, which reveals the functionality of the pipelined processor to resolve the data dependency.

Figure 5.2 shows two console outputs of the passed test and the failed test. If any combination yields an incorrect result, the program jumps to the routine TEST_FAIL. Otherwise, the program keeps executing the TEST_PASS. In the TEST_PASS routine, programs write one to the register 3, while, in the TEST_FAIL routine, programs write the value to the register 3 to indicate which test routine yields mistake in computation. A small block of logic in the test-bench continuously checks the value in register 3 through back-door access. If the none-one value is detected, the test-bench terminates the simulation. Developers then can open the waveform to trace the incorrect operation and fix the implementation.

The superscalar RISC-V processor passed all testcases that are related to the defined rv32im instruction set as the regression testing. Three groups of testcases are the following:

- rv32ui: There are 39 testcases such each testcase is responsible for one basic 32-bit

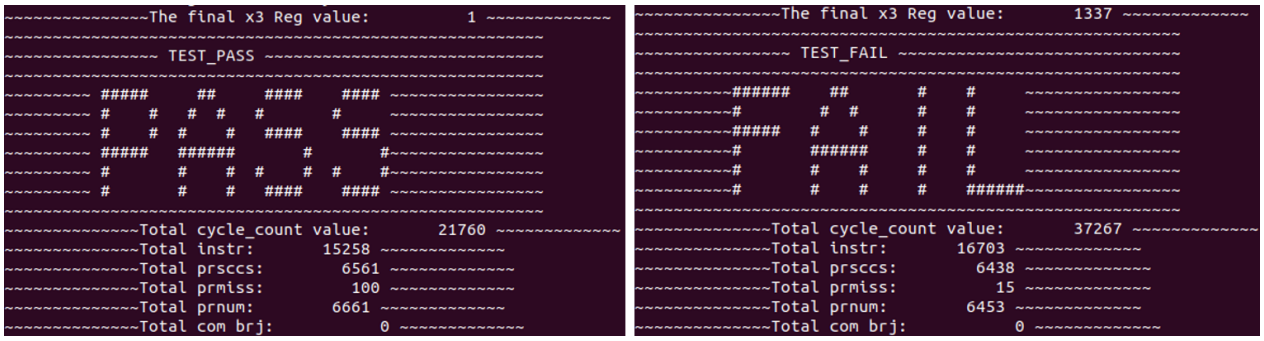

Figure 5.2: Console outputs of ISA tests.

RISC-V instruction, such as ADD, JAL, and LW.

- rv32um: The 8 multiplication/division instructions in the M extension are tested individually in both arithmetic and functional perspectives.

- rv32mi: This set tests the machine-level interrupt and exception handlers with CSR instructions. Passed exceptions include illegal instruction, miss-aligned addresses, and miss-aligned CSR addresses.

After the regression test succeeds, the superscalar RISC-V processor meets the RISC-V ISA specification, which is compatible with the standard GNU toolchain to enable the abstraction level to the C programming language.

5.1.2 Toolchain-flow

Even though each individual instruction performs correctly, some computation-intensive test-cases are required to debug and evaluate the hardware implementation. Due to the nature of the RISC assembly language, it is inefficient to develop the computation-intensive program directly in assembly code. Because the RISC-V GCC is available, some complex programs are designed in the C programming language to leverage usability.

Figure 5.3 illustrates the work-flow from C codes to hardware processing. Supposed that the all C source codes are stored in the same folder, the first command for converting C source codes to assembly codes is

```
$ riscv-none-embed-gcc -O2 -march=rv32im -mabi=ilp32 -S *.c.
```

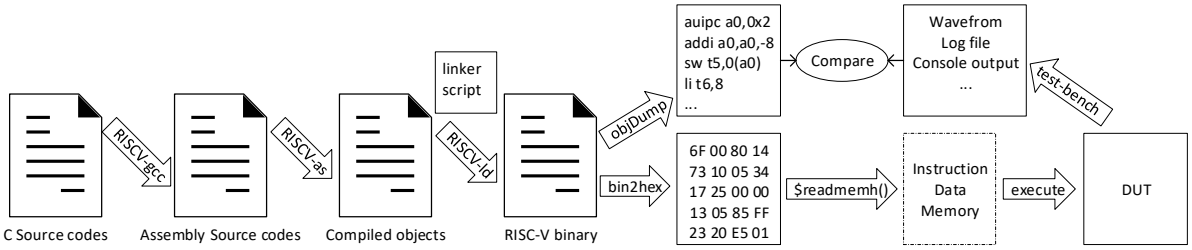

Figure 5.3: Work-flow from software code to hardware output.

This command generates an assembly source file for each input C source file. The wildcard "*.c" includes every C code inside the working directory as the input source to compile. For example, the "main.s" is the compiled assembly source of the "main.c" code, and the "function.s" is the complied assembly source of the "function.c" code. The "-O2" flag enables the optimization more for code size and execution time for the GCC compiler.

Next step, the RISC-V assembler converts every assembly source into object file, which the command is

`$ riscv-none-embed-as -march=rv32im *.s startup.s`.

This command generates the object file for each wildcard "*.s" assembly source. The entry code is presented in the "startup.s". During the execution, a program starts with running the entry code. When the entry portion is finished, the main function starts to execute in the "main.c". Because there is no thread or firmware implemented to the design, the entry code just initiates every general purpose register to zero.

Then, the RISC-V link editor, or linker, combines every object file into one executable binary, which the command is

`$ riscv-none-embed-ld -static -T linker.script *.o -o bin`.

This command links each wildcard "*.o" object file into one binary file "bin". The "linker.script" defines how the linker combines and places a different portion of codes together. In the linker script, the entry code (.startup.o(.text)) is defined at the beginning of the instruction memory to "0x0000_0000". The rest of the instructions (.text) are placed after the entry code, including the main function, and other subroutines. Read-only data (.rodata), normal data (.data), initialized data (.sdata) and BSS segments (.bss) sections are located in the data memory. Therefore, those sections are defined in the data memory by configuring

the starting address to “0x1000_0000”.

Next step, the executable RISC-V binary file “bin” is converted to Verilog HEX file by the RISC-V objcopy, and to readable text in assembly form by the RISC-V objdump.

`$ riscv-none-embed-objcopy -O verilog bin hex.txt`.

This command turns the binary file “bin” into Verilog HEX file “hex.txt”. The machine codes are aligned in byte-width hex-decimal value in the form that the right-most hex value has the smallest address. The file starts at “@0000_0000” to indicate that the following contents are instructions and the content after “@1000_0000” is data, which is controlled by the linker script. Instructions and data are stored to the corresponding memories through back-door access by the Verilog macro “$readmemh()”.

`$ riscv-none-embed-objdump -S bin -Mno-aliases --disassemble-all >> dump.txt`.

This command disassembles the binary file “bin” to view it in assembly form. The “-Mno-aliases” flag removes every pseudo-instruction. For example, the typical MOVE instruction, which moves the value from one register to another, does not exist in the instruction set. However, the move operation in RISC-V ISA is aliased to ADDI instruction, which adds the source value with 0 and stores the result to another register. Without the pseudo-instruction, it is clear to investigate which instruction is processing in the debug-flow. The output of this command, the dissembled binary, is redirected to the text file “dump.txt”.

5.1.3 C programs

Because the software does not depend on any kernel layer, the RISC-V C program uses several customized macro instead of the standard system call interface, such as display function (printf) and termination (return 0). Those two macros are defined as below,

`#define DISPLAY_CHAR(chr) *((int*)(disp_addr)) = chr`, and

`#define FINISH_PROGRAM *((int*)(finish_addr)) = 1`,

where the disp_addr and finish_addr are defined as the constant values, 0x2005_0000 and 0x2005_0004 receptively, with the type of volatile unsigned int.

When a program tries to display a string, each character in the string array is passed to the DISPLAY_CHAR macro until the string reaches the null terminator ’\0’. The compiler treats the macro as a store operation, which source value is the character and the external address is

the disp_addr. At the same time on the hardware boundary, the memory port writes a 32-bit data to the host trigger address space (0x2xxx_xxxx). The address 0x2005_0000 triggers the display function so that the test-bench displays the 32-bit data as character format on the simulation console to realize the printf function.

The FINISH_PROGRAM macro follows the same approach with different trigger addresses. In this case, the termination function in the test-bench is triggered to finish the simulation. Other macro functions as the basic software environment include display integer, display hex-decimal, start timer, and end timer, with each function associative to a unique trigger address.

