POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Analyzing and Extending Instruction Sets for Efficient Cryptographic Computation

Supervisors

Prof. Guido MASERA

Ing. Alessandra DOLMETA

Ing. Valeria PISCOPO

Candidate

Andrea BONINO

June 2025

Abstract

Embedded systems and IoT devices are designed to perform specific tasks within strict limits on energy consumption and computing power. To ensure secure communication, these devices require cryptography, nowadays including post-quantum algorithms designed to resist emerging threats. However, the complexity of such algorithms often makes their implementation inefficient on constrained platforms. Although hardware optimisations have proven effective in this domain, they typically require deep algorithm-specific knowledge and manual intervention.

This work proposes an alternative approach: CIRCO (Custom Instruction RISC-V Code Optimizer). This tool automatically analyses the assembly code of an application to identify patterns of instructions that can be merged into new, custom RISC-V-compliant instructions. Unlike conventional approaches, CIRCO focuses on logic and arithmetic patterns, avoiding changes to memory or control flow instructions. The CIRCO flow allows user interaction, enabling iterative exploration and guided optimisation.

The software's potential has been tested on a real application: Kyber, a post-quantum cryptographic algorithm that has been widely studied for optimisation using the traditional approach. Due to Kyber's register-limited nature, the performance improvement achieved is modest (around 3 %), but its flexibility allows it to be used on top of an already developed solution.

Beyond Kyber, CIRCO can serve as a starting point for exploring optimisations in other applications, including non-cryptographic ones, supporting the user in the process. Its software's versatility and modular design suggest a promising direction for future research in custom instruction generation.

Keywords: Hardware optimization, RISC-V, Post-Quantum Cryptography

Acknowledgements

Grazie a chi mi ha accompagnato in questo percorso
Grazie a chi festeggia con me questo traguardo
Grazie a chi mi aspetta per una nuova partenza

Table of Contents

List of Tables

List of Figures

Acronyms

CIRCO

Cusom Instruction RISC-V Code Optimizer

IoT

Internet of Things

RISC

Reduce Istruction Set Computer

ASM

Assembly

PQ

Post Quantum

PQC

Post Quantum Cryptography

NIST

National Institute of Standards and Technology

KEM

Key Encapsulation Mechanism

Chapter 1

Introduction

1.1 Embedded systems and IoT devices

Embedded systems are specialized computing units designed to perform specific tasks and are typically integrated into larger, more complex environments. Similarly, IoT (Internet of Things) devices combine electronic components—such as sensors and processors—with a native, non-electronic context. Sensors acquire data from the environment, while embedded processors enable network connectivity and communication, allowing the device to exchange information in real time.

These systems are fundamental to modern technology, enabling smart infrastructure, wearable health monitors, industrial automation, and much more. However, they are inherently constrained by two critical factors: limited computational power and strict energy budgets [1]. Unlike general-purpose computers, which are designed for multitasking and high throughput, embedded and IoT devices are tailored for compactness, simplicity, and cost-effectiveness. Integrating full-scale computing platforms into such systems is not only unnecessary but also counterproductive—these platforms require more physical space, introduce higher power demands, and significantly increase production costs.

Moreover, many of these devices operate on battery power or energy harvesting, making energy efficiency not just desirable but essential. Even small inefficiencies can drastically reduce operational lifespan or require frequent maintenance, which is especially problematic in remote or large-scale deployments.

To overcome these challenges, both software and hardware must be carefully co-designed. Applications need to be highly optimized to reduce computational overhead, and hardware architectures must be streamlined to include only what is strictly necessary. This tailored approach ensures that the systems meet performance goals while staying within power and size constraints.

1.2 Cryptography and Post Quantum Algorithms

Cryptography is the study of methods and algorithms that secure communication and data transfer by making them inaccessible to third parties [2].

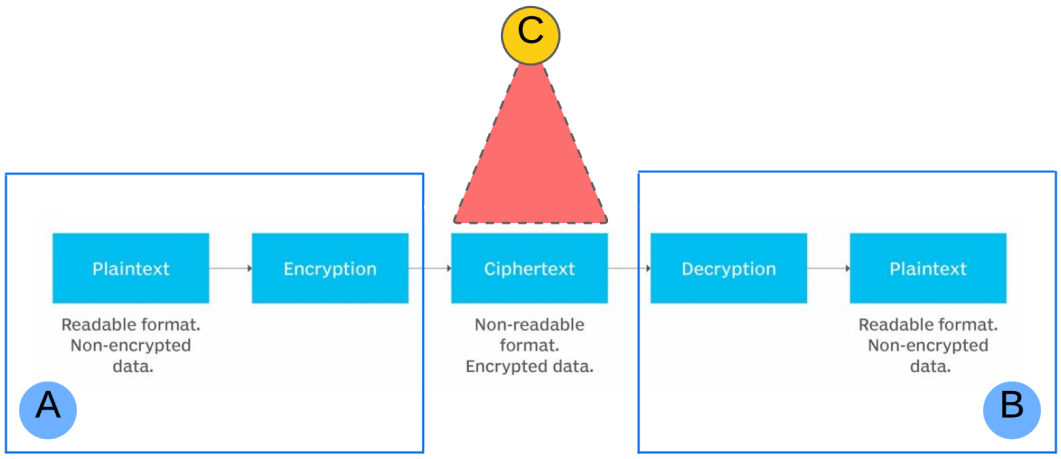

Figure 1.1: Scheme of a cryptographic communication

Figure 1.1 shows the general concept of cryptography.
Consider a scenario where Alice (A) wants to send a confidential message to Bob (B). If she transmits the message in its original form, known as plaintext, anyone with access to the communication channel could intercept and read it. To protect the message, Alice can encrypt the plaintext using a cryptographic key, producing a ciphertext. This ciphertext, which appears unintelligible without the key, is then transmitted to Bob.

Upon receiving the ciphertext, Bob uses a corresponding key to decrypt it and recover the original plaintext. Although an attacker may still intercept the ciphertext during transmission, modern cryptographic algorithms are designed so that, without the decryption key, recovering the original message is computationally infeasible using current technology.

However, this security model is threatened by the emergence of quantum computers. Quantum algorithms can efficiently solve certain mathematical problems, like integer factorization and discrete logarithms, which underpin the security of many widely used cryptographic systems [3]. As a result, much of today's public-key cryptography could become vulnerable in a post-quantum world.

To address this risk, researchers have developed post-quantum cryptographic algorithms, designed to remain secure even against adversaries equipped with powerful quantum computers [4, 5]. A major focus today is optimizing these new algorithms—reducing their computational and memory complexity—so that they can be feasibly implemented on constrained platforms such as embedded systems and IoT devices.

While the exchange described above primarily relates to the problem of key encryption—further explored in Chapter 4.1, which discusses both symmetric and asymmetric cryptography—cryptographic algorithms serve several other essential roles. For instance, they ensure data integrity and authentication. Cryptographic hash functions generate fixed-size digests that can be used to verify whether data has been altered. Extendable-output functions (XOFs), such as SHAKE, offer flexible output lengths and are especially useful in applications like key derivation, pseudorandom generation, and masking [6].

Together, these cryptographic tools form the backbone of secure communication in both classical and post-quantum contexts, making them more critical than ever.

1.3 Optimisation of an application

Optimise an application refers to the process of improving its efficiency and effectiveness, leading to better performance and resource use. In electronic systems, this means reducing the weight of the devices' constraints, such as limited computational power, low memory or power consumpation.

There are three categories of optimisations:

- Software optimisations try to make the application more efficient by changing what it does. It intervenes in the algorithm applied, the use of memory, and the internal structure of the program.
- Hardware optimisations try to make the application more efficient by changing the device or the environment it run on. To do so, they change the resources

of the devices, add new computation structures and modify their organisation

- Hybrid optimisations are a combination of hardware and software optimisations

Software optimisations are quite hard to archieve on post quantum cryptographic algorithms. The code is designed to be complex and difficult to simplify. They make of their complexity a strong point to resist attacks, characteristics difficult to maintein by changing algorithms or software code. On the other hand, hardware optimisation can be very effective, in particular on Embedded systems. When a microcontroller is present, modifying the hardware, adding new structures, and widening the instruction set on which the microcontroller can operate can make the system more specific and efficient in the execution of the application.
The flow which is usually followed to implement an optimisation as the one mentioned, is the following:

- The application is studied to find bottlenecks, frequently used functions, and instruction patterns
- New specific hardware is developed for the application
- New hardware is synthesised and tests are performed on the new architecture [7]

The figure 1.2 schematises this approach. This approach has been proven to be very effective in obtaining large speed-ups and reducing power consumption.
On the other hand, it requires deep knowledge of the application and is not very flexible: the same pattern may appear in other code or functions, but a new case study would be required.

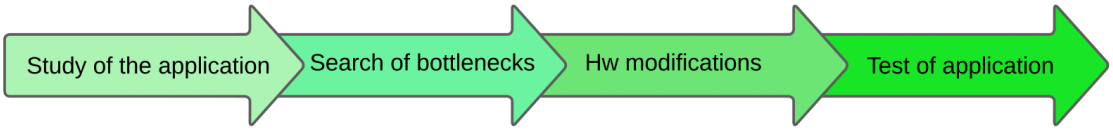

Figure 1.2: Flow of a hardware optimisation

1.4 Thesis purpose

This project aims to develop a tool, written in Python, that disassembles functions of an application and analyses their assembly code to identify instruction sequences suitable for extension as custom RISC-V instructions. The goal is to create optimized, RISC-V-compliant extensions that improve performance for specific cryptographic operations, advancing efficiency and accelerating computations in high-complexity algorithms. In other words, and in opposition to what is described in Section 1.3, the tool should work using a generic approach that limits the need for human effort and knowledge of the specific operations of the algorithm. This will eventually provide information for starting a standard case study based on the results obtained.

1.5 Thesis organization

The thesis will be organized as follows:

- **Chapter 1** is a short introduction about the motivations and the purpose of this thesis;
- **Chapter 2** describes the tools and software that have been used during this work. This includes the description of RISC-V and its instructions, the X-HEEP tool, and some information regarding compilation and the assembly code.
- **Chapter 3** introduces CIRCO, the tool developed in this work. Section 3.1 provides a global overview, while the subsequent sections explain the individual components in detail, focusing on the aspects addressed and the issues resolved.
- **Chapter 4** presents Kyber, a post-quantum cryptographic application. The algorithm is described in Section 4.1, while in the following sections, optimisations are applied to the algorithm. First, in Section 4.2, a study that follows the traditional approach is reported; then CIRCO is applied to Kyber step by step (Section 4.3).
- **Chapter 5** compares the results of the different optimisations. Possible improvements of the tool are described in Section 5.2.
- **Chapter 6** is the conclusion of the thesis, providing reflections on this work

while summarizing all the outcomes.

Chapter 2

Tools and Environment

2.1 RISC-V

RISC-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. The Github repository is accessible at `https://github.com/riscv`.

This processor family is open source, which means that anyone knows the exact composition of the hardware and can change it to generate a new optimised version for their applications. They can then freely produce, sell, and test the new processor without the need for permissions or licenses.
RISC-V is a pipelined architecture. Instruction execution is divided into consecutive stages. At each clock cycle, the instruction proceeds to the next stage, and that stage begins processing the next instruction.
The number of pipeline stages can vary depending on the specific implementation of RISC-V, but a typical configuration includes five stages, with the following breakdown of instruction execution:

- **Instruction Fetch (IF)** stage: the CPU retrieves the next instruction from memory.
- **Instruction Decode (ID)** stage: the instruction is decoded to determine the operation and operands. If the operands are registers, they are read from the register file.
- **Execute (EX)** stage: the ALU (Arithmetic Logic Unit) of the processor

executes the operation specified by the instruction.

- **Memory (MEM)** stage: if the instruction reads from or writes to memory, the memory access is performed.
- **Write Back (WB)** stage: the result of the operation is saved in the register file at the location specified by the destination register.

The ALU contains the hardware structures required to execute any computation defined by the instruction set architecture (ISA). When more complex computations are required, they are divided into multiple instructions executed sequentially, with partial results stored in the register file.
The RISC-V ISA is modular, meaning it is based on a core set of instructions with optional extensions. This allows users to add extension instructions that the processor can decode, along with hardware structures in the ALU to execute them, making the system more efficient for application-specific tasks.
The open-source nature of RISC-V includes dedicated opcodes that enable the addition of custom instructions.

The architecture used in this project is RV32IMC.
Memory addresses are 32 bits long, and the base integer instruction set (I) works on registers of 32 bits. Other instructions already present are from the M extension (Standard Extension for Integer Multiplication and Division) and the C extension (Standard Extension for Compressed Instructions).

In this work, it has been decided to implement instructions that are compliant with the RISC-V standard extension, which means they respect the following requirements:

1. The ISE must align with the wider RISC-V design principles. This means it should favor simple building-block operations and use instruction encodings with at most 3 source registers and 1 destination register. This avoids the cost of a general-purpose register file with more than 3-read ports or 1-write port.
2. The ISE must use the RISC-V general-purpose scalar register file to store operands and results, rather than any vector register file

3. The ISE must not introduce special-purpose architectural state, nor rely on special-purpose micro architectural state (e.g., caches or scratch-pad memory)

The tool focuses on logic and arithmetic patterns, searching for new instructions to implement in order to speed up execution. Each new instruction introduces a dedicated structure in the execution stage for its computation.

RISC-V defines different instruction formats.
In the base ISA, arithmetic or logic instructions between registers are decoded in R-format. They have two source registers that are used in the computation of the result, and then saved in the destination register. This format is the primary choice for the implementation of custom instruction by the tool.
Another format that will be used is R4.
In R4 instructions, the function field of 7 bits is split to include a third source register, and the remaining two bits still classify instructions. The additional register allows the tool to widen the set of instructions that can be merged.
In Figure 2.1, a comparison of the two encodings is reported.

