POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design and Implementation of Instruction Fusion Techniques in a RISC-V Out-Of-Order core

Supervisors

Prof. Maurizio ZAMBONI

Candidate

Chiara RIATTI

October 2024

Abstract

This thesis explores the design and implementation of instruction fusion techniques in a RISC-V Out-of-Order core. Instruction fusion is a microarchitectural technique that aims at optimising performance by merging multiple instructions inside the processor into a single operation, without modifying the Instruction Set Architecture. The study explores the potential of instruction fusion in optimizing resource utilization, reducing instruction count, and improving execution efficiency within the Lagarto Ox core, developed at the Barcelona Supercomputing Center. The work involves the design of a scalable infrastructure capable of supporting various fusion idioms, initially focusing on compressed instructions with plans for future support of normal-sized instructions. Verification and validation are conducted through custom testbenches, while performance evaluations demonstrate significant reductions in execution cycles and instruction counts. The study concludes with an analysis of the physical design impact, emphasizing the trade-offs in area and frequency for implementing instruction fusion in a RISC-V architecture.

Keywords: Instruction fusion, Out-of-Order core, RISC-V, Microarchitectural technique.

Acknowledgements

I would like to thank my supervisor, professor Maurizio Zamboni, for his support in this work.

Moreover, I would like to thank Jonnatan Mendoza and Francesc Moll, my Master's thesis advisors at the Barcelona Supercomputing Center, for giving me the opportunity to work on this project and all the support during my time in the Barcelona.

Table of Contents

List of Tables IV

List of Figures V

Acronyms VIII

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Instruction Set Architecture . . . 1
  - 1.1.2 Instruction Fusion . . . 3
- 1.2 Objectives . . . 5

2 Background 6
- 2.1 State of the art . . . 6
- 2.2 Computer architecture . . . 12
- 2.3 Methodology . . . 18

3 Design and Implementation 22
- 3.1 Instruction fusion . . . 22
  - 3.1.1 Fusion candidates . . . 23
  - 3.1.2 Fused instruction format . . . 26
- 3.2 Fusion support in the Front-End . . . 28
  - 3.2.1 Fetch stage: Fusion Detector . . . 28
  - 3.2.2 Fetch stage: testbench . . . 37
  - 3.2.3 Decode stage: Fusion Decoder . . . 39
  - 3.2.4 Decode stage: testbench . . . 40
  - 3.2.5 Load pairs . . . 41
- 3.3 Fusion support in the Back-End . . . 46
  - 3.3.1 Integer Execution: Fusion ALU . . . 46
  - 3.3.2 Integer Execution: testbench . . . 49
  - 3.3.3 Memory Execution . . . 49

3.3.4 Memory Execution: testbench 52
3.4 Exception handling 52
3.4.1 Fetch stage 53
3.4.2 Reorder Buffer 53
3.4.3 Program Counter 57
3.5 Entire core 58

4 Verification and Performance 60
4.1 ISA tests 60
4.2 Benchmarks 62
4.2.1 Performance 63

5 Physical design impact 66
5.1 Area 66
5.2 Frequency 71

6 Conclusions 72
6.1 Future work 73

Bibliography 75

List of Tables

3.1 Driving signal of the multiplexer to select the fused instruction output 35
3.2 Replace mask and Invalid mask generation 36
3.3 Register constraints 37
3.4 Register constraints for Load Pairs idiom 45
3.5 Operation supported by the Fusion ALU 48

List of Figures

1.1 Abstraction Layers . . . 2

2.1 Total dynamic instruction count from paper [10] . . . 7
2.2 Total dynamic instruction bytes fetched from paper [10] . . . 7
2.3 Fusion candidates discussed in [10], taken from paper [5] . . . 10
2.4 Overview of pipeline of Helios architecture to support Fusion from paper [5] . . . 10
2.5 Fusion Target of XiangShan Processor from paper [15] . . . 11
2.6 Fusion decode unit implemented in XiangShan Processor from paper [15] . . . 11
2.8 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm from [18] . . . 15
2.9 HLIB engineering cycle . . . 19
2.7 Lagarto Ox core diagram . . . 21

3.1 Supported fusion sequences . . . 25
3.2 R4 format . . . 26
3.3 F_R4 format . . . 27
3.4 F_R4 fields . . . 27
3.5 Registers specified by the 3-bit encoding, taken from [22] . . . 28
3.6 Architecture of the Fetch stage . . . 29
3.7 Focus on F2 stage . . . 30
3.8 Focus on signals connection inside Fusion Detector: how the input signals are connected with instantiation #0 of c_detector . . . 31
3.9 Detection of two Scaled Load sequences and one Load Effective Address . . . 32
3.10 Detection of four fusible sequences made of two instructions each . . . 32
3.11 Focus on output replace and invalid mask of Fusion Detector module 33
3.12 Example of input sequence and generation of the replace and invalid masks . . . 33
3.13 Reduction from the 7 masks to to final ones relative to Figure 3.12 . . . 34

3.14 Selection of the fused instruction output 34
3.15 Parallel detection 36
3.16 Test bench flow of the Fusion Decoder 38
3.17 Decode stage 39
3.18 Scaled Index family control signals 40
3.19 Example of control signals structure 41
3.20 Output of python script from fused instruction to original instructions. 41
3.21 Percentage of fused micro-ops considering all or just memory fusion idioms, taken from paper [5] 42
3.22 Load Pairs fusion sequences 43
3.23 F_R4 format with Load Pairs 43
3.24 Opcode of Scaled Index family and Load Pairs family 44
3.25 F_R4 fields of the Load Pairs family 44
3.26 Load Pairs family control signals 45
3.27 Pipeline, focus on integer execution 46
3.28 Integer FU generator 47
3.29 SystemVerilog implementation of 32-bit LEA operation 48
3.30 SystemVerilog implementation of 64-bit LEA operation 49
3.31 Block diagram of Memory Execution 50
3.32 Effective address calculation in IL instruction, for 32-bits configuration 51
3.33 Effective address calculation in IL instruction, for 64-bits configuration 51
3.34 Effective address calculation in SL instruction, for 32-bits configuration 52
3.35 Effective address calculation in SL instruction, for 64-bits configuration 52
3.36 Block scheme of Reorder Buffer 54
3.37 Completed vector boundaries 55
3.38 Completed vector boundaries, corner case of circular buffer 55
3.39 Modified block scheme of the Reorder Buffer 57
3.40 Program Counter update for Indexed Load and Scaled Load operations 58
3.41 Lagarto Ox core diagram with instruction fusion support 59

4.1 Performance results of the Lagarto OX core without Instruction Fusion 62
4.2 Retired Instructions count for the four benchmarks 64
4.3 Cycles count for the four benchmarks and the ad hoc benchemarks 65

5.1 Area breakdown graph of Fetch Stage. 66
5.2 Fetch Stage area comparison 67
5.3 Area breakdown graph of Decode Stage 68
5.4 Decode Stage area comparison 68
5.5 Area breakdown graph of Front-End 69
5.6 Integer execution area comparison 69

5.7 Memory execution area comparison 70
5.8 Overall area comparison 70
5.9 Critical paths comparison 71

Acronyms

ISA
: Instruction Set Architecture

RISC
: Reduced Instruction Set Computer

CISC
: Complex Instruction Set Computer

CPU
: Central Processing Unit

ALU
: Arithmetic Logic Unit

ROB
: Reorder Buffer

PC
: Program Counter

LEA
: Load Effective Address

IL
: Indexed Load

SL
: Scaled Load

Chapter 1

Introduction

This chapter aims to present the motivation behind the choice of implementing the instruction fusion technique and the objectives of this thesis work.

1.1 Motivation

1.1.1 Instruction Set Architecture

In a computer, the processor is the physical heart where operations occur. It interprets and performs commands from the computer's software. The interface between hardware and software is the instruction set architecture (ISA). The ISA defines which instructions the processor can execute. Consequently, a CPU implementing a particular ISA can execute all instructions defined by that architecture. Similarly, software designed for a specific ISA will run on any processor that supports it. The same instruction set architecture can be supported by different processors; however, the specific design and implementation, known as microarchitecture, can vary between processors [1]. x86-64 and RISCV are ISAs. Skylake is an Intel microarchitecture, Zen is an AMD microarchitecture, and they both support the same ISA, x86-64.

In the abstraction hierarchy, the ISA is between the microarchitecture and the code, as shown in Figure 1.1. The ISA defines all the necessary elements to allow the processor to communicate with the software: the operations that the processor can execute, the memory management, the supported registers and data formats. Two main paradigms that dominate the current CPU development context are distinguished by their architectural complexity: the Reduced Instruction Set Computer and the Complex Instruction Set Computer. Some examples of RISC ISAs are MIPS, RISC-V and ARM. On the other hand, x86 is a CISC ISA used by most

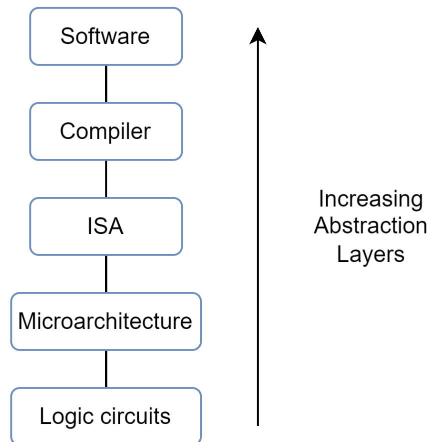


Figure 1.1: Abstraction Layers

Intel and AMD processors.

The Complex Instruction Set Architecture (CISC) supports complex instructions that can be carried out through multiple clock cycles. This results in denser code, since a complex operation is encoded in a single instruction, reducing memory usage and improving cache performance. On the other hand, the decoding of the instructions is more complicated and complex operations have to be broken down at microarchitectural level into simpler instructions called micro-op in order to actually be executed.

The Reduced Instruction Set Architecture (RISC) uses simple instructions that can be executed in a single cycle. The decoding and execution are simple and that allows to reduce power consumption. Opposite to the CISC paradigm, executing a complex instruction requires the processor to carry out each individual basic step of a complex operation separately. The simplicity of the RISC ISA does not constrain microarchitectural design choices aimed at enhancing performance. Instead, it offers significant flexibility, allowing designers to directly exploit the ISA's simplicity for efficient processing or implement advanced hardware techniques such as out-of-order execution, multiple issue, and, lastly, instruction fusion. Instruction fusion serves as a mechanism to bridge the gap between RISC and CISC paradigms by internally synthesizing more complex operations within a RISC processor.

All of the popular commercial ISAs are proprietary. However, RISC-V is a free and open ISA that is based on the original Reduced Instruction Set Computer architectures. It is structured as a small base ISA with a variety of extensions [2]. This is significant because it allows smaller device manufacturers to build hardware

without having to pay royalties. Both for low-power embedded systems and high performance.

1.1.2 Instruction Fusion

Instruction fusion is a microarchitectural technique used to improve the performance of modern processors by combining multiple instructions in a single operation. It is important to distinguish between these matters due to the varying terminology [3] :

- MicroFusion is when multiple micro-ops from the same assembly instruction are merged into a single micro-op. It happens inside the processor's pipeline. MicroFusion can take place in the processors where the backend breaks down a single instruction into micro-op. Fused micro-ops count as one, from the decoders to the reservation station in order to save pipeline bandwidth. However, they can not be executed so they are split at the execution stage. Examples are in Intel and AMD CPUs [4].
- MacroFusion is when multiple ops from different assembly instructions are merged into one micro-op. This is made by the decoding pipeline inside the CPU. Both Micro and Macro fusion can be supported at the same time.

This thesis concentrates on MacroFusion. The purpose of instruction fusion is to "maximize resource utilization by saving resources such as Reorder Buffer, Scheduler, and Load/Store Queue entries" [5], as well as executing more work with fewer bits, consequently saving power.

Commercial processor with instruction fusion

The technique for fusing instructions is owned by Intel and is protected by a patent filed in December 2000 [6]. X86 processors support both MicroFusion and MacroFusion. MicroFusion combines load-ALU operations and ALU-store operations. MacroFusion takes compare operations followed by a branch operation and fuses them together. The advantages of the technique are an increase in the rename and retire bandwidth, more storage for instructions in-flight, and power savings by representing more work in fewer bits [7].

Other examples of instruction fusion in a commercial CISC architecture are AMD Zen microarchitecture. Zen 1 and Zen 2 support the fusion of a CMP or TEST instruction immediately followed by a conditional jump into a single instruction. Zen 3 is able to fuse an arithmetic or logic instruction immediately followed by a conditional jump. This applies to instructions such as CMP, TEST, ADD, SUB,

AND, OR, XOR, INC, and DEC, along with all conditional jumps. [4]

Arm also supports some macro-op fusion operations in recent microarchitectures. An example is Arm Cortex-A78C Core 1 that supports the fusion of the following instructions [8]:

- CMP/CMN (immediate) + B.cond
- CMP/CMN (register) + B.cond
- TST (immediate) + B.cond
- TST (register) + B.cond
- BICS (register) + B.cond
- NOP + any instruction

In all the processors proposed, the instruction pairs must be adjacent to each other in the program code in order to be fused. It is possible to see that in all the commercial processors listed here the instruction fusion regards compare-and-branch sequences, that typically require two instructions both for ARM and x86, and, based on Intel's estimation, they make up 15% of all instructions.
In contrast, in a RISC-V architecture, this kind of fusion is not needed because it already supports a special dedicated instruction format for conditional branches. The RISC-V Instruction Set Manual states "The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC and Xtensa ISA), rather than use condition codes (x86,ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS)." [9]

Instruction Fusion in RISC-V core

The use of macro-op fusion in RISC-V was proposed in a 2016 Berkeley paper [10]. The reason for implementing instruction fusion in a RISCV architecture is that the instructions of this ISA are simple, and some instructions do not fully utilize the available resources or do not need all the internal resources that the format allows them to claim. The resources are better exploited and bandwidth is internally saved by combining the instructions into a more complex one that the processor can still handle. The front-end detects the fusible instructions; if the requirements are met, a fused instruction is created and sent through the pipeline in the same manner as a regular instruction, but with just one instruction now present rather than the original's several. MacroFusion can be seen as the opposite to the process

of generating many micro-ops from a single ISA instruction that takes place in CISC cores.
Some examples of commercial RISC-V processors that implement instruction fusion are Ventana's Veyron V1 [11] and SiFive's P870 [12].

1.2 Objectives

The objective of this master's thesis is to design, implement, and test instruction fusion support in the Lagarto Ox Out-of-Order core, while preserving a scalable infrastructure to implement further idioms of fusion.
More detailed objectives are:

- Investigate existing instruction fusion techniques: research the current state of the art of the usage of instruction fusion techniques focusing in its support for RISC-V processors
- Design custom instruction fusion techniques for RISC-V: design instruction fusion architectural support tailored to out-of-order execution and RISC-V processors, considering its unique characteristics and usage patterns
- Integration into Lagarto Ox Core: RTL development and integration into the Out-of-Order RISC-V Lagarto OX core
- Verification and validation: develop a verification framework in cocotb for the derived modules of the instruction fusion feature
- Performance evaluation: performance evaluation of the fusion techniques in terms of execution efficiency, reduced instruction count, reduced cycle count, and overall performance
- Trade-off: examine the impact on the area requirements and frequency.

Chapter 2

Background

In this chapter are reported the study of instruction fusion techniques present in RISC-V processor and the theoretical background needed to understand how this technique can be implemented inside the Lagarto Ox core developed at the Barcelona Supercomputing Center (BSC).

2.1 State of the art

The use of macro-op fusion in RISC-V was proposed in 2016 in the paper titled "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V" paper [10].
The objective of the article is to use instruction fusion to increase RISC-V's code density and performance without compromising the simplicity that makes it a Reduced Instruction Set Computer. They analyze the dynamic instruction counts and bytes fetched for widely used proprietary ISAs, including ARMv7, ARMv8, IA-32, and x86-64, in comparison with the free and open RISC-V RV64G and RV64GC ISAs using the SPEC CINT2006 benchmark suite.

In Figure 2.1, the total dynamic instruction count is shown for each of the ISAs, normalized to the x86-64 instruction count. RV64G executes 16% more instructions than x86-64, 3% more than IA-32, 9% more instructions than ARMv8, and 4% fewer instructions than ARMv7. With the use of instruction fusion, the instruction count for RV64GC is reduced by 5.4%.
Figure 2.2 presents data on the total dynamic instruction bytes fetched. The RV64G architecture, with its fixed 4-byte instruction size, fetches 23% more bytes per program than x86-64. Contrary to expectations, the x86-64 architecture exhibits a relatively low instruction density, with an average of 3.71 bytes per instruction.

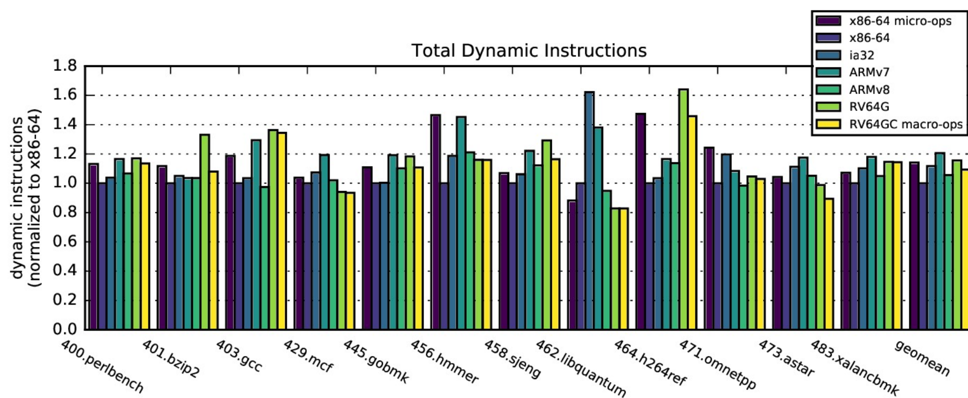


Figure 2.1: Total dynamic instruction count from paper [10]

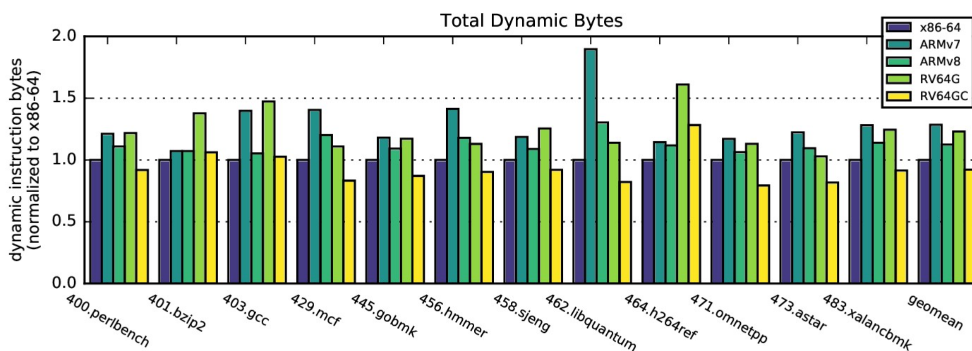


Figure 2.2: Total dynamic instruction bytes fetched from paper [10]

Similarly to RV64G, both ARMv7 and ARMv8 architectures employ a fixed 4-byte instruction size. The use of the RISC-V "C" Compressed ISA extension in RV64GC results in an average of 3.00 bytes per instruction, 8% fewer dynamic instruction bytes than x86-64. Moreover, RV64GC retrieves significantly fewer bytes compared to both ARMv7 and ARMv8 architectures.