Several computation-intensive programs are tested by the superscalar processor to verify the stability in real applications and to evaluate the performance. Recursive functions are complementary in such scenario, which usually requires relative small code size but high computation complexity. Moreover, the complexity level is controlled by several parameters as the initial state, which is capable of verifying different final results without re-programming the software. Three famous recursive algorithms are listed below:

- Ackermann function [31]: This function is defined as recursive iterations with two non-negative integers as:

$$A(m,n) = \begin{cases} A(0,n) = n+1 \\ A(m+1,0) = A(m,1) \\ A(m+1,n+1) = A(m,A(m+1,n)). \end{cases}$$

  The number of iterations grows rapidly for small inputs. The Ackermann function is small in the code size, but brings exponential computation iterations, which efficiently tests the processor's durability in recursive branches and load/store operation regarding to the stack pointer.

- Towers of Hanoi [32]: This function takes its origin from a puzzle game. In the game, there are three sticks with multiple disks of different sizes staked in ascending order. The goal of this game is to move all disks to another stick renaming the same ascending order of disk sizes. This game is abstracted into a mathematical function with fully

recursive implementation. The number of disks becomes the parameter to scale the complexity. The more disks are placed on the stick, the more steps are required to solve the puzzle.

- Tarai function [33]: This function is a simple recursive function that is often used as a benchmark to evaluate the compiler's optimization for recursion. The algorithm is defined as:

$$tarai(x,y,z) = \begin{cases} tarai(tarai(x-1,y,z), tarai(y-1,z,x), tarai(z-1,x,y)) & \text{if y} < \text{x}, \\ y & \text{otherwise.} \end{cases}$$

  In each recursive iteration in the Tarai function, each operand is the result of another Tarai function and the position of operands switch around, as it is shown in the equation. Therefore, the Tarai function yields deep recursion tests to evaluate the calling and addressing speed.

Other procedure-based computation functions are programmed to the processor, including matrix multiplication, software-emulated multiplication, and sorting array of integers (qsort, rsort).

All the programs and functions are developed in separate source codes and they are compiled into both RISC-V executable binary and standard Linux executable binary to verify the results generated by the RISC-V core. For example, the matrix multiplication code is located in "matmul.c" and test matrices are defined in its header file "matmul.h". The main function of the RISC-V program "main.c" uses the matrix multiplication function in "matmul.c" by including the header file "matmul.h". When the computation is finished, the RISC-V main function displays the results by the display macro.

On the other hand, a standard main function noted as "main_gold.c" also refers to the same source code "matmul.c" and uses the same test matrices. The results of matrix multiplication are then printed by the built-in printf function. The "main_gold.c" program is compiled by the standard GCC and executed in the bash shell. The results generated by the Linux executable are the reference to compare with the RISC-V results. Figure 5.4 presents two console outputs of the testing RISC-V processor and the standard Linux executable.

Note that the output results are consistent; therefore, it is proved that the RISC-V processor functions correctly in the test program.

```
00000395,00000375,000001DF,000002FF
000006E1,00000392,00000325,00000481
00000568,00000473,000002DE,000002ED
00000585,00000204,00000285,000003B4

      1402 clks
prnum:          373, prsuccess:          305, prmiss:          68
(prcom:          373)
#instr:         1801
```

Console output of RISC-V processor

```
jih292@jih292-Ubuntu:~/xin/testcase/app/matmul$ ./a.out
395,375,1df,2ff
6e1,392,325,481
568,473,2de,2ed
585,204,285,3b4
jih292@jih292-Ubuntu:~/xin/testcase/app/matmul$
```

Console output of standard Linux executable

Figure 5.4: RISC-V program result vs standard program result.

5.2 Vector Software

Because the current RISC-V compiler does not support V-extension and the Cambricon compiler is not available, the vector software is programmed in pure assembly language. Assembly codes are stored in an Excel file, in which columns represent instruction type, destination register, source registers, and immediate values. A customized assembler is developed in Matlab scripts based on text processing, which converts the assembly codes into executable hex file by matching and replacing.

The forward inference of LeNet-5 [20] is implemented on this prototype to reveal the benefit of SIMD computation in the vector co-processor. Table 5.1 shows a summary of the CNN model. All activation functions are the rectified linear units (ReLU). The 8-bit fix point number is selected as the data format of each element.

Table 5.1: CNN structure

Layers	Data Sizes		
input (1@32×32)	input	parameters	output
conv (6@28×28, K:6@5×5)	1024	156 (5×5×6+6)	4704
pool (6@14×14, K:2×2)	4704		1176
conv (16@10×10, K:16@5×5)	1176	2416	1600
pool (16@5×5, K:2×2)	1600		400
fc (120)	400	48120	120
fc (84)	120	10164	84

Conventional convolution layers involve a three-dimension feature map, noted as (row, column, depth), and four-dimension weights, noted as (row, column, input_depth, output_depth) to generate the output feature map. To maximize the continuous pattern in memory access and parallel computation in depth-wise, the 3D feature maps are flattened in the order of depth, column, row. The 4D weights are flattened in the order of input_depth, output_depth, column, row.

Figure 5.5 shows the computation of the first block results of the output feature map in a convolution layer. The numbers inside each data indicate the offset addresses in memories. Two static memory spaces are reserved to hold the temporary results of the dot product and the accumulation results, with both the same sizes of the output depth.

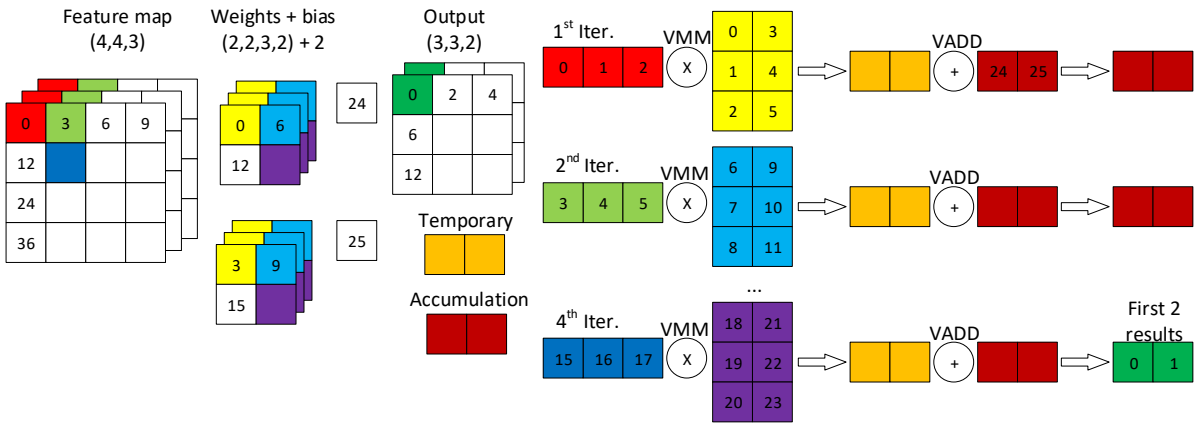

Figure 5.5: Data flow of convolution computation.

Before the first iteration, two bias elements are copied to the accumulation space as the starting point of the computation. It requires four iterations to finish the 2×2 kernel. In the last iteration, the elements in the accumulation space are the final two results, in depth-wise, of the output feature map.

5.2.1 Layer kernels

The generalized assembly kernels of the convolution layers and max-pooling layers are presented in Figure 5.6, which supports variable sizes of data and parameters, kernel sizes, and different strides.