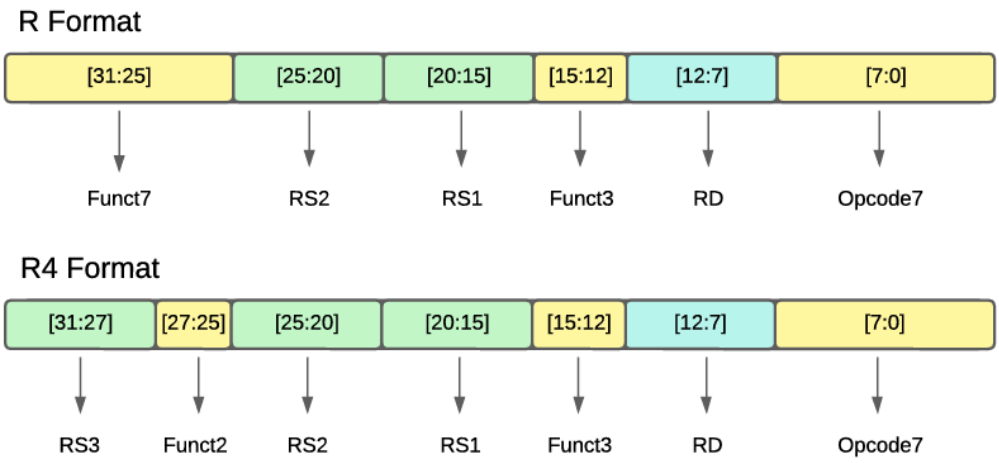

Figure 2.1: R and R4 formats

2.2 X-HEEP

X-HEEP (eXtendable Heterogeneous Energy-Efficient Platform) is a RISC-V microcontroller described in SystemVerilog. It provides a simple but complete MCU (CPU, common peripherals, memory, etc.) predisposed to be extended with an accelerator or a coprocessor [8].
The configuration of the CPU subsystem with the specific core CV32E40PX supports the CV-X-IF (Core-V eXtension Interface) standard.
The structure of X-HEEP is shown in 2.2.
Through this interface, a coprocessor can receive instructions, register values, and status information about the execution of applications on the main core.
The signals used in the interface are reported in Table 2.1. The packet data signals are put together with ready/valid signals that are used to implement the synchronisation between the main core and the coprocessor, ensuring integrity in the execution flow.
In Figure 2.2, the interface scheme is shown, while in 2.3 the timing diagram of a communication is reported.

Signal class	Signal name	Src->Dst	Description
Issue	issue_valid	RISC-V -> coproc	The RISC-V want to issue a new instruction to the coprocessor
	issue_request	RISC-V -> coproc	Packet data with instruction, [instruction ID and Hardware ID]
	issue_ready	Coproc -> RISC-V	The coprocessor is ready to elaborate a issue
	issue_respond	Coproc -> RISC-V	Packet data to reply to issue (accept bit, write back is necessary, need to read specific register)
Commit	commit_valid	RISC V -> coproc	Commit valid to the coprocessor, it tells that the instruction is definitive (not speculative)
	commit	RISC V -> coproc	Packet data with information on commit kill command , [instruction ID and Hardware ID]
Register	register_valid	RISC V -> coproc	The RISC-V has a register value ready to be sent to coprocessor
	register	RISC V -> coproc	Packet data with register value
	register_ready	Coproc -> RISC-V	The coprocessor is ready to receive the register value
Result	result_ready	RISC V -> coproc	The RISC-V is ready to receive the result
	result	Coproc -> RISC-V	Packet data with results or exceptions
	result_valid	Coproc -> RISC-V	The coprocessor has result ready

Table 2.1: Signal of CV-X-IF interface

When a custom instruction that is not present in the ISA of the main core but described in the coprocessor is detected, the coprocessor substitutes the main ALU by computing and providing the result to be saved in the register file at the next clock cycle.
This make the coprocessor tightly coupled with the RISC-V core. The coprocessor shares resources such memory and register files with the main core and enables the possibility of adding custom instruction without altering the pipeline or the toolchain of the RISC-V core. The support for Verilator and QuestaSim allows having a tool to compile an application on RISC-V and to simulate its code behavior

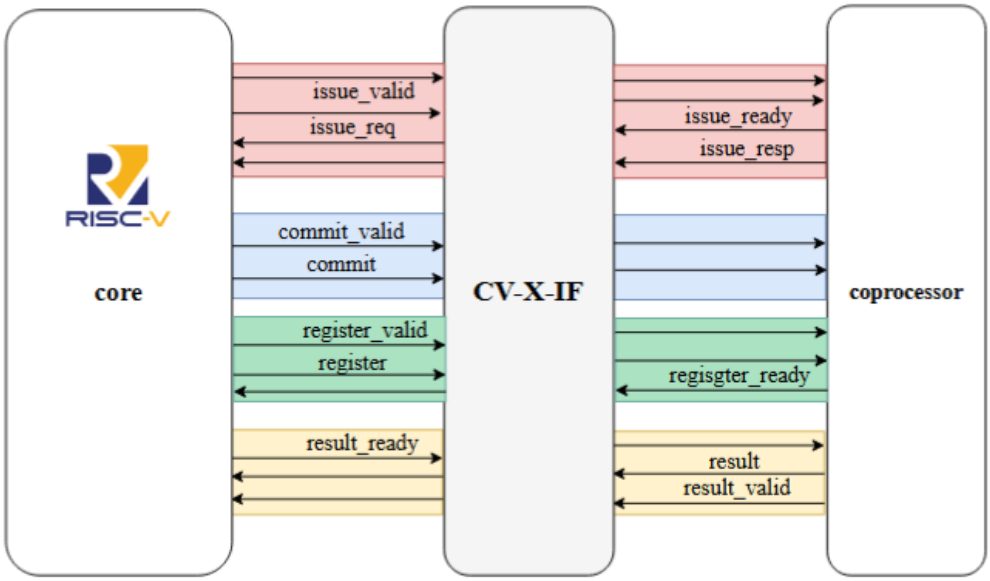

Table 2.2: Structure and connection of CV-X-IF interface

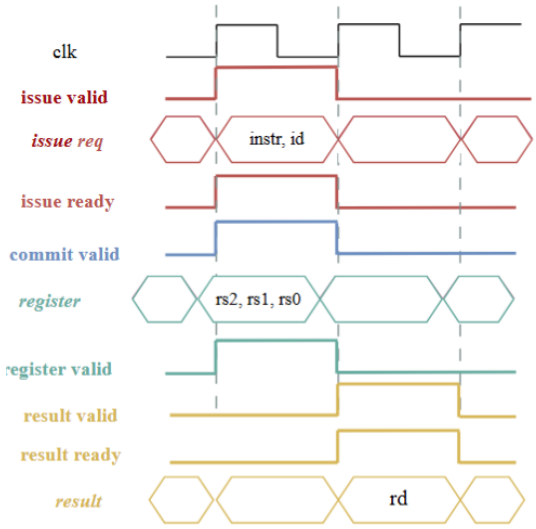

Table 2.3: Timing diagram of the issue of an instruction

before and after custom modification.
It is also possible to have a cycle-accurate counter to compare the two simulations.

The full documentation of X-HEEP can be found at `https://x-heep.readthedocs.io`.

X-HEEP project repository is at `https://github.com/esl-epfl/x-heep`.
CV32E40PX project repository is at `https://github.com/esl-epfl/cv32e40px`.
XIF project repository is at `https://github.com/openhwgroup/core-v-xif`.

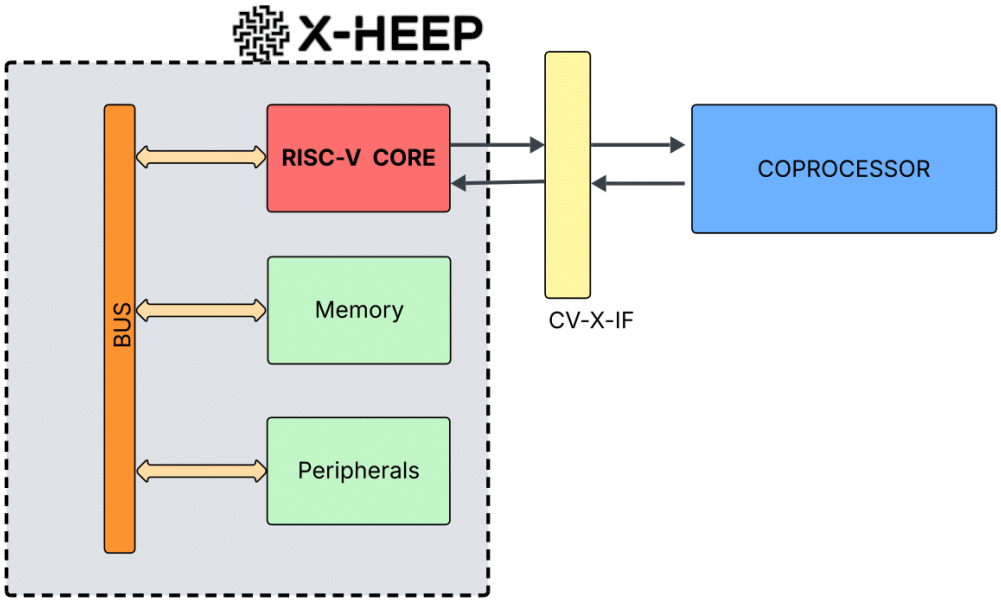

Figure 2.2: Block scheme of X-HEEP and coprocessor

2.2.1 Coprocessor

The coprocessor used in this project was developed by the VLSI Lab at the Politecnico di Torino, inspired by the coprocessor developed in CVA6 repository which is accessible at `https://github.com/openhwgroup/cva6`.
The block scheme of its structure is shown in Figure 2.3.
When a custom instruction is fetched by the RISC-V core, the following steps are performed:

1. The interface issues the instruction with a validation signal to the coprocessor.
2. The decode stage (ID) of the coprocessor acknowledges the issue, decodes the instruction, and raises the signal for the write-back of the operation to the interface.
3. The instruction selected and operands from the decode phase are sent to the implementation blocks of the custom instructions.

4. The selected block produces the result that is sent to the commit stage.
5. The commit stage forwards the result to the XIF.
6. The result returns to the RISC-V core and is written back to the register file.

The operands can be the values of the registers from the register file of the RISC-V core or the values present in the instruction for short immediate operands.
The structure has been updated to carry on consecutive custom instructions, with the possibility of internal forwarding of the output result from the commit stage.

A pipeline register is present in the decode stage to separate the issue and decode stages from the commit and computation stages.

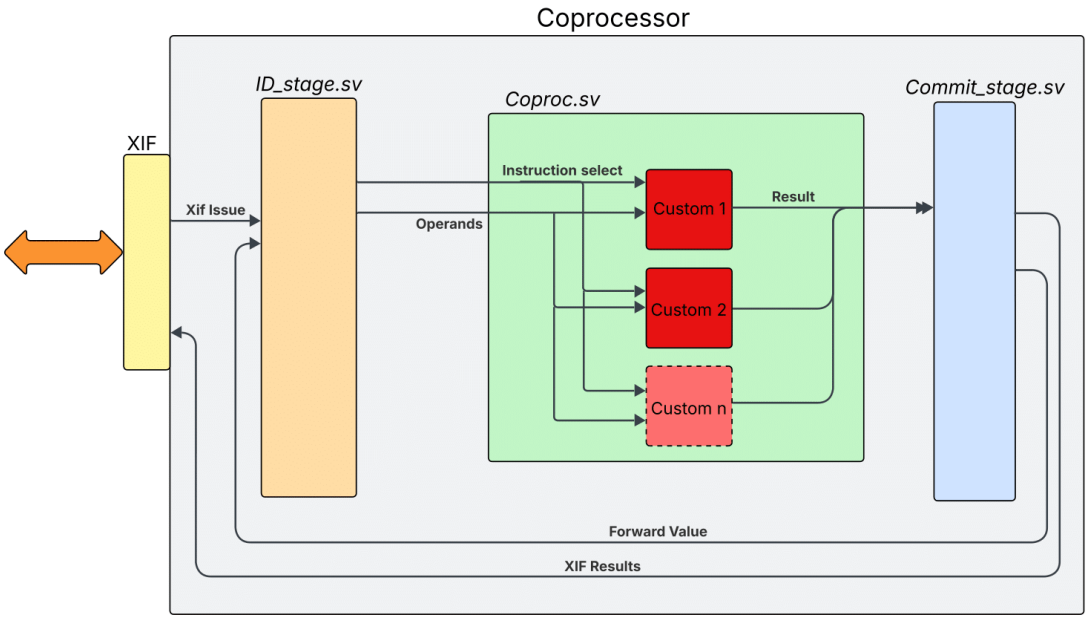

Figure 2.3: Block diagram of the coprocessor's structure

2.3 Compilation, assembly code and ASM inline

A processor is able to execute machine code, which contains only the instructions defined in the processor's Instruction Set Architecture (ISA), written in binary.
An application written in a high-level language, such as C, must be compiled in order to be executed. The compilation process, illustrated in Figure 2.4, is a series of transformations that, starting from the source code, generates the machine

code. The collection of software tools used for these transformations is known as a toolchain.
For a C application, the steps are as follows:

1. Preprocessing: The preprocessor reads each source file (.c), interprets the directives (the lines starting with "#"), and substitutes the corresponding text. In this phase, the header files (.h) are included. The output file is still a .c file.

2. Compilation: The compiler translates the C code into assembly language, a low-level language specific to the ISA of the target processor.

3. Assembly: The assembler takes the assembly files and generates object files, which are written in machine code.

4. Linking: The linker combines all the object files from the application and the used libraries into a single executable file.