The RISC-V "C" Compressed ISA extension encodes frequently used instructions into 16-bit formats, which occupy half the space of a standard RISC-V instruction word. This reduction in code size increases fetch bandwidth and aims to enhance both energy efficiency and overall performance. However, due to the limited space within a 16-bit format, only a subset of the most common instructions, such as add, load, and branch, can be compressed. "RVC programs are 25% smaller than RISC-V programs, fetch 25% fewer instruction bits than RISCV programs, and incur fewer instruction cache misses" [13].
Given these advantages, it is logical to pair the compressed extension with instruction fusion to further maximize code density and minimize the instruction count.

This combination leverages the strengths of both techniques, achieving denser, more efficient code while optimizing the performance benefits of RISC-V systems.

With the results of a 5.4% decrease in the effective instruction count when macro-op fusion is implemented in conjunction with the compressed instruction set extension, the authors reach the goal of giving the RISC-V ISA denser code and achieving more work per instruction, without modifying the ISA.

The support for three macro-op fusion has been implemented in the in-order single core Rocket [14]. In the decode step, the two instructions are fused together if fusion is feasible. The result of the first instruction is kept visible in case the second instruction in the fused pair causes an exception.

The work [10] includes an analysis of which idioms increase the instruction count and could serve as strong candidates for fusion in a RISC-V architecture. The analysis concludes that, unlike other proprietary RISC ISAs such as ARMv7 and CISC architectures such as x86-64 and IA-32, RISC-V uses additional instructions for indexed memory operations, unsigned integer array indexing, and library routines such as memset and memcpy.
The three macro-op fusion implemented are:

- Load effective address

  slli rd, rs1, imm
  add rd, rd, rs2

  It computes the effective address of a memory location and places the address into a register. The typical use case is an array offset that is shifted to a data-aligned offset and then added to the array base address.

- Indexed load

  add rd, rs1, rs2
  ld rd, 0(rd)

  It is an extremely common idiom, that in x86-64 and ARM is implemented as a single instruction, but RISC-V requires up to three instructions to have the same behavior. This idiom loads data from an address calculated by adding two registers. It can also be combined with the Load effective address to generate a fusion of three instructions.

- Clear upper word

  slli rd, rs1, 32
  srli rd, rd, 32

  It zeros the upper 32-bits of a 64-bit register.

Some interesting idioms analyzed by the authors [10] but not implemented are reported here. An example is the Wide Multiply/Divide idioms: multiplication in RISC-V generates a product of size 2*XLEN. Two separate instructions are needed to get the full 2*XLEN of the product:

MULH[[S]U] rdh, rs1, rs2
MUL rdl, rs1, rs2

In order to create immediate values larger than the standard 12 bits that most RISC-V instructions may use, the Load Upper Immediate instruction is utilized. This pair loads a 32-bit immediate into a register:

lui rd, imm[31:12]
addi rd, rd, imm[11:0]

ARMv8 employs load-pair and store-pair instructions to transfer up to 128 contiguous bits of data from memory into two separate registers (or vice versa) within a single operation. In RISC-V, this functionality can be mimicked by combining consecutive load (or store) instructions that access adjacent memory addresses. However, these instruction sequences require two write ports on the register file, in case of the load pairs, three read ports in case of the store pairs.

ld rd1, imm(rs1)
ld rd2, imm+8(rs1)

The complete list of all the idioms analyzed in the article is reported in Figure 2.3.

The article also highlights how crucial the compiler is to achieve the intended outcome. A fusion-aware compiler, which has the task of increasing the number of fusible pairings in the compiler-generated code, will ensure that all the work of supporting macro-op fusion yields results.
Finally, the application will determine the benefits of this technique. There will not be any improvements if the code of the benchmarks do not come with any of the fusible opportunities supported by the processor.

add rd, rs1, rs2 *ld rd, 0(rd)*	*lui rd, imm[31:12]* *addi rd, rd, imm[11:0]*
ld rd, imm(rs1) *add rs1, rs1, 8*	*auipc t, imm20* *jalr ra, imm12(t)*
slli rd, rs1, {1,2,3} *add rd, rd, rs2*	*mulh[[S]U] rdh, rs1, rs2* *mul rdl, rs1, rs2*
slli rd, rs1, 32 *srli rd, rd, 29/30/31/32*	*div[U] rdq, rs1, rs2* *rem[U] rdr, rs1, rs*
lui rd, imm[31:12] *ld rd, imm[11:0](rd)*	*auipc rd, symbol[31:12]* *ld rd, symbol[11:0](rd)*
ld rd1, imm(rs1) ***ld rd2, imm+8(rs1)***	***st rs2, imm(rs1)*** ***st rs3, imm+8(rs1)***

Figure 2.3: Fusion candidates discussed in [10], taken from paper [5]

The article "Exploring Instruction Fusion Opportunities in General Purpose Processors" [5], published in 2022, proposes a focus on the fusion of memory instructions. Instruction fusion is done for consecutive instructions, which is the operation of fusing instructions that are consecutive in the dynamic execution of the stream, and contiguous instructions, which is the fusion of memory operations that access contiguous memory bytes. The authors aim at increasing the number of fused memory instructions by relaxing these two constraints. They focus on implementing the fusion of load pairs and store pairs. It consists of the fusion of two loads even if they access non-contiguous data as long as that data fits within a specific memory region. This approach can be considered a run time fusion, in contrast to the static time fusion analyzed in the previous article. Run-time fusion consists of a speculation on the possibility that those instructions are really fusible because it requires knowledge that is not available until the execution stage, above all the effective address. In order to implement the run-time fusion, a mechanism of prediction and recovery is needed, it is implemented in the Helios architecture as it is possible to see in Figure 2.4.

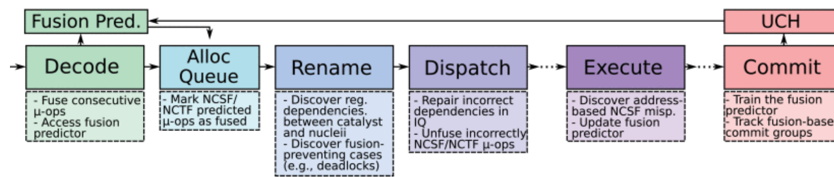


Figure 2.4: Overview of pipeline of Helios architecture to support Fusion from paper [5]

Helios allows to fuse an additional 5.5% of dynamic instructions, bringing a 14.2% performance uplift over no fusion.

Another implementation of instruction fusion in RISC-V is the paper “Accelerate Bit Manipulation in XiangShan Processer Using RISC-V B Extension and Instruction Fusion” (2022) [15]. The purpose is to exploit instruction fusion to accelerate bit manipulation. Instruction fusion is implemented in the decode unit. They propose 10 new instruction fusion candidates after profiling the benchmarks and finding the frequent idioms for their specific application. They are summarized in Figure 2.5.

Target	Instruction pairs	description
szewl1	slli r1, r0, 32 + srli r1, r1, 31	left shift zero-extended word by 1 bit
szewl2	slli r1, r0, 32 + srli r1, r1 30	left shift zero-extended word by 2 bits
szewl3	slli r1, r0, 32 + srli r1, r1 29	left shift zero-extended word by 3 bits
sh4add	slli r1, r0, 4 + add r1, r1, r2	left shift by 4 bits and add
sr29add	srli r1, r0, 29 + add r1, r1, r2	right shift by 29 bits and add
sr30add	srli r1, r0, 30 + add r1, r1, r2	right shift by 30 bits and add
sr31add	srli r1, r0, 31 + add r1, r1, r2	right shift by 31 bits and add
sr32add	srli r1, r0, 32 + add r1, r1, r2	right shift by 32 bits and add
oddaddw	andi r1, r0, 1 + addw r1, r1, r2	add one if odd (in word format)
orh48	andi r1, r0, -256 + or r1, r1, r2	or operation with high 48 bits

Figure 2.5: Fusion Target of XiangShan Processor from paper [15]

The instructions that can be fused are detected in the decode stage by a specific Fusion Decode Unit. This unit checks every two adjacent instructions to verify that they match one of the supported pairs. If there is a match, the first instruction is replaced with the fused instruction and the second is discarded. To keep track of the fused instruction a bit is added to the control signal. This architecture is shown in Figure 2.6.

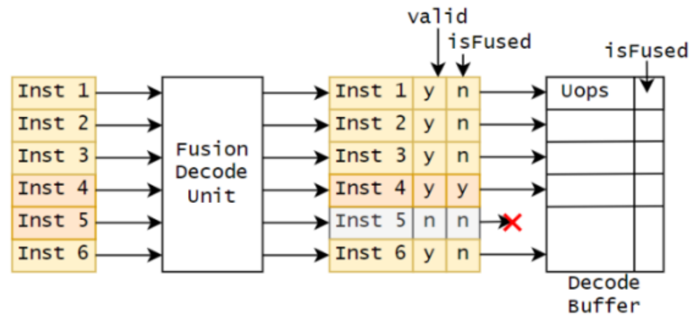


Figure 2.6: Fusion decode unit implemented in XiangShan Processor from paper [15]

To evaluate the results, they use Coremark benchmark. It shows that dynamic instructions reduce by 18.14%, and the performance improves by 12.09% and a reduction of 10.74% of cycle count.

2.2 Computer architecture

This project was developed while working on the on-going project of the third release of the Lagarto family processors. Lagarto Ka [16] and Lagarto Ox are the second and third generations of the Lagarto core family developed at BSC, as part of the DRAC project. The DRAC project has as its primary goal the design, verification, and fabrication of a high-performance processor that integrates several accelerators and other features in the same system-on-chip, using RISC-V technology. [17]

Lagarto Ox Core

Lagarto Ox is a reconfigurable core design implementing the RVGC Instruction Set Architecture. It supports an 11-stage pipeline with Out-of-Order execution and in-order commit, with the following stages:

1. Instruction Fetch 1
2. Instruction Fetch 2
3. Instruction Fetch 3
4. Instruction Decode
5. Register Renaming
6. Dispatch/Allocation
7. Issue
8. Read Registers
9. Execution
10. Write-Back
11. Commit

The Lagarto Ox core diagram is shown in Figure 2.7. The Front-End of the processor is responsible for maintaining a continuous stream of instructions in the pipeline by fetching a new set of instructions from the Instruction Cache during

each cycle. These instructions are then decoded, renamed, and dispatched to the Back-End to its respective instruction scheduler. Register renaming assigns each instruction a *rob id* and resolves false dependencies between instructions, keeping only the true dependencies. The Front-End also incorporates a branch predictor and a recovery mechanism. These components enable the effective execution of speculative instructions, allowing the processor to make educated guesses about the direction of branches and recover accurately if the speculation proves incorrect.

The Back-End issues instructions out-of-order from each instruction queue to the read register stage, where sources are satisfied from the physical register file or the bypass network to later arrive at the execution stage on its respective functional unit. Once instructions are complete, they deliver data to the writeback bus for back-to-back execution and register file writeback. Later, the reorder buffer will commit instructions in program order as they are completed. [16]

Out-of-Order core

Out-of-Order execution is a technique designed to exploit instruction-level parallelism, minimize stalls, and enhance overall processor performance by overcoming the limitations imposed by data dependencies inherent in classical In-Order execution. In the In-Order execution model, when an instruction requires data from memory, all subsequent instructions are stalled until the necessary data is retrieved and ready for use. This stalling occurs even if the subsequent instructions do not depend on the data being fetched, leading to inefficiencies and reduced performance. Out-of-Order execution addresses this inefficiency by allowing instructions to execute as soon as their operands are available, irrespective of their original sequence in the program. This approach allows instructions that are independent of the stalled instruction to proceed, thus increasing overall throughput. The execution order, therefore, is determined by the readiness of the required operands, rather than by the program's instruction order.

This technique is based on the Tomasulo algorithm. This algorithm employs several key structures, including reservation stations, a reorder buffer, and common data buses (CDB).

Reservation Stations

Reservation stations serve as a waiting area for instructions whose execution is temporarily delayed. These stations monitor the availability of the required operands, enabling the instruction to be issued as soon as all dependencies are resolved and the operands are ready.
In the reservation station the instruction is referred to with the value of its tag.

After an instruction completes the execution, its result is broadcasted with the CDB, any instruction or register that was waiting for that result will pick it up.

Registers Rename

When instructions are executed out of sequence relative to their original program order, three types of data hazards may arise:

- Read-After-Write (RAW) Hazard: this is a true dependency that occurs when an instruction attempts to read a value before a preceding instruction has written it. If not managed properly, this leads to incorrect data being read. To avoid this, instructions must be executed in their original program order.
- Write-After-Write (WAW) Hazard: it occurs when an instruction attempts to write a value before a prior instruction has completed its write. The result is that the final value stored may be incorrect. WAW hazards can be mitigated using register renaming, which ensures that each write operation targets the correct version of the register.
- Write-After-Read (WAR) Hazard: it arises when an instruction tries to write a value before a previous instruction has read it, potentially causing the previous instruction to read the wrong value. Like WAW hazards, WAR hazards are addressed using register renaming.

WAW and WAR hazards are not true dependencies because they are a result of the limited number of registers available in the register file. Tomasulo's algorithm avoids data hazards by using register renaming. Physical registers are mapped to logical registers so that different instructions can use the same logical register without conflict.
On the other hand, RAW hazards are solved by ensuring that instructions with true data dependencies wait in reservation stations until all their required operands are available.

Tomasulo's Algorithm

Figure 2.8 shows the basic structure of a Tomasulo-based processor. The instructions are sent from the instruction unit to the instruction queue, from which they are issued in order to the reservation stations. If a reservation station is available, the instruction and its renamed operands, source registers and tag, are inserted into the reservation station.
While the instruction waits in the reservation station, it checks the common data bus for the tag of its sources. If the tag is found, it gets the value for the sources and keeps it in the reservation station, solving one of the sources. When all the operands are available, the instruction can be issued, effectively executing the

instructions out of order.
When the instruction is ready, it gets dispatched to its functional unit. After the computation is completed, the result is broadcasted on the common data bus to solve the source value of the instructions at the reservation stations, the tag is also broadcasted to the register file [18].

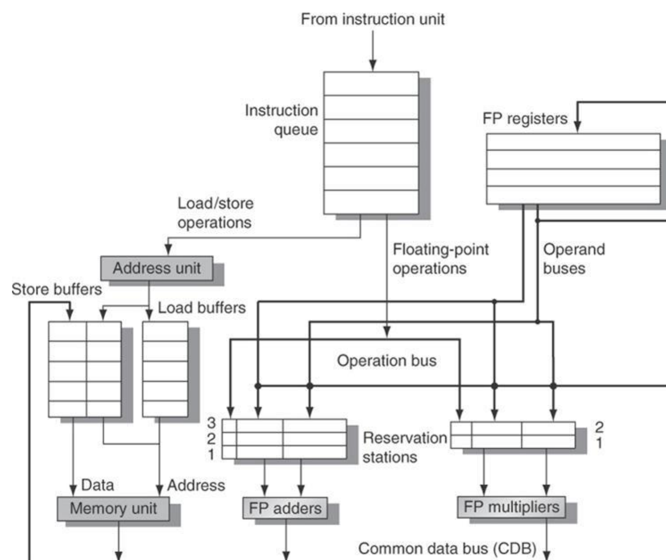


Figure 2.8: The basic structure of a MIPS floating-point unit using Tomasulo's algorithm from [18]

Reorder Buffer

The primary purpose of the ROB is to allow instructions to be executed out of order while ensuring that their results are committed to the architectural state of the processor in program order.
The Reorder Buffer monitors in-flight instructions as soon as they come out from the Dispatch State, going into the Back-End of the processor. Once the instruction meets the criteria for wake-up and is selected, it passes from the Issue Stage to the Execute Stage. At this point, when an instruction completes execution, its result is stored in a reservation station, which the ROB keeps track of, along with a tag (*rob id*) indicating its program order and a completion bit per entry. Once an instruction is ready to be committed, it is committed respecting the program order. The ROB assigns incrementing *rob ids* to instructions while they are still in program order. When an instruction reaches the head of the ROB and is ready to be committed, the actual commit process occurs. Since the instructions within the ROB are stored in the order they appear in the program, the commit process

naturally respects the original program order. This ensures that, even if instructions are executed out of order, they are committed in the correct sequence, maintaining the consistency and correctness of the program's execution.

The ROB is implemented as a circular buffer, the oldest instruction is pointed by the rob head, and the newest is pointed by the rob tail. On one side there is the ROB as just the memory, on the other the FIFO controller implements the control system for the circular buffer. The module receives read and write requests and replies by granting such requests and providing indexes for the operations to take place in the actual memory. The module evaluates the number of available and consumed slots based on the state of the write and read pointers, and grants the write and read requests. Slots need to be read before they can be rewritten, giving a limitation to how far the pointers can be from each other, and thus assuring a size limitation for the FIFO. The module has two status flags that indicate when the FIFO is full or empty, and two status signal that indicate the number of available slots and the number of consumed slots.

Precise Exception

Exceptions are handled when detected, contrary to interrupts that are handled when convenient. The architectural state should be consistent when the exception is ready to be handled. This means that all the previous instructions should be completely retired, and no later instruction should be retired. To implement the mechanism of precise exception in an out-of-order execution, the instructions are dispatched and executed out of order but the retirement should happen in order. Basically, reorder the instructions before making the results visible to the architectural state, using the Reorder Buffer.

Explicit Renaming

Tomasulo's algorithm is implemented to support out-of-order execution, but the implementation in the core is not precisely the one described by the algorithm. The reservation station, as they are described, are too complex, thus they are implemented differently while maintaining the same conceptual behavior. Lagarto Ox splits the reservation station job between the Reorder Buffer, the Rename Unit, the Issue Queues. The mechanism used is called Explicit Renaming.

Tomasulo's algorithm provides Implicit Register Renaming, where the registers are renamed to the reservation station tags. Explicit Renaming uses a physical register file that is larger than the number of registers specified by the ISA. What is needed is a translation table to map the ISA registers to the physical ones, and a mechanism to know which physical registers are free and can be assigned. It allows

to remove all WAW and WAR hazards, decouples renaming from scheduling, allows the implementation of out-of-order execution and makes precise exception easier since all that is needed to do to get the precise state of the execution is to undo the table mapping. As instructions are renamed, their register specifiers are explicitly updated to point to physical registers located in the Physical Register File [19].
The MIPS R10k, Alpha 21264, Intel Sandy Bridge, ARM Cortex A15 cores and the Berkeley Out-of-Order Machine (BOOM) are all example of explicit renaming out-of-order cores [20].

Recovery mechanism

Context recovery refers to the techniques used to handle and recover from instruction level errors, generally mis-speculations of control and data flow instructions that may occur during the execution among the out-of-order processor pipeline.
Speculative execution involves the processor speculatively executing instructions before it is known whether they are actually needed. If it turns out that the instructions are not needed (e.g. because the branch is mispredicted), their effects are discarded; this process will involve the context recovery of the processor up to a point where the state is consistent with respect to the required recovery point.

Checkpoints are snapshots of the processor's state (e.g. program counter, register file, etc.) taken at certain points in time, typically before a branch instruction. If a branch is mispredicted, the core can restore the state from the last correct checkpoint, effectively rolling back to the point before the misprediction occurred. Upon detecting the misprediction, usually after the branch instruction is resolved, the recovery mechanism:

- Flushes the pipeline: all instructions in the pipeline following the mispredicted branch are invalidated.
- Restores the checkpoint: the processor's state is rolled back to the checkpoint taken before the branch, and execution resumes from the correct path.