CONV code:

```
//x10 input address, x12 input depth, x13 weight address,
//x15 kernal size, x16 bias addr->x13 + weight sizes,
//x17 accumulation address->x10 + input sizes,
//x18 output size, x19 output depth, x20 stride x->stride × x12,
//x21 stride y->x12 × (input width - x15 + 1),
//x22 input iter, x23 weight iter,
//x24 temporary address->x16 + x19, x25 X-looper,
//x26 Y-looper, x27 output row looper,
//x28 weight stride->x12 × x19, x29 output width,
//x30 output stride of next row->x15 × x12 + input width × x12
× (stride - 1),
//x31 output external address
L4:addi x27,x29,0              //init output row looper
L3: vcopy x17,x19,x16          //init bias to accumulation space
    addi x23,x13,0             //init weight iter
    addi x22,x10,0             //init input iter
    addi x26,x15,0             //init y looper
L2:addi x25,x15,0              //init x looper
L1:vmm x24,x19,x22,x23,x12     //depth of input × weights
   vadd x17,x19,x17,x24        //accumulate the result
   addi x25,x25,-1             //x--
   beq x25,x0,#L0              //branch to next row
   add x22,x22,x12             //input++
   add x23,x23,x28             //weight++
   jal x0,x0,#L1               //start to next point
L0:addi x26,x26,-1             //y--
   beq x26,x0,#Le              //branch if accumulation done
   add x22,x22,x21             //input+=next row
   add x23,x23,x28             //weight++
   jal x0,x0,#L2
Le:vstore x31,x19,x17          //output depth result done
   add x31,x31,x19             //bump output addr
   sub x18,x18,x19
   beq x18,x0,#ret             //all output ready layer done
   addi x27,x27,-1             //output row--
   beq x27,x0,#Lf              //branch if next output row
   add x10,x10,x20             //next starting addr of input
   jal x0,x0,#L3
Lf:add x10,x10,x30             //next starting addr of input
   jal x0,x0,#L4
```

Max-pooling code:

```
//x10 input address, x12 output depth,
//x15 kernal size,
//x17 temporary address->x10 + input sizes,
//x18 output size, x20 stride x->stride × x12,
//x21 stride y->x12 × (input width - x15 + 1),
//x22 input iter,
//x26 Y-looper, x27 output row looper,
//x30 output stride of next row->x15 × x12 + input width × x12
× (stride - 1),
//x31 output external address
L4:addi x27,x29,0              //init output row looper
L3:vloads x17,x12,x0           //init zeros to compare position
   addi x22,x10,0              //init input iter
   addi x26,x15,0              //init y looper
L2:addi x25,x15,0              //init x looper
L1:vgtm x17,x12,x22,x17        //depth-wise comparision
   addi x25,x25,-1             //x--
   beq x25,x0,#L0              //branch to next row
   add x22,x22,x21             //input++
   jal x0,x0,#L1               //start to next point
L0:addi x26,x26,-1             //y--
   beq x26,x0,#Le              //branch if one kernel done
   add x22,x22,x21             //input+=next row
   jal x0,x0,#L2
Le:sub x18,x18,x12
   vstore x31,x12,x17          //output depth result done
   add x31,x31,x12             //bump output addr
   beq x18,x0,#ret             //all output ready layer done
   addi x27,x27,-1             //output row--
   beq x27,x0,#Lf              //branch if next output row
   add x10,x10,x20             //next starting addr of input
   jal x0,x0,#L3
Lf:add x10,x10,x30             //next starting addr of input
   jal x0,x0, #L4
```

Figure 5.6: Generalized assembly kernels for convolution and max-pooling.

To lift the utilization of internal memory banks, the temporary space is allocated after the parameters, and the accumulation space is allocated after the input feature map. The flow of the max-pooling kernel is similar to the flow of convolution. Multiple data items in depth-wise are compared simultaneously by the VGTM instruction. Because the data items are already flattened in memories, one VMM instruction is enough to compute the fully-connected layer.

5.2.2 Optimizations

The assembly kernel assumes that the input feature maps are fully loaded into one memory bank and parameters into another bank before the calculation starts. However, given that each memory bank has a capacity of 4KB, layers must be separated into different parts if the sizes of their parameters or input data exceed the memory limitation (4096-byte). Besides, due to the flexibility of the hardware and instruction set, double buffering and loop unrolling can be achieved by rearranging the assembly codes, without changing the hardware architecture, to increase the performance.

Layer separation

In a nutshell, the parameter and the output sizes must be smaller than (4096 – output_depth). The memory of size output_depth is reserved for temporary and accumulation data respectively in the input feature map bank and the parameter bank.

To adjust the parameter size, the original layer is sliced in the dimension of output_depth. For example, the first fully connected layer in Table 5.1 of size (400,120) is divided into 15 small fully connected layers of each size (400,8). In this case, the required memory space reduces to 3216-bytes, which is consisted of 3200-bytes of weights, 8-bytes of bias, and 8-bytes of temporary data. Each separate layer contributes 8 final results with reuse of the same input feature map. Every 8 final results are gathered in another spare bank. After all 15 small layers are finished, the 120 final results are grouped in the spare bank, which is then translated to the input feature map bank for the next layer.

In another case of the first pooling layer, the input size exceeds the capacity limit. The input feature map is separated into two parts with each size of 2352(6@28×14). The pool-

ing assembly kernel executes twice by considering the two input feature maps of the top [1:14]×28×6 part and the bottom [15:28]×28×6 part. The control of the output addresses can rejoin the output feature map for the next layer.

Double buffering

Because the output of the previous layer is the input of the next layer, except for the final results in the last layer, it is unnecessary to store the temporary results back to the main memory. To reduce the computation cycles, the double buffering is achieved by careful hand-coding, which hides the loading cycles inside the computation cycles.

The VSTORE instruction is replaced to VCOPY and the output address is changed to another internal memory bank. As a result, at the beginning of the next layer, the loading cycles of the input feature map are omitted. Also, the loading cycles of the weights for the next layer can be hidden inside the calculation cycles of the previous layer.

In Figure 5.7, row A shows the normal data flow, while, row B shows the optimized data flow. In the optimized data flow, the computation portion requires memory accesses of bank_0 (input feather map), bank_1 (weights and bias), and bank_2 (gathering output). Before the processor jumps to the convolution kernel to start the calculation, a VLOAD instruction that loads the next layer's parameters to bank_3 is inserted.

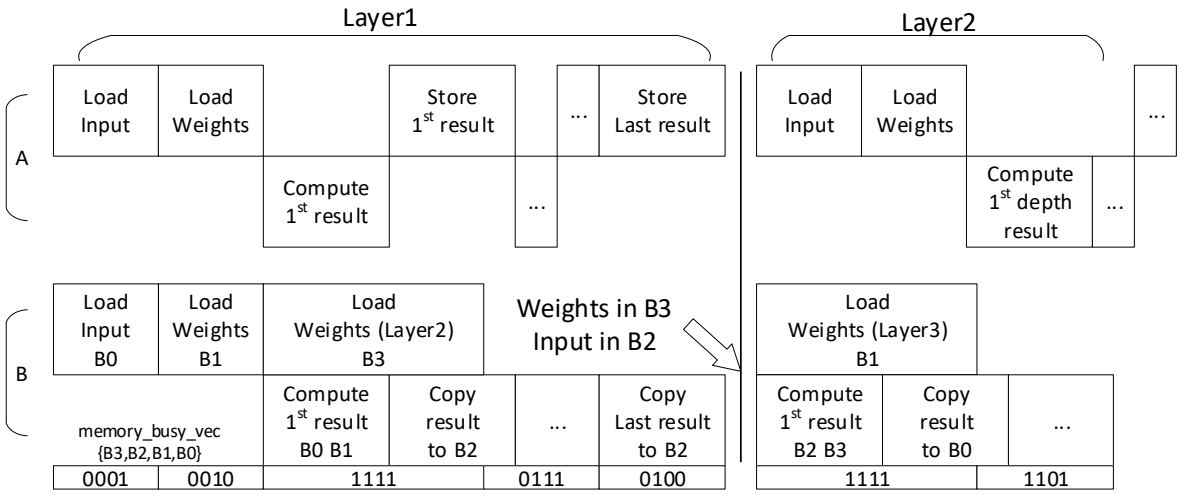

Figure 5.7: Data flow of double buffering.

In this case, the loading portion and the computation portion are independent so that they can execute in parallel. At the end of the first layer, both input feature maps and parameters of the next layer are already presented at the internal memory, in bank_2 and bank_3 respectively. Compared with row A, the conventional load-store flow, the data flow of row B improves the throughput by dropping the loading cycles after the first layer.

Loop unrolling

The general convolution code uses VMM instruction to calculate the corresponding results of the output depth. However, because the first layer has only one channel, it is inefficient to use the dot product loop. In the dot-product unit, with the vector length of 1, the rest of the 7 operands are masked to 0, and the accumulation stage is also wasted by adding one result with 7 zeros.

The same results can be calculated by the element-wise multiplication. To quickly recap, the vector-scalar-multiplication instruction VSMUL, takes a vector element-wisely multiplied with a scalar. In this approach, the weights of the output depth become the input vector and the corresponding point of the feature map becomes the input scalar. The computation time reduces from 6 cycles, three loops of 2 cycles dot-product, to 1 cycle of element-wise multiplication, as shown in Figure 5.8.

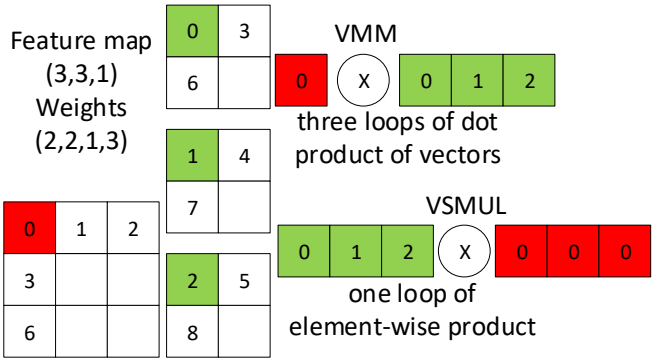

Figure 5.8: Comparison between dot-product and element-multiplication

5.3 Summary

This chapter introduces software development collaborating with the hardware design.