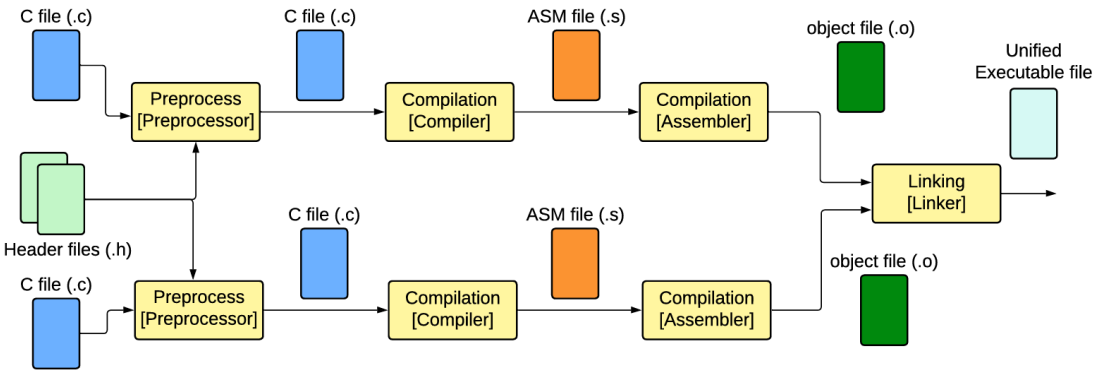

Figure 2.4: Compilation process of C source code

One possible format for the executable file is the ELF (Executable and Linkable Format). In this format, in addition to the machine code, other information is stored, such as debugging information, symbol tables, relocation data, and section headers that describe how code and data are organised within the file.
From the ELF file, a hex file can be generated. This final file contains the memory content to be loaded onto the embedded system.

The GCC toolchain [9] has been used to obtain the elf file and with the objdump command is possible to return the assembly, in the process called disassembly.
The reason that assembly code is used is because is a human-readable format and, in general, each instruction in assembly corresponds to a single instruction

in machine language. In other words, there is a one-to-one relationship between assembly instructions and the instruction set architecture, allowing the tool to study hardware optimisation from assembly code.

Code including modifications and custom instructions should be recompiled to correctly run the new application. ASM inline can be used to incorporate assembly language instructions directly into C source files.
An example of a function written with ASM inline is the following:

```
void example_function(){
    asm volatile(
        "asm_instruction \n\t"
        "asn_instruction \n\t"
        [...]
        "asm_instruction \n\t"
    );
}
```

The compiler just translates the function into machine code and integrates it with the rest of the code.
ASM inline allows the programmer to add a new custom instruction that is not known by the compiler by using the ".insn" directive.
An example is given below.

```
".insn r 0x6b, 0x1,0x2, s3,s4,s5,s6  \n\t"
//r4 instruction in asm-inline, in the example:
//0x6b -> opcode7
//0x1 ->funct3 field for r4 instruction
//0x2 ->funct2t field for r4 instruction
//s3 -> destination register
//s4,s5,s6 -> source registers
```

A more in-depth description of how the tool rewrites the C source files is provided in 3.2.4, while the description of the commands used to compile the applications is provided in 3.2.1.

Chapter 3

The tool CIRCO

In this chapter, the CIRCO tool is introduced.
Section 3.1 outlines the core concept underlying the tool, while Section 3.2 details its implementation and the data structures involved in the process.
Throughout the chapter, filenames, programs, and procedural steps are referenced with respect to the flow diagram shown in Figure 3.2.

3.1 CIRCO: a general overview

CIRCO (Custom Instruction RISC-V Code Optimizer) is the tool developed in this work. It automatically analyses a C application's assembly code to identify patterns of instructions that can be merged into new, custom RISC-V-compliant instructions. The tool generates a solution: a set of possible custom instructions that can be added to the ISA to improve its performance in the execution of the application. Then, it implements this solution by rewriting the C source and header files as a recompilable version using asm inline.

The software is written in Python and is divided into three separate parts that must be run in sequential order to get the optimised version of the source code. Each part uses files for configuration and application data import, and produces files with partial results of the execution as output. These files are human-readable and users can modify them to interact directly with the successive phase, changing the behaviour of the software and exploring different solutions or implementations. This makes the tool flexible, as it can be used as a first approach to a new application

or on top of an already developed optimised version.

Understanding how CIRCO works can be done by starting from a simplified problem. Let us imagine having a function with the following code:

```
instruction A
instruction B
instruction C
```

Figure 3.1 illustrates some alternative optimisations.
At this level of description of the instructions, it is not possible to distinguish which optimisations are possible or preferable. In a realistic analysis, when multiple instructions are taken, many more constraints must be considered.

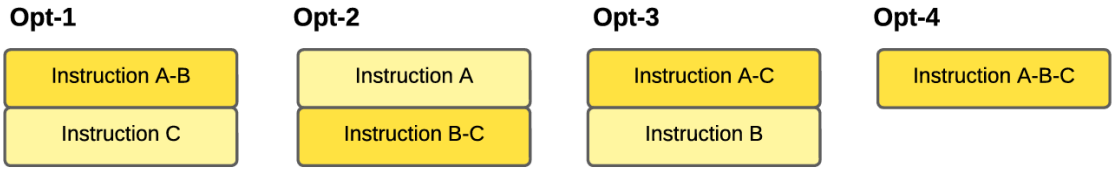

Figure 3.1: Possible union of instructions

Not every instruction can be merged without altering the standard structure and execution of the RISC-V architecture. Different implementations of load/store or branches would not be RISC-V compliant.
In this work, only arithmetic or logic instructions are considered as candidates for instruction unions. Such a union can be implemented by adding a hardware block to the coprocessor, which executes the individual instructions of the union either in parallel or in sequence in a single clock cycle. These instructions have source and destination registers.
Custom instructions must also have source and destination registers. The number of destination registers should equal one, while the number of source registers cannot exceed three (by implementing R4-format instructions) to adapt to the standard ports of the RISC-V register file.

Now, another example is reported, where the information about registers is present:

```
lw r2, 15
and r1,r2,r3
or r2, r1, r4
xor r2, r2, r4
```

In this case, the second and third instructions can be merged into a new instruction that computes the OR operation between the output of an AND operation and a register. The custom instruction would result in "Custom1 r2, r2, r3, r4" which translates to "OR (AND (r2, r3), r4)".
This optimisation would eliminate the partial result of the AND, so it is possible only if there is no successive operation that uses the register 'r1' as source.
The same applies to the third and fourth instructions with a custom instruction with the syntax "XOR (OR (r1, r4), r2)".

This simple example shows that working unions may not be implemented in the same solution. This creates what will be called a conflict: the first and second unions share an instruction. If the first union is applied there would not be the third instruction to be merged with the fourth. If conflicts are not solved correctly, the code will not execute the same operations as the original application.

The possibility of merging two or more instructions depends on multiple factors:

- **Instruction semantics and context**: The functional behavior of the instructions to be merged and any instructions in between or following them must allow for a safe and meaningful combination.
- **Register constraints**: The total number of registers involved must be limited, so that they can be encoded within the available fields of a custom instruction's opcode format.
- **Data availability**: All registers used by the merged instruction must be properly loaded with valid data before the execution of the first instruction in the sequence.
- **Data flow preservation**: If the merge removes an intermediate result (i.e., a value that would have been written to a register), that value must not be required by any subsequent instructions.

During the analysis of the assembly code, CIRCO takes into consideration all the constraints and solves conflicts.
By doing so, it may have to make choices. In this case, it tries to maximise the number of unions, and if the solutions have the same number of optimisations, it sets one as the first solution and flags all the others as equivalent alternatives.
The solutions are reported in the results files, and the user can modify the configuration files to explore different solutions or confirm the result to obtain the updated version of the source code.

3.2 The flow

Figure 3.2 shows the flow to analyse an application.

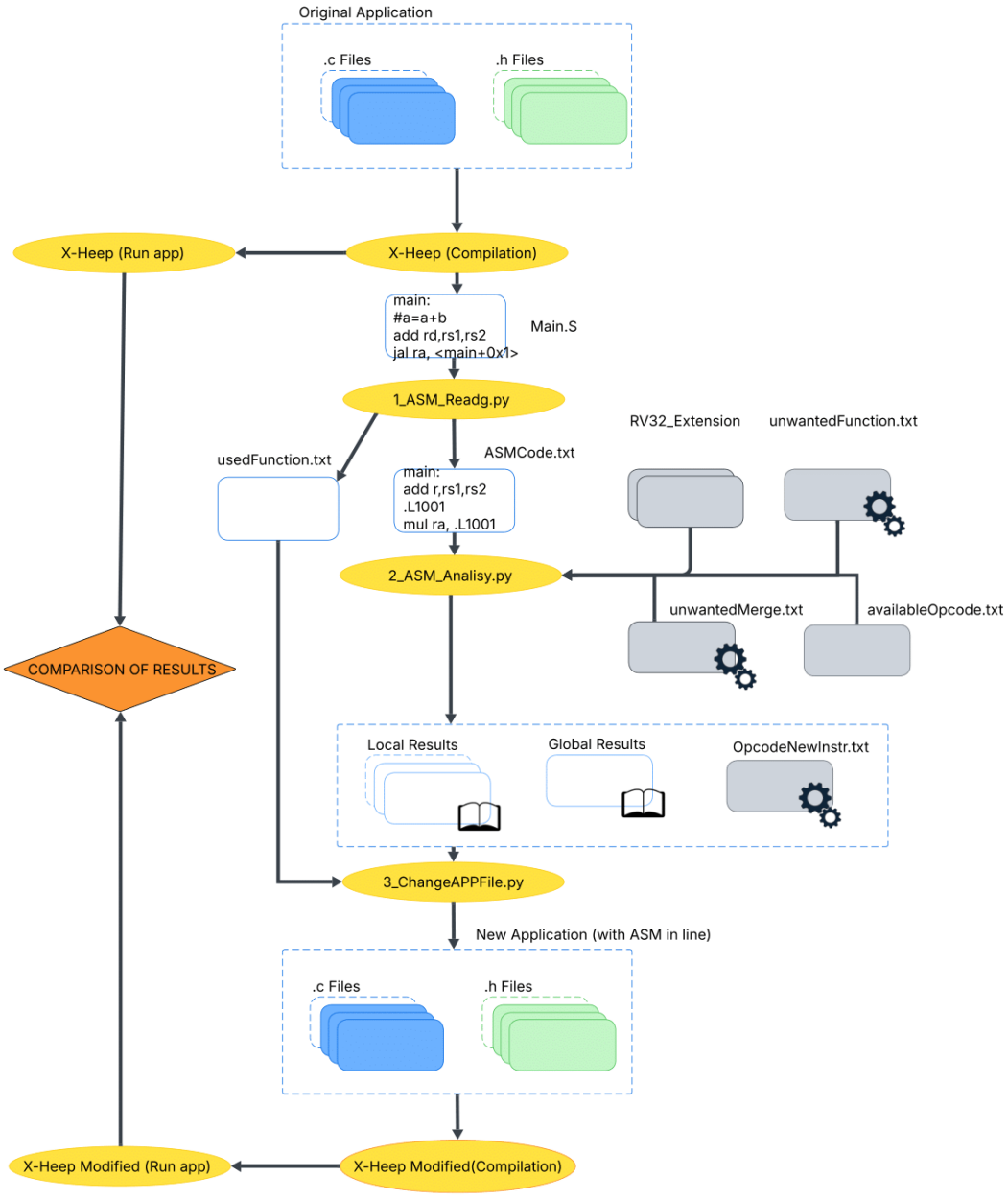

Figure 3.2: Flow of optimisation of an application with CIRCO

Starting from the compilation of the original code, the three parts of CIRCO (1_ASM_Read, 2_ASM_Analysis, 3_ChangeAPPFile) are executed, reaching at the end the optimised version.

The most important files and dependencies with the program are highlighted.
A gear symbol is placed near files that can be modified to interact with the flow of the tool, while the book icon is near files that provide information on the application and solutions found.

3.2.1 Compilation of the application

X-HEEP has been used to compile the application source code. X-HEEP uses a series of Python scripts to generate the desired configuration of the MCU, so, after setting up the environment, the microcontroller can be generated with the command:

```
make mcu-gen [...] CPU=cv32e40px [...]
```

Additional arguments can be added to modify the default values of the configuration.

The application source code can be added to the software folder, then the compilation is done with the make command:

```
make app PROJECT=<directory of source code in software folder>
```

X-HEEP calls the GCC toolchain with the command:

```
riscv32-unknown-elf-gcc -march=rv32imc [...] -O2 -nostdlib
[...] <linker options> <library includes> [...]
```

The same script calls for the objdump of the object file to obtain the disassembly. The command is of this type:

```
riscv32-unknown-elf-objdump -d  <path of object file>
> <path to destination of disassembly file>
```

The destination of the compilation is a system without an operating system, which means that all functions used from libraries must be included directly in the object code. Other functions, such as GPIO functions or peripheral handlers, are also included in the object file.

3.2.2 Part 1 : ASM Read

The first part of CIRCO (scheme in Figure 3.2) is about reading and modifying the assembly code obtained from X-HEEP.
The input file is the Assembly code obtained from the compilation of the application code. The output files from this part are:

- ASMCode.txt, a cleaned version of the assembly file.
- usedFunction.txt, a list of the function called by each analised function.

An example of the structure of a function in the assembly code is given below.

```
0000067a <FunctionName>:
************************************************
* COMMENTS from C File
************************************************
int16_t functionName(int32_t a) {
    int16_t t;              //C Code lines
    t = (int16_t)a * b;

     67a: 787d                lui a6,0xfffff
     67c: 30180793           addi a5,a6,769
     680: 831ff0ef           jal ra,231c <functionName2>
     684: 02e888b3           mul a7,a7,a4
     688: a809                 j 67c <functionName+0x4>
     88a: ret
```

The first line indicates that the subsequent operations constitute the assembly of the named function.
The hexadecimal number at the start of the line indicates the memory address at which the function's code begins. These addresses will change when the body of the function is modified and recompiled.
Comments and references to the compiled C files are also omitted from the program output.