The ROB plays the vital role of identifying the precise exception point, meaning that the core can determine which instruction caused the exception by referring to the ROB.

Instruction Queues

The instruction queue in the Lagarto Ox core is a parametric out-of-order queue. The queue aims to solve the necessity of out-of-order instruction issue.
Instruction sources are stored in Content-Addressable Memories (CAMs), where the rest of instruction data is stored in Random-Access Memories (RAMs) in

the same index. Destinations are completed through the write-back bus and set dependent instructions as ready using the CAMs and ready source vectors. When an instruction has all its sources ready can be issued if there is a functional unit ready at execution stage to execute it.

The instruction queue stores instructions coming from dispatch. Here, the instructions wait until its operands are ready. Register destinations are completed through the write-back bus, which are compared to instruction register sources using CAMs, and set as ready the matching ones. When an instruction has all its sources ready, and there is a functional unit in the execution stage ready to compute it, the issue logic selects it and issues the data from the queue.

2.3 Methodology

In this section, the used of System Verilog is explained, as well as the use of the HLIB hardware library and its engineering cycle.

System Verilog

SystemVerilog is a hardware description and verification language that extends the capabilities of Verilog. SystemVerilog combines features from both HDLs and hardware verification languages (HVLs). This means that designers can design hardware and verify it within the same framework.
It is built on Verilog's syntax and semantics, adding new features to enhance the modeling of hardware designs. It introduces new data types, such as unpacked arrays, the definition of interfaces, which simplify the connection between different modules in a design. Moreover, SystemVerilog allows modules to be parameterized to create reusable and flexible hardware components.
For what concerns hardware verification, it supports assertions, used to check whether specific conditions hold true at particular points in time during the execution of a design, constained random verification, functional coverage and object-oriented programming.

HLIB

HLIB is a general-purpose open-source SystemVerilog hardware library, developed at the BSC. It provides a collection of commonly used, highly parameterized modules designed to minimize micro-architectural hardware implementation overhead

while improving code maintenance, modularity, and readability. Additionally, HLIB aims to simplify the verification process by defining a clearly defined engineering cycle and the use of the cocotb framework.

The objective is to cut RTL design development time by offering a comprehensive methodology, code guidelines, conventions, and tools, alongside a set of hardware modules that adhere to these standards.

The engineering cycle

To include a new module in the library, HLIB follows a specified engineering cycle shown in Figure 2.9. At the end of the cycle, the module will be fully documented in order to be easily accessed by everyone.

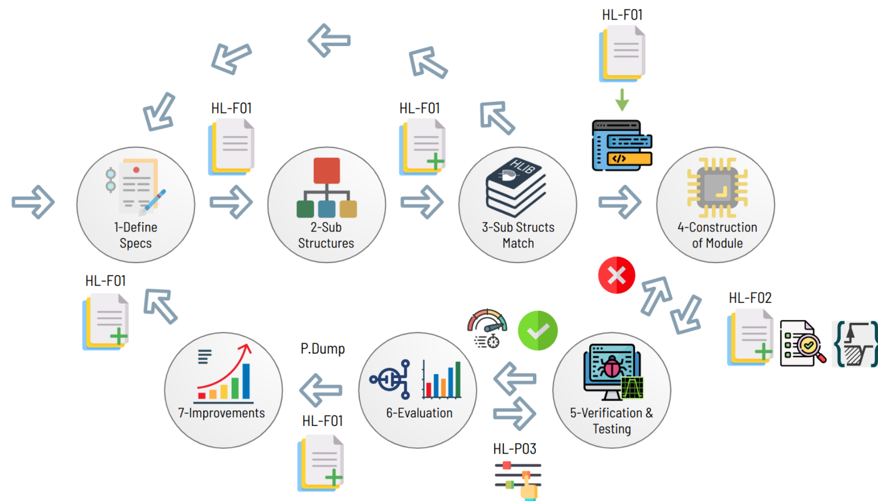


Figure 2.9: HLIB engineering cycle

The cycle starts with the definition of the specifications related to the new module. Then, all the submodules must be defined and their specifications must be documented. At this point, the actual RTL design can take place, putting in the comments the information collected in the previous steps.
Once the module implementation is finished, a proper test bench has to be implemented using the Library infrastructure, which relies on cocotb. The test bench is used to verify the specificated features and behavior of the module but can also help to estimate the performance. After the verification process is completed, that is, the module passes the test benches checks and standard linting test, the module

is ready to be synthesized with CI support. An industrial level tool-chain, such as Synopsys Design Compiler or Cadence Genus, is used to generate the synthesis, the resultant netlists will be used for performance, area, frequency, and scalability analysis.
The last point of the cycle is the evaluation of all the possible improvements that can be done in the current module after the evaluation of the synthesis results.

cocotb

During the development of an HLIB module, the verification process is performed through cocotb test benches. Cocotb is an open source COroutine-based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL. It is possible to write complex test benches using high-level code with python, cocotb will decouple the testing from the cycle-accurate simulator. Cocotb allows users to interface directly with the toplevel module as the Design Under Test (DUT) [21].

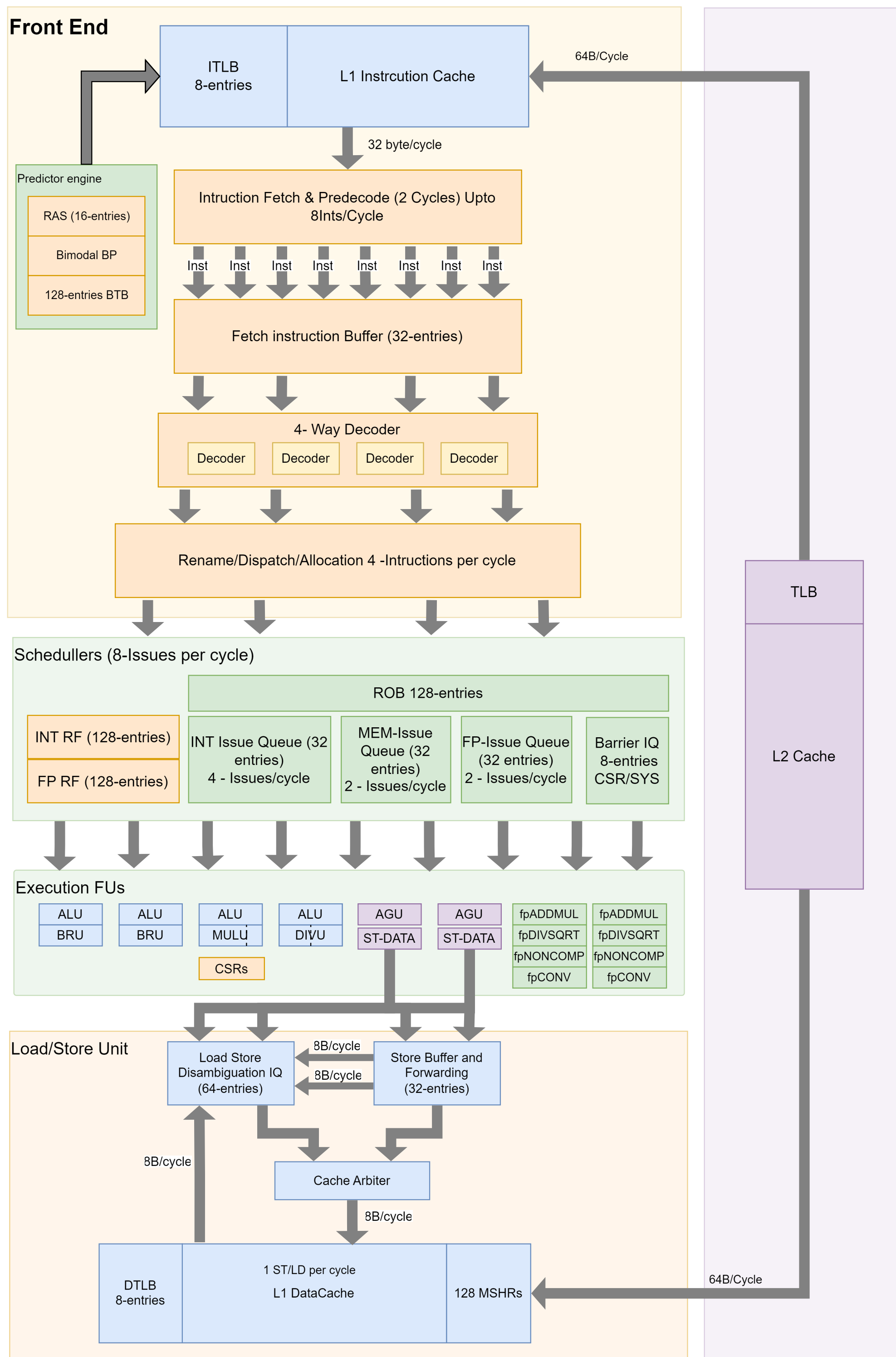


Figure 2.7: Lagarto Ox core diagram

Chapter 3

Design and Implementation

This chapter details the design and implementation of the RTL code for introducing the support of instruction fusion in the Lagarto Ox core.
The following sections will cover the design considerations, methodology, implementation of the RTL code, and how the code was verified and validated.

3.1 Instruction fusion

The subject of this work concerns the RTL implementation of instruction fusion in the RISC-V Out-of-Order Lagarto Ox core.
The fusion technique implemented is static time fusion, based on the work proposed in the 2016 Berkley article [10]. The instructions are considered fusible if they are adjacent in the code and in the same fetch window. The resulting fused instruction is encoded on 4 bytes, the size of an uncompressed instruction. At the moment, the fusible candidates are made of only compressed instructions, however, the design choices take into account the possibility of supporting the uncompressed instructions in future implementations.

To be fused, the instructions have to show some dependencies, in particular for all the three fusion candidates implemented the destination register has to be the same. If a sequence of instructions presents the same destination register, they can be interpreted as partial computations that contribute to the same final result, so it makes sense to group them into a single computation, consequently saving all the work of storing the intermediate results and the propagation of the instructions in the pipeline. This is the case for the RISC-V simple ISA that needs step-by-step computations to carry out a more complex operation.

3.1.1 Fusion candidates

The choice of which instructions fusion to support is based on the considerations made in the Berkley article [10]. The Indexed Load idiom is an interesting addition to the RISC-V architecture that can require up to three instructions compared to one instruction of x86, ARM and MIPS. The Indexed Load affects the calculation of the effective address of a load instruction, consequently it makes sense to give support for the fusion of the Load Effective Address idiom and the Scaled Load idiom, which is the union of the first two.

Load Effective Address (LEA)

The LEA idiom calculates the effective address of a memory location and puts the address in a register [10].

slli rd, rs1, imm
add rd, rd, rs2

The resulting fused instruction has two source registers, one immediate and one destination register.

f_lea rd, rs1, rs2, imm

In order to be fusible the two instructions have to satisfy the following requirements:

- the immediate has to be a value smaller than 32, thus it will be encoded on 5 bits, because that is the maximum value of shifting position that can take place
- the destination registers of the two instructions have to coincide
- the first source register and the destination register of the add instruction have to coincide

The implemented instruction fusion supports only compressed instructions, and all possible combinations of *c.slli* and *c.add, c.addw* are supported. However, the format defined for these instructions has also been made to support uncompressed versions of these instructions, so that they can be easily integrated.

Indexed Load (IL)

The IL idiom loads data from an address calculated by adding two registers.

add rd, rs1, rs2
ld rd, 0(rd)

The resulting fused instruction has two source registers and one destination register.

f_il rd, rs1, rs2

The constraints needed to be satisfied for the instructions to be fusible are:

- the immediate value of the load has to be zero since the calculation of the offset is executed by the add instruction
- the destination register of the two instructions has to coincide
- the source and destination registers of the load instruction have to coincide

All possible combinations of *c.add, c.addw* and *c.ld, c.lw* are supported. Again, the format of the generated fused instructions is made so that it can also support the fusion of not compressed instruction since they will have two different source registers and one destination, whereas with compressed there is only one source register, the destination register acts like a source.

Scaled Load (SL)

The SL idiom loads data from an address calculated by adding one register to a shifted register value.

slli rd, rs1, imm
add rd, rd, rs2
ld rd, 0(rd)

The resulting fused instruction has two source registers, one immediate and one destination register.

f_sl rd, rs1, rs2, imm

In order to be fusible, the three instructions have to satisfy the following requirements:

- the immediate has to be a value smaller than 32
- the destination registers of the three instructions have to coincide

- the first source register and the destination register of the add instruction have to coincide
- the source and destination registers of the load instruction have to coincide

All the possible combinations of *c.slli, c.add, c.addw* and *c.ld, c.lw* are supported. Again, the format is designed to accommodate also the future support of uncompressed instructions.
In summary, the core supports the fusion of the instruction sequences found in Table 3.1. The table also shows the constraints regarding the registers.

F_LEA	F_IL	F_SL
c.slli rd, rd, imm(5) c.add rd, rd, rs2	c.add rd, rd, rs2 c.ld rd, 0(rd)	c.slli rd, rd, imm(5) c.add rd, rd, rs2 c.ld rd, 0(rd)
c.slli rd, rd, imm(5) c.addw rd, rd, rs2	c.add rd, rd, rs2 c.lw rd, 0(rd)	c.slli rd, rd, imm(5) c.addw rd, rd, rs2 c.ld rd, 0(rd)
	c.addw rd, rd, rs2 c.ld rd, 0(rd)	c.slli rd, rd, imm(5) c.add rd, rd, rs2 c.lw rd, 0(rd)
	c.addw rd, rd, rs2 c.lw rd, 0(rd)	c.slli rd, rd, imm(5) c.addw rd, rd, rs2 c.lw rd, 0(rd)

Figure 3.1: Supported fusion sequences

The instruction fusion possibilities made of the three idioms, with the support for only compressed instructions, end up being a total of 10 fusible sequences.
The support can be easily extended to the uncompressed instructions since the format accommodates them as well.
The three idioms implemented for the fusion are grouped as the **Scaled Index** family of fusion instructions. This is done to group them since they conceptually have the same function and to differentiate them from other idioms that can supported in the future, with the aim of being capable of enabling and disabling a particular family of fusion support based on the necessity.

3.1.2 Fused instruction format

The three idioms composing the Scaled Index family have been encoded in the same format. The instructions need, at most, two source registers, one immediate of 5 bits and one destination register.

A new format called F_R4 has been defined, it has been designed starting from the standard R4 format. R4-type instructions specify three source registers (rs1, rs2 and rs3) and a destination register (rd). This format is only used by the floating-point fused multiply-add instructions [22].

The abbreviations used in the table of instruction formats are:

- opcode: 7-bits operation code.
- func3: unsigned immediate for 3-bits function code.
- func2: unsigned immediate for 2-bits function code.
- rd: destination register number for operand x.
- rs1: first source register number for operand x.
- rs2: second source register number for operand x
- imm(5): unsigned 5-bit immediate for operand x.

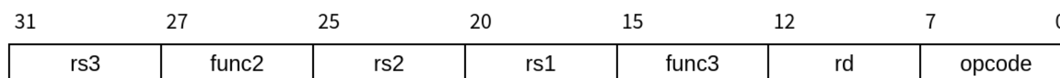


Figure 3.2: R4 format

The encoding definition described lies outside the standard specification of the RISC-V ISA, but by adding an extra bit to the inter-stage buffer between the fetch and decode stages, the full 32-bit encoding space can be utilized without adhering to a specific instruction format. This approach offers significant flexibility in the design, enabling the support of an extended range of fusible idioms and facilitating more versatile internal decoding processes. While it is technically feasible to exceed the 32-bit instruction size to encode additional information, this would introduce increased complexity to the processor's buses, affecting both compressed and non-compressed cases. Such complexity can potentially undermine the simplicity and efficiency of a RISC ISA, consequently affecting the energy/performance ratio. However, it is worth mentioning that this design choice could be appropriate

for certain domain-specific architectures, which are not the focus of the current processor design.

This new format tailored for fused instruction is based on an already existing one because it is convenient to be able to reuse the decoder already present in the architecture. By respecting the fields of the opcode, func2, and func3 it is possible to decode the fused instruction without many modifications. The rest of the fields are for the registers and immediate values and can be modified without any problem.

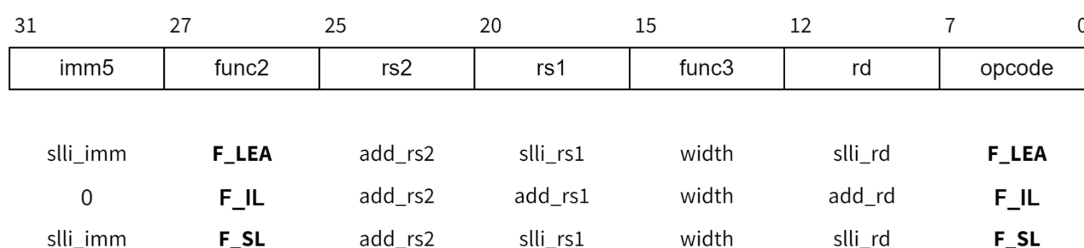
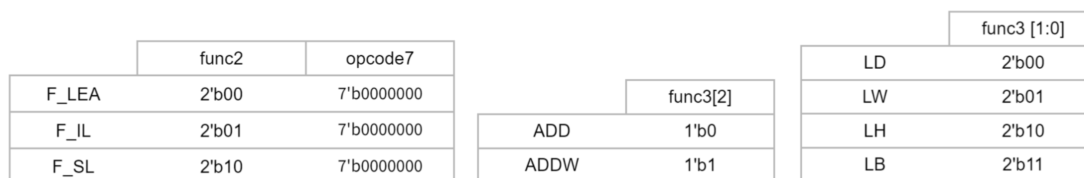


Figure 3.3: F_R4 format

The F_R4 format, reported in Figure 3.3 has the same structure as the R4 format but the rs3 field is substituted by the immediate value on 5 bits.
The opcode is common for the entire family and it is 7'b0. To differentiate the three idioms within the family, the field func2 is used. The field func3 encodes the width of the instructions that make up the original sequence. The encoding is shown in Figure 3.4.
This format is general for all combinations of compressed and uncompressed instructions that are part of the Scaled Index family. The only limitation is that only double word shift left logic is considered.



	func2	opcode7
F_LEA	2'b00	7'b0000000
F_IL	2'b01	7'b0000000
F_SL	2'b10	7'b0000000

	func3[2]
ADD	1'b0
ADDW	1'b1

	func3 [1:0]
LD	2'b00
LW	2'b01
LH	2'b10
LB	2'b11

Figure 3.4: F_R4 fields

It is interesting to note that the instructions *addw, ld, lw* in compressed format are encoded with the version of the 3-bit registers, so the source and destination registers can only be s0, s1, a0-a5 [22], as shown in Figure 3.5.

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

Figure 3.5: Registers specified by the 3-bit encoding, taken from [22]

3.2 Fusion support in the Front-End

The Front-End of the core is made of Fetch, Decode, Rename, Dispatch and Allocation stages. To support instruction fusion, some modifications in the Fetch and Decode stage are needed. The fusible sequences are detected in the last stage of Fetch, contrary to other designs seen before that place the detection in the Decode stage. It is a design choice: the added complexity is moved from the Decode stage to Fetch, this allows to save entries in the queue between Fetch and Decode.