The test-bench circuit and environment are presented in this chapter, as well as the compilation and test work-flows of the design. The superscalar RISC-V processor passes the ISA regression test with all defined instructions.

The RISC-V GNU toolchain flow is introduced. With the support of the compiler, the processor can execute the program that is developed in C programming language. Several computation-intensive programs prove the functionality of the processor.

The usage of the vector co-processor is shown. Vector programs with the customized vector instruction set are written in assembly code and converted to executable by the customized assembler. Software optimizations of SIMD instructions for the specific LeNet-5 CNN are also introduced to increase the performance, including layer separation, double buffering, and loop unrolling.

Chapter 6

Results and Analysis

This chapter provides the implementation results of the dual-issue superscalar RISC-V processor and the vector co-processor.

Section 6.1 presents the performance results of the general-purpose RISC-V processor regarding several benchmark programs. The proposed design is compared with another design, Ridecore [12], which has the most similar architectures and techniques. Moreover, the FPGA synthesis reports are also presented in the section. Analyses according to those implementation results are included.

Section 6.2 presents the performance results of the vector co-processor to handle the inference of the LeNet-5 [20] model. The same network structure is also executed by the scalar processor only to reveal the performance improvement of the SIMD calculation. The processing units' utilization of different types of layers is compared and analyzed. The FPGA synthesis reports are included to evaluate the area/performance trade-off.

6.1 Superscalar Processor

The superscalar processor that is described in Chapter 3, is implemented in SystemVerilog hardware description language. The design with several programs is simulated on Synopsys VCS. The functionality of the RISC-V processor is verified by the ISA regression test and the output results that the same source codes generate in the bash shell. Test programs from Ridecore [12] are directly compiled from C codes by the RISC-V GNU toolchain. The compiled hex files are fed to instruction cache by backdoor access. Those programs are evaluated by the proposed design and Ridecore [12].

6.1.1 Performance

Table 6.1 presents the detailed log files on different test programs. Although Ridecore is also a dual-issue superscalar RISC-V processor with the same number of pipeline stages, the execution time of the same program on the proposed design is shorter than the required cycles on the Ridecore. Figure 6.1 shows the comparison of instruction per cycle (IPC) and Figure 6.2 shows the comparison of prediction hit rates among every test program. All programs can be found in the software directory of [12]. On average, the proposed design achieves 1.126 IPC and 74.87% prediction hit rate, which improves 18.9% on average IPC and 4.92% on average prediction hit rate than the Ridecore.

Table 6.1: Log files of software execution on different test programs

	Proposed				Ridecore [12]			
testcase	instr[1]	clk[2]	prsccs[3]	prnum[4]	instr	clk	prsccs	prnum
ackermann	33799	29677	5820	7262	33802	39733	3641	7263
charout	210701	158771	51996	52541	210703	210197	51995	52542
cprime	100792	151561	11491	19428	100802	162168	10815	19429
komachi	1592209	1514186	166487	296467	1592211	1753771	151823	296468
stirling	30541	30136	5443	7874	30549	37931	4573	7875
matmul	1801	1402	305	373	1803	1704	299	374
combinant	39080	40208	5750	9647	39088	42606	7288	9468
hanoi	7017	5137	1461	1598	7018	6462	1462	1599
stencil	3290	2435	557	619	3291	2932	557	620
tarai	240779	219888	19310	31530	240782	219522	19348	31531

[1] number of commited instructions

[2] total machine cycles

[3] number of successful prediction

[4] number of branch instructions

Although the two processors execute the same compiled RISC-V binary, the total committed instructions of Ridecore are slightly more than those of the proposed design among all

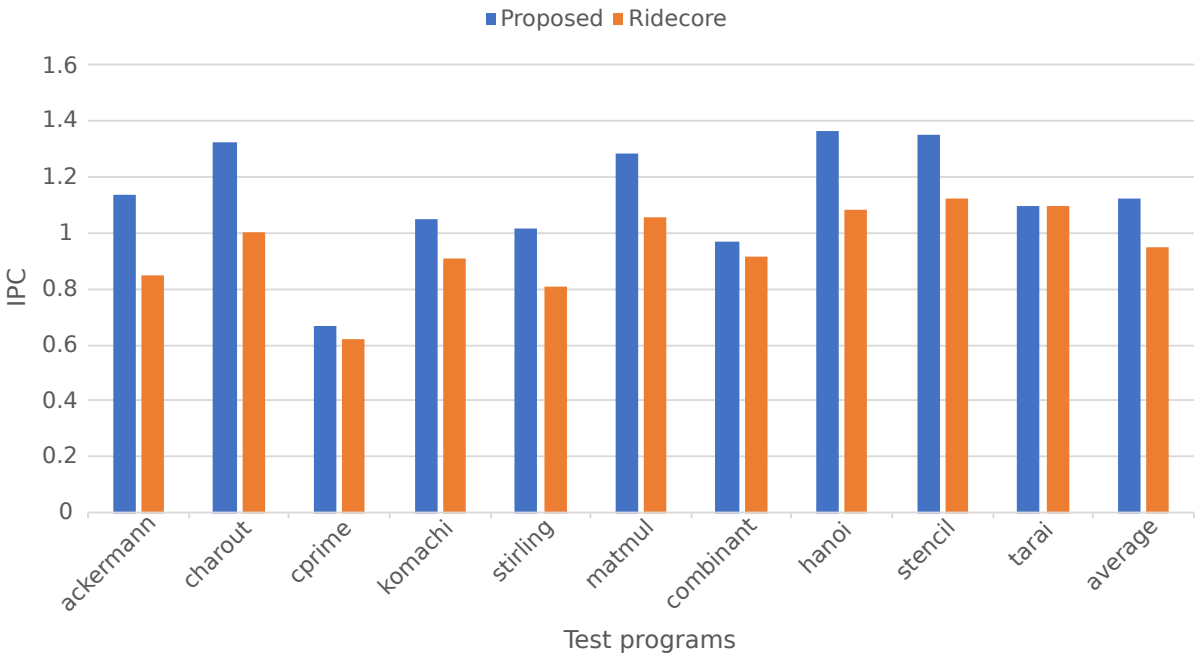

Figure 6.1: Comparison of instructions per cycle.

the programs. The reason is that two processors have different commit schedule and external system bus, which leads to a small amount of latency difference in termination signal received by the host logic.

From the perspective of host logic, the actual arriving time of the write signal varies from different system bus protocols and memory hierarchy. That latency effect on prediction hit rate and IPC is negligible since the differences are small compared with the total numbers. Although two designs both use Gshare branch prediction, the proposed design achieves a better prediction hit rate on average.

The branch prediction module in the proposed design receives branch outcomes and target addresses after the execution stage. The latency between execution and commit depends on how many instructions are ahead of the branch instruction in the commit buffer. During that time interval, branch predictors can generate better TAKEN/UNTAKEN results based on more recent branch outcomes, and the branch target buffer reduces the cold start miss chance. Moreover, the Gshare branch prediction technique yields better prediction accuracy with more and latter past branch outcomes.