Regarding the instruction lines, the structure is as follows:

<memory address:> <raw opcode> <disassembled instruction> <#comment>

The program has been designed to work mainly with disassembled instructions.

The same information is embedded in the opcode bytes of the instruction (e.g. the name of the instruction, the destination register, the source registers, or the immediate values), but following this approach could make the code less clear, necessitating a decoding phase to report partial results and allow the user to interact with the program's overall flow.
Working directly with opcode bytes becomes necessary when custom instructions are included in the C source code using inline assembly and the .insn directive, as explained in 2.3. In such cases, the compiler outputs only the opcode bytes of the instruction; it does not assign a name, define a syntax, or display the registers involved.
For example, the directive

```
.insn r 0x6b, 0x01, 0x0, r, a1, a1, a2, a3/n/t
```

would appear in the assembly code as

```
<addressmemory>: 3461ab6            0x3461ab6
```

with "3461ab6" the opcode of the instruction .
The opcode is the only available information. The instruction must be decoded internally during the analysis of the assembly code.

The second problem that this part of the tool solves is adjusting the jump and branch references.
As shown in the example, the file contains references to other instructions by their memory addresses.
The length and position of the code may change after recompilation. Comments provided by the compiler can be used to reconstruct the code.

When jumping to a function, the name of the called function is reported in '<>' and the address must be substituted with the name of the function.

For branches, the destination address is written as an offset from the start of the function. The program adds a label to each branch's destination and creates a dictionary of the found addresses. When writing the output, the labels are added to the code, and their names are substituted in the instructions that reference them. To avoid conflicts with labels already present and generated from code that has not been optimised, label names start from a high number (1000), with the first label being named .L1000.
The output of the example code would be as follows:

```
Function Name:
lui a6,0xfffff
Label .L1005:
addi a5,a6,769
jal ra, <Function called name>
mul a7,a7,a4
j .L1005
ret
```

3.2.3 Part 2 : ASM Analysis

The second part of the program is designed to find instructions that can be merged. After identifying this set, it determines the maximum subset of custom instructions that can be present in the final code simultaneously without altering its behaviour: it resolves conflicts and user constraints.
Finally, it saves the results in a format that can easily be used by the next stage to rewrite the application's source file.
As can be seen from the scheme in 3.2, it uses the following files as input:

- ASMCode.txt, the cleaned version of the assembly code generated by Part 1.
- Extensions file description, containing the characteristic of the instruction already present in the architecture.
- unwantedFunction.txt, a file listing the functions that the user does not want to be analysed.
- unwantedMerge.txt, a file listing the custom instruction that the user wants to exclude from implementation.
- availableOpcode.txt, a file listing the opcodes that can be assigned to the new instructions.

This part generates as output files:

- Local Results files, one for each function analised, they contain the union to implement in the solution
- Global Results file, one summary file of the custom instructions in the entire application
- OpcodeNewInstruction.txt, a file listing the opcode assigned to each custom instructions

Reading of function assembly

The analysis is performed on a function-by-function basis.
However, the compiled output file contains not only the application code: initialisation functions, handler functions, interrupt service routine functions, and included standard library functions are also present.
To avoid the analysis of functions that are not part of the application or that the user does not want analysed, a configuration file is provided.

This file, named unwantedFunction.txt, lists the names of functions that the programme skips. For all others, the programme reads the assembly code from ASMCode.txt, the output file of the first part of CIRCO.

While the first part of the programme involves methodically cleaning the assembly code by removing unnecessary information and replacing addresses with labels, this part requires characterising the instructions.
Not all instructions are the same, and not all instructions can be merged or moved without altering the behaviour of the code.
Each instruction has its own syntax, and the meaning of the registers can differ depending on that syntax.

For each extension used in the RISC-V architecture, a .csv file must be provided with the description of the instructions. In this case, the architecture uses the base integer, multiplication, and compressed extensions, which are described in RV32I.csv, RV32M.csv, and RV32C.csv.

Table 3.1 shows the description of some instructions of the integer extension.

Name	Field	Description	Can be group?	Can be moved?	Is commutative?	Takes two clock cycles?
ADD	rd,rs1,rs2	Add	Yes	Yes	Yes	No
SUB	rd,rs1,rs2	Subtract	Yes	Yes	No	No
SLLI	rd,rs1,imm	Shift Left Logical Immediate	Yes	Yes	No	No
SW	rs2,offset(rs1)	Store Word	No	Yes	No	No
BNE	rs1,rs2,offset	Branch Not Equal	No	No	No	No
BLT	rs1,rs2,offset	Branch Less Than	No	No	No	No
LW	rd,offset(rs1)	Load Word	No	Yes	No	Yes
[...]						

Table 3.1: CSV file description of ISA

In general, the following rules are used to describe the instructions:

- All instructions can be moved, except for branch or jump instructions.
- Instructions that can be merged are only those that perform an arithmetic or logic operation and whose operands can be written in the dedicated 5 bits of the R4 format. This includes registers or limited immediates (e.g. immediate shift).
- Loading from the stack (e.g.lw) takes two clock cycles before the new value is accessible by a following instruction.

- Commutative instructions are those that have two registers as sources, and switching the inputs leads to the same result without any other changes.

In the assembly code, pseudo-instructions may be present. These are special commands that are not part of the official instruction set but are translated into one or more standard instructions. Typically, this is done to make the assembly more human-readable.
An example of a pseudo-instruction is the mv operator in the instruction mv r1, r2, which the assembler translates into add r1, r2, x0, where x0 is the register containing the value zero.
A separate file is dedicated to pseudo-instructions (RV32pseudo.csv); its structure is the same as that used for standard extensions.

Finally, the assembly may include custom instructions. In this case, only the opcode byte is reported in the assembly code, and a decode phase is necessary to retrieve the registers involved.
A dedicated file is provided, including the fields for the instruction opcode and its format. The 0s and 1s in the code fields define the instruction, while the Xs represent the register bits.
The following table reports the structure of the RV32Custom.csv file:

Name	Syntax	Description	Can be group?	Can be moved?	Is commutative?	Takes two clock cycles?	Code
Cus1	R4	Custom instructionR4	No	Yes	Yes	No	XXXXX00XXXXXXXXX001XXXXX1101011
Cus2	R4	Custom instructionR4	No	Yes	Yes	No	XXXXX01XXXXXXXXX001XXXXX1101011

Table 3.2: CSV file description of custom instructions

The tool reads the lines of each function. For each one, it creates an InstructionClass, which includes the name of the instruction and the registers as parameters.
Finally, it appends the class to the function list, which is analysed by the union algorithm.

Union Algorithm

Each instruction is analysed, starting from the beginning to the end of the function. There are two possibilities:

- If the instruction cannot be merged (e.g. branches or loads), it is skipped and the analysis moves to the next one.

- If it can be merged, it is selected as the first element of a possible union. Two sets of registers are allocated: one stores the destination registers of the instructions in the union list, and the other stores the source registers.

In order to be added to the union list, an instruction must meet this set of criteria:

- It has to be a mergeable instruction.
- It should not follow a jump or branch instruction, as these instructions alter the execution flow of the code. This means that an instruction placed immediately after a jump or branch may not be executed, depending on the control flow. Including such an instruction in the union could change the program's behavior by executing an instruction that, in the original code, would not have been executed when the jump or branch is taken.
- The number of registers added to the custom instruction should be lower than the maximum number of source registers minus the one already used by the instructions in the union (the representation of the union instruction must be possible with maximum three source registers).
- If it adds registers to the source set, they have to be ready before the first instruction present in the union list. If the load is from stack, it takes two clock cycles to execute, so the instruction before should also not be a load of these registers.

If all the criteria are met, the instruction is added to the union list and the source and destination register sets are updated. If it is not and the instruction uses a register that would disappear with the union, the union set is updated, removing the already added instruction and its registers.

The source and destination sets shrink during the analysis of following instructions because if a register is used as destination it loses correlation with the previous usage and can be freely used from that point on. When all the registers in these sets have been reused as destinations, each following instruction is independent and the search for union instructions ends.

From the sets of possible unions, a tree is built with the dependencies of the instructions. Each subtree of this structure is a possible union.

The algorithm continues by taking the following instruction to be the first part of a new union until the end of the function.

The result is a list of possible unions, with each union containing the indices of instructions that belong to that union.

Generation of the new custom

The class NewInstruction has been used to describe custom instructions. Figure 3.3 shows a possible instance of the class.
It has as main parameters the operation, the destination registers, and one or two operands.
The operands can be objects of the class as well.
This approach enables the construction of a complex instruction starting from its primitives.
The first step is to take the last instruction of the set and use it to generate a new instruction class. Then, the preceding instructions are analysed and added by replacing the source registers of the custom instruction object produced as output by the last instruction. The usage of a class and its methods simplifies functions such as the generation of new names and syntaxes or the comparison with other custom instructions.
A dictionary maintains the correspondence between the actual register and some tokens ("_A_", "_B_", "_C_"), and it is updated any time the instruction is updated (e.g. change of an operand).
The syntax is generated with these tokens, and the print is implemented with a recursive function: the function prints the operation and the operands of the object; if an operand is an object, it calls its print method.

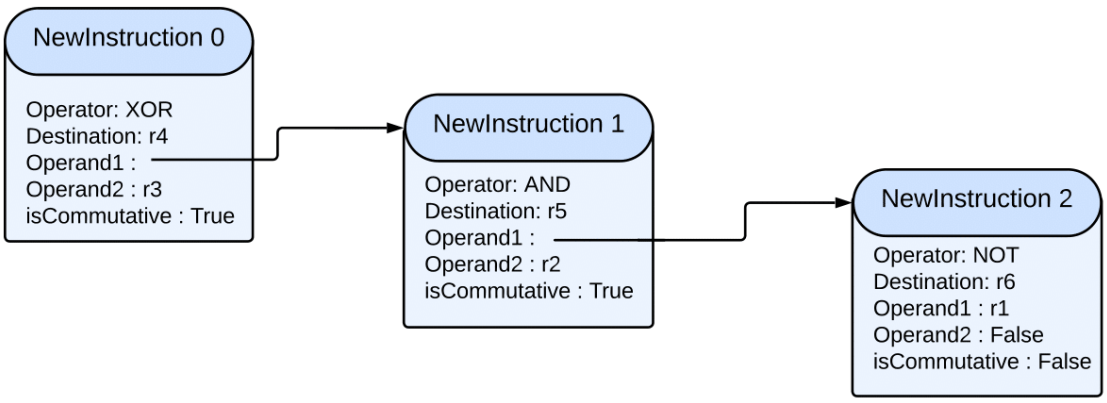

Figure 3.3: Structure of a custom instruction

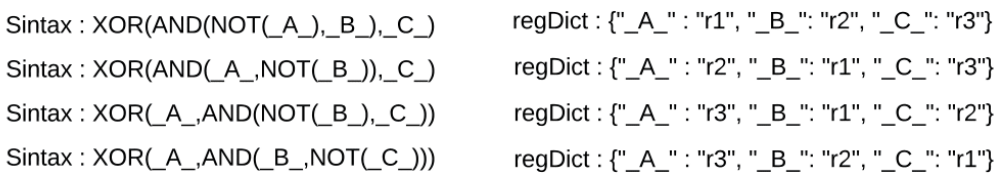

Figure 3.4: Different syntaxes for the same custom instruction

As can be seen in Figure 3.4, different syntaxes may represent the same logical instruction, only with exchanged registers. Noticing this allows the tool to group identical instructions, avoiding double implementations and producing clearer results.
To do so, the commutative parameter is included as information in each instruction described in the ISA.
When a new instruction is generated, it is compared to all the others: each possible permutation is applied to determine if they are the same instruction. If so, the structure is saved as the one already known.

Conflicts resolution

The lists of possible unions may not be included in the same solution. This is because they may share one or more instructions.
The union lists are divided into conflict sets.
Unions present in a conflict set share at least one instruction with another union in the same set, and they are independent of all the other conflict sets.
Each set has at least one solution: a subset of unions that can be simultaneously present in the new code without altering its behaviour.

The algorithm used for searching unions works in such a way that the following conflicts can occur:

- A in B (e.g. unions [1, 2] and [1, 2, 3]).
  One union includes the other; only one of them can be applied to the code.
- End-end (e.g. unions [3,4] and [2,4]).
  Unions share the last instructions; only one of them can be applied to the code.
- Begin-begin (e.g. unions [1,2] and [1,3]).

Unions share the beginning instructions. In this case, when one merge is applied, one or more source registers used by the upper part of the second union are no longer present. The possibilities are either both optimisations present or none of them.

- Begin-end (e.g. unions [1,2] and [2,3]).
  One union shares the last instructions with the beginning ones of another. Only one of them can be applied to the code.

The scheme in Figure 3.5 shows the division into conflict sets of a union list and the correlation between instructions.

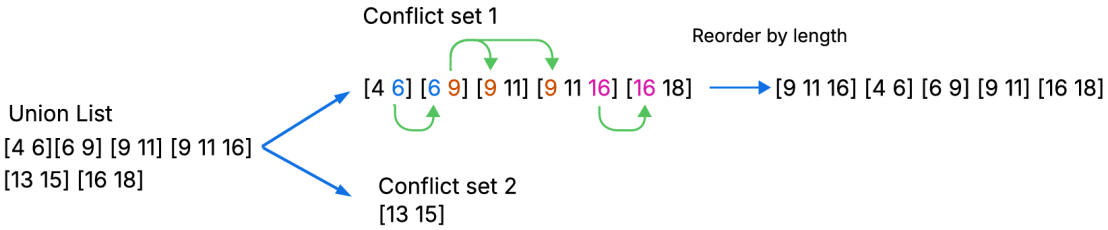

Figure 3.5: Example of division into conflict sets

During this phase, constraints on which custom instructions may appear in the final code are also considered.
The user may want to exclude some custom instructions for exploring different solutions or because there are too few occurrences and their implementation is not convenient.
By writing the names of the unwanted custom instructions in the file unwanted-merge.txt, they will not be present in the solution.