3.2.1 Fetch stage: Fusion Detector

The Fetch Unit in the Lagarto Ox processor supplies a constant stream of instructions to the pipeline by requesting a new set of instructions from the Instruction Cache every cycle. Each instruction can be a 16 or 32-bit boundary due to the "C" extension support. The Fetch Unit can provide up to 8 16-bit instructions per cycle or any combination of instructions that fit the 128-bit read bus bandwidth of the full associative instruction buffer.

In the Fetch stage, the fusible sequences defined in Figure 3.1 are identified and encoded into a single fused instruction. The detection and encoding takes place in the F2 stage of fetch, where the Fusion Detector module identifies and processes the fusible sequences. The F2 stage is the last stage of Fetch. By detecting the fusion opportunities here, it is possible to reduce the fetch-decode inter-stage queue entries, because there will only be one instruction and not the original sequence. This optimization significantly benefits the queue, especially when detecting sequences of uncompressed instructions will be introduced: a sequence of three instructions of 4-byte each (Scaled Load idiom) will become a single 4-byte instruction, saving 4 entries in the queue (each queue entry is 2 bytes, thus a compressed instruction takes one entry and an uncompressed instruction two).

The supported instruction fusion process combines two or three compressed 2-byte instructions into a single 4-byte instruction. When two instructions are fused, there is no reduction in the number of entries in the fetch-decode queue. This

is illustrated in Figure 3.10, where four sequences of two instructions are fused without any reduction in queue entries. However, when three instructions are fused, one entry in the queue is saved, as depicted in Figure 3.9. This effect is limited to the fetch-decode inter-stage queue; in subsequent pipeline stages, such as the Decode, Rename stage, Reorder Buffer, and Instruction Queues, one or two entries are saved due to the fusion in every structure, depending on whether two or three instructions were fused.
The saved entry is marked as invalid, as illustrated in Figure 3.9. The writing mechanism of the queue typically stops at the first invalid instruction, so if an empty entry is gained through fusion, writing stops, and the following valid entries must wait to enter the queue until the next cycle. However, the fetch queue supports an unordered write mechanism that enables the writing of all valid instructions without being halted by an invalid one. This feature makes it possible to increase efficiency by taking advantage of the empty slots obtained with the fusion.

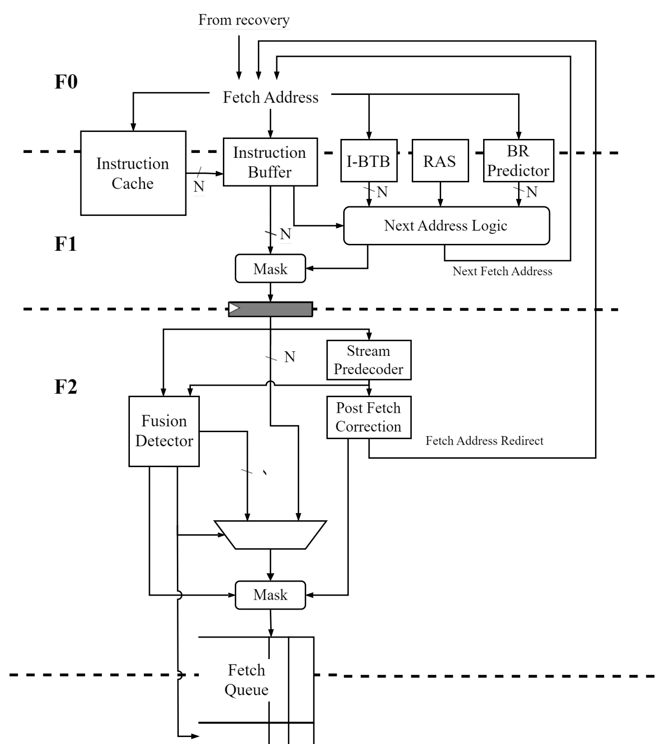


Figure 3.6: Architecture of the Fetch stage

In the F2 stage, the predecoding of instructions is carried out to support the "C" extension. Prior to this module, the data is propagated as 8 entries of 16 bits each, which can represent any combination of 16-bit or 32-bit instructions. The key role of the predecoder is to discern compressed instructions.

The predecoder, combined with the post-fetch correction module, is also used to identify and correct jump instructions.
The write enable signals for the fetch queue are generated by combining the mask of incoming valid signals with the corrections generated during the F2 stage. The detection of the fusion opportunities takes place in this stage, parallel to the post fetch correction, with the Fusion Detector module, as shown in Figure 3.6.

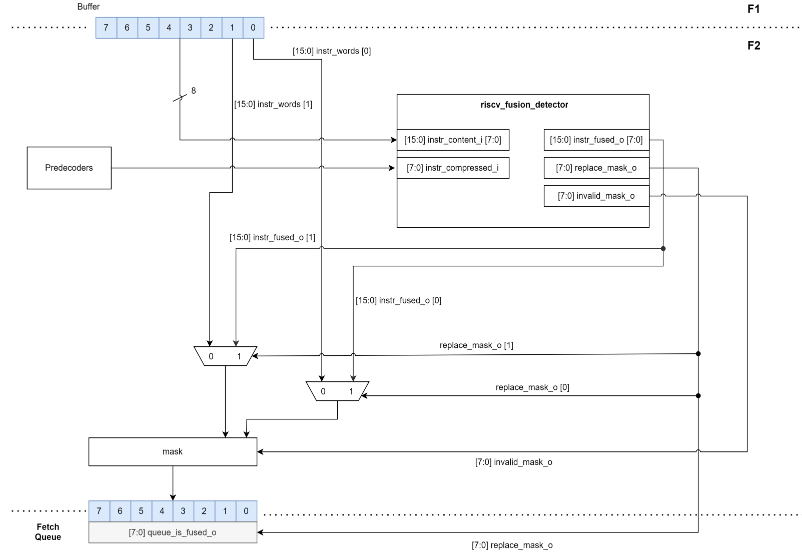


Figure 3.7: Focus on F2 stage

The Fusion Detector module generates fused instructions when a fusible sequence is detected. Consequently, there are two instruction flows: the normal flow and the fused instructions. For each entry in the Fetch Queue, the correct flow must be accurately selected. To achieve this, the Fusion Detector provides a mask, called *replace_mask*, that drives the multiplexer responsible for selecting the appropriate entry. This mask indicates the positions of valid fused instructions and is also used to indicate the fused instruction in the Fetch Queue, effectively adding an extra bit.
When three compressed instructions are fused into a single 32-bit instruction, one entry in the queue becomes vacant. This vacant entry must be invalidated using an additional mask generated by the module, called *invalid_mask*, which is integrated with the existing masks that generate the write enables of the inter-stage queue. As previously analyzed, considering the possibility of unordered writes, this invalid entry will be filled with the next valid instruction, if one is available. This mechanism is shown in Figure 3.7.

Fusion detector

The Fusion Detector is a combinational module that takes as input 8 entries of 16 bits each, corresponding to the fetch bandwidth. Each entry can either be a compressed instruction or part of a normal-sized instruction, which consists of two consecutive entries. This information is provided by *instr_compressed_i* input vector.
The Fusion Detector module instantiates 7 *c_detector* modules.

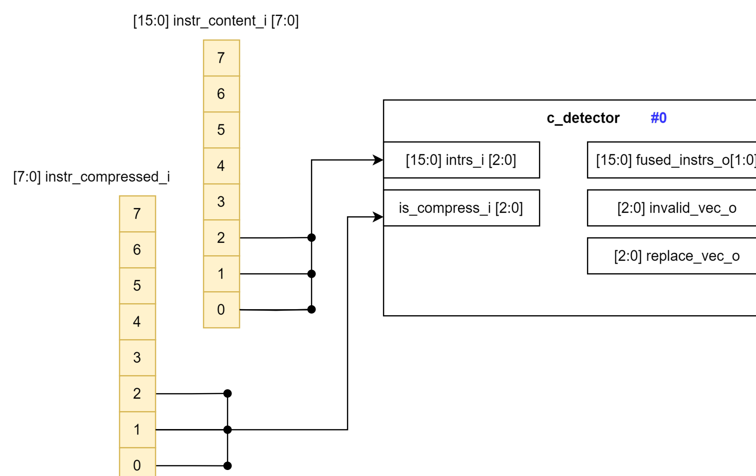


Figure 3.8: Focus on signals connection inside Fusion Detector: how the input signals are connected with instantiation #0 of c_detector

The detection occurs within this module. Each *c_detector* takes three consecutive entries as input, as the longest fusible sequence comprises three compressed instructions, and verifies if they match one of the fusible sequences supported.
In Figure 3.8, the input signal connections for the initial instantiation of the *c_detector* module are illustrated. For each subsequent instantiation, the input signals are taken by shifting one position in the input vector. Specifically, the first instance (#0) uses input data from positions 0 to 2, the second (#1) uses input data from positions 1 to 3 and this pattern continues incrementally for each successive instantiation. The final *c_detector* receives two valid entries as input, with the last input set to 0 since there are no entries left. This configuration implies that no sequence of three instructions can be detected, but sequences of two instructions are still detected normally.
With seven instances of *c_detector*, every possible starting point of a fusible sequence is examined, as shown in Figure 3.9, where two Scaled Load and one Indexed Load are detected. Another example is in Figure 3.10, where 4 fusible sequences of two instructions are detected. These two cases are examples of the maximum

throughput achievable, with three instructions in the output in the first case and four instructions in the second, instead of the original eight.

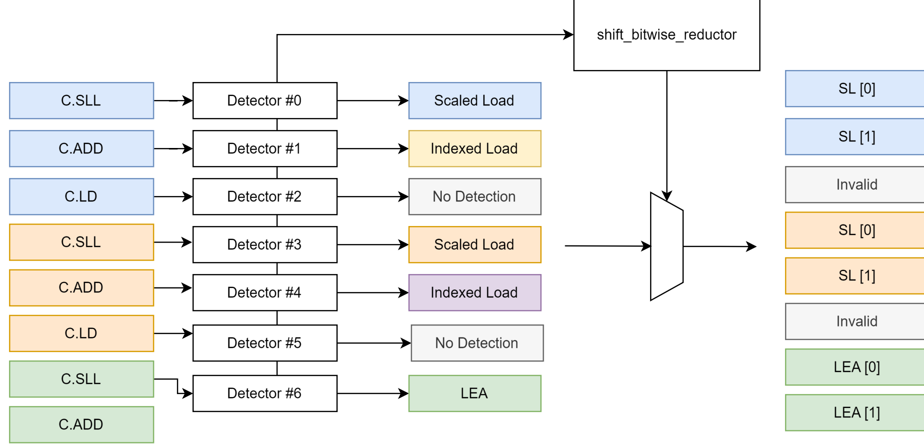


Figure 3.9: Detection of two Scaled Load sequences and one Load Effective Address

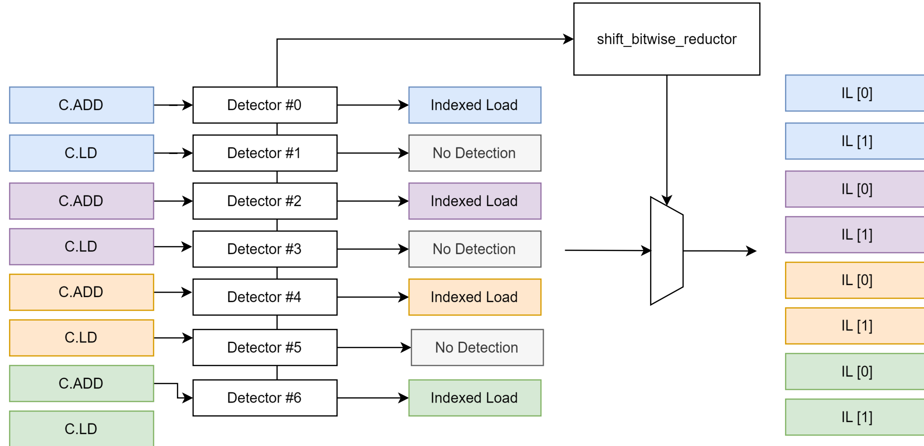


Figure 3.10: Detection of four fusible sequences made of two instructions each

Each *c_detector* generates a replacement mask, an invalidation mask, and a fused instruction of 32 bits, which occupies two 16-bit slots, as shown in Figure 3.11.

To generate the final masks, two instantiations of the *shift_bitwise_reductor* submodule are used: one for generating *replace_mask_o* and one for generating *invalid_mask_o*.
The submodule is in charge of abstracting the generation of reductions in control

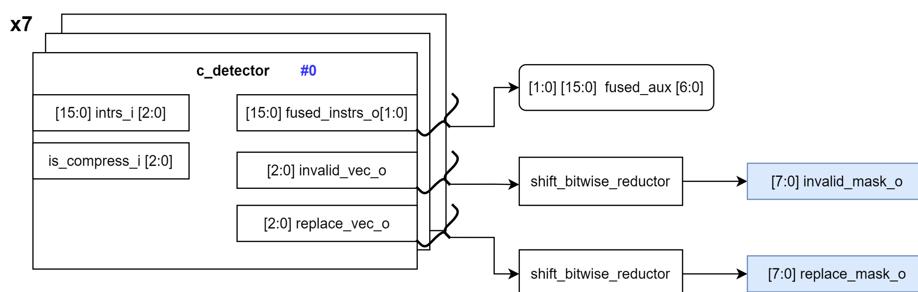


Figure 3.11: Focus on output replace and invalid mask of Fusion Detector module

logic when partial results are obtained in batch processing logic styles and need to be reduced and shifted with a specific amount to correspond to the elements processed in each batch. In batch processing logic styles, partial results from multiple operations need to be reduced and shifted appropriately to ensure they correspond accurately to the processed elements in each batch.

[15:0] instr_content_i [7:0]		from c_detector: invalid_vec	replace_vec	Fusion
0	C.SLL	100	111	**SL**
1	C.ADD	000	011	**IL**
2	C.LD	000	000	No
3	C.SLL	100	111	**SL**
4	C.ADD	000	011	**IL**
5	C.LD	000	000	No
6	C.SLL	000	011	**LEA**
7	C.ADD			

invalid_mask_o	0010_0100
replace_mask_o	1111_1111

Figure 3.12: Example of input sequence and generation of the replace and invalid masks

To generate the two masks an OR bitwise reduction is used. Every *c_detector* generates the masks, but at the end only one mask is needed. As an example, Figure 3.12, shows on the left side the masks generated by every c_detector, based on the fusible instruction found, and on the right the final expected masks that will be the output of the Fusion Detector module.
Figure 3.13 shows how the 7 masks are positioned and why an OR reduction is needed: every time there is a 1 in the batch, there should be a 1 in the final mask.

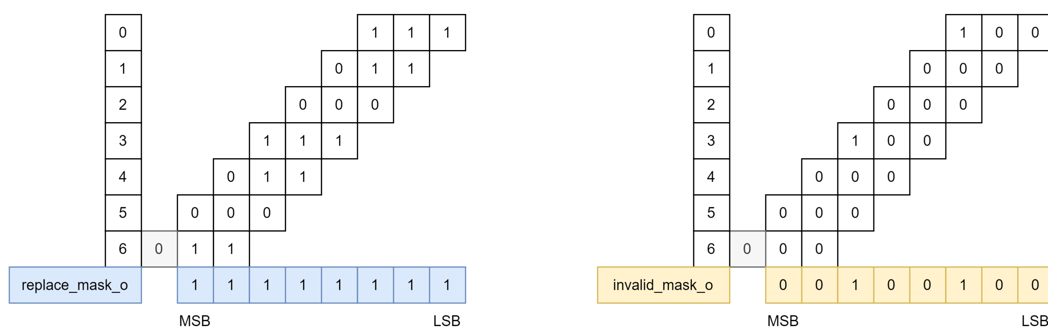


Figure 3.13: Reduction from the 7 masks to to final ones relative to Figure 3.12

Finally, Figure 3.14 shows how the final instruction is selected. To select the appropriate 'fused slots' from the *c_detector*, six multiplexers are used. The first and last slots of the fused instruction output are always fixed, while each of the remaining slots will either be the second half of the previous fused instruction or the first half of the current fused instruction. The multiplexers used to perform this selection are controlled by the signal *sel_mux*, which is set to 1 when *replace_mask_o[i]* is greater than *replace_mask_o[i-1]*, as summarize in Table 3.1. Those are the only possible combinations of signals, each replacement mask coming from the c_detector tells if there are no matches, 2 instructions fused or 3 instructions fused.

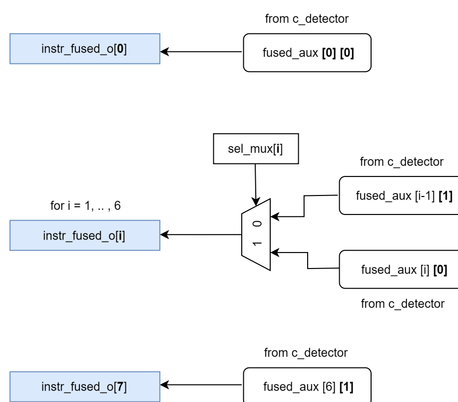


Figure 3.14: Selection of the fused instruction output

The Fusion Detector has been designed as a scalable module to accommodate the detection of additional fusion sequences. This scalability can be achieved through two primary approaches. The first approach involves increasing the complexity of the *c_detector* module. The second approach, illustrated in Figure 3.15, involves allocating multiple detectors operating in parallel. In this parallel configuration, the same input instruction sequence is processed by two sets of detectors, each

replace_mask_o[i]	replace_mask_o[i-1]	sel_mux[i]
000	000	0
000	011	0
011	000	1
011	111	0
111	000	1

Table 3.1: Driving signal of the multiplexer to select the fused instruction output

tasked with recognizing different families of fusible instructions. The outcomes from these detectors must then be appropriately selected. In both cases, the bitwise reduction mechanism together with the replace mask generation are employed to select the wanted outcome, giving the opportunity to prioritise the outcome of certain detectors over others and resulting in a hierarchy among fusion families, where certain fusion sequences are given precedence.
Moreover, ambiguity in detection is avoided, as fusible instruction sequences are identified beginning with their first instruction. However, overlapping can occur, an example is the case of Scaled Load and Indexed Load sequences: when a detector recognises the Scaled Load sequence the following one will always detect a Indexed Load sequence. This overlap is resolved using the above mentioned mechanism, where priority is given to the Scaled Load sequence over the Indexed Load sequence, as the former fuses three instructions rather than just two.

C detector

The *c_detector* is a combinational module, designed to detect the 10 fusible candidates in Figure 3.1, from a sequence of three compressed instructions. Upon identifying a match, it generates the fused instruction, the replacement mask, and the invalidation mask.
The inputs to the module consist of three 16-bit instructions and a 3-bit vector indicating whether each instruction is compressed. The outputs include a 32-bit fused instruction, which is generated as two 16-bit instructions, a 3-bit replacement mask indicating which input instructions were part of the fusion, and a 3-bit invalidation mask indicating which input instructions are no longer valid due to their inclusion in the fusion process, leaving them outside the two halfword slots. The invalidation mask is essential because when a sequence of three instructions is fused, the resulting fused instruction occupies two slots. Consequently, the slot previously occupied by the last instruction of the sequence becomes vacant and must be invalidated. The two masks cannot take any value, the only possible cases are summarized in Table 3.2.