The improvement on average IPC is achieved by not only the better prediction hit rate

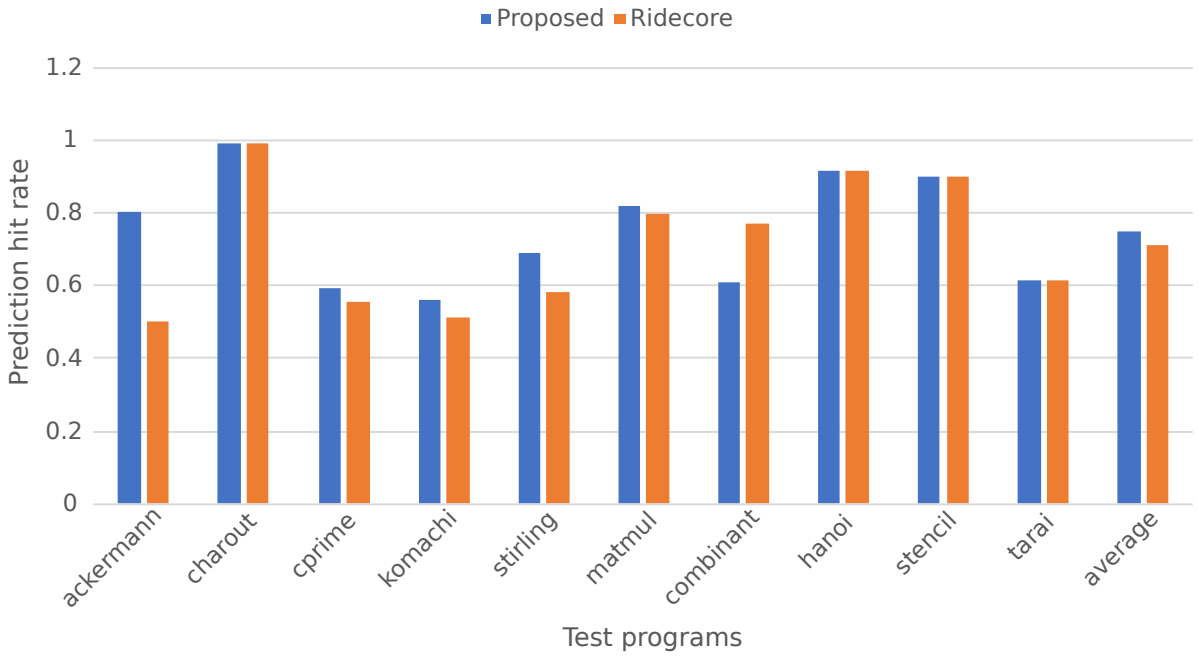

Figure 6.2: Comparison of prediction hit rates.

but also the modified speculation logic and busy counters. In Ridecore, the speculation bits clear synchronously on the prediction success cycle, which takes one stall cycle in every stage before reservation stations. In the proposed design, speculation bits clear asynchronously so that new instructions can keep pushing into reservation stations without a stall cycle to yield higher utilization.

In program `charout`, since the branch hit rates are the same on two processors, the proposed processor runs faster by the number of successful predictions. In other words, the required number of cycles on the proposed processor is 158,771 that is close to the number of cycles, 210,197, minus the number of success prediction, 51,995, on Ridecore.

On the other hand, by adopting the 4-bit busy counters, the proposed design can keep dispatching up to 8 instructions (the MSB generates stall logic on the dispatch stage) that have the same destination. In program `Komachi`, register a5 is constantly modified according to the compiled assembly code. During the execution, Ridecore must stall the dispatch logic to wait for the previous instructions modifying the register a5. The proposed design with the busy counters, however, can continuously dispatch those instructions that modify the register a5, which improves the pipeline utilization to bring higher IPC.

To make the general evaluation of the performance, the processor is assessed by the Dhrystone benchmark program. The Dhrystone benchmark [34] was first proposed in 1984. It has become the standard representative of the performance in general-purpose processors. The latest version (2.1) is evaluated in the proposed design to compare the performance with other processors, including RISC-V ISA and other ISAs.

The benchmark is written in C programming language and is compiled by the same flow as other test programs. There are 8 processing functions to test the common software flow, such as procedure calls, pointer indirections, and variable assignments. One iteration of those functions is called one Dhrystone. The Dhrystone benchmark performance is presented as the number of Dhrystones per second.

According to common standards in the industry, the Dhrystone benchmark should refer to the Dhrystone benchmark of the VAX 11/780 [35], which achieves 1757 Dhrystones per second. The Dhrystone result is calculated by the measured Dhrystones per second, dividing by 1757, and reported as "DMIPS", how many times faster than the VAX 11/780.

The Dhrystone source codes are copied from the official website. The system call functions, including printf() and time() functions, are overridden by the customized macros to monitor the result without the firmware layer.

The Dhrystone program with 2,000 iterations is tested on the proposed design. There are 1,004,011 instructions committed during the execution. The complication options of the RISC-V GNU toolchain include the "-O2" flag that turns on every optimization flag specified by "-O", and "-march=rv32i" flag that forces to use the software emulated division instead of the division instruction in the "M" extension. There are 225,946 successful branch predictions of the total 253,999 branch predictions, which yields the prediction hit rate of 88.96%.

The program is executed with two builds, one with an ideal data cache (D$) which assumes the cache size is infinite with a 100% hit rate. The other one is directly connected to the data memory. With a data cache, the processor can handle speculative store operations to enhance the performance. By default, the load/store reservation station issues the instruction without speculation to guarantee that miss prediction does not happen in the external data memory. As a result, the speculative store instruction must wait for the related branch outcome, which produces waste cycles during the execution.

However, with the data cache, the processor can handle speculative store operations without affecting the data memory. The store instruction only changes the data in the data cache, which will write back to the external data memory as soon as the store instruction is committed. In this case, the mispredicted store instruction does not pollute the data memory by keeping the incorrect data in the data cache.

The proposed design requires 870,812 cycles to finish the 2,000 iterations without the data cache and requires 792,934 cycles with the ideal data cache, which yields 1.3072 DMIPS/MHz and 1.4356 DMIPS/Mhz respectively. Because most of the predictions are correct, the capability of speculative store operations improves the performance by 9.82%. Figure 6.3 presents the Dhrystone benchmark results of other processors.

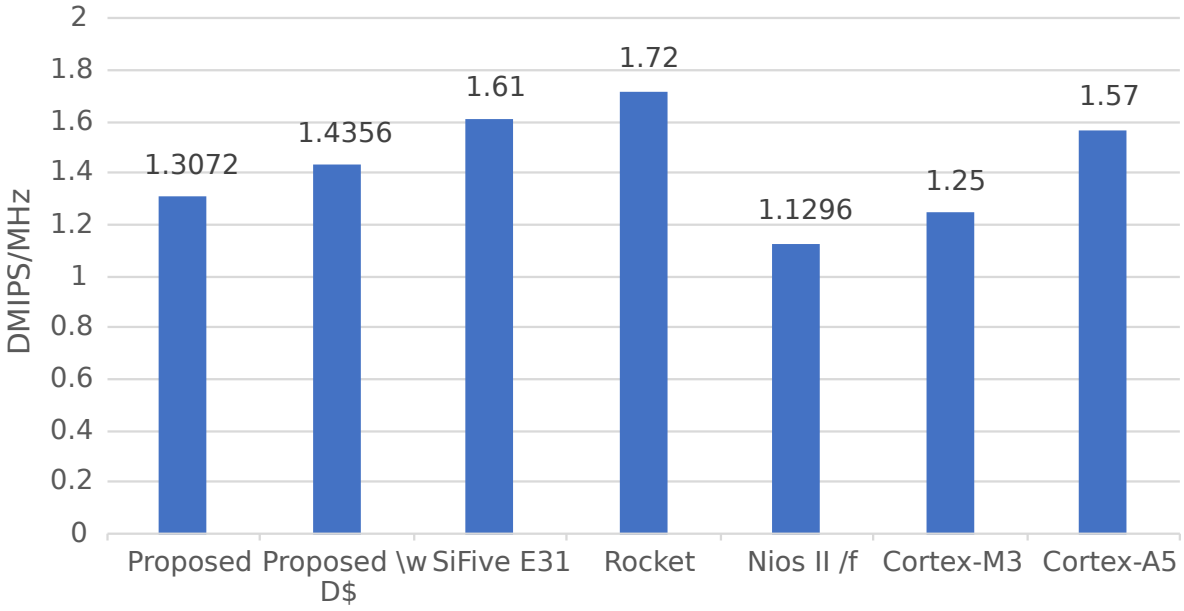

Figure 6.3: Dhrystone benchmarks of several commercial processors.

The SiFive E31 [36] is a 5-6 stage in-order processor with the rv32imac specification. It is equipped with many peripherals, including a platform interrupt controller, debug interface, and an advanced memory subsystem. The Rocket [11] is a 64-bit 6 stage in-order processor that further supports floating-point operations including fused-multiply-add. Both the SiFive E31 and the Rocket cores are the RISC-V processors. For other processors with different ISAs, Nios II [37] is a soft-processor on Intel's FPGA, whose instruction set is based on its own specification. The DMIPS/MHz result is reported based on the fast implementation on

the Arria 10 board. The Cortex-M3 [38] and Cortex-A5 [39] are two popular commercial ARM processors that have been widely used in embedded applications and cellphones.

6.1.2 Synthesis

For evaluation purposes, two designs are implemented on the Arria 10 FPGA board. The synthesis reports are presented in Table 6.2. To make a fair comparison on cores, the store buffer exclusive in Ridecore and the CSR buffer exclusive in the proposed design are removed before the synthesis.