The metric used to prefer one solution over another is the number of unions applied, with the tool trying to maximise this number. If more than one solution has the same number of optimisations, it selects the first solution with that number met as taken and the others as alternative solutions.
Unions in the conflict set are ordered in such a way that, when possible, the algorithm takes the solution that merges more instructions.

The solving of a conflict set is complex. A combination approach has been chosen. Starting from the number of unions in the set, a test vector of the same length is created. This vector contains only 1s or 0s, with each index assigned to one union in the conflict set.

A '1' means that the union is applied; a '0' means that it is not. Compliance with conflicts is tested later.
If n is the number of unions in the conflict set, the test vector can be seen as the digits of the binary number x on n bits.
The first test vector is $x = 2^n - 1$ (all ones). If the constraints are not met, x is decreased by one (all ones but the last), and so on until a solution is found.
Test vectors that have fewer unions than the best solution already found are discarded without verifying constraints on conflicts.
A solution is always found: in the worst case it will be the one with all zeroes, meaning that there are no unions of the ones selected in the conflict set.
In Figure 3.6, an example is reported.

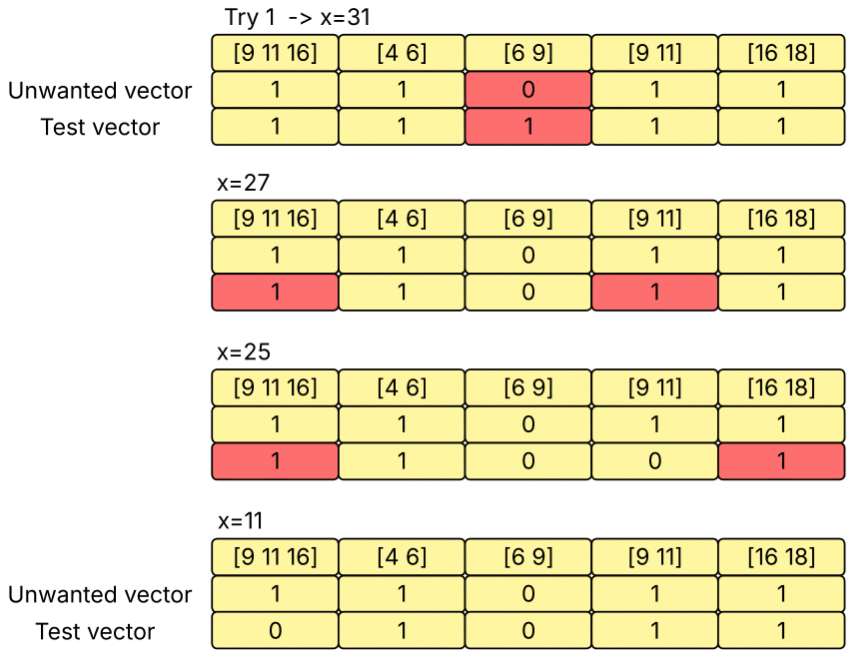

Figure 3.6: Example of attempts to solve a conflict set

In this case, the third union would translate to a custom instruction that the user has marked as unwanted.
In the first attempt ($x = 31$) with all unions present, the unwanted constraints are not met; x is reduced and the new attempt is processed.
When $x = 27$, the problem with the third union is solved, but there is a conflict

between the first and fourth union: since the fourth instruction is included in the first, they cannot both be implemented.
At attempt 25, there is a begin-end conflict between the first and the last.
Finally, when $x = 11$, the test vector does not generate any conflicts, and the solution is found with three out of five unions implemented.
This solution is saved in the result file.
In this case, there are no equivalent three-union solutions.

Saving of results

The assembly analysis ends with the results of each conflict set being saved.
For each analysed function, a dedicated result file is generated. It contains, for each union, the indices of the primitive instructions, the syntax of the new instruction, and the registers involved.
When more than one solution is present for the same conflict set, the alternative solutions are commented.
The names of the local result files are '<functionName>_unionInstr.txt'.
An example of a local result file is the following:

```
# [instructions of the union]
# Syntax of the instruction
# Destination register, Tokens source registers dictionary

#Conflict found - Unions taken
[1, 2]
OR(AND(_A_)(_B_))(_B_)
a4, '_A_': 'a4', '_B_': 'a3'
#Conflict - Alternative Solution
#[2, 3 ]
#XOR(AND(_A_)(_B_))(_C_)
#a4, '_A_': 'a4', '_B_': 'a3', '_C_': 'a5'
    [...]
```

These files are called 'local' to differentiate them from the 'global' results file, which is a summary of the optimisations for the entire application.
In the global file, each new instruction has an entry, including its syntax and the number of recurrences in the entire application, followed by details of the recurrences in each function.
The name of the global results file is 'GlobalResults.txt'.
An example of the structure of the global results file is the following:

```
OR(AND(_A_)(_B_))(_B_): 3
    FunctionName1(2)
    FunctionName2(1)

XOR(AND(_A_)(_B_))(_C_): 6
    FunctionName3(6)
    [...]
```

The file OpcodeNewInstruction.txt is filled with the syntaxes of each new instruction, and an opcode is assigned from the AvailableOpcode.txt file. The user can manually change the fields of the opcode to match the description of instructions already implemented.

3.2.4 Part 3: Change application files

The third and final part of the tool takes as input

- the local results files generated from the analyse of the assembly
- OpcodeNewInstruction.txt, the list of opcode assigned to each custom instructions
- The application source code (C files and headers)

to generate the new version of C code including the custom instructions.

The program reads each C file and, whenever it finds a function, it looks for a local results file that includes the name of that function. If so, it reads its assembly code from the ASMcode.txt file (the simplified disassembly of the application from Part 1), modifies the assembly by adding all the custom instructions present in the local results, and replaces the body of the function with inline assembly.
The tool stores each instruction in a list together with its index number.
The optimisations are read from the local results file. For each of them, the merged instruction indices are removed from the list, and the custom instruction is inserted in place of the primitive instruction with the lowest number.
The .insn statements are built using the fields in the file OpcodeNewInstruction.txt generated in Part 2.
If there are no optimisations for the function, the code of the function is copied from the source file.
Comments and directives are copied as they are into the new C files.

The header files are also modified.
The compiler does not parse the inline assembly, but only copies it into the object code, translating each instruction with its opcode bytes.
The body of the rewritten function consists of inline assembly only, so the compiler may consider the function as unused and exclude it from the compiled version. The same can happen to function calls within the inline assembly: the compiler does not detect the calls and may exclude the apparently unused functions.
The used attribute is added to each modified function or to functions called by a modified function.
The information about which functions are called by a given function is saved in the usedFunction.txt file during the cleaning of the assembly file in Part 1.
The tool reads the header files and rewrites the declaration of those functions, adding "__attribute__((used))" at their end.

3.2.5 Modification of coprocessor

The coprocessor is in charge of executing custom instructions.
If an instruction is present in the compiled code, but not correctly described in the coprocessor, the microcontroller will stall.

The decode phase of the coprocessor is described as a for loop during which each custom instruction is compared with the issued instruction.
If they are equal, the instruction is selected.
A mask is applied in such a way that only the fields characterising the instruction are used to define compatibility, excluding the register bits from the verification.
The Verilog code is shown in Figure 3.7.

```
logic [NbInstr-1:0] sel;

for (genvar i = 0; i < NbInstr; i++) begin : gen_predecoder_selector
  assign sel[i] = ((CoproInstr[i].mask & instruction) == CoproInstr[i].instr);
end
```

Figure 3.7: For loop of the coprocessor's decode stage

Including a new instruction means adding a new unique case to the custom instruction list.
The definition of this list is in the coprocessor package coproc_pkg.sv, and the relevant part is shown in Figure 3.8.

```
'{
  instr: 32'b00000_01_00000_00000_010_00000_1101011,  // CUSTOM opcode
  mask:  32'b00000_11_00000_00000_111_00000_1111111,
  resp : '{accept : 1'b1, writeback : 1'b1, register_read : {1'b1, 1'b1, 1'b1},
                                    op_type : 2'b10, insn: custom_instruction, done : 1'b1},
  opcode : CUSTOM
},
```

Figure 3.8: Description of a custom instruction

Finally, the operations of the custom instructions must be described. The source

registers are used to compute the value of the destination register.
A case structure is used to assign this value to the result when the specific custom instruction is selected.

The file coproc.sv, where this final step is implemented, is shown in Figure 3.9. Examples of the description of custom instructions are reported in Section 4.3, where the custom instructions for Kyber are added to the coprocessor.

```
  //--------always-present signals declarations ------------------------------------
  coproc_pkg::out_t out;
  logic [31:0] a1, b1, c1;
  logic [4:0] rs1_immediate, rs2_immediate, rs3_immediate;

  logic [31:0] custom_instruction_result;

  //-------- input declarations ---------------------------------------------------
  //operands avaiable for the custom instruction
  assign a1 = rs_values_i.rs1_0;
  assign b1 = rs_values_i.rs2_0;
  assign c1 = rs_values_i.rs3_0;
  assign rs1_immediate=rs_immediate_i.rs1_immediate;
  assign rs2_immediate=rs_immediate_i.rs2_immediate;
  assign rs3_immediate=rs_immediate_i.rs3_immediate;

  assign custom_instruction_result = (a1 + a2 + a3);
                                //description of the custom operation with operands
  //-------- output assignment ----------------------------------------------------
always_comb begin
    // Default assignment (optional)
    out.rd1 = '0;
    // Case statement to evaluate selector
case (insn_i)
        coproc_pkg::cus_instruction: begin
          out.rd1 = custom_instruction_result;
        end
  [...]
endcase
```

Figure 3.9: Description of the new custom instruction operators

3.2.6 Run of the application

The execution of the application is carried out by the X-HEEP environment and simulated with Questasim.
Using the command:

```
make questasim-sim
```

the hardware description of the microcontroller, peripherals, and coprocessor is compiled.

After the compilation of the application, as described in Section 3.2.1, the command

```
make verilator-sim
```

starts the simulation.

Any output of the system is printed via the simulated UART and saved in the output file uart.out.
In addition to verifying that the new code functions as intended and produces the correct results for the input tests, the Control and Status Register Memory Cycle (CSR_MC) can be used to measure the clock cycles required for execution.
This register is initialised before the call to the application functions by writing zero to it, and is read before the return from main.
Comparing the value from the original version with the one obtained using custom instructions provides information about the optimisations achieved in the new code.

Chapter 4

Optimisation of Kyber

In this chapter, CIRCO is tested on an application, the post-quantum algorithm Kyber.
In Section 4.1, Kyber is presented with a brief description of how it works.
In Section 4.2, the study of some optimisations following the traditional approach is reported.
In Section 4.3, CIRCO is applied step by step with a focus on the choices made and partial results.
The comparison between the results of the two optimisations is reported in Chapter 5, along with a description of the limitations of the tool and possible improvements.

4.1 CRYSTAL-Kyber

CRYSTAL-Kyber (commonly referred to as Kyber) is a post-quantum cryptographic (PQC) algorithm designed to withstand potential attacks by quantum computers in the future. It was the first algorithm to be selected by the National Institute of Standards and Technology (NIST) as part of the standardization process for PQC algorithms.

Kyber is a Key Encapsulation Mechanism (KEM).

In asymmetric cryptography, a pair of keys is used to exchange a message: a public key is shared openly with anyone who wants to communicate with the receiver of the message, and a private key is known only to the recipient.
Secure communication works by having the sender use the recipient's public key to

encrypt the message into ciphertext. Then, the sender transmits the ciphertext to the recipient, who can use their private key to decrypt the ciphertext and retrieve the original message.
The strength of asymmetric cryptography is that it is not necessary to exchange a shared key between those who want to send and those who want to receive a message, as required by symmetric cryptography.
On the other hand, symmetric algorithms balance the secure key distribution problem with easier implementation, efficiency, and speed, making them suitable for encrypting large volumes of data [10].

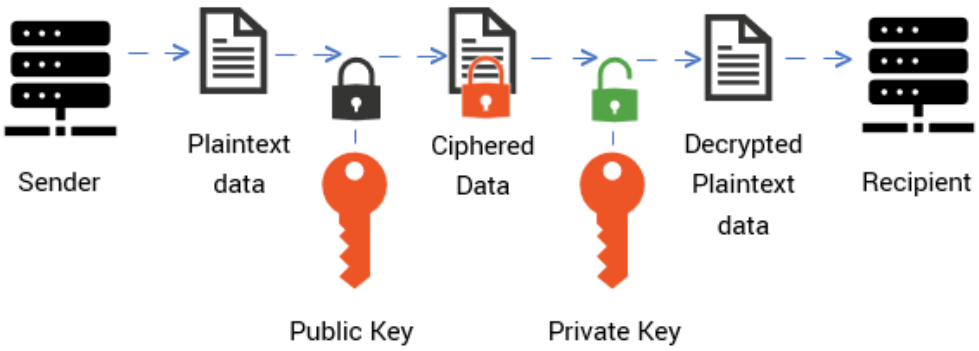

Figure 4.1: Scheme of principle of asymmetric cryptography [11]

In KEM cryptography, the aim is not to transmit a generic message but to establish a shared secret key between the communicating systems.
Public and secret keys are used to obtain and decrypt the ciphertext.
This ciphertext is called "encapsulation" and contains a secret key that can later be used to share messages with symmetric cryptography.