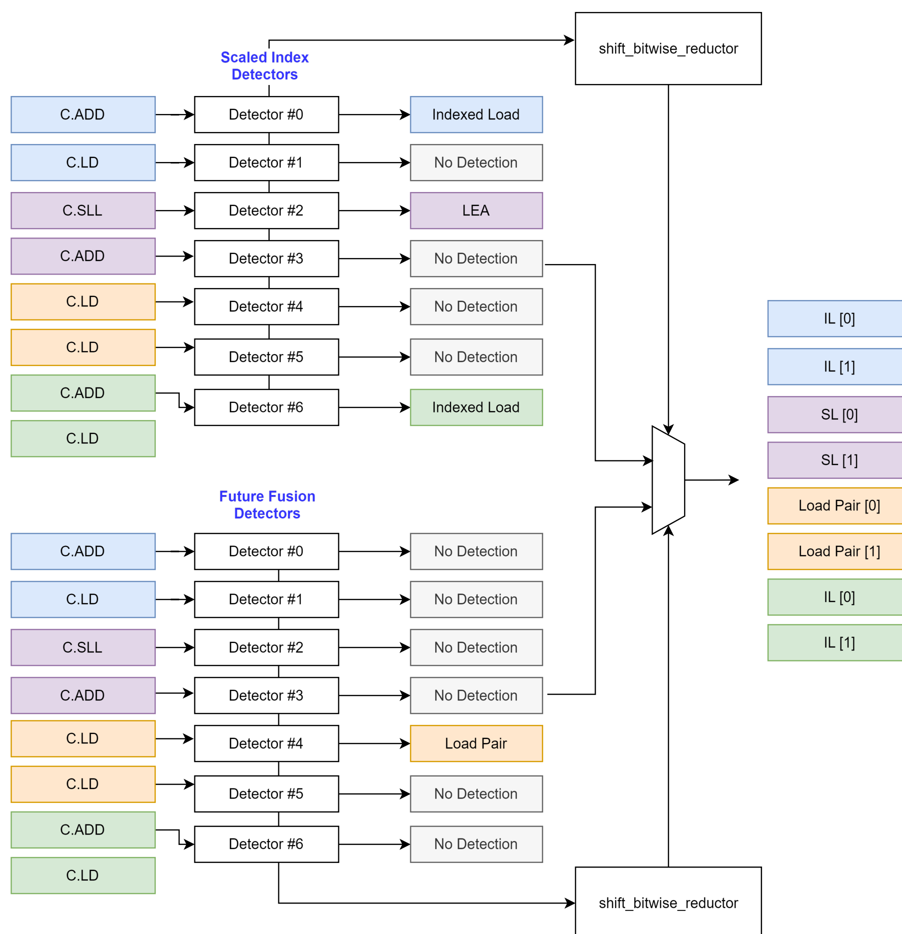


Figure 3.15: Parallel detection

	replace_mask_o	invalid_mask_o
No matches	000	000
2 instructions fusion	011	000
3 instructions fusion	111	100

Table 3.2: Replace mask and Invalid mask generation

Functionally, the module behaves as a decoder. It searches for a matching pattern and, rather than generating control signals, encodes the fused instruction and the associated masks. The detector's structure employs *if statement* to verify whether an instruction is compressed and *case statement* to identify the specific instructions

of interest.

Upon identifying an instruction sequence that matches one of the supported fusible sequences, an additional *if statement* ensures that the registers of these instructions are coherent with the fusion requirements, explained in Subsection 3.1.1 and summarized in Table 3.3.

F_LEA	F_IL	F_SL	
add rd = ld rd	add rd = add rs1	add rd = add rs1	slli rd = ld rs1
ld rs1 = ld rd	slli rd = add rs1	slli rd = add rs1	ld offset = 0
ld offset = 0	slli imm5 = 0	slli rd = ld rd	slli imm5 = 0

Table 3.3: Register constraints

If the register coherence check is successful, the fused instruction is generated along with the two masks, as previously described. The generated instruction is encoded according to the format outlined in Figure 3.3 and 3.4.
In the event that the register check fails or no fusible instructions are detected, all outputs are set to zero.
The *c_detector* is implemented so that if a Load Effective Address idiom is detected, it will check also the next instruction to see if it is a Scaled Load, and if a match is found the output is the Scaled Load idiom.

3.2.2 Fetch stage: testbench

Fusion Detector and C detector have been tested with ad hoc testbenches inside the cocotb framework.
Initially, the behavior of the C detector was evaluated by providing the ten supported fusion candidates as input. It was verified that these candidates were correctly detected and that the generated masks and output instructions conformed to the specifications outlined in Table 3.2 and the defined format in Figure 3.3 and 3.4, by comparing the output of the DUT to the expected output using assertions, as reported in Figure 3.16.
Subsequent tests were performed to ensure that incorrect sequences were not detected erroneously. Random sequences of instructions were tested, resulting in no fusion detections. Additionally, sequences that matched fusible candidates but did not meet the register requirements were also tested, confirming that these were correctly identified as non-fusible.

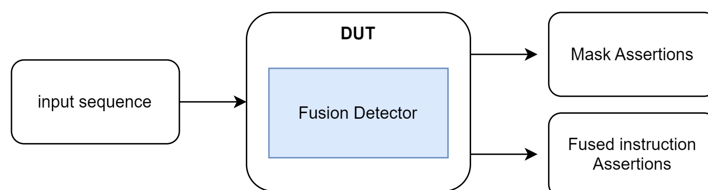


Figure 3.16: Test bench flow of the Fusion Decoder

Then, the Fusion Detector was tested. The previous test already verified the correctness of the detection, now the focus is to verify that the seven outputs of the C detectors are correctly selected and generate the output. Four tests were carried out to cover different scenarios, with the same flow used for the *c_detector* and present in Figure 3.16:

- The example reported in Figure 3.12, where a SL is detected in position 0 and 3, and a LEA in position 6
- The sequence of 4 repetition of *c.add, c.ld*, as in Figure 3.10 to verify that 4 IL are detected and correctly generated
- The sequence of 2 repetition of *c.slli, c.add, c.ld*, and *c.add, c.ld*, to verify that 2 SL and one IL are detected
- The sequence presented before with the second repetition of the SL idiom with the wrong registers, to verify that one SL and one IL are detected, while the second SL should not be detected because of the registers requirements.

After verifying the correct operation of the Fusion Detector module, it was integrated into the Fetch stage. The signals were connected as illustrated in Figure 3.7, with the additional bit *is_fused*, which indicates that the entry is a fused instruction, included in the signals transmitted from the Fetch stage, through the inter-stage queue, to the Decode stage. Furthermore, unordered writes in the inter-stage queue were enabled.
To ensure that this integration did not disrupt the normal functionality of the Fetch stage, the fetch test bench was executed and verified to work correctly.

3.2.3 Decode stage: Fusion Decoder

In the decode stage, the 4-wide instruction decoder is instantiated and capable of handling any combination of 16-bit and 32-bit instructions. It generates control information for all the different parts of the pipeline. Additionally, it detects invalid instructions that will be reported to the exception collector.
The fused instructions are 32 bits long and managed as an extension of the processor's executable instructions. However, this management is purely internal, as the fused instructions do not constitute an extension of the Instruction Set Architecture.

In the decode stage, a new decoder specialized for fused instructions operates in parallel with the standard decoder. This Fusion Decoder mirrors the structure of the original decoder, with the generation of immediate values and register information tailored to the requirements of the fused instructions.

To determine the appropriate output between the normal decoder and the fused decoder, a multiplexer is utilized, which is controlled by the *is_fused* flag, the information coming from the fetch stage, shown in Figure 3.17.

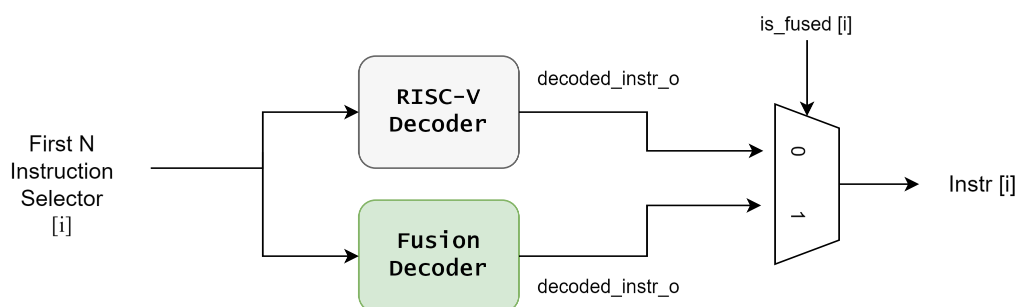


Figure 3.17: Decode stage

Two scenarios can arise during the decoding process. In the case of a normal instruction, the RISC-V decoder will correctly decode and process the instruction, while the fused decoder will output an 'illegal' instruction. Reversely, when the instruction is a fused instruction, the fused decoder will produce the correct decoded output, and the RISC-V decoder will generate an 'illegal' instruction.

Definition of the Control signals

The Load Effective Address instruction is executed within the integer execution unit because all original instructions are processed there. To support this operation, it is necessary to enhance the Arithmetic Logic Unit to perform two calculations

simultaneously (shift and load). The chosen approach involves incorporating a new ALU specialized for arithmetic fusion operations. Thus, this operation is issued to the Integer Queue and to the functional unit FUSION_ALU_UNIT.

Indexed Load and Scaled Load instructions are executed within the memory execution unit because they both involve a load operation derived from the original fused instructions. The memory execution unit already has the infrastructure necessary to manage load operations. Therefore, the modification required is to enhance the complexity of calculating the effective address. Consequently, both instructions are issued to the Memory Queue: IL is executed in the usual memory functional unit, while SL is executed inside a specialized functional unit MEM_SHIFT_UNIT.

instruction	valid	use_src	rf_src	use_dst	rf_dst	queue_id	use_imm	imm_ext_typ	use_pc	func_unit	instr_op	unsigned_op	width	illegal_instr
F_LEA_D	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	INTEGER_QUEUE	1	IMM_19_0		FUSION_ALU_UNIT	F_LEA_OP		_DD_	
F_LEA_W	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	INTEGER_QUEUE	1	IMM_19_0		FUSION_ALU_UNIT	F_LEA_OP		_DW_	
F_IL_DD	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	0	IMM_19_0		MEM_UNIT	F_IL_OP		_DD_	
F_IL_DW	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	0	IMM_19_0		MEM_UNIT	F_IL_OP		_DW_	
F_IL_WD	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	0	IMM_19_0		MEM_UNIT	F_IL_OP		_WD_	
F_IL_WW	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	0	IMM_19_0		MEM_UNIT	F_IL_OP		_WW_	
F_SL_DD	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_SHIFT_UNIT	F_SL_OP		_DD_	
F_SL_WD	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_SHIFT_UNIT	F_SL_OP		_WD_	
F_SL_DW	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_SHIFT_UNIT	F_SL_OP		_DW_	
F_SL_WW	1	11	INTEGER_RF-INTEGER_RF	1	INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_SHIFT_UNIT	F_SL_OP		_WW_	

Figure 3.18: Scaled Index family control signals

The control signal information is present in the spreadsheet in Figure 3.18. From this spreadsheet the package containing the control signals is automatically generated with a Python script. Each instruction has a unique structure that contains its control signals in a human-readable way.
The control signals produced upon the detection of the *slli + addw* sequence are shown in Figure 3.19, and it is possible to see the correspondence with the data defined in the spreadsheet. There is also an additional signal, enclosed within an *ifdef* statement, that is used only in simulation for debugging purposes.

3.2.4 Decode stage: testbench

To test the fusion decoder, all 10 fused instructions were provided as input. The testbench verifies that the debug signal output from the decoder matches the expected value of the debug signal. This debug value is part of the control signals generated for every instruction, and it is unique, so it is sufficient to verify its correctness. This signal is used solely for simulation purposes to facilitate debugging, as illustrated in Figure 3.19.
The input instructions were generated manually according to the format specified

```
parameter instr_ctrl_t F_LEA_W_CTRL = '{
    `ifdef SIMULATION
    instruction: F_LEA_W_DBG,
    `endif
    valid: 'b1,
    use_src: 'b11,
    rf_src: {INTEGER_RF, INTEGER_RF},
    use_dst: 'b1,
    rf_dst: INTEGER_RF,
    queue_id: INTEGER_QUEUE,
    use_imm: 'b1,
    imm_ext_typ: IMM_19_0,
    func_unit: FUSION_ALU_UNIT,
    instr_op: F_LEA_OP,
    width: _DW_,
    default: '0
};
```

Figure 3.19: Example of control signals structure

in Figure 3.3 and validated before being used on the test bench. A Python script was used for this validation, and it turned out to be valuable during the debugging process as well.

The Python script, given a fused instruction in hexadecimal or binary format, generates all the information about the original instructions. Specifically, it identifies which fused instruction it is, the corresponding original instructions, the registers involved, and the immediate value. It is useful in ensuring the correctness and facilitating the debugging of the entire pipeline.

```
Opcode Scaled Index family
Instruction : F_LEA
Original instructions: add + ld
Immediate value :  3
rd :  15
rs1 :  15
rs2 :  13
```

Figure 3.20: Output of python script from fused instruction to original instructions.

3.2.5 Load pairs

After implementing support for the Scaled Index family, the objective was to extend this support to include Load Pairs as well. The reason behind this extension lies in the potential to enhance instruction fusion capabilities by incorporating a broader range of instruction sequences.

As discussed in Section 2.1, it is possible to replicate in a RISC-V architecture the load-pair instruction supported in ARMv8 by fusing consecutive load instructions that access contiguous memory addresses. This case is a static time fusion because it occurs during the decode stage, as it is not necessary to know the exact address of the load, it is sufficient that the pair of loads share the same base register and have immediate values that allow consecutive memory access [10]. Then, 128 bits can be accessed with single 4-byte instruction, thus a single memory access, by fusing two load word instructions, as shown in the sequence below.

Ld rd1, imm (rs1)
Ld rd2, imm + 8 (rs1)

Load Pair rd1, rd2, imm (rs1)

The reason to explore memory instruction fusion is supported by the 2022 article [5], which demonstates, reported in Figure 3.21, that memory pairing idioms are more common than other idioms, and furthermore, they also provide larger performance benefits as they not only reduce IQ/ROB pressure but also LQ and SQ pressure.

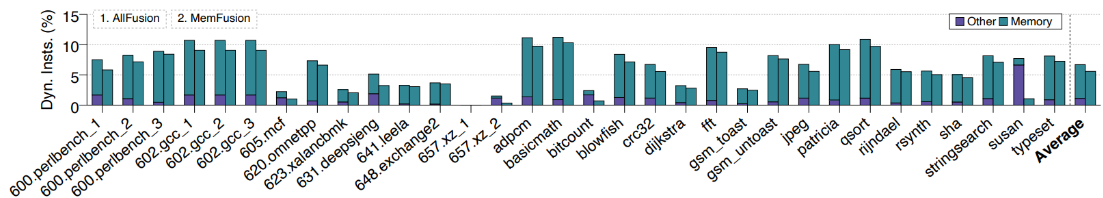


Figure 3.21: Percentage of fused micro-ops considering all or just memory fusion idioms, taken from paper [5]

However, only the Front-End implementation was developed, as supporting Load Pairs in the Back-End proved to be non-trivial and exceeded the scope and time constraints of this thesis. Unlike the Scaled Index family, Load Pair instructions require two different destination registers rather than a single destination. This requires handling multiple rename destinations and accommodating multiple write-back operations in the Load/Store Unit.

Although the support for this instruction idiom remains incomplete, the modifications introduced in the Front-End provide valuable insights. They demonstrate that the support implemented for the first family of fusion is indeed scalable, making the extension to this new fusion family relatively straightforward.
The Load Pairs fusion family comprises consecutive load instructions that target memory at contiguous positive locations. Consistent with the previously mentioned

design approach, only compressed instructions were considered, specifically the *ld* and *lw* instructions in all possible combinations. Two consecutive loads share the same source register, with appropriate offsets, but have two distinct destination registers. For fusion to occur, the load operations must access contiguous memory addresses. Consequently, if the first operation is a load double-word instruction, the offset must be eight, while if it is a load word instruction, the offset is four. This is summarized in Figure 3.22.

F_LP
c.ld rd1, imm(rs1) c.ld rd2, imm**+8**(rs1)
c.lw rd1, imm(rs1) c.lw rd2, imm**+4**(rs1)
c.ld rd1, imm(rs1) c.lw rd2, imm**+8**(rs1)
c.lw rd1, imm(rs1) c.ld rd2, imm**+4**(rs1)

Figure 3.22: Load Pairs fusion sequences

The format selected to support this fusion family is the custom-designed F_R4 format. In this configuration, the destination register field is assigned to the destination register of the first load instruction, while the second source field, given that there is only one source, will be interpreted as the second destination register. This format is illustrated in 3.23.

31 imm5	27 func2	25 rs2/**rd2**	20 rs1	15 func3	12 rd/**rd1**	7 opcode 0
slli_imm	**F_LEA**	add_rs2	slli_rs1	width	slli_rd	**F_LEA**
0	**F_IL**	add_rs2	add_rs1	width	add_rd	**F_IL**
slli_imm	**F_SL**	add_rs2	slli_rs1	width	slli_rd	**F_SL**
Imm load 1	**F_LP**	rd2	rs1	width	rd1	**F_LP**

Figure 3.23: F_R4 format with Load Pairs

The opcode defining these fusion sequences is 7'b0000001. The func3[2] field is not used, whereas the func2 and func3[1:0] fields are employed to specify the width of the two load instructions that constitute the fused sequence, as depicted in Figure 3.24 and 3.25.

	opcode7
F_LEA	7'b0000000
F_IL	7'b0000000
F_SL	7'b0000000
F_LP	7'b0000001

Figure 3.24: Opcode of Scaled Index family and Load Pairs family

First Instruction	func2
LD	2'b00
LW	2'b01
LH	2'b10
LB	2'b11

Second instruction	func3 [1:0]
LD	2'b00
LW	2'b01
LH	2'b10
LB	2'b11

func3[2]
x

Figure 3.25: F_R4 fields of the Load Pairs family

After establishing the encoding for the resulting fused instruction, the detector must be modified to support this new family of fused instructions. The existing *c_detector* module has been modified to detect these sequences by increasing its complexity. This approach was chosen because the detector remains sufficiently simple, thus there is no need for parallelization. Moreover, the Load Pair sequences do not overlap with the already supported fusion sequences, making their detection within the existing module a straightforward task. The implementation required only the addition of specific *case statements* without necessitating changes to the mask generation or multiplexer selections.
The two instruction families do not overlap, as they share no common starting point for fusion. However, the final instruction in a Scaled Load or Indexed Load sequence could potentially be a load, which might appear to form a fusible sequence with the subsequent instruction. Nevertheless, one of the constraints for the Load Pair fusion is that the source register must differ from the destination register. This requirement is the opposite of what is needed for the Scaled Load and Indexed Load sequences, where the source and destination registers are the same. As a result, there is no ambiguity between the two fusion families.

The detection process involves verifying that the base register is the same for both loads, ensuring that the offset results in adjacent memory accesses, confirming that the first load's destination register differs from its source register, and checking that

the immediate values are correct according to the specific sequence requirements. These requirements are summarized in Table 3.4.