Table 6.2: FPGA synthesis reports

	Ridecore[1][12]	Proposed[2]
Timing constraint	50Mhz	50Mhz
ALMs	25640	23045
Registers	14657	13492
DSP/M20K	12/5	9/2
Core dynamic power	98.60mW	84.03mW
Fmax[3]	53.76Mhz	62.85Mhz

[1] excluding store buffer.

[2] excluding CSR buffer.

[3] slow 900mV 100C model.

Even though the proposed design has additional features, including integer division and machine-level privileged instructions, the proposed design takes advantage in every perspective, area, power, and performance.

With respect to logic blocks and registers, the proposed processor saves 10.1% of the adaptive logic modules (ALMs) and 7.9% of registers. This saving is primarily contributed by the merge of the latest value column in the renaming register file, into the result column in the commit buffer.

In Ridecore, each renaming register file contains not only the busy status and renaming destination but also the latest value that the renaming destination is referring to. In the

proposed design, every renaming register file does not contain the value, which is alternatively located in the commit buffer as the result column. The dispatched instruction fetches the latest value from the commit buffer with the renaming destination as the address. According to the detailed synthesis report, the merged commit buffer saves 2,889 ALMs compared to standalone commit buffer and renaming register file.

The in-order processing of branch instructions simplifies the branch recovery logic by omitting the branch dependency. In the case of out-of-order branch handling, an additional module is required to decide the flush logic upon the correct prediction. In other words, the latest branch outcome may not be the oldest branch operation, therefore, the matched speculative bit cannot directly turn off until the oldest branch outcome is ready.

Moreover, the additional features do not introduce much hardware overhead. The interrupt and exception handle logic in the CSR front module is implemented as a latch to detect the interrupt and system instructions. The multi-cycle integer division unit based on the finite-state-machine also has relatively small logic complexity.

The less complex architecture saves 5.6% of the power consumption in the core dynamic power. The organized multiplication arithmetic unit saves 3 integer multiplier blocks (DSP) during the hardware compilation. The storage of the latest values in the register renaming file is translated to on-board memory (M20K) by the Quartus compiler in the Ridecore. Therefore, the proposed design lacks three M20K blocks.

The critical path is related to the branch/jump unit. The starting registers of the critical path are located after the branch/jump reservation station, which holds the operands and opcode of the issued branch instructions to the processing unit. The ending registers are located in either branch speculation tag generator module or every entry in each reservation station, varying from different synthesis runs.

The computational logic in the critical path includes the calculation of the branch outcome inside the branch unit and the recovery logic based on the outcome.

In the clock cycle that the branch outcome is ready, the speculation tag generator module must release the corresponding speculation tag if the outcome is correct, or recover the corresponding speculation tag if the outcome is incorrect. At the same time, the reservation stations must turn off the corresponding speculative bit or flush the speculative instructions.

The control logic related to branch operations has the longest latency path in the processor.

As a result, the simplified branch recovery logic also shortens the critical path, which achieves 16.9% higher clock frequency in the proposed design compared with Ridecore. By adding a new set of registers that hold the branch outcome, the proposed processor can achieve 104.74Mhz as the maximum clock frequency.

6.2 Vector Co-processor

The vector co-processor that is described in Chapter 4 is implemented in SystemVerilog hardware description language. The proposed vector co-processor is coupled to the previous superscalar processor as one combined system. The functionality of the proposed design is verified by results comparison in the forward inference of the LeNet-5 CNN model. Parameters of the LeNet-5 model are generated by Matlab scripts and are computed in the same way as the 2-D convolution, max-pooling, and FC in the built-in functions. The results calculated by the scripts are the correct references to guarantee that the hardware functions correctly.

The input feature map and parameters are converted to Verilog memory hex file from Matlab arrays as the content in the data memory. The LeNet-5 software is programmed directly in assembly language and converted to Verilog memory hex file by the customized assembler as the content in the instruction memory. To evaluate the vector co-processor, a C program that computes the same LeNet-5 model is executed on the superscalar processor only for reference.

6.2.1 Performance

Table 6.3 presents the results of the machine cycles to finish each layer and instruction code sizes of each approach. The base column shows the machine cycles to finish each layer in the LeNet-5 model, which machine codes are compiled from C program by the RISC-V compiler with the flags "-march=rv32im" and "-O2". The computation flow in the standard RISC-V ISA is limited in fully scalar operation because of its general-purpose property. The inference of the LeNet-5 has the longest latency and the largest code size.

The vector column shows the machine cycles to finish each layer with the extended SIMD

Table 6.3: Machine cycles to finish each layer

	base[1]	vector	vector_opt
conv5×5	905383	475425	161033
pool2×2	55235	7224	6078
conv5×5	1825364	141823	140720
pool2×2	18691	1408	1323
fc	366002	12734	12031
fc	248690	3570	3426
total	3419365	642184	324611
code sizes in bytes	7568	696	1452

[1] rv32im instruction set only.

instructions. This normal vector program employs the basic load-compute-store flow with the convolution and max-pooling assembly kernels provided in the previous section. Because the same type of kernels can share the same assembly kernel, for example, the computation of the first and third layers uses the same block of instructions, the normal vector approach has the smallest code size. Moreover, due to the advantage of SIMD parallelism in the data and computation-intensive tasks, the normal vector program achieves 4.32× throughput improvement compared with the scalar approach.

The vector optimized column shows the machine cycles to finish each layer under the optimizations mentioned in the previous section, including double buffering and loop unrolling. The normal vector approach and the optimized vector approach share the same hardware architecture, however, differ in the software only. With the double buffering, every layer reduces the computation cycles that are required to load the corresponding input feature map and parameters. The loop unrolling optimization adopted in the first layer significantly reduces the computation cycles. The optimized vector program further achieves 9.53× throughput improvement compared with the scalar approach.

On the other hand, the optimizations bring additional codes to switch the internal addresses in double buffering, and a specific computation kernel for the unrolled first layer. As a result, the optimized vector program requires larger code size than the normal vector pro-

gram, but it is still smaller than the scalar program, due to the extended vector instructions that are specific for the machine learning tasks.

Figure 6.4 shows the normalized improvement in three types of layers with the delicate vector program, compared with the scalar program. According to the figure, the fully-connected layer has the best results among other layers. The reason is that the computation flow can be simplified to one vector-multiply-matrix operation in an FC layer, which has the most continuous data patterns of both input vector and input matrix.

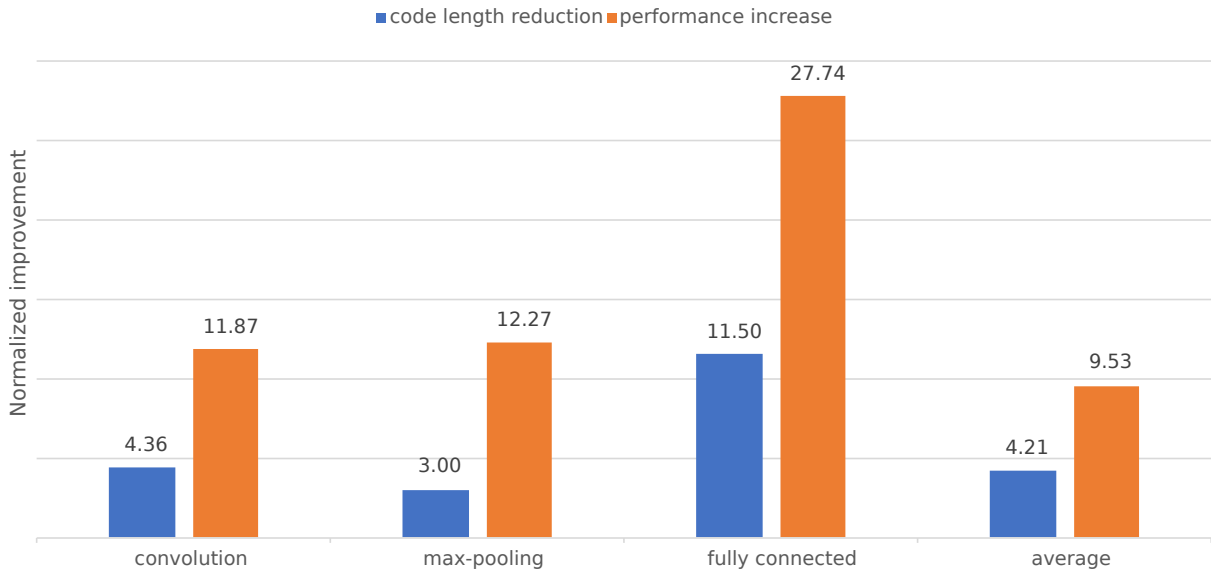

Figure 6.4: Normalized improvements in CONV, Max-pooling and FC layers.