The high-level functions of Kyber are:

- KeyGen() : generates the public and private keys (pk,sk)
- Encrypt(pk) : generates a random value shared key (ss) and encrypts it to ciphertext (ct) using the public key (pk)
- Decrypt(ct,sk) : decrypts ciphertext (ct) to plaintext (ss) using private key (sk)

To have deterministic behavior in Kyber, the functions have been modified not to use random values, but "Known Answer Test" (KAT) vectors. These test vectors

are default values that simulate random output but in a controlled manner.
The prototypes of the "derandomized" functions are reported in Figure 4.2.
The Kyber source code used in this work can be obtained from GitHub at `https://github.com/PQClean/PQClean`

```
int PQCLEAN_KYBER512_CLEAN_crypto_kem_keypair_derand(uint8_t *pk, uint8_t *sk,
                                                     const uint8_t *coins);

int PQCLEAN_KYBER512_CLEAN_crypto_kem_keypair(uint8_t *pk, uint8_t *sk);

int PQCLEAN_KYBER512_CLEAN_crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,
                                                 const uint8_t *pk, const uint8_t *coins);

int PQCLEAN_KYBER512_CLEAN_crypto_kem_enc(uint8_t *ct, uint8_t *ss, const uint8_t *pk);

int PQCLEAN_KYBER512_CLEAN_crypto_kem_dec(uint8_t *ss, const uint8_t *ct, const uint8_t *sk);
```

Figure 4.2: Application Programming Interface of Kyber

From the implementation point of view, Kyber uses modular arithmetic with modulus $q = 3329$ and polynomials of high degree (256). The private key is a vector of polynomials, while the public key is composed of a matrix A and a vector t, both of which are also polynomials.
Public and secret keys are correlated by the equation

$$t = As + e \tag{4.1}$$

where e is a random polynomial vector with small coefficients.

The encryption procedure calculates two values u and v

$$u = A^T r + e_1 \tag{4.2}$$

$$v = t^T r + e_2 + m \tag{4.3}$$

where:

- m is the plaintext, transformed into a polynomial and scaled modulo q by division by 2
- e_1, e_2 are two random polynomials
- r is a random polynomial vector

The original message can be retrieved by the recipient, applying (4.5):.

$$m_r = v - s^T u \tag{4.4}$$

$$m_r = e^T r + e_2 + m + s^T e_1 \tag{4.5}$$

By rounding the coefficients of this last polynomial up or down, it is possible to recover the original message.

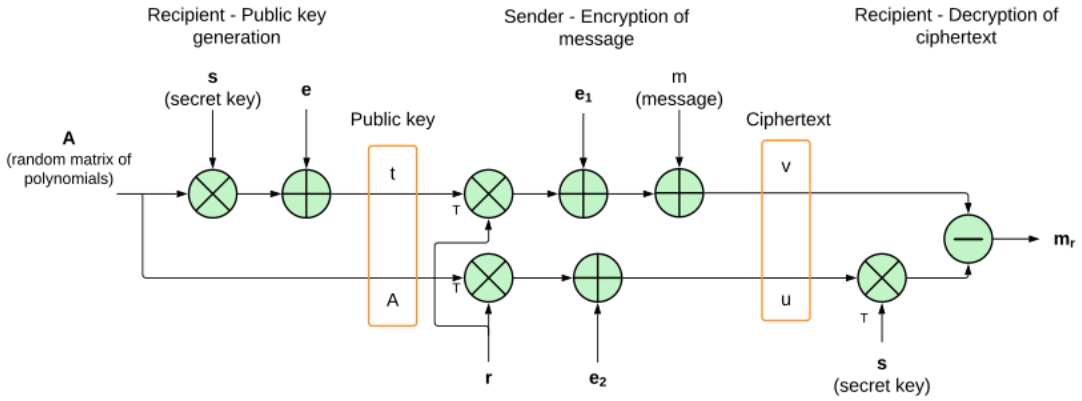

Figure 4.3: Block diagram of Kyber algorithm

These equations form the core of Kyber, which works with the module-learning-with-errors (MLWE) setting, which should be computationally infeasible, even for a quantum computer [12].
At the same time, it requires the arithmetic logic unit of the system to work with modulo arithmetic and randomness generation.
These are two computationally intensive tasks for a common RISC-V architecture.

Kyber has different versions with different levels of security: Kyber512, Kyber768 and Kyber1024.
Kyber512, the version analyzed in this work, is characterized by a polynomial length equal to two, a size of 800 bytes for the public key, 1632 bytes for the private key, and 768 bytes for the ciphertext.

4.2 Optimisation with a standard approach

The objective was to study optimisations of the Kyber algorithm on an Application-Specific Instruction Processor (ASIP)[13].
The work was supported by the use of ASIP Designer, a tool developed by Synopsys that assists the user throughout all the steps of hardware optimisation.

In ASIP Designer, it is possible to import a model for the processor (e.g. RISC-V) and upload the application source code.
The software then simulates the code, analyses it, and profiles the application with a focus on identifying which functions are the most used and responsible for the number of clock cycles.
The tool provides reports on which basis hardware optimisations can be studied.
ASIP Designer allows the user to modify the processor model, add custom instructions, and verify improvements.
The description of custom instructions is done using a high-level language (nML), which the RTL generator software translates into hardware.
Finally, the tool generates the HDL files that can be used to synthesise the new processor.

From the profiling report of Kyber, the functions selected for optimisation are the following:

- KeccakF1600 state permute function: with 29.03% of the total number of clock cycles spent in this function, it is the most computationally intensive function of Kyber.
- Montgomery reduction function: with 16.36% of the execution time.
- Barrett reduction function: with 8.04% of the total clock cycles.

It was decided to divide the problem into two optimisations: one for Keccak and the other for both reduction functions.

4.2.1 Keccak

Keccak is a family of sponge functions[14].
Sponge functions are a type of algorithm that process input data of any length ("absorbing" it into a state vector) and produce an output of a desired length ("squeezing" it out from the state).

The one used in Kyber is Keccak-F1600. It uses a 1600-bit state vector, organised as a matrix of five rows by five columns, each element being 64 bits in length.
Between each step of absorbing input and squeezing output, a function called permutation is applied to the state.
A permutation is a set of logical and algebraic steps repeated n times (in Keccak-F1600, $n = 24$).
These transformations are designed to provide diffusion and confusion, making the permutation highly non-linear. The figure 4.4 shows Keccak structure.

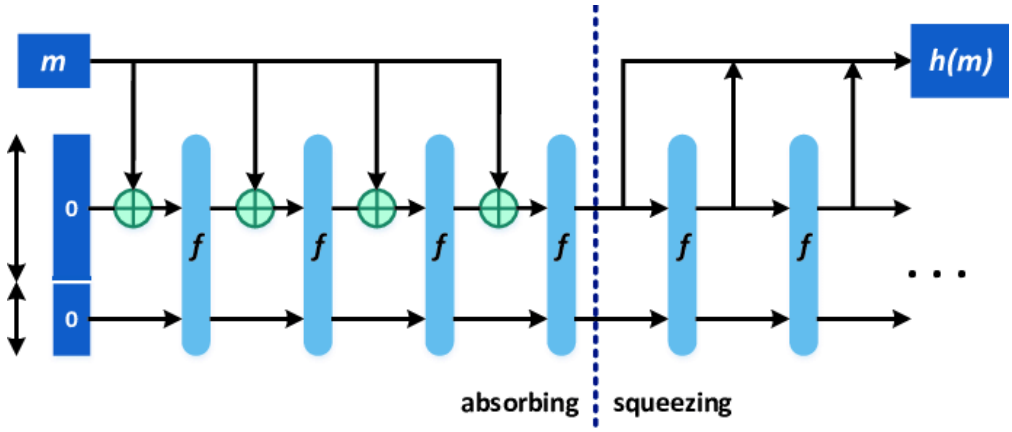

Figure 4.4: Keccak Sponge function structure[15]

In Kyber, Keccak is used extensively as a pseudo-random function (PRF) and extendable-output function (XOF). It is used to generate random polynomials for the public key, error vector and secret key, for hashing of messages and, in the decryption phase, to retrieve the encapsulated key.

The problems with the execution of the permutation function are mainly due to the fact that it is register-limited on RISC-V. It uses twenty-five 64-bit values each time it applies a change to the state vector: RISC-V does not have enough registers to store all these values, so a lot of time is wasted swapping the state components and partial results in and out of stack memory. The operations on the state vector are quite simple (XORs and shifts) but they are scattered among load and store instructions.

The optimisation implemented was a complex structure made of a custom register file of twenty-five 64-bit locations and a logic matrix block.
In Figure 4.5 a scheme of the hardware is reported.

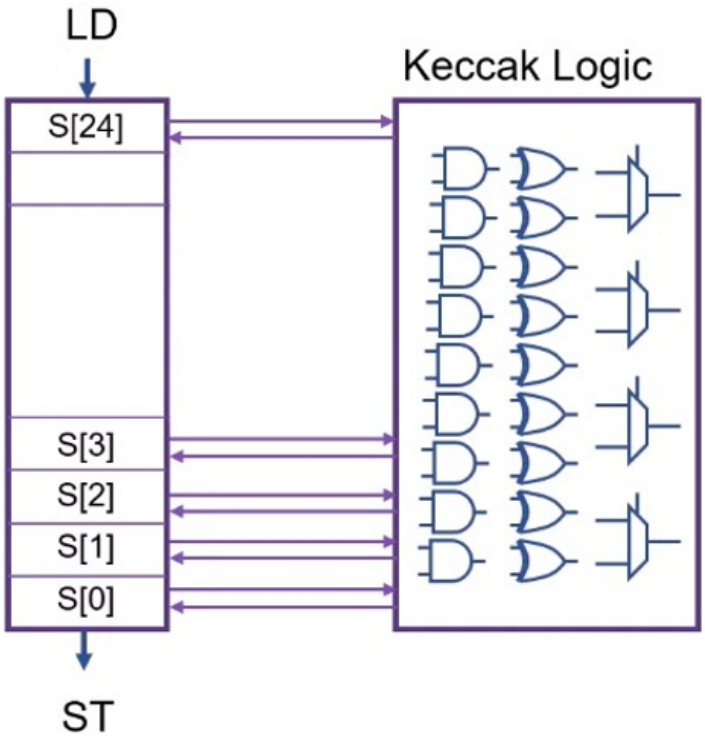

Figure 4.5: Scheme of the Keccak optimization structure

During the execution of the permutation, the first step is the load of the state into the custom register. The logic and arithmetic operation are then executed in the matrix block within a single clock cycle, with the output of the block that is saved as the new state in the same custom register file. To complete the execution, the values of the state register are saved in RISC-V memory using a series of stores.

Three different custom instruction were added to the ISA:

- A custom instruction for the load and one for the store of directly 64 bit from memory. This reduces the number of clock cycles required to load the Keccak state register by half.

- The Keccak instruction which starts the computation of the permutation and saves the results in the the registers of the state.

4.2.2 Reduction functions

Barrett and Montgomery reduction functions are both related to modular arithmetic: Barrett reduction is an algorithm designed to optimise the calculation of $a \bmod n$, Montgomery reduction is used to perform modular multiplication.
Both substitute the division operation with much simpler moltiplications.

These functions are translated into a set of instructions in RISC-V architecture. The optimisation involves grouping these operations together and executing them in one single clock cycle.

Two new customs were added to the ISA:

- One for Barrett reduction. It computes the modulo of the source register number.
- One for Montgomery reduction. It takes the product of a times b as input and applies the algorithm to return the modulo of the product.

4.2.3 Results

The results of the two optimisations are reported in 4.1.

Version	Clock Cycles Reduction [%]	Area Increase [%]
Custom Reduction	34,27%	6%
Custom Keccak	28,87%	102%

Table 4.1: Results of traditional optimisation on Kyber

These results are used for comparison in 5.1 with the optimisations found by CIRCO during its analisys.

4.3 Optimisation with CIRCO

4.3.1 CIRCO flow

Before running the tool, some modifications to the source code must be made.
As described in Subsection 3.2.4, the compiler will try to optimise the assembly code and exclude functions and variables that it does not recognise as used. To avoid this, ___attribute___((used)) must be added to constant vectors present in functions the user wants to analyse (e.g. ntt_zeroes).

For functions, instead, the software reads the header files and adds the attribute on its own where it is needed, but it is still necessary to have all the prototypes declared.
The source code obtained from the GitHub repository misses the declaration of the Keccak functions in the header file, so it is necessary to add them first.
After these changes, the flow can start from the compilation of the application.
This phase does not require any special precautions; X-HEEP runs gcc commands and returns the assembly file main.S, the disassembled version of the main.elf object file.
The disassembly file contains many functions different from main and application functions, such as handlers (timer and GPIO), initialisation functions, standard library functions, etc.
The aim is not to optimise these functions, so the list of excluded functions is written in the configuration file unwantedFunction.txt.
At this point, it is possible to include the main.S file in CIRCO and run Part1_ASMRead to obtain the cleaned version of the assembly code, ASMCode.txt.
The opcodes selected for the custom instructions of this code are the ones with funct7 field equal to 0x6B. If the file availableOpcode.txt is filled as follows, the tool automatically assigns an opcode to each new instruction.

```
Opcode7 funct2 funct3
0x6B    0x1    0x1
0x6B    0x1    0x2
0x6B    0x1    0x3
0x6B    0x1    0x4
0x6B    0x1    0x5
0x6B    0x1    0x6
0x6B    0x1    0x7
0x6B    0x2    0x1

[...]
```

For each different funct7 field, there are 32 available instructions from each combination of the funct2 and funct3 fields.
Since there are enough available codes with respect to the new custom instructions implemented, it has been decided to leave the code 0x0 unused.