F_LP
ld1 rd != ld1 rd
ld1 rs1 = ld2 rs1
ld2 imm = ld1 imm + offset

Table 3.4: Register constraints for Load Pairs idiom

Regarding the decoder, integrating support for the new fusion sequences is relatively straightforward. It primarily involves adding the logic necessary to detect these sequences and defining the appropriate control signals, as illustrated in Figure 3.26.

instruction	valid	use_src	rf_src	use_dst	rf_dst	queue_id	use_imm	imm_ext_typ	use_pc	func_unit	instr_op	unsigned_op	width	illegal_instr
F_LP_DD	1	1	INTEGER_RF	11	INTEGER_RF-INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_UNIT	F_LP_OP		_DD_	
F_LP_WW	1	1	INTEGER_RF	11	INTEGER_RF-INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_UNIT	F_LP_OP		_WW_	
F_LP_DW	1	1	INTEGER_RF	11	INTEGER_RF-INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_UNIT	F_LP_OP		_DW_	
F_LP_WD	1	1	INTEGER_RF	11	INTEGER_RF-INTEGER_RF	MEMORY_QUEUE	1	IMM_19_0		MEM_UNIT	F_LP_OP		_WD_	

Figure 3.26: Load Pairs family control signals

3.3 Fusion support in the Back-End

The Back-End is composed of the instruction queues and four datapaths: integer, floating point, memory, and system.
The fused instructions that belong to the LEA idioms are issued to the integer datapath, whereas the ones that belong to the IL and SL idioms belong to the memory datapath.

3.3.1 Integer Execution: Fusion ALU

The integer execution module receives operations from the integer issue queue and processes them in the functional units.
In the integer execution, shown in Figure 3.27, operations proceed through the following stages:

- Issue: the operations are received by the pipeline
- Register Request (RR): all operations request registers to read from the Register File, it can be performed in the same cycle of Issue.
- Execute (Exe): the execution of the instruction is performed within the functional unit
- Complete (CMPLT): the result is registered, and target control operations take place

Figure 3.27: Pipeline, focus on integer execution

The Integer Functional Unit Generator module generates the necessary instances of integer functional units required for processor execution. Given a specified number of ports, this module instantiates functional units on each port according to the parameters ALU_FU_VEC, MULU_FU_VEC, DIVU_FU_VEC, and BRU_FU_VEC.
For example, with the configuration:

$$ALU_FU_VEC = 4'b1011,\ MULU_FU_VEC = 4'b1000,$$
$$DIVU_FU_VEC = 4'b1000,\ BRU_FU_VEC = 4'b0101$$

the Integer Functional Unit Generator will create an arrangement as follows, it can also be seen in Figure 3.28:

- ALU (Arithmetic Logic Unit) on ports 0, 1, 3
- MULU (Multiply Unit) on port 3
- DIVU (Divide Unit) on port 3
- BRU (Branch Unit) on ports 0, 2

This configuration means that port 0 can handle both ALU and BRU operations, port 1 can handle ALU operations, port 2 can handle BRU operations, port 3 can handle ALU, MULU, and DIVU operations.

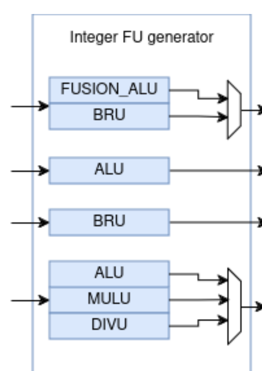


Figure 3.28: Integer FU generator

Fusion ALU

A new arithmetic logic unit has been designed to execute fused instructions. Currently, the only fused instruction that it supports is the Load Effective Address instruction. This new ALU is implemented as an extension of the existing ALU, incorporating support for all standard operations as well as the additional LEA operation.

The new parameter, ALU_FUSION_VEC, has been added to the Integer Functional Unit Generator. The Fusion ALU supports all operations of the standard ALU, with the added capability of executing LEA instructions. Therefore, only one of the two ALUs can be allocated to a given port at a time. The constraint is that a fusion ALU can be allocated only if the normal ALU parameter is enabled; in this case, only the fusion ALU is allocated to that port. If only the fusion ALU is enabled, it cannot be allocated. As shown in Figure 3.28.

The Fusion Arithmetic Logic Unit (FUSION_ALU) is a combinational module that performs a series of operations involving operand 1, operand 2, and/or an immediate value. The supported operations include all those of the standard ALU, with the addition of LEA, as detailed in Table 3.5.

OP	Name
HL_ADD	Addition
HL_SUB	Substraction
HL_XOR	Logical HL_XOR
HL_OR	Logical HL_OR
HL_AND	Logical HL_AND
HL_SRA	Arithmetic Right Shift
HL_SRL	Logical Right Shift
HL_SLL	Logical Left Shift
HL_SLT	Set if Less Than
HL_F_LEA	**Load Effective Address**

Table 3.5: Operation supported by the Fusion ALU

The LEA operation is encoded using the signal HL_F_LEA, and the operation size, whether it is *add* or *addw*, is encoded in the input signal size_i.

As the ALU, it can be configured for 32-bit size operations only or 64-bit size operations included:

- 32-bit size : only 32-bits operations are allowed, as a consequence there is no need to extend the result. The LEA operation is reported in Figure 3.29

- 64-bit size : operations are performed on 64 bits. If the addition is on 32 bits, the result is sign-extended to 64 bits. The LEA operation is reported in Figure 3.30

```
HL_F_LEA : begin
    result_o = (data_rs1_i << imm_i) + data_rs2_i;
end
```

Figure 3.29: SystemVerilog implementation of 32-bit LEA operation

```
HL_F_LEA : begin

    if (op_size_i == HL_WORD) begin
        result_o[31:0] = {data_rs1_i<< imm_i}[31:0] + data_rs2_i[31:0];
        result_o[63:32] = {32{result_o[31]}};
    end
    else begin
        result_o = (data_rs1_i << imm_i) + data_rs2_i;

    end
end
```

Figure 3.30: SystemVerilog implementation of 64-bit LEA operation

3.3.2 Integer Execution: testbench

The module has been validated using a comprehensive testbench. This testbench is an extended version of the one previously used for the ALU module, which incorporates additional support for LEA operation.

Two distinct sets of tests were conducted: one for 32-bit operations and the other for 64-bit operations. Each test set comprised 1000 random operations. The operands included a mix of signed and unsigned values, as well as varying sizes.
This approach ensured that all possible operations supported by the Fusion ALU were thoroughly tested.

3.3.3 Memory Execution

The memory execution includes stages analogous to those in integer execution, specifically the stages of Register Read, Execute, and Complete. The effective address of the memory operations is calculated in the Address Generation Unit (AGU). The AGU is the initial stage of the Load Store Unit (LSU) in the Lagarto Ox core.

The memory execution handles the fused instructions Indexed Load and Scaled Load. These instructions are made of two parts: an arithmetic one responsible for computing the effective address and a load operation that utilizes this address.
For Indexed Load instructions, the arithmetic component involves a straightforward addition, where the effective address is computed by summing two source registers. On the other hand, the arithmetic operations for Scaled Load instructions are the same as those performed by the Load Effective Address instruction.

To support these new operations, the calculation of the effective address has been relocated in two different units in the Execution stage. The MEM_SHIFT unit

is responsible for handling Scaled Load operations, executed by the module *fusion_sl_fu*. Meanwhile, the standard address calculation is conducted in the MEM unit, which now also accommodates the calculation of Indexed Load addresses. This calculation is similar to the usual effective address computation, involving an addition, but can also handle the *addw* operation. For this purpose, a distinct module called *fusion_il_fu* is utilized. The memory execution's block diagram is reported in Figure 3.31.

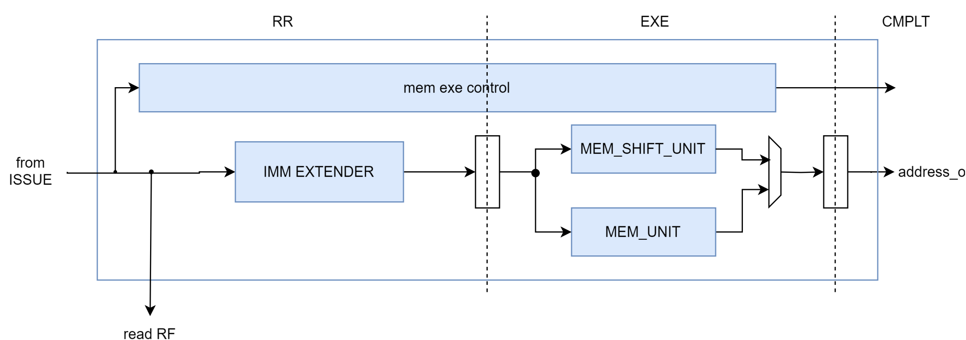


Figure 3.31: Block diagram of Memory Execution

IL functional unit

The Fusion Indexed Load Unit executes the arithmetic component of the fused instruction. The outcome of this operation is the effective address used by the load in the Indexed Load fused operation.
The effective address is computed as the addition of the two source registers:

$$rs1 + rs2$$

This unit can be configured for 32-bit size operations only, or 64-bit operations, with the support of both *add* and *addw* operations.
In the first case, only 32-bit operations are permitted, and there is no need to extend the result. The code is reported in Figure 3.32.
In the second case, the operations are conducted on 64 bits. If the addition is an *addw* operation performed on 32 bits, the result is sign-extended to 64 bits. The code is reported in Figure 3.33

```
assign result_o = data_rs1_i + data_rs2_i;
```

Figure 3.32: Effective address calculation in IL instruction, for 32-bits configuration

```
always_comb begin
    if ((op_size_i == HL_WW) || (op_size_i == HL_WD)) begin
        result_o[31:0] = data_rs1_i[31:0] + data_rs2_i[31:0];
        result_o[63:32] = {32{result_o[31]}};
    end
    else begin
        result_o = data_rs1_i + data_rs2_i;
    end
end
```

Figure 3.33: Effective address calculation in IL instruction, for 64-bits configuration

SL functional unit

The Fusion Scaled Index Unit is responsible for the arithmetic operations associated with the Scaled Load fused instruction. The outcome of this operation is the effective address utilized by the load in the Scaled Load fused operation.

The effective address is computed as:

$$(rs1 << imm) + rs2$$

A parameter allows to configure the module to execute the operation in 0 or 1 cycles. This feature was implemented to verify and ensure timing compliance when the effective address calculation is not divided into separate stages. Performing both the shift and the addition within the same cycle could potentially create a critical path, especially since the result of this operation is required by the Translation Lookaside Buffer (TLB) as the effective address within the same clock cycle.

- 0 cycle: the shift calculation and the load operation are completed within a single cycle
- 1 cycle: the shift calculation is performed in the first cycle, followed by the addition in the second cycle

As the previous module, this unit can be configured for either 32-bit size operations only, Figure 3.34 , or for 64-bit size operations, Figure 3.35.

```
result_o = intermediate_result_q + data_rs2_q;
```

Figure 3.34: Effective address calculation in SL instruction, for 32-bits configuration

```
always_comb begin
    if ((op_size_q == HL_WW) || (op_size_q == HL_WD)) begin
        result_o[31:0] = intermediate_result_q[31:0] + data_rs2_q[31:0];
        result_o[63:32] = {32{result_o[31]}};
    end
    else  begin
        result_o = intermediate_result_q + data_rs2_q;
    end
end
```

Figure 3.35: Effective address calculation in SL instruction, for 64-bits configuration

3.3.4 Memory Execution: testbench

The two modules underwent validation through testbenches, following a process similar to the validation of the Fusion ALU.
For the *fusion_il_fu* module, 2 sets of tests were executed with the 32-bits and 64-bits configurations. For each configurations, multiple iterations of the test were conducted using random input operations selected from the four operations that belong to the IL idiom. Similarly, the *fusion_sl_fu* module was tested using the same methodology, with additional tests for the two possible cycle configurations.

After the successful validation of these modules, a testbench at the memory execution level was conducted to verify successful integration. In this phase, the IL and SL operations were incorporated into the range of input signals for the memory execution testbench and then tested.

3.4 Exception handling

Among the instructions that compose the fusion candidates, the only one that can generate an exception is the load instruction and it is always the last instruction in the sequence that can be fused.
If the load instruction generates an exception, the preceding instructions in the fused sequence should still be committed. This introduces a new scenario where an instruction that generates an exception can also commit, which was not possible before.

3.4.1 Fetch stage

In this stage the exceptions that the load instruction can generate are page fault and access fault.
An instruction access fault occurs when the instruction's memory address points to a location that the processor is not allowed to access. This can happen if the instruction tries to access memory outside the bounds of the allocated segment, a read-only segment, or an invalid address.
An instruction page fault occurs when the instruction's memory address is part of a block of memory that has not been made available yet because the page is not currently mapped to the physical memory.
If an exception is detected, the instructions must not be fused, so the input of the fusion detector is invalidated.
Referring to the Fetch stage block diagram in Figure 3.6, instruction page faults are detected in stage F1 of fetch, originating from the instruction cache, and they invalidate the fusion detector in stage F2. Instruction access faults are detected in stage F2.

3.4.2 Reorder Buffer

When a fused instruction generates an exception, it needs to be committed and the exception handled. This is a new scenario, up until now an instruction that generated an exception could not have been completed.
Every cycle up to commit-bandwidth instructions can committed if they have completed the execution and the ROB head is advanced. The mechanism is shown in Figure 3.36.

Upon completion of an instruction execution, a corresponding bit is set in the *completed_vector*. This vector is addressed using the *rob_id* of the instructions. The FIFO controller module manages all control signals of the Reorder Buffer, which functions as a register file. The FIFO controller generates addresses to access the *completed_vector*, with a size equal to the commit bandwidth starting from the *rob_head*. Information regarding the completion of instructions within the commit window is then saved in the *rdy_vector*. This signal is assigned to *commit_rdy_o*, which is sent to the Load-Store Data Unit to enable the commitment of those instructions. It also serves as the read enable signal for the ROB and the FIFO controller. The former produces the physical registers of the committing instructions needed for renaming, while the latter advances the ROB head and generates valid commit signals sent to the rename stage.
If an instruction is not completed when the vector is read, the ROB head will

advance to that instruction and remain there until either the instruction is completed or it is detected as an exception by the exception collector. The exception collector, a dedicated module, compares the ROB head with the *rob_id* of the instruction that generated the exception. Upon detecting a match, the exception handling process is initiated.

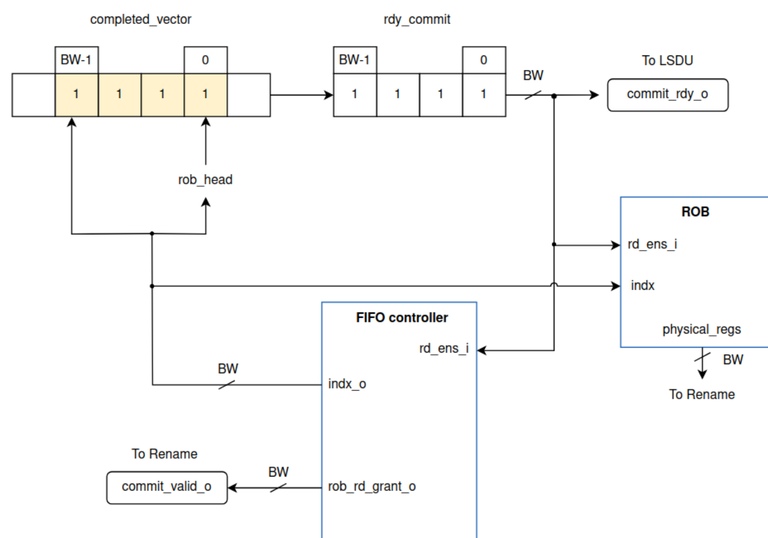


Figure 3.36: Block scheme of Reorder Buffer

This mechanism needs modifications in the new scenario introduced by fused instructions. If a fused instruction generates an exception, it might never reach the head of the ROB because it is also marked as completed and will thus be committed. Consequently, the exception might never be handled.

To generalize, this scenario where an instruction can complete but also generate an exception will be referred to as the 'partially completed' case. Detecting the partially completed case, which can occur only with fused instructions, requires meeting two conditions:

- The instruction in the commit window has an exception
- The instruction is marked as completed

Instead of using a vector to track whether an instruction is fused, which would require an additional vector of the same size as the ROB, a detection method is implemented.

The first step is to detect if an instruction within the commit window has encountered an exception. This involves identifying whether the rob_id of the instruction

that can generate an exception, provided by the exception collector, falls within the commit window.

The commit window consists of consecutively increasing *rob_ids*, since instructions need to be committed in order. The lower bound of the commit window is the ROB head and the upper bound is determined by the ROB head plus the commit bandwidth, which represents the number of instructions in the commit window. As shown in Figure 3.37.

If the *rob_id* of the instruction that has encountered an exception is within the commit window, the position of the instruction within the commit window is saved. This index is then used to generate the necessary masks for commit and exception handling.

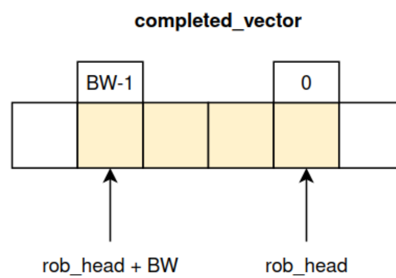


Figure 3.37: Completed vector boundaries

Since the Reorder Buffer is a circular buffer, a corner case must be considered for detection. This occurs when the commit window spans the end and beginning of the queue, the situation is summarised in Figure 3.38. In such cases, the current detection logic fails. Additional logic is required to handle this scenario and correctly detect exceptions within the commit window.

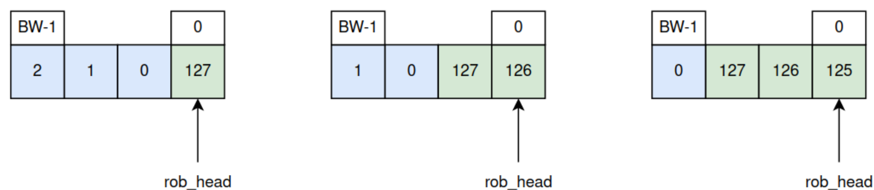


Figure 3.38: Completed vector boundaries, corner case of circular buffer

Completed instructions must be committed only up to the partially completed one, which will be the last instruction that can be committed within the current commit

bandwidth. The *commit_valid_mask* achieves this by disabling the commit of instructions that come after the partially completed one in the commit window. The final signal is *commit_rdy_o*.
This mask also serves as the read enable for the ROB to obtain the physical registers needed for committing the instructions. Additionally, it enables the read operation for *commit_valid_o* through *fusion_xcptn_rob_rd_grants_generator*, which uses the same logic as the ROB controller for generating read grants but tailored for the partially completed case.

The *controller_rd_ens_mask* is used to invalidate the read enable signal of the ROB controller for the partially completed instruction and the subsequent instructions in the commit window. This is necessary to prevent the ROB head from advancing past the partially completed instruction, as the ROB head must remain at this instruction for the exception to be detected and handled. This mechanism prevents unnecessary advancement of the ROB head and it is possible to implement because the ROB allows separate read and write operations. The ROB needs to be read to obtain the data necessary to commit the instruction, but the ROB head should not advance past the partially completed instruction.
An extra port has been added to the module that handles the reset of the 'complete' vector to reset the partially completed instruction when it is detected. The 'complete' signal is reset to ensure that the instruction is committed only once. This reset occurs when the instruction is detected, allowing it to be committed, and takes effect in the next cycle.
This reset is crucial because if the instruction is not at the ROB head when marked as completed and generates an exception, it will reach the head in the next cycle. If the complete signal is not reset, the instruction would be committed again.
The Reorder Buffer after the modification in shown in Figure 3.39.