For example in the first FC layer, the computation flow can be simply deployed as one VMM instruction of 1×400 vector multiplied with 400×120 matrix. Even though the layer is divided into 15 parts, that each one is 1×400 vector multiplied with 400×8 matrix, to fit the limitation of internal memory space, the 400 continuous input data keeps iterating in the multiplication sequencer without switching to other operations.

In contrast, the computation flow of convolution layers is paralleled in the depth-dimension. The VMM instruction in the third layer, convolution 5×5, iterates with the input vector of size 6, in which case the dot-product-8 unit must bypass 2 multipliers to handle the size of 6. There are 25 VADD instructions for accumulation after each of the 25 VMM instructions that are required to finish one 5×5 kernel. Therefore, the multiplication sequencer cannot keep

processing in convolution layers, which lowers the resource utilization of the dot-product-8 unit and yields less performance increase in the convolution layers.

Table 6.4 shows the mandatory multiply-accumulate (MAC) operations in convolution layers and fully connected layers. The MAC operations present the computation complexity in different layers. In the proposed vector co-processor design, all multiplication operations take place in the dot-product-8 unit, in which the theoretical computation bandwidth is 8 MAC operations per cycle. Similarly, the data bandwidth is limited by the shared external memory bus, in which the bandwidth is 32-bit so that the theoretical data bandwidth is 4 operands per cycle.

Table 6.4: Throughput of MAC operations

	MAC operations	Clock cycles	MAC-opr/cycle
layer_1 conv5 (1-6@28×28)	117600	161033	0.730285097
layer_3 conv5 (6-16@10×10)	240000	140720	1.705514497
convolution	357600	301753	1.185075211
layer_5 fc (400-120)	48000	12031	3.989693292
layer_6 fc (120-84)	10080	3426	2.942206655
fully connected	58080	15457	3.757520864

In convolution layers, the throughput is limited by the relatively smaller scale of the vectorization. The looping scheme of the convolution is paralleled in either input depth or output depth. However, two convolution layers of this typical LeNet-5 model can only be vectorized into 6 parallel computation. The overheads of each VMM instruction dilute the MAC operation throughput, which includes 1 cycle of addresses initialization and 1 cycle of the first pipeline stage in the dot-product unit.

On the other hand, the overheads of each VMM instruction in fully connected layers become insignificant, compared with the input sequence of 400 elements. Because of its largest number of channels, the layer_5 has the best throughput, which hits the upper bound of the data rate, 4 operands per cycle. According to the waveform, the computation portion of the layer_5 is ended before the loading portion of the layer_6. The double buffering optimization assumes that the loading cycles of the next layer are smaller than the computation cycles

of the current layer. The layer_5 is divided into 15 parts. By the completion of one part of the calculation, the parameters of the next part are still being fetched in the vector load sequencer. The calculation of the next part must wait until the corresponding parameters are ready in the internal memory bank. Therefore, the throughput is limited by 4 MAC operations per cycle.

6.2.2 Synthesis

The proposed design is implemented on the Arria 10 FPGA board for hardware assessment. Table 6.5 provides the synthesis reports of the co-processor coupled with the previous superscalar processors of different pipeline stages.

Table 6.5: FPGA synthesis reports with vector co-processor

	scalar	scalar+vec	scalarpiped[1]+vec
Timing constraint	50Mhz	50Mhz	100Mhz
ALMs	23045	52813	82475
Register	13492	49034	49527
DSP/M20K	9/2	17/16	17/16
Core dynamic power	84.03mW	265.84mW	598.08mW
Fmax[2]	62.85Mhz	60.73Mhz	104.74Mhz

[1] piped branch unit and branch prediction + excessive Quartus compiler.

[2] slow 900mV 100C model.

The vector co-processor is tightly merged with the superscalar processor. It is hard to differentiate the boundary of the vector co-processor. Specifically, a part of the decoder for the vector instructions is located in the ID stage; the source registers of the vector extension are shared with the scalar processor; the commit buffer expands the read bandwidth for the vector instructions. Because the vector co-processor cannot perform independently, it is compiled together with the scalar processor as a full design.

The vector co-processor introduces 1.29× more of the ALMs and 2.63× more of the

registers. Most of those logic blocks are placed in the interconnect module of the processing units in the second stage of the vector co-processor, which appoints the connection between the 6 master ports from sequencers and the 4 slave ports from internal memory banks. The fully associative 6-4 connections yield 24 possible combinations so that the multiplexers of the interconnect module require heavy logic blocks to implement.

The additional 8 DSP blocks are programmed as the multipliers in the dot-product unit. The additional 14 M20K blocks are programmed as the internal memory banks with a total size of 16KB. With the increased logic blocks of ALMs and registers, the vector co-processor consumes 2.16× more power. The critical path is still located in the control logic of the branch/jump unit in the scalar processor. The vector co-processor does not effectively decrease the operating clock frequency and the maximum clock frequency is still dependent on the scalar processor side.

To further reveal the operating speed of the full system, the vector co-processor is coupled with the modified superscalar processor, which adds two additional pipeline stages, one after the branch/jump unit to hold the branch outcome and the other in the IF stage to hold the predicted next PC. Those two pipeline stages reduce the critical path in the original superscalar processor to achieve higher operating clock frequency. The synthesis result of this design is shown in the fourth column in Table 6.5.

The strict timing constraint maintains the max clock frequency of 104.74Mhz. However, the ALM blocks increase significantly compared with the origin scalar+vec design. This reveals that the hardware compiler is reaching the limitation of meeting the timing constraint by increasing the logic usage.

According to the detailed compilation report, the path with the longest latency begins at the newly added register after branch/jump unit and ends in every entry in the reservation stations, which is responsible to flush the speculative instructions upon the miss prediction cycle. In other words, the critical path is still located in the prediction recovery logic of the scalar processor, which limits the maximum operating frequency to 104.74Mhz.

From the perspective of the vector co-processor, every issued vector instruction from the host processor is not speculative and is resolved in the vector reservation station. Therefore, the co-processor, without the prediction recovery logic, does not meet the timing roofline.

Table 6.6: Performance and energy efficiency over the LeNet-5 model

	scalar	scalar+vec	scalarpiped+vec
Inference cycles (cycles/image)	3419365	324611[1]	324611[1]
Operating freqency[2](Mhz)	62.85	60.73	104.74
Throughput (images/second)	18.38060576	187.0855	322.663126
Power consumption[3](mW)	84.03	265.84	598.08
Energy efficiency (performance/watt)	218.7386143	703.7521	539.4982712

[1] optimized vector program.

[2] maximum clock frequency.

[3] core dynamic power.

Table 6.6 shows the real-time inference throughput of the LeNet-5 model and the energy efficiency in the proposed designs. The superscalar RISC-V processor can process 18.38 images per second in real-time with the CNN model. With the extended vector instructions and SIMD computation, the vector co-processor can process 187.09 images per second and 322.66 images per second, which provides 10.18× and 17.55× real-time throughput, respectively with the basic processor and the best-effort processor.

However, by considering the power consumption, the best-effort design with the highest throughput does not lead to the best energy efficiency, due to the significant increase of the ALM blocks and the excessive compiler scheme. The vector co-processor coupled with the normal superscalar processor achieves the best energy efficiency among all implementations.

Chapter 7
Conclusion

7.1 Conclusion

This thesis work presents the hardware implementations of a dual-issue superscalar RISC-V processor with out-of-order execution and a SIMD vector co-processor with customized vector instructions. The proposed superscalar processor is targeted to achieve high performance in the field of general-purpose tasks, while the proposed vector co-processor, with the extended vector instructions, is targeted to further enhance the performance specifically in the machine learning area.

In the proposed superscalar processor, the Tomasulo algorithm is implemented in the hardware architecture to enable the out-of-order execution. The Gshare branch prediction technique is applied in the instruction fetch stage. With 5 backups of the renaming register file, the processor can speculatively execute instructions to reduce the waste cycles that are caused by the branch operations. The busy counters in the renaming register file bring better instruction throughput compared to the conventional one-bit busy status. The processor, with the busy counters in the renaming file, can continuously dispatch instructions that keep modifying the same destination register to fulfill the utilization of every pipeline stage. By rearranging the latest value column in the traditional register renaming file to the result column in the commit buffer, the hardware complexity is reduced to save the area and power consumption. Moreover, the simplified prediction recovery scheme shortens critical paths to reach higher operating clock frequency. Compared to a similar design, the proposed RISC-V processor improves average instruction throughput by 18.9% and average prediction hit rate by 4.92%. Additionally, the proposed processor reaches 16.9% higher operating frequency with the additional support of machine-level exception and integer multiplication/division.