The first analysis, without limiting the kinds of instructions the tool can merge, is

performed by leaving the configuration file UnwantedMerge.txt empty.
The Part2_ASMAnalysis reads the code and returns the local and global results files.
Some considerations can be made from the global results file.
A section of the file is reported here, while the full version is in Appendix A.2.

```
XOR(XOR(_A_)(_B_))(_C_): 20
    KeccakF1600_StatePermute(20)

OR(SLLI(_A_)(_B_))(_C_): 16
    KeccakF1600_StatePermute(15)
    rej_uniform(1)

OR(OR(_A_)(_B_))(_C_): 1
    PQCLEAN_KYBER512_CLEAN_poly_getnoise_eta1(1)

SUB(ADD(_A_)(_B_))(_C_): 1
    PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(1)

XOR(OR(_A_)(_B_))(_C_): 8
    KeccakF1600_StatePermute(8)

[...]
```

The number of new custom instructions is high, but the majority of them have a low recurrence count.
Even though the reduction in clock cycles of a single union depends on how many times the function is called or whether the instructions are inside a for/while loop, adding a dedicated custom instruction and the corresponding hardware would increase area with a negligible effect on performance.
It has been decided to select and implement only custom instructions with a recurrence higher than 3, and to mark all others as unwanted for merging.
Furthermore, the recurrence of the selected instructions may slightly increase due to the fact that conflicts between unions would prioritise the remaining customs when searching for a solution that complies with unwanted merge constraints.

Two different solutions have been explored:

- Optimisation 1 (Opt1): only high-recurrence custom instructions that do not use complex architectures such as adders and multipliers.

- Optimisation 2 (Opt2): all high-recurrence custom instructions, including those that use adders and multipliers.

A summary table of the new custom instructions included in the different versions is reported in Table 4.2.

Custom Instruction	Opt1	Opt2
OR(SRLI(_A_)(_B_))(_C_)	X	X
OR(SLLI(_A_)(_B_))(_C_)	X	X
XOR(OR(_A_)(_B_))(_C_)	X	X
XOR(XOR(_A_)(_B_))(_C_)	X	X
ADD(SLLI(_A_)(_B_))(_C_)		X
SRAI(ADD(_A_)(_B_))(_C_)		X
MUL(SRAI(_A_)(_B_))(_C_)		X
SLLI(MUL(_A_)(_B_))(_C_)		X
ADD(MUL(_A_)(_B_))(_C_)		X
ADD(ADD(_A_)(_B_))(_C_)		X

Table 4.2: Custom instructions list of the two versions

To proceed with the first solution, the file unwantedMerge.txt is filled with all other instructions present in the global results, except for those selected.
New local and global results files are generated. The local results file for the Keccak function is also reported as an example in Appendix **??**.
Running part3_ChangeAPPFile generates the new C and H files. The functions that contain the custom instructions are written in inline assembly, as shown for the Keccak function in Figure 4.6

```
void KeccakF1600_StatePermute(uint64_t *state) {
    [...]
    Asu = state[24];
    for (round = 0; round < NROUNDS; round += 2) {
        //    prepareTheta
        BCa = Aba ^ Aga ^ Aka ^ Ama ^ Asa;
        BCe = Abe ^ Age ^ Ake ^ Ame ^ Ase;
    [...]
}
```

```
void KeccakF1600_StatePermute(uint64_t *state) {
asm volatile(
    [...]
    "lw t2,212(sp)\n\t"
    "lw s9,188(sp)\n\t"
    "lw s10,8(sp)\n\t"
    ".insn r 0x6B, 0x1, 0x1, a6, a3, a2, s4\n\t"
    "xor    a0,a0,a1\n\t"
    "xor    a2,s8,a5\n\t"
    "xor    a1,s1,t3\n\t"
    "lw a5,216(sp)\n\t"
    "xor    t3,t4,t5\n\t"
    "lw t5,232(sp)\n\t"
    [...]
    );
}
```

Figure 4.6: Keccak C file before and after implementation with inline assembly

Before running the application with the new source code, the coprocessor must be

modified to include the custom instructions that have been identified.
The description of the instructions and their operations must be added.
The opcodes assigned by the tool are saved in the configuration file OpcodeNewInstruction.txt.
In this case, it contains:

```
0x6B 0x1 0x1 XOR(XOR(_A_)(_B_))(_C_)
0x6B 0x2 0x1 XOR(OR(_A_)(_B_))(_C_)
0x6B 0x3 0x1 OR(SLLI(_A_)(_B_))(_C_)
0x6B 0x4 0x1 OR(SRLI(_A_)(_B_))(_C_)
```

These opcode fields must coincide with those in the description. For each custom instruction, a new structure of the type described in Figure 3.8 must be included. All new custom instructions are single-cycle instructions without requirements on other instructions; they will be executed on the clock cycle following the issue and, at their completion, will return the destination register for the write-back. Therefore, the bits of the response structure should not be modified from the example.
The mask bits are used to compare only the bits of the opcode byte that characterise the instruction, excluding the register bits. For R4 instructions, the mask is always "00000_11_00000_00000_111_00000_1111111".

What differs among the custom instructions are the instruction fields (opcode) and the name.
The description of two of the four custom instructions is reported here:

```
'{
instr: 32'b00000_01_00000_00000_001_00000_1101011,  //cus_xor_xor
mask:  32'b00000_11_00000_00000_111_00000_1111111,
resp : '{accept : 1'b1, writeback : 1'b1,
        register_read :{1'b1, 1'b1, 1'b1},
        op_type : 2'b10, insn: cus_xor_xor_1_r4, done : 1'b1},
opcode : CUSTOM
},
'{
instr: 32'b00000_01_00000_00000_010_00000_1101011,  //cus_xor_or
mask:  32'b00000_11_00000_00000_111_00000_1111111,
resp : '{accept : 1'b1, writeback : 1'b1,
        register_read : {1'b1, 1'b1, 1'b1},
```

```
        op_type : 2'b10, insn: cus_xor_or_1_r4, done : 1'b1},
opcode : CUSTOM
},
```

The operation description is done in the coprocessor file "coproc.sv".
For the instructions selected the description is the following:

```
  assign a1 = rs_values_i.rs1_0;
  assign b1 = rs_values_i.rs2_0;
  assign c1 = rs_values_i.rs3_0;
  assign rs1_immediate=rs_immediate_i.rs1_immediate;
  assign rs2_immediate=rs_immediate_i.rs2_immediate;
  assign rs3_immediate=rs_immediate_i.rs3_immediate;

  assign cus_xor_xor_1_r4_result = a1 ^ b1 ^ c1;
  assign cus_xor_or_1_r4_result = (a1 | b1) ^ c1;
  assign cus_or_slli_1_r4_result =(a1 << rs2_immediate) | c1;
  assign cus_or_srli_1_r4_result =(a1 >> rs2_immediate) | c1;
```

Finally, it is possible to compile the application code and run the application in the modified X-HEEP environment.

To implement the second solution, it is necessary to return to the file unwantedMerge.txt and remove all the custom instructions that were excluded by the first optimisation but are present in the second one.
At this point, it is possible to run the CIRCO analysis program one last time.
The description of the instructions implemented in the first solution is already present in the coprocessor file with the old opcodes.
To maintain that assignment, before running Part 3, it is possible to modify the file opcodeNewInstr.txt and manually adjust the opcodes assigned by the tool to those instructions.

Now it is possible to generate the source code files for this solution by running Part 3, and to describe the remaining instructions in the processor as done for the first optimised version.

Finally, the new files obtained can be imported into X-HEEP, compiled, and run.

4.3.2 Clock cycles count

The main.c of the application resets the memory cycle register before calling the Kyber API and prints the value after their return.
In this way, the simulation of the application returns the number of clock cycles that the processor takes to execute all Kyber steps.
In Table 4.3, the results for the different implementations are reported.
The clock cycle counts refer only to the execution of the function `Keygen()`. Similar results are obtained for the other two parts of the algorithm.

Version	# of Custom Instr.	Total Recurrences	Clock Cycles	Clock Cycles Reduction [%]
Kyber-Original	/	0	1009979	/
Kyber-Opt1	4	56	991898	1,80%
Kyber-Opt2	10	109	971043	3,85%

Table 4.3: Clock cycles count of Kyber for CIRCO optimisations

4.3.3 Synthesis

The X-HEEP environment used is designed to run synthesis on FPGA with the support of Vivado software.
The board used is the ZCU104 Evaluation Board.

The result is the utilisation report. The key information that can be extracted from this report is the number of Configurable Logic Blocks (CLBs) used by the design.

All X-HEEP descriptions are synthesised, including the coprocessor, in the three scenarios: without custom instructions, with those of the first optimisation (without adders and multipliers), and the second optimisation with more complex operators.

Since the different designs differ only in the presence of the custom instructions, all the CLBs used are the same except for the number of look-up tables LUTs and multiplexers, which are used in the synthesis of the new combinational parts.
These changes are reported in Table 4.4.

As expected, the number of LUTs increases with the addition of more custom instructions, and the Opt2 solution uses a higher number of arithmetic blocks (CARRY8) to implement adders.
Muxes are used to combine different LUTs when the function is too complex. The

Version	Component	Number [#]	Increase with respect to Original [%]
Kyber-Original	Look Up Table (logic)	30556	/
	Muxes	3694	/
	CARRY8	817	/
Kyber-Opt1	Look Up Table (logic)	31219	2,17%
	Muxes	3633	-1,65%
	CARRY8	817	0,00%
Kyber-Opt2	Look Up Table (logic)	31462	2,97%
	Muxes	3647	-1,27%
	CARRY8	829	1,47%

Table 4.4: Configurable Logic Block usage in the different synthesis

reduction in their number between versions is small and is probably due to different optimisations of the design by Vivado.
In any case, these increases are small in percentage terms. The reason is that there is no complex architecture and the description of the custom instructions uses simple operators.

Chapter 5

Results and comparisons

5.1 Comparison of optimisations

Table 5.1 summarises the results for the optimised versions of Kyber previously described.
The area comparison has been omitted due to the different synthesis targets.

Version	Clock Cycles Reduction [%]
CIRCO-Opt1	1,80%
CIRCO-Opt2	3,85%
Traditional-Keccak	28,87%
Traditional-Reduction	34,27%

Table 5.1: Summary of result for optimised versions of Kyber

From this, it is possible to understand whether the methodology can be improved to obtain better results and the characteristics of the solutions themselves.

By looking at the reduction in the number of clock cycles, it can be seen that CIRCO's optimisations are not truly comparable with those achieved through manual analysis and ad hoc optimisations.

Let us start with what has been done for the Keccak permutation function.
In this case, the main problem of the function is the dimension of the state vector, which is big enough to make the function register limited on RISC-V, needing a

constant swap in and out of register between each logical and arithmetic operation. The solution from the study of the function is to add a register file and implement an instruction that does not use the standard RISC-V register as input and output but the new one implemented. It is important to note that all of this violates the RISC-V compliance requirement.
It should not be surprising that CIRCO does not see it as a possible optimisation. Without the possibility of adding registers, there is not much room for more efficient solutions that the one found by CIRCO.

The tool attempted to address this and partially succeeded.
The solution "Opt1" can be considered CIRCO's optimisation for Keccak.
In this solution, multipliers and adders are excluded from the custom instructions. Only shift, OR, and XOR operators remain, which are heavily used in Keccak.
Table 5.2 reports the global result file for CIRCO's first optimisation, "Opt1".
Out of fifty-six new instructions, fifty-four are used in Keccak.
The tool is limited in finding more instructions due to the high number of load and store operations: registers used in the instructions are loaded just before their usage, as they are needed for other operations before.

Custom Instruction	Total recurrences	Recurrences in KeccakF1600
OR(SRLI(_A_)(_B_))(_C_)	12	11
OR(SLLI(_A_)(_B_))(_C_)	16	15
XOR(OR(_A_)(_B_))(_C_)	8	8
XOR(XOR(_A_)(_B_))(_C_)	20	20
TOTAL	56	54

Table 5.2: Summary of result report of opt1 solution from CIRCO

It is also important to note that the solution with a custom register file doubles the area of the microcontroller. This may be unacceptable for embedded system implementations.

The cases of the reduction functions are different. The Barrett and Montgomery C code is reported in 5.1. The C code translates to the ASM code in Figure 5.3

In the traditional approach, the custom instructions encapsulate the entire functions and execute in a single clock cycle.
A similar optimisation should also be identifiable by CIRCO.
However, it is not recognised due to the design choice of generating only generic

```c
int16_t PQCLEAN_KYBER512_CLEAN_montgomery_reduce(int32_t a){
    int16_t t;

    t = (int16_t)a * QINV;
    t = (a - (int32_t)t * KYBER_Q) >> 16;
    return t;
}
```

```c
int16_t PQCLEAN_KYBER512_CLEAN_barrett_reduce(int16_t a){
    int16_t t;
    const int16_t v = ((1 << 26) + KYBER_Q / 2) / KYBER_Q;

    t  = ((int32_t)v * a + (1 << 25)) >> 26;
    t *= KYBER_Q;
    return a - t;
}
```

Figure 5.1: Barrett and Montgomery reduction C code

```
<montgomery_reduce>:
lui   a4,0xfffff
addi  a5,a4,769
mul   a5,a0,a5
addi  a4,a4,767
slli  a5,a5,0x10
srai  a5,a5,0x10
mul   a5,a5,a4
add   a0,a0,a5
srai  a0,a0,0x10
ret
```

```
<barrett_reduce>:
lui   a5,0x5
addi  a5,a5,-321
mul   a5,a0,a5
lui   a4,0x2000
add   a5,a5,a4
lui   a4,0x1
addi  a4,a4,-767
srai  a5,a5,0x1a
mul   a5,a5,a4
sub   a0,a0,a5
slli  a0,a0,0x10
srai  a0,a0,0x10
ret
```

Table 5.3: ASM code of Montgomery and Barrett functions

custom instructions, meaning that all operands must be encoded in the instruction opcode.
Constants are not hardcoded in hardware. For example, a “shift 0x10” operation requires a source field for the value 0x10. The same applies to instructions with immediate operands.
In CIRCO, such instructions were excluded from merging because the immediate value would not fit within a composed instruction. Examples include ADDI, XORI, ORI, and LUI (if the value is stored in a register used only in the next instruction). This prevents the tool from identifying the entire blocks of Montgomery and Barrett functions as single custom instructions, limiting it instead to merging only pairs of instructions.