In summary, the steps needed to handle the fused instruction exception are:

- detect if the instruction in the commit window has an exception and if it is marked as completed
- generate two masks to handle the exception. *commit_valid_mask* ensures the instruction is ready to be committed, by generating the *commit_rdy_o* signal and the *commit_valid_o* signal that comes from the reading of the ROB with a duplicated logic. The second mask is *controller_rd_ens_mask*, it prevents advancing the ROB head unnecessarily, as the ROB allows separate read and write operations. Since the ROB has to be read, in order to get the data necessary to commit the instruction, the ROB head should not advance past the fused instruction
- once the instruction that is both completed and generates an exception is

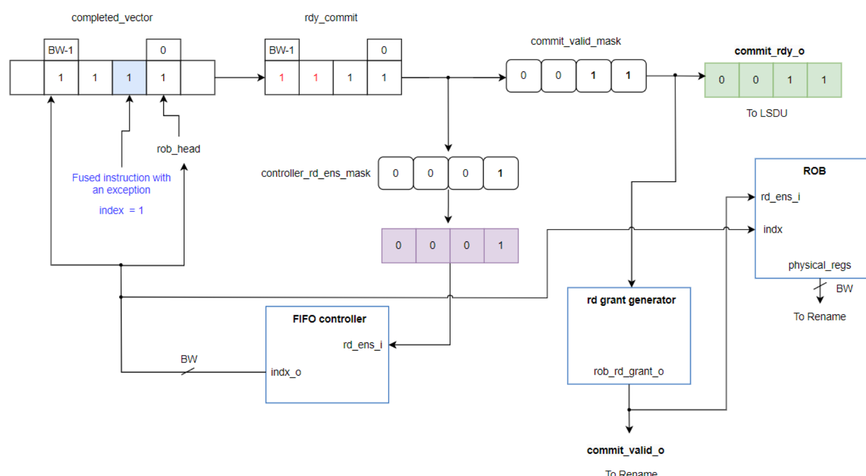


Figure 3.39: Modified block scheme of the Reorder Buffer

detected the instruction can be committed and the ROB head has to be moved to point to that instruction so that the exception can be detected

- the 'complete' signal has to be reset to ensure that the instruction is committed only once. This reset is given when the instruction is detected, so it can be committed, and it will be effective the next cycle.

3.4.3 Program Counter

The last issue that remain to correctly implement the precise exception mechanism regards the Program Counter.

When a sequence of instructions are fused, the PC of the resulting instruction will be the one that belonged to the first instruction in the sequence. When an instruction generates an exception, the PC information is needed to handle the exception. In case of a fused instruction, the last operation in the sequence can generate an exception, which is the load operation in the IL and SL fused instructions. However, the PC of the fused instruction is not the one that generated the exception.
In the case of Indexed Load, the PC is the one referring to the *slli* instruction, but *load* is the instruction that can produce the exception, so the rigth PC value is the PC of the fused instruction incremented by two, since the instructions in the sequence are compressed.
In case of Scaled Load, the load instruction is the third in the sequence, so the PC must be updated as PC incremented by four, reported in Figure 3.40.

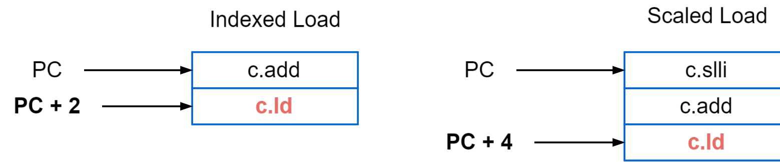


Figure 3.40: Program Counter update for Indexed Load and Scaled Load operations

This mechanism takes place only when the exception that has been detected as valid and has to be taken care of by going from the datapath to the core system level. If the exception is valid and the source is one of the related fused instructions, the PC is updated accordingly.

3.5 Entire core

The presence of the key components related to instruction fusion can be observed across different stages of the Lagarto OX core diagram, in Figure 3.41. The Fusion Detector placed in the Fetch stage, the Fusion Decoders in the Decoder, the Exception Commit Fusion Handler in the Reorder Buffer, the Fusion ALUs and the Fusion Calculation, responsible of the effective address calculation in the Functional Units. Moreover, the Rename stage requires Multiple Rename Destination Support to support load pairs fusion, complemented by the Load Pairs Write-Back Handler within the Load/Store Unit. These components are highlighted in gray in the current design schematic, indicating that they are yet to be implemented but are planned for future development.

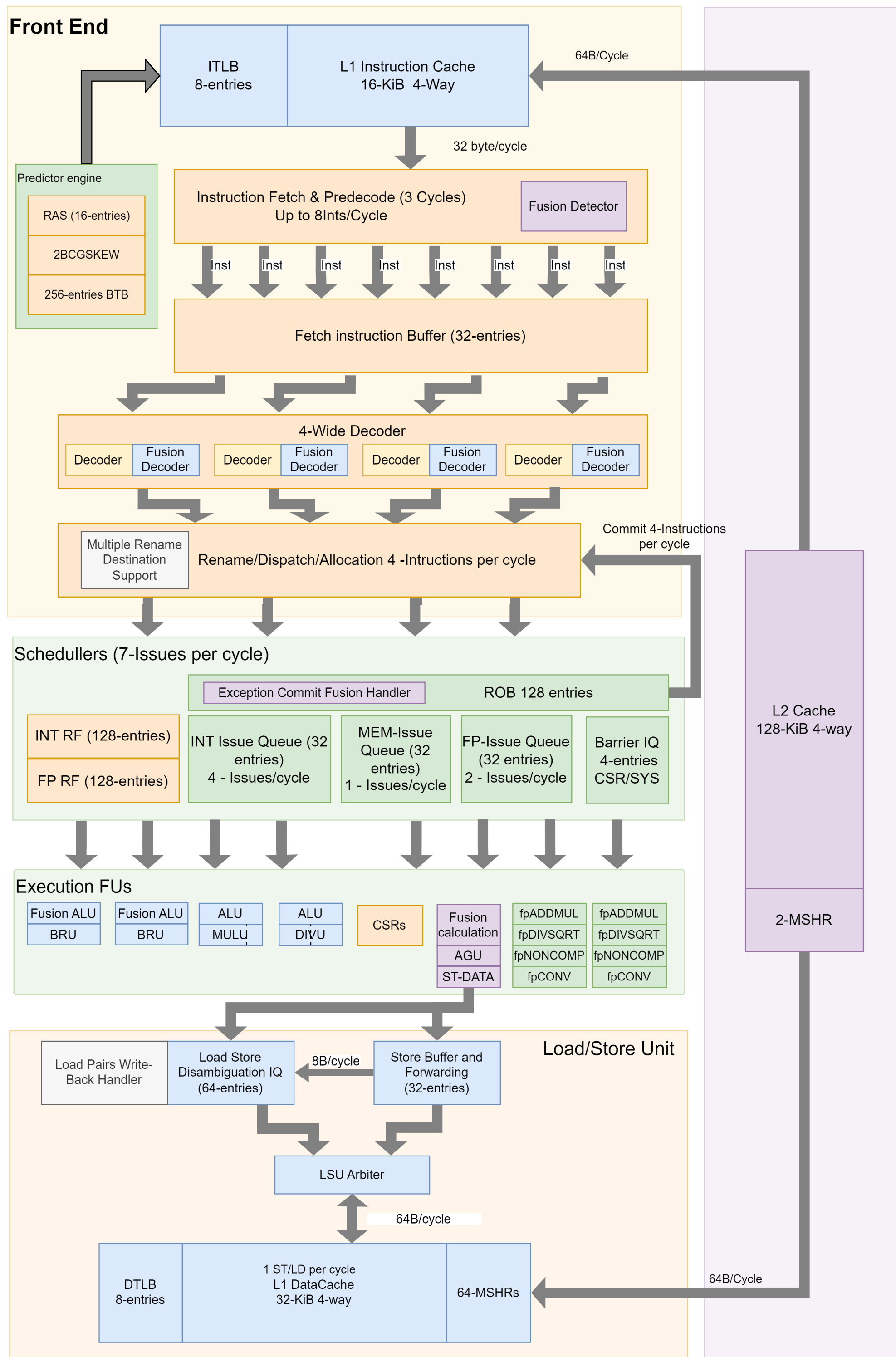


Figure 3.41: Lagarto Ox core diagram with instruction fusion support

Chapter 4

Verification and Performance

The verification and performance evaluation of the core are critical steps in validating the integration of instruction fusion. It is important to ensure that the core adheres to the ISA standards, operates correctly, and achieves the desired efficiency and effectiveness.
The initial step involved the verification of each new implementation to ensure its correctness. This was accomplished through testbenching, conducted at the level of each modification, utilizing the cocotb framework.
Then, the ISA tests and benchmarks are conducted. The aim is to carry out testing to ensure that the new feature do not affect the normal functioning of the system and the new operations are executed correctly.

4.1 ISA tests

Architectural testing is a nonfunctional testing technique employed to validate whether a developed system conforms to the prescribed standards. In this context, the golden reference is the RISC-V ISA standard. The tests are conducted using a Test Virtual Machine (TVM) for RISC-V, an environment specifically designed to facilitate the testing and verification of RISC-V cores, software, and systems. The TVM provides a controlled and isolated environment in which developers can run tests and simulations to validate the functionality, performance, and compliance of RISC-V implementations.

Test programs for RISC-V are written in a single assembly language file, processed through the C preprocessor, allowing the use of standard assembly directives. The architectural tests executed are the User-Level TVMs available at the provided repository [23]. These tests are executed in environments with both physical and virtual memory configurations, and different execution environment levels such as

machine, supervisor and user. The ISA tests are composed of MACROs that define the testing framework. Variables are passed to these MACROs, which structure the test into two potential outcomes: a failure branch and a success branch. Upon completion of each test, the result is compared with the expected data, directing the flow into one of these branches based on whether the result matches the expected outcome.
The 'test case' MACRO serves as the foundational structure within each test's MACRO. The variables passed to this MACRO include the test number, the register where the expected result is stored, the expected result itself, and the instructions to be executed. The MACRO loads the expected result into the specified register, executes the instruction provided as a parameter, and then compares the outcome to the expected result passed as a parameter. This process ensures that each test is systematically verified against its expected behavior.
For example, in the case of an ISA test for a load instruction, the test uses the 'test case' MACRO, providing parameters such as the test number, the specific instruction (e.g., ld), the base address, the offset, and the expected result. The instruction then loads the result from the address calculated as the base address plus the offset and compares the loaded data with the expected result provided to the MACRO. This structure ensures that each test is systematically executed and evaluated for correctness.

Due to instruction fusion, which introduces new internal instructions, the standard test set was extended. Specific ISA tests were added to evaluate new instruction idioms such as LEA , IL, and SL. The test have been designed following the ISA test structure, enabling the possibility to have the test executed with virtual memory and the different execution environment levels. This extension ensures comprehensive testing of the instruction set, validating both standard and newly introduced instructions.
The Load Effective Address test MACRO takes as parameters the two source values, the immediate, and the expected result. The test case MACRO performs the shift and addition operations, subsequently comparing the computed result to the expected value. This MACRO is then utilized with various data inputs and differing operand widths for the addition operation, specifically add and addw.
The Indexed Load and Scaled Load tests are built upon the load instruction MACRO, as they involve a load operation. The primary modification is the inclusion of the necessary arithmetic instruction to calculate the effective address, as specified by the fused instruction. This adjustment ensures that the tests accurately reflect the behavior of the IL and SL operations within the architecture.

4.2 Benchmarks

Benchmarking a RISC-V core involves evaluating its performance across various metrics and scenarios to understand its efficiency, speed, and resource usage.

The benchmarks used are axpy, bubblesort, common, coremark, dhrystone, fibonacci, histogram, matmul_fp, matrix_mul, median, mem_copy, mem_flush, memcpy, mm, mt-matmul, mt-memcpy, mt-vvadd, multiply, peak_flops, peak_mips, peak_flops_mix, peak_flops_div, pmp, qsort, rsort, spmv, spmv_fp, towers, vec-daxpy, vec-memcpy, vec-sgemm, vec-strcmp, vvadd.
The performance of the core without the support of instruction fusion is reported in Figure 4.1, where the number of cycles and retired instructions are reported, as well as the Instructions Per Cycle (IPC) parameter.

RISCV			
BENCHMARK	**CYCLES**	**INSTRUCTIONS**	**IPC**
axpy	25814	24258	0,940
bubblesort	25701047	23040117	0,896
dhrystone	131978	187582	1,421
fibonacci	3009844	3565491	1,185
histogram	4568	4076	0,892
matrix_mult	610572	1499939	2,457
median	60595	44923	0,741
mem_copy	29051	25073	0,863
mem_flush	7654	4447	0,581
mm	9929	24704	2,488
multiply	15756	24155	1,533
peak_flops	11450	20094	1,755
peak_mips	56665	200075	3,531
qsort	585025	847183	1,448
rsort	879866	1711457	1,945
spmv_fp	62440	109644	1,756
spmv	21420	34521	1,612
towers	6634	4282	0,645
vvadd	31766	70126	2,208
GEOMEAN			**1,353**
GEOMEAN NO PEAK			**1,259**

Figure 4.1: Performance results of the Lagarto OX core without Instruction Fusion

The current compiler struggles to schedule instructions in a manner that enables them to be fused effectively. The fusion technique implemented requires specific patterns of instructions that are close to each other in the instruction stream. However, compilers might not always generate these patterns due to optimization strategies

like instruction scheduling and register allocation, reducing the fusion opportunities. As of the latest GCC versions, there is no built-in support for instruction fusion. However, future developments are anticipated, with the RISE project planning to introduce such support [24], as well as Ventana that is implementing support for ten specific fusion idioms within GCC [25].
Without fusion-aware instruction scheduling, the potential performance improvements from instruction fusion cannot be realized. This limitation is evident in the benchmark results, where instruction fusion is observed in only four tests. This outcome highlights the critical importance of compiler support for maximizing the benefits of instruction fusion.
The reason why only four benchmarks show significant results is that the fusion idioms implemented are not as prevalent as others, such as load pairs, which would likely demonstrate more results, as analyzed in [5]. Another significant factor is the lack of compiler support, which causes to miss even the limited fusion opportunities that currently exist.

To address this limitation, two sets of benchmarks have been developed for each supported fusion idiom. The first set, called 'Ideal', consists of loop iterations executed 10000 times with instructions scheduled precisely as required for fusion. The second set, called 'Loop', mirrors the ideal case but includes the additional step of storing the result within the loop, creating a scenario more reflective of real-world applications. The necessity for these tests arises from the lack of significant results in standard benchmarks, and they provide valuable insights into the upper bounds of performance gains achievable through this optimization technique and the selected sequences.
The compiler issue is further demonstrated by the fact that the benchmarks are written in C, yet it is necessary to force assembly language using an *asm* statement. This is required because the compiler is unable to schedule the instructions correctly to create opportunities for fusion. This reliance on manually crafted assembly code highlights the urgent need for compiler enhancements to support instruction fusion effectively.

4.2.1 Performance

The performance is evaluated based on the number of cycles and the retired instruction count. Between the standard benchmarks, four present opportunities for the supported instruction fusion. *histogram* benchmark presents a 12.4% reduction in the retired instruction count, but there is no improvement in the number of cycles. *spmv fp* benchmark, double-precision general matrix multiplication benchmark, presents a 3.3% reduction in the retired instruction count and a 2.5% reduction in the number of cycles. *spmv* benchmark, double-precision sparse matrix-vector

multiplication benchmark, presents a 2.2% reduction in the retired instruction count and a 0.8% reduction in the number of cycles. *rsort* benchmark, that uses quicksort to sort an array of integers, presents a 3.6% reduction in the retired instruction count and no improvement in the number of cycles.

The comparison in the number of retired instruction is shown in Figure 4.2, while the comparison of the number of cycles is shown, together with the ad hoc benchmarks, in Figure 4.3.

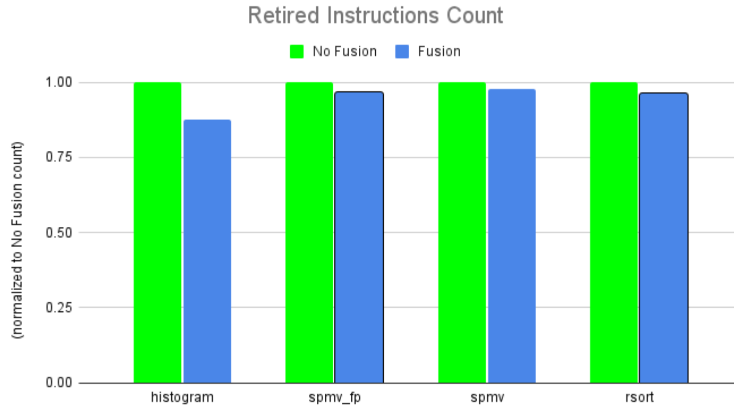


Figure 4.2: Retired Instructions count for the four benchmarks

These benchmark results demonstrate that the 10 fusible instruction sequences supported achieve the goal of reducing the retired instruction count, when they are present in the code. The reduction in the number of cycles is low or not even present because of the dependencies of the fused instruction with the following instructions. This is the reason why it is necessary to look at the performance of the ad hoc benchmarks to have a clear picture of the potential of the implemented technique.

The Load effective address benchmark achieves more than 50% reduction in the number of cycles in the ‘Ideal’ test. This is the upper bound: two instructions are fused into one, there are no dependencies, and the number of cycles is reduced by half. The ’Loop’ tests achieves 5.4% of reduction.

The Indexed Load benchmark shows around a 25% reduction for both the ‘Ideal’ and the ‘Loop’ test. Like in the previous case, two instructions are fused into one, but the dependencies related to the load instruction do not allow a 50% reduction to be reached.

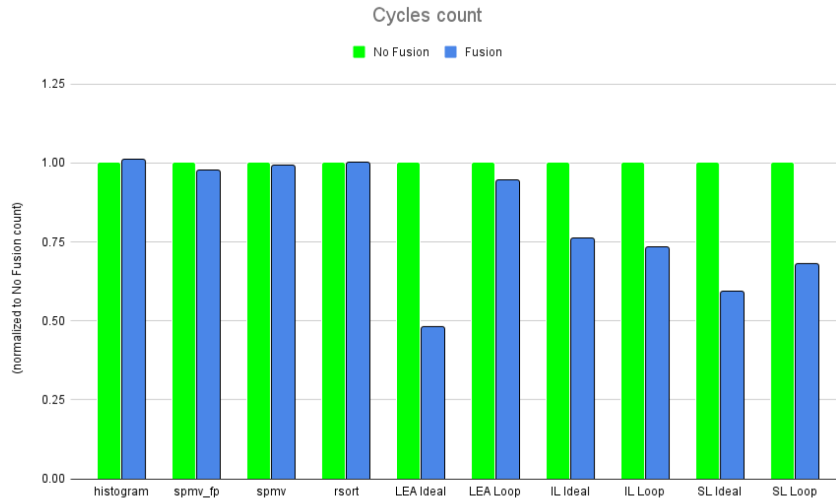


Figure 4.3: Cycles count for the four benchmarks and the ad hoc benchemarks

The Scaled Load benchmark shows a 40% reduction in the 'Ideal' test, and a 32% reduction in the 'Loop' test. This idiom fuses three instructions, the performance is better than the Indexed Load benchmark, and it shows that a study of the dependencies is important.
The performance result regarding the cycles count is reported in Figure 4.3.

Chapter 5

Physical design impact

This final stage of the work presents the results of the synthesis and their analysis. For the RTL synthesis, the Genus Synthesis Solution tool was used. The simulation was done with 7 nm technology from TSMC. The synthesis parameters analyzed are the area [μm^2] and the frequency [GHz].