In the proposed vector co-processor, the SIMD architecture is adopted to increases the performance of the computation and data-intensive tasks. A customized SIMD instruction set is proposed based on the Cambricon ISA and is mapped to the standard 32-bit RISC-V instruction format. Compared to the Cambricon ISA, the proposed vector extension unifies the internal address mapping to emphasize the flexibility of the instruction set. Following the specification of the proposed vector instruction set, the co-processor consists of the vector instruction board, the wrapped internal memory banks, and the corresponding processing units. The instruction board merges the functionalities of the reservation station and commit buffer, which can solve the data dependency and provide instruction-level parallelism in the processing units. The wrapped memory bank of the true-dual-port memory block supports one-cycle misaligned memory access in hardware to simplify the sequencers and leverage the memory utilization. In the case study of the LeNet-5 model, the normal vector program achieves 4.32× throughput improvement and the delicate vector program with software optimizations achieves 9.53× improvement, compared to the basic C program. The vector co-processor with the superscalar processor can handle 187.09 images per second, which provides 10.18× real-time throughput and 2.22× energy efficiency compared with the RISC-V processor alone.

In conclusion, the fully-tested superscalar processor may be a reference model of the central control unit for future FPGA applications. The vector co-processor specifically enhances the performance of CNN tasks with decent versatility for future-proofing. Future developers may add new customized instructions and accelerators for other specific tasks by following the outline of this work.

7.2 Future Work

There are several topics of this work to research in the future. One group of the extensive works is related to optimizing the superscalar processor. The other group is related to refining the machine learning specific co-processor.

The superscalar RISC-V processor in this thesis only supports bare-metal computation without any firmware layer. Several features should implement on the current processor to

bring better usability. In the RISC-V privileged specification, the supervisor-level and user-level privileged CSRs are also provided for hardware design. The firmware kernel should execute in the supervisor mode. The RISC-V programs, on the other hand, should execute in the user mode, which entry addresses and heap pointers are controlled by the supervisor mode. The memory virtualization feature with organized peripherals to support the memory hierarchy is mandatory for those two privileged modes.

At the same time, other common RISC-V standard extensions can be defined in the processor, including the compressed "C" extension, the floating-point "F" extension, and the atomic "A" extension, to expand the compatibility.

The proposed vector co-processor only tests the performance on a basic CNN model. However, the instruction-based accelerator is versatile for many different neural network models and different types of layers. Other popular layers, including squeeze-and-excite, inception, depth-wise convolution, RNN, and LTSM, can be evaluated by the design.

The element format in the vector co-processor is defined as an 8-bit integer. However, fixed point numbers are inadequate for real machine learning applications with a relatively small data range. By keeping the width of each element to 8-bit, the data formats of Minifloat [40] and Posit [41] can be investigated on the processing units to produce the accurate result with a real set of neural network parameters.

References

[1] Y. Chen, H. Lan, Z. Du, S. Liu, J. Tao, D. Han, T. Luo, Q. Guo, L. Li, Y. Xie, and T. Chen, "An Instruction Set Architecture for Machine Learning," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3331469

[2] M. M. Mano, *Computer System Architecture (3rd Ed.)*. USA: Prentice-Hall, Inc., 1993.

[3] A. Khan, K. M. Khare, D. K. Mishra, R. S. Gamad, D. V. Ghodke, and V. K.Senecha, "FPGA based embedded controller using soft core processor for Ion Source Applications," in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, 2018, pp. 592–596.

[4] W. Xie, C. Zhang, Y. Zhang, C. Hu, H. Jiang, and Z. Wang, "An Energy-Efficient FPGA-Based Embedded System for CNN Application," in *2018 IEEE International Conference on Electron Devices and Solid State Circuits (EDSSC)*, 2018, pp. 1–2.

[5] G. A. M. Sborz, G. A. Pohl, F. Viel, and C. A. Zeferino, "A Custom Processor for an FPGA-based Platform for Automatic License Plate Recognition," in *2019 32nd Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2019, pp. 1–6.

[6] C. Wolf, "PicoRV32 - A Size-Optimized RISC-V CPU," https://github.com/cliffordwolf/picorv32, 2019.

[7] SI-RISCV, "The Ultra-Low Power RISC Core," https://github.com/SI-RISCV/e200_opensource, 2019.

[8] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2017.

[9] 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.

[10] pulp-platform, "Ariane RISC-V CPU," https://pulp-platform.github.io/ariane/docs/home/, 2019.

[11] Y. Lee, "RISC-V "Rocket Chip" SoC Generator in Chisel," EECS Department, University of California, Berkeley, Tech. Rep., Jun 2015. [Online]. Available: https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015.pdf

[12] M. Fujinami, S. Mashimo, T. V. Chu, and K. Kise, "RIsc-v Dynamic Execution CORE," https://github.com/ridecore/ridecore, 2017.

[13] A. Waterman, "Design of the RISC-V Instruction Set Architecture," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[14] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129, Jul 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html

[15] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[16] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[17] 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, 2020.

[18] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[19] Y. Chang, P. Lin, S. Cheng, K. Chan, Y. Zeng, C. Liao, W. Chang, Y. Wang, and Y. Tsao, "Robust anchorperson detection based on audio streams using a hybrid I-vector and DNN system," in *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*, 2014, pp. 1–4.

[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] A. Howard, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for MobileNetV3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[22] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: Hardware-aware neural network design," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2018, pp. 1719–171 909.

[23] T. Elsken, J. H. Metzen, and F. Hutter, "Neural Architecture Search: A Survey," *J. Mach. Learn. Res.*, vol. 20, pp. 55:1–55:21, 2019.

[24] J. Yu, G. Ge, Y. Hu, X. Ning, J. Qiu, K. Guo, Y. Wang, and H. Yang, "Instruction Driven Cross-Layer CNN Accelerator for Fast Detection on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3283452

[25] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, 2002.

[26] S. Mcfarling, "Combining Branch Predictors," Tech. Rep., 1993.

[27] R. Nair, "Optimal 2-bit branch predictors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 698–702, May 1995.

[28] M. Lu, *Arithmetic and Logic in Computer Systems*, ser. Wiley Series in Microwave and Optical Engineering. Wiley, 2005. [Online]. Available: https://books.google.ca/books?id=ABZ4bgU75pQC

[29] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the Devil in the Details: Delving Deep into Convolutional Nets," *CoRR*, vol. abs/1405.3531, 2014. [Online]. Available: http://arxiv.org/abs/1405.3531

[30] SIFIVE, "riscv-tests," https://github.com/riscv/riscv-tests, 2019.

[31] Y. Sundblad, "The Ackermann function. a theoretical, computational, and formula manipulative study," *BIT Numerical Mathematics*, vol. 11, no. 1, pp. 107–119, Mar 1971. [Online]. Available: https://doi.org/10.1007/BF01935330

[32] A. M. Hinz, S. Klavzar, U. Milutinovic, and C. Petr, *The Tower of Hanoi - Myths and Maths*. Birkhäuser Basel, 2013.

[33] T. Ishiu and M. Kikuchi, "The Termination of the Higher-Dimensional Tarai Functions," *Inf. Process. Lett.*, vol. 115, no. 2, p. 125–127, Feb. 2015. [Online]. Available: https://doi.org/10.1016/j.ipl.2014.08.001

[34] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Commun. ACM*, vol. 27, no. 10, p. 1013–1030, Oct. 1984. [Online]. Available: https://doi.org/10.1145/358274.358283

[35] G. H. Goble and M. H. Marsh, "A Dual Processor VAX 11/780," *SIGARCH Comput. Archit. News*, vol. 10, no. 3, p. 291–298, Apr. 1982. [Online]. Available: https://doi.org/10.1145/1067649.801738

[36] The SiFive E31 Standard Core. SiFive. [Online]. Available: https://www.sifive.com/cores/e31

[37] Nios II Processors for FPGAs. Intel Corporation. [Online]. Available: https://www.intel.ca/content/www/ca/en/products/programmable/processor/nios-ii.html

[38] (2012, 29 August) Cortex-M3. ARM Holdings. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m3

[39] (2011, 7 July) Cortex-A5. ARM Holdings. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-a/cortex-a5

[40] C. A. Cermelli, D. G. Roddier, and C. C. Busso, "MINIFLOAT: A Novel Concept of Minimal Floating Platform For Marginal Field Development," Toulon, France, p. 8, Jan 2004, iSOPE.

[41] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du, "Training Deep Neural Networks Using Posit Number System," *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, Sep 2019. [Online]. Available: http://dx.doi.org/10.1109/SOCC46988.2019.1570558530