If we accept to remove this constraint and modify the instruction merging algorithm, such complex instructions could be identified, increasing performance.

This would benefit algorithms like Kyber, where modular arithmetic is applied and many operations use constant values.

Finally, while the traditional approach may appear easy to implement—especially when using tools like ASIP Designer—it should be noted that Kyber is a well-known and widely studied algorithm. For this reason, the code is well-organised into blocks and independent functions, such as the Barrett and Montgomery algorithms.
Without such a structure, the ASIP Designer report would be much less clear, and finding effective optimisations would be significantly harder.
CIRCO, on the other hand, would still perform consistently regardless of code organisation.

5.2 Future works

As described in 5.1, to achieve better tool performance, some limitations on custom instructions should be removed.
This includes the possibility of adding registers to the architecture or implementing highly specific instructions that include constants hardwired into the computation.
These changes would allow the algorithm to find many new instructions and may require a change in the metric used to select which instructions to merge.
Currently, this selection is based on the number of custom instructions in the final solution. A better approach might involve an initial execution of the algorithm with annotation of how many times each instruction is executed.
In this way, the tool could select the instructions that contribute the most to reducing the number of clock cycles, providing the user with some pre-implementation insight into the optimisations.

Finally, to maintain human-readable code, the complete rewriting of the C code with inline assembly should be avoided.
This is not an easy task to automate.
C variables can be linked to registers, but understanding from the C code which assignments and operations generate specific assembly instructions is not trivial, even with disassembly of the object code.
In general, CIRCO has proven to have potential, and further tests should be conducted on different application from Kyber.

Chapter 6

Conclusions

The new approach developed in CIRCO works.
An entire application is analysed in a few minutes automatically, with a flow that gives the user the possibility to explore different optimisations and obtain information on the application.
The tool analyses the assembly of the functions and correctly rewrites them with inline assembly that can be compiled again to include new custom instructions.
The optimised version of the application shows an improvement in performance, which is visible from the reduction in the number of clock cycles during execution.
The optimisations are global to the entire application, with the custom instructions included in each part of the code where the same pattern is found. From the comparison with the study of the algorithm using the standard approach, the results highlight some critical issues.
This tool cannot replace the traditional approach to hardware optimisation. Some optimisations, like the one implemented for Keccak, require complex architectures. These structures have been proven to be very effective but cannot easily be derived solely from the assembly code sequence.
The criteria applied in the search for custom instructions are a key parameter.
The idea of having only generic custom instructions that include all the operands in the instruction opcode prevents the tool from finding complete custom instructions for Barrett and Montgomery reductions. Even though CIRCO did not perform well with Kyber due to the nature of the application, the tool's flexibility suggests that it could perform better with other cryptographic algorithms or applications in different fields. In conclusion, the new approach is interesting and has potential,

but it needs further investigation and development.
The tool, with the right refinements, can become a strong support tool for optimising any type of application in cooperation (and not in substitution) with traditional optimisation methods.

Appendix A

Report from CIRCO

A.1 Global results file of Kyber without any unwanted merges

```
AND(NOT(_A_))(_B_): 2
 KeccakF1600_StatePermute(2)

OR(SRLI(_A_)(_B_))(_C_): 12
 KeccakF1600_StatePermute(11)
 PQCLEAN_KYBER512_CLEAN_polyvec_tobytes(1)

OR(SLLI(_A_)(_B_))(_C_): 16
 KeccakF1600_StatePermute(15)
 rej_uniform(1)

XOR(OR(_A_)(_B_))(_C_): 8
 KeccakF1600_StatePermute(8)

XOR(XOR(_A_)(_B_))(_C_): 20
 KeccakF1600_StatePermute(20)

XOR(AND(_A_)(_B_))(_C_): 1
 KeccakF1600_StatePermute(1)
```

```
ADD(SLLI(_A_)(_B_))(_C_): 6
 rej_uniform(2)
 PQCLEAN_KYBER512_CLEAN_ntt(4)

SRLI(SLLI(AND(_A_)(_B_))(_C_))(_C_): 1
 rej_uniform(1)

SRLI(ADD(_A_)(_B_))(_C_): 1
 PQCLEAN_KYBER512_CLEAN_ntt(1)

MUL(SRAI(_A_)(_B_))(_C_): 14
 PQCLEAN_KYBER512_CLEAN_ntt(1)
 PQCLEAN_KYBER512_CLEAN_basemul(5)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(8)

SLLI(MUL(_A_)(_B_))(_C_): 7
 PQCLEAN_KYBER512_CLEAN_ntt(1)
 PQCLEAN_KYBER512_CLEAN_basemul(5)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(1)

SRAI(ADD(_A_)(_B_))(_C_): 6
 PQCLEAN_KYBER512_CLEAN_basemul(5)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(1)

SRAI(SLLI(NEG(_A_))(_B_))(_B_): 1
 PQCLEAN_KYBER512_CLEAN_poly_basemul_montgomery(1)

SRLI(SLLI(_A_)(_B_))(_B_): 1
 PQCLEAN_KYBER512_CLEAN_polyvec_tobytes(1)

ADD(_A_)(AND(SRAI(_A_)(_B_))(_C_)): 1
 PQCLEAN_KYBER512_CLEAN_polyvec_tobytes(1)

SRLI(SLLI(ADD(_A_)(_B_))(_C_))(_C_): 1
 PQCLEAN_KYBER512_CLEAN_polyvec_tobytes(1)

ADD(ADD(_A_)(_B_))(_C_): 13
```

```
 PQCLEAN_KYBER512_CLEAN_poly_getnoise_eta1(1)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(12)

AND(SRLI(_A_)(_B_))(_C_): 1
 PQCLEAN_KYBER512_CLEAN_poly_getnoise_eta1(1)

OR(OR(_A_)(_B_))(_C_): 1
 PQCLEAN_KYBER512_CLEAN_poly_getnoise_eta1(1)

SUB(ADD(_A_)(_B_))(_C_): 1
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(1)

ADD(MUL(_A_)(_B_))(_C_): 7
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(7)
```

A.2 Global results file of Kyber of Opt1 solution

```
OR(SRLI(_A_)(_B_))(_C_): 12
 KeccakF1600_StatePermute(11)
 PQCLEAN_KYBER512_CLEAN_polyvec_tobytes(1)

OR(SLLI(_A_)(_B_))(_C_): 16
 KeccakF1600_StatePermute(15)
 rej_uniform(1)

XOR(OR(_A_)(_B_))(_C_): 8
 KeccakF1600_StatePermute(8)

XOR(XOR(_A_)(_B_))(_C_): 20
 KeccakF1600_StatePermute(20)
```

A.3 Glocal results file of Kyber of opt2 solution

```
OR(SRLI(_A_)(_B_))(_C_): 12
```

```
 KeccakF1600_StatePermute(11)
 PQCLEAN_KYBER512_CLEAN_polyvec_tobytes(1)

OR(SLLI(_A_)(_B_))(_C_): 16
 KeccakF1600_StatePermute(15)
 rej_uniform(1)

XOR(OR(_A_)(_B_))(_C_): 8
 KeccakF1600_StatePermute(8)

XOR(XOR(_A_)(_B_))(_C_): 20
 KeccakF1600_StatePermute(20)

ADD(SLLI(_A_)(_B_))(_C_): 6
 rej_uniform(2)
 PQCLEAN_KYBER512_CLEAN_ntt(4)

MUL(SRAI(_A_)(_B_))(_C_): 14
 PQCLEAN_KYBER512_CLEAN_ntt(1)
 PQCLEAN_KYBER512_CLEAN_basemul(5)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(8)

SLLI(MUL(_A_)(_B_))(_C_): 7
 PQCLEAN_KYBER512_CLEAN_ntt(1)
 PQCLEAN_KYBER512_CLEAN_basemul(5)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(1)

SRAI(ADD(_A_)(_B_))(_C_): 6
 PQCLEAN_KYBER512_CLEAN_basemul(5)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(1)

ADD(ADD(_A_)(_B_))(_C_): 13
 PQCLEAN_KYBER512_CLEAN_poly_getnoise_eta1(1)
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(12)

ADD(MUL(_A_)(_B_))(_C_): 7
 PQCLEAN_KYBER512_CLEAN_indcpa_keypair_derand(7)
```

A.4 Snippet of local results file of KeccakF1600 function of Opt1 solution

```
[1549, 1546]
OR(SRLI(_A_)(_B_))(_C_)
a5, {'_A_': 'a5', '_B_': '0x3', '_C_': 'a4'}
[1547, 1545]
OR(SLLI(_A_)(_B_))(_C_)
a2, {'_A_': 'a2', '_B_': '0xd', '_C_': 'a3'}
[1521, 1520]
OR(SLLI(_A_)(_B_))(_C_)
s2, {'_A_': 's2', '_B_': '0x1c', '_C_': 's3'}
[1506, 1505]
OR(SLLI(_A_)(_B_))(_C_)
s6, {'_A_': 's6', '_B_': '0x15', '_C_': 's7'}
[1496, 1495]
OR(SRLI(_A_)(_B_))(_C_)
s8, {'_A_': 's8', '_B_': '0x15', '_C_': 's9'}
[1206, 1200]
XOR(OR(_A_)(_B_))(_C_)
t5, {'_A_': 't5', '_B_': 's7', '_C_': 'a4'}
[1195, 1192]
XOR(OR(_A_)(_B_))(_C_)
t0, {'_A_': 't0', '_B_': 's6', '_C_': 's5'}
[1184, 1182]
XOR(OR(_A_)(_B_))(_C_)
t4, {'_A_': 's8', '_B_': 't4', '_C_': 'a5'}
#Conflict found - Unions taken
[1174, 1171]
OR(SRLI(_A_)(_B_))(_C_)
t2, {'_A_': 'a4', '_B_': '0x1f', '_C_': 's9'}
#Conflict - Alternative Solution
#[1177, 1174]
#XOR(OR(_A_)(_B_))(_C_)
#t2, {'_A_': 't2', '_B_': 's9', '_C_': 'a3'}
[...]
```

Bibliography

[1] Sparsh Mittal. «A survey of techniques for improving energy efficiency in embedded computing systems». In: *International Journal of Computer Aided Engineering and Technology* 6.4 (2014), pp. 440–459. DOI: 10.1504/IJCAET.2014.065419 (cit. on p. 1).

[2] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 3rd ed. CRC Press, 2020 (cit. on p. 2).

[3] Peter W. Shor. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer». In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172 (cit. on p. 3).

[4] Lily Chen and et al. *Report on Post-Quantum Cryptography*. Tech. rep. NIST IR 8105. National Institute of Standards and Technology (NIST), 2016. DOI: 10.6028/NIST.IR.8105. URL: https://doi.org/10.6028/NIST.IR.8105 (cit. on p. 3).

[5] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. «Post-quantum key exchange—A new hope». In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016, pp. 327–343. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim (cit. on p. 3).

[6] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Tech. rep. FIPS PUB 180-4. U.S. Department of Commerce, Aug. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (cit. on p. 3).

[7] R. Banakar, A. Raghunathan, and N. K. Jha. *Hardware/Software Co-Design for Embedded Systems*. Springer, 2010 (cit. on p. 4).

[8] *X-HEEP: eXtensible Hardware Explorer and Educational Platform*. https://x-heep.readthedocs.io/en/latest/. Accessed: 2025-06. 2025 (cit. on p. 10).

[9] RISC-V Software Collaboration. *RISC-V GNU Toolchain*. https://github.com/riscv-collab/riscv-gnu-toolchain. 2025 (cit. on p. 14).

[10] Christof Paar, Jan Pelzl, and Tim Güneysu. *Understanding Cryptography: From Established Symmetric and Asymmetric Ciphers to Post-Quantum Algorithms*. 2nd ed. Springer, 2024. ISBN: 978-3662690062 (cit. on p. 39).

[11] Accessed: 2025-06. 2024. URL: https://www.researchgate.net/figure/Decryption-Scheme-1-Request-the-address-of-the-image-to-be-decrypted-2-Decrypt-from_fig2_393370948 (cit. on p. 39).

[12] Roberto Avanzi et al. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism ML-KEM*. Tech. rep. FIPS 203. CRYSTALS-Kyber: an IND-CCA2 secure module-lattice-based key encapsulation mechanism. National Institute of Standards and Technology, Jan. 2024. URL: https://doi.org/10.6028/NIST.FIPS.203 (cit. on p. 41).

[13] Inc. Synopsys. *Tsec: an ASIP for Post-Quantum Cryptography (Kyber case study with ASIP Designer)*. ASIP eUpdate Newsletter, Synopsys ASIP Designer U-2023.06. Describes the "Tsec" example processor: a RISC-V baseline extended with custom Keccak/XOF instructions to accelerate Kyber, yielding 8× speed-up at 1.8× area cost. Oct. 2023 (cit. on p. 42).

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. «Keccak sponge function family main document». In: *Submission to NIST (Round 3)* (2011). URL: https://keccak.team/files/Keccak-main-1.3.pdf (cit. on p. 42).

[15] Jawad Haj-Yahya, Ming Ming Wong, Suman Sau, and Anupam Chattopadhyay. «A New High Throughput and Area Efficient SHA-3 Implementation». In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351761. URL: https://www.researchgate.net/figure/Keccak-sponge-construction_fig1_324956629 (cit. on p. 43).