5.1 Area

Fetch Stage synthesis

The result of the synthesis of the Fetch stage is summarized in Figure 5.1. The area is 40085 μm^2.

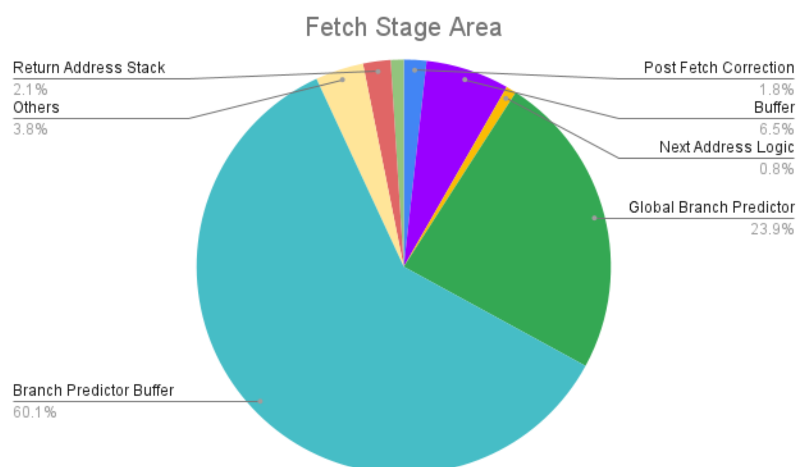


Figure 5.1: Area breakdown graph of Fetch Stage.

The majority of the area within the Fetch stage is occupied by memory-demanding structures. The largest modules in this stage are the Branch Predictor Buffer and the Global Branch Predictor, followed by the inter-stage Buffer. The Branch Predictor Buffer includes the Branch Target Buffer, which stores target program counters in a 256-entry array used during branch prediction. The Global Branch Predictor houses the pattern history table, a memory array of 256 entries that records all pattern histories and recovery information for the global history. Finally, the Instuction Buffer is a fully associative buffer that stores two cache lines of 256 elements.

The Fusion detector occupies an area of 176 μm^2, its impact can be seen in Figure 5.2: the overhead in terms of area is 0.27%, thus it is not even shown in the area breakdown graph because it is too small.

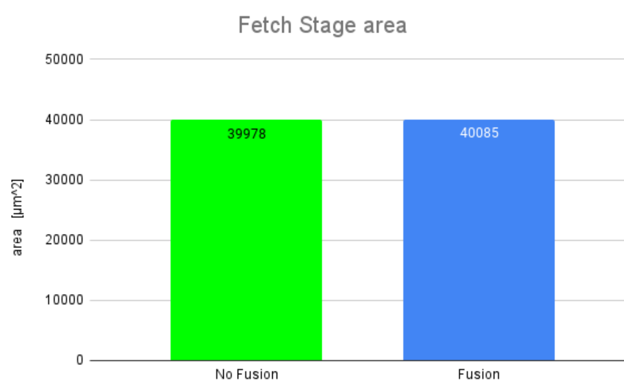


Figure 5.2: Fetch Stage area comparison

Decode stage synthesis

The area breakdown of the Decode stage is reported in Figure 5.3. The area of the Fusion Decoder is 66.27 μm^2, with an overhead of 9.16%, reported in Figure 5.4.

In Figure 5.5, it is possible to see that the Decode stage represents a small portion of the area with respect to the Fetch stage and the ROB. This explains why, even if the increase in the area of the Decode stage is around 10%, it is still a negligible overhead.

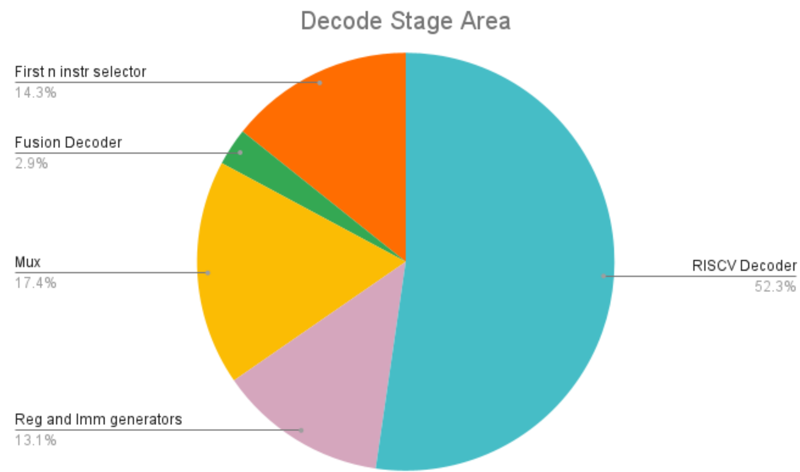


Figure 5.3: Area breakdown graph of Decode Stage

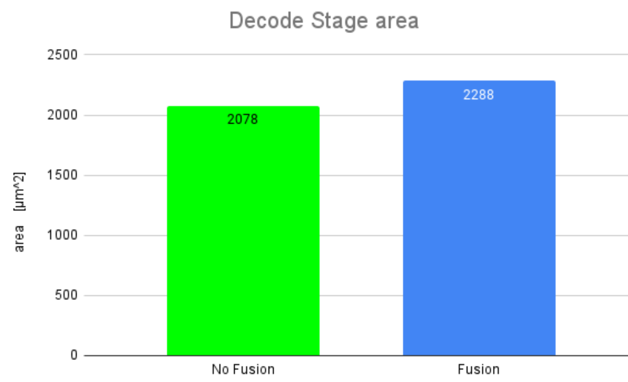


Figure 5.4: Decode Stage area comparison

Integer execution synthesis

The integration of the Fusion ALU costs the Integer execution an increase of 13.63% in area, reported in Figure 5.6.

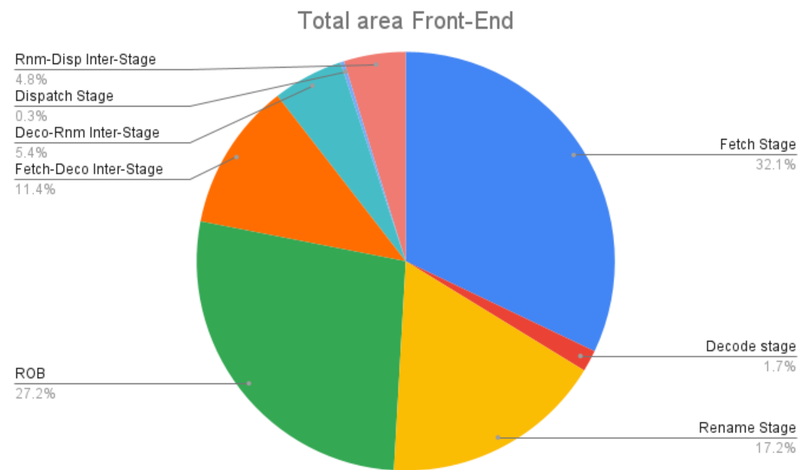


Figure 5.5: Area breakdown graph of Front-End

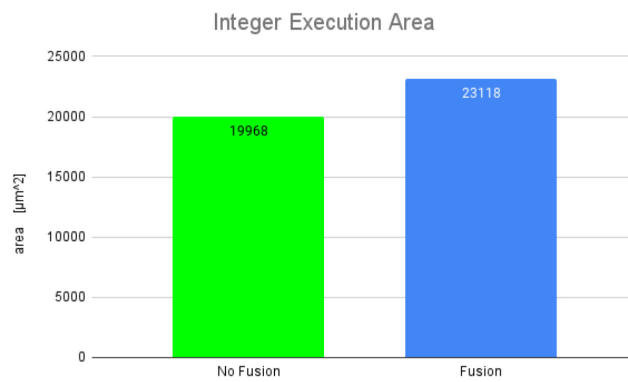


Figure 5.6: Integer execution area comparison

Memory execution synthesis

The Memory execution stage now calculates the effective address, moving the complexity from the Address Generation Unit to the functional units (MEM_UNIT and MEM_SHIFT_UNIT). The 91.15% increase in area, reported in Figure 5.7, is explained by the fact that the Memory execution unit was an empty module before the fusion support introduction.

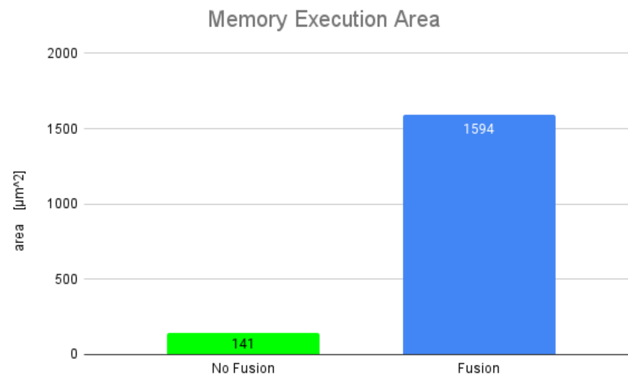


Figure 5.7: Memory execution area comparison

Core synthesis

Overall, there are no increases in the area of the Reorder Buffer. The final area of the core with the support for instruction fusion has increased by 3.32%, reported in Figure 5.8.

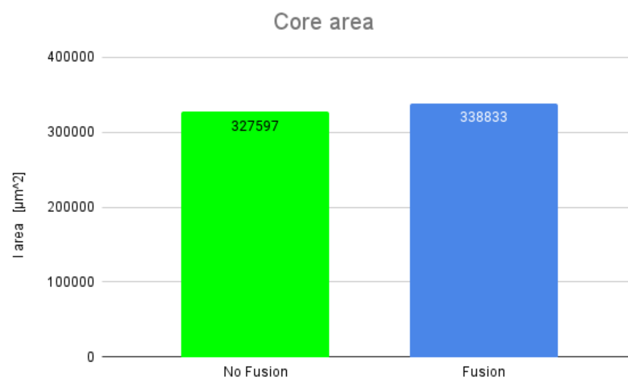


Figure 5.8: Overall area comparison

5.2 Frequency

The tool is configured with a target frequency of 2 GHz. The introduction of the instruction fusion support did not change the already existing critical paths. The comparison is summarized in Figure 5.9.
The critical paths have increased by around 1%, the frequency reached in the best case is 1.57 GHz. In the worst case 0.97 GHz.

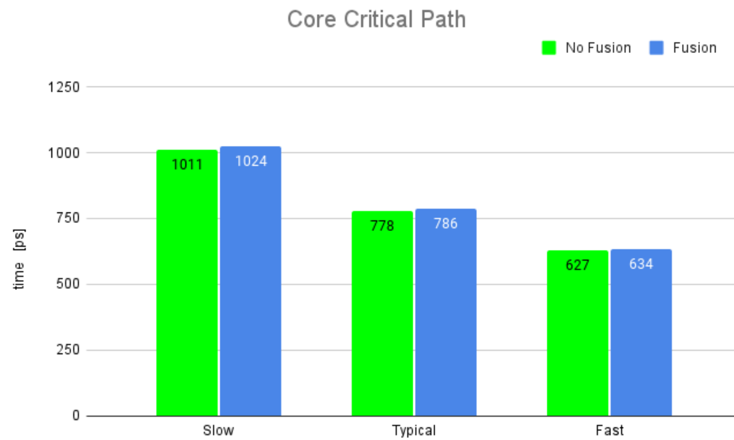


Figure 5.9: Critical paths comparison

Overall, the impact of the support for instruction fusion is negligible in area and frequency when compared with the potential performance improvement. There is also to consider that the reduction in the retired instruction count, with the saved bandwidth in the queues and modules, has an impact on energy consumption, that was not evaluated due to the time constraints of this work, but it is a point to consider for future development.

Chapter 6

Conclusions

This thesis aimed to investigate and develop instruction fusion techniques in a RISC-V Out-of-Order core. Through a detailed analysis involving the study of the state of the art and the architecture of the Lagarto Ox core, the following conclusions were reached:

- deep understanding of the instruction fusion opportunities in a RISC-V core
- implementation of three instruction fusion sequences, Load Effective Address, Indexed Load, Scaled Load, in the Out-of-Order core
- implementation of the precise exception for the fused instructions
- verification of the correct behavior of the fused instructions
- verification of the performance by running benchmarks. The potential gain in the retired instruction count is up to 12% in the benchmarks. While the reduction in the cycles count is shown to be 50% in the ideal case, that serves as upperbound, and up to 2.5% reduction in the benchmarks
- verification of the area and frequency overhead. Both the area and frequency overhead is negligible with the increase of total area by 3.32% and the increase of the critical paths by 1%

The observed increase in performance is constrained by the specific characteristics of the application, particularly due to the presence of the supported instruction fusion sequences within the application code. Moreover, this performance enhancement is further limited by the compiler's ability to recognize and optimize these sequences. For maximal performance gains, the compiler must not only identify and leverage instruction fusion opportunities but also optimize register usage to effectively accommodate the instruction fusion possibilities.

This work contributes to understanding that instruction fusion techniques represent a compelling optimization strategy to improve processor performance, particularly in RISC-V cores.

This conclusion is supported by the results, which demonstrate that the area and frequency overhead incurred by implementing instruction fusion is negligible. Moreover, even without the support from the compiler, the benchmark results reveal that instruction fusion can significantly enhance performance, as evidenced by reductions in both retired instruction counts and execution cycles, and also contribute to energy savings.

6.1 Future work

Some of the potential future explorations are discussed in this section.

- **Load Pairs support**: the implementation of Load Pairs support in the Back-End presents significant challenges. Unlike the Scaled Index family of fusion, Load Pairs instructions require multiple destination registers rather than a single one. This necessitates the management of multiple renamed destinations, adding complexity to the renaming process. Furthermore, the Load/Store Unit must be capable of handling multiple write-backs for a single instruction that accesses contiguous parts of the cache line.

  At this level, the Load/Store Disambiguation Unit demands the most effort. This module plays a crucial role in ensuring that there is no conflict between the load and store instructions. For example, if a load is scheduled before a store but accesses the same address, there is a risk that the load could retrieve incorrect data. The LSDU checks for dependencies between loads and stores to prevent such scenarios.

  With the introduction of Load Pairs fusion, the LSDU will need to track fused instructions as loads that access a larger memory size. This adjustment requires careful modifications to the unit to ensure that it can handle the increased complexity of managing larger memory operations while maintaining data integrity and avoiding collisions between memory accesses.

- **Normal-size instruction fusion support**: in this work, all fusible sequences are composed of compressed instructions. Extending the support of the Scaled Index family to normal-sized instructions only requires modifications to the detector, since the necessary infrastructure is already in place.

- **Energy consumption evaluation**: the next step is providing an energy consumption evaluation, conducting a comprehensive power analysis of both static and dynamic power estimation, to have a complete picture of the performance and overhead of the instruction fusion implementation.

Bibliography

[1] ARM. *Instruction Set Architecture (ISA).* URL: https://www.arm.com/glossary/isa (cit. on p. 1).

[2] Andrew Waterman. «Design of the RISC-V Instruction Set Architecture». In: 2016. URL: https://api.semanticscholar.org/CorpusID:63861396 (cit. on p. 2).

[3] Denis Bakhvalov. *Microbenchmarking fused instruction.* URL: https://easyperf.net/blog/2018/02/04/Micro-ops-fusion (cit. on p. 3).

[4] Agner Fog. *The microarchitecture of Intel, AMD, and VIA CPUs An optimization guide for assembly programmers and compiler makers.* URL: https://www.agner.org/optimize/microarchitecture.pdf (cit. on pp. 3, 4).

[5] Sawan Singh, Arthur Perais, Alexandra Jimborean, and Alberto Ros. «Exploring Instruction Fusion Opportunities in General Purpose Processors». In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2022, pp. 199–212. DOI: 10.1109/MICRO56248.2022.00026 (cit. on pp. 3, 10, 42, 63).

[6] Nathaniel Hoffman Ronny Ronen Alexander Peleg. «System and method for fusing instructions». US Patent US6675376B2. Intel Corp. Dec. 2000 (cit. on p. 3).

[7] Intel. «Intel® 64 and IA-32 Architectures Optimization Reference Manual». In: chap. 3.4.2.2. URL: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf (cit. on p. 3).

[8] *Arm Cortex-A77 Core Software Optimization Guide.* Arm. Nov. 2, 2020. Chap. 4.13 (cit. on p. 4).

[9] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Tech. rep. UCB/EECS-2014-54. May 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html (cit. on p. 4).

[10] Christopher Celio, Palmer Dabbelt, David A. Patterson, and Krste Asanović. *The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V*. 2016. arXiv: 1607.02318 [cs.AR]. URL: https://arxiv.org/abs/1607.02318 (cit. on pp. 4, 6–10, 22, 23, 42).

[11] clamchowder. *Hot Chips 2023: Ventana's Unconventional Veyron V1*. 2023. URL: https://chipsandcheese.com/2023/09/01/hot-chips-2023-ventanas-unconventional-veyron-v1/ (cit. on p. 5).

[12] clamchowder. *Hot Chips 2023: SiFive's P870 Takes RISC-V Further*. 2023. URL: https://chipsandcheese.com/2023/09/03/hot-chips-2023-sifives-p870-takes-risc-v-further/ (cit. on p. 5).

[13] Andrew Waterman. «Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed». MA thesis. EECS Department, University of California, Berkeley, May 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-63.html (cit. on p. 7).

[14] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html (cit. on p. 8).

[15] Fawang Zhang, Dan Tang, and Ye Cai. «Accelerate bit manipulation in XiangShan processer using RISC-V B extension and instruction fusion». In: *International Conference on Cloud Computing, Internet of Things, and Computer Applications (CICA 2022)*. Ed. by Warwick Powell and Amr Tolba. Vol. 12303. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series. July 2022, 123032Q, 123032Q. DOI: 10.1117/12.2642009 (cit. on p. 11).

[16] DRAC project. *Lagarto Ka: The High Performance Core for Drac*. 2021. URL: https://drac.bsc.es/en/bsc2 (cit. on pp. 12, 13).

[17] *DRAC project*. URL: https://drac.bsc.es/en (cit. on p. 12).

[18] J.L. Hennessy, D.A. Patterson, and K. Asanović. *Computer Architecture: A Quantitative Approach*. Computer Architecture: A Quantitative Approach. Kaufmann, 2012. ISBN: 9780123838728. URL: https://books.google.it/books?id=v3-1hVwHnHwC (cit. on p. 15).

[19] John Kubiatowicz. *Graduate Computer Architecture Lecture 8, Explicit Renaming Precise Interrupts*. Feb. 2010. URL: https://people.eecs.berkeley.edu/~kubitron/cs252/lectures/lec08-dynasched3.pdf (cit. on p. 17).

[20] *The Rename Stage - RISCV-V BOOM documentation*. URL: https://docs.boom-core.org/en/latest/sections/rename-stage.html (cit. on p. 17).

[21] *cocotb - Python verification framework*. URL: https://www.cocotb.org/ (cit. on p. 20).

[22] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 20191214-draft, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019 (cit. on pp. 26–28).

[23] *RISCV-software-src/RISCV-tests, GitHub*. URL: `https://github.com/riscv-software-src/riscv-tests` (cit. on p. 60).

[24] Jeff Law. *Fusion Support (GCC) - RISE project*. URL: `https://lf-rise.atlassian.net/wiki/x/dRSD` (cit. on p. 63).

[25] Philipp Tomsich. *RISC-V: Add instruction fusion (for ventana-vt1)*. URL: `https://gcc.gnu.org/pipermail/gcc-patches/2021-November/584404.html` (cit. on p. 63).