POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Design and verification of a RISC-V superscalar CPU

Tesi di Laurea Magistrale in
Electronics Engineering - Ingegneria Elettronica

Author: **Marco Vitali and Sebastiano Vittoria**

Student ID: 990037 and 990278
Advisor: Prof. Davide Zoni
Co-advisors: Dr. Andrea Galimberti
Academic Year: 2022-23

Abstract

In recent decades, the market developed in a way that the majority of the applications require a specific range of performance, area, and power consumption. This necessity brought to the continuous research for computational efficiency, leading to the specialization of computer architectures and different CPU architectures have been developed to satisfy specific market needs. The RISC-V Instruction Set Architecture, known for its simplicity and flexibility, serves as the foundation for this CPU design, while the superscalar and dual-issue concepts aim to enhance instruction throughput, thereby improving overall computational efficiency. This thesis presents the comprehensive design, implementation, and experimental evaluation of a superscalar RISC-V dual-issue central processing unit (CPU) on an Artix-7 FPGA, a popular choice in the academic field. The architectural choices and design considerations specific to RISC-V are meticulously discussed, emphasizing the integration of dual-issue capabilities to exploit parallelism in instruction execution. The presented CPU features a 7-stage pipeline composed of Instruction Fetch, Decode, Issue, Execution, Reorder Buffer, and Register File units. The architecture is verified with a custom tool based on the RISC-V golden model, supporting an instruction granularity method where every committing instruction is verified. To evaluate the proposed superscalar RISC-V dual-issue CPU, a comprehensive set of benchmarks is employed, encompassing a diverse range of real-world applications and workloads. Performance metrics such as execution time, resource utilization, and energy efficiency are analyzed to provide an overview of the CPU's capabilities. This thesis contributes to the existing body of knowledge by providing insights into the design challenges, trade-offs, and performance implications associated with superscalar RISC-V dual-issue architectures. Overall, this research underscores the significance of efficient instruction execution in advancing the capabilities of RISC-V processors in the era of complex computing workloads.

Keywords: CPU, superscalar, dual-issue, RISC-V, computer architecture, verification

Abstract in lingua italiana

Negli ultimi decenni, il mercato si è sviluppato in modo tale che la maggior parte delle applicazioni richiede una specifica gamma di prestazioni, area e consumo energetico. Questa necessità ha portato alla continua ricerca di maggiore efficienza computazionale. Di conseguenza, sono state sviluppate diverse architetture CPU per soddisfare queste specifiche esigenze di mercato. L'Instruction Set Architecture RISC-V, noto per la sua semplicità e flessibilità, funge da fondamento per il design di questa CPU, mentre i concetti di superscalar e dual-issue mirano a migliorare il throughput delle istruzioni, migliorando così l'efficienza computazionale complessiva. Questa tesi presenta il design completo, l'implementazione e la valutazione sperimentale di una CPU RISC-V superscalare dual-issue su un FPGA Artix-7, scelta popolare nel campo accademico. Le scelte architetturali e le considerazioni di design specifiche per la CPU sono discusse meticolosamente, sottolineando l'integrazione delle funzionalità di dual-issue per sfruttare il parallelismo nell'esecuzione delle istruzioni. La CPU discussa presenta una pipeline a 7 stadi composta da unità di Fetch, Decode, Issue, Execute, Reorder Buffer e Register File. L'architettura è verificata con un'infrastruttura personalizzata basata sul golden model di RISC-V, supportando un metodo in cui ogni istruzione viene singolarmente verificata. Per valutare la CPU RISC-V superscalar proposta, viene impiegato un set di benchmark, che comprende una vasta gamma di applicazioni del mondo reale e carichi di lavoro. Le metriche di performance come il tempo di esecuzione, l'utilizzo delle risorse e l'efficienza energetica vengono analizzate per fornire una panoramica delle capacità della CPU. Questa tesi contribuisce all'attuale corpus di conoscenze fornendo approfondimenti sulle sfide di progettazione, i compromessi e le implicazioni delle prestazioni associate alle architetture superscalari RISC-V dual-issue. In generale, questa ricerca sottolinea l'importanza di un'efficiente esecuzione delle istruzioni nel promuovere le capacità dei processori RISC-V in un'era di carichi di lavoro informatici complessi.

Parole chiave: CPU, superscalare, dual-issue, RISC-V, architetture dei calcolatori, verifica

Contents

1 | Introduction

In recent decades, the market has been developing in a way that the majority of the applications require the hardware to be in a specific range of interest in terms of performance, area, and power consumption, which represent the main quality metrics to identify the optimal solution to target. This necessity brought continuous research for computational efficiency, leading to the specialization of computer architecture, and, as time went on, different CPU architectures were developed to satisfy specific market needs.

The final goal of the majority of commercial design flows is the production of a System-on-Chip (SoC) to be inserted in the market, with a time to market that is always reducing. For this reason, many agile SoC development frameworks, like Chipyard [3], were created, which allow the generation of different CPUs to be used as a base for the needed customization. Some of these are open-source, enabling academic groups to work with pre-existing cores, with the possibility to parametrically compose the architecture required for a specific application.

The accelerating pace of technological advancement has been indelibly marked by Gordon Moore's empirical assertion in 1965, known as Moore's Law [21], which postulated that the number of transistors on a microchip would double approximately every two years. However, as the physical limitations of silicon-based transistor scaling began to manifest, the industry found itself facing a profound challenge to sustain this exponential growth. It is within this context that Robert Dennard's scaling [16] principles became a guiding force, predicting that as transistors shrink, their power density remains constant, leading to a proportional increase in performance without a corresponding rise in power consumption. The pursuit of smaller, faster, and more energy-efficient semiconductor devices has encountered challenges in recent years, as the miniaturization of transistors approaches the atomic scale. The once dependable cadence of doubling transistor density every two years has become increasingly difficult to maintain, prompting a paradigm shift in how the industry conceptualizes and engineers computational devices. The Moore's Law narrative has evolved from a straightforward prediction of transistor count to a broader conversation about the nature of computational power and efficiency. The dichotomy between Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC)

architectures has long been a focal point in processor design. CISC architectures, with their rich instruction sets, aim to reduce the number of instructions per program, theoretically leading to more efficient code execution. On the other hand, RISC architectures streamline instruction sets, emphasizing simplicity and efficiency in execution.
Being open-source, the RISC-V Instruction Set Architecture [24] is the most adopted in the academic research field. Its modularity and extendibility allow for a high degree of customization in terms of ISA extensions or even the creation of brand-new instructions, thanks to the free spaces left in the opcodes list. The simpler nature of a Reduced Instruction Set is also optimal for the development of more logically complex pipelines, to boost the performances with the cost of a minor increase in area and power consumption. One of the approaches is to allow the concurrent propagation of more than one instruction at a time through the each stage of pipeline, creating the distinction between scalar and superscalar CPUs.
A superscalar CPU presents more complex logic and, as a consequence, a higher area occupation and power consumption but with the advantage of completing, ideally, multiple instructions per clock cycle. If designed carefully, this type of CPU can represent a good middle ground between low-cost and high-performance solutions. Often, FPGAs represent the best target for the development of a design, thanks to their re-programmability, consequentially reducing the time to market of a product.
In an era where the inexorable march of technological progress is measured not only by raw computational power but increasingly by the judicious use of resources, the concept of computational efficiency assumes paramount importance. Striving for more computational efficiency requires the optimization of the performance of computing systems while minimizing resource utilization, including power consumption and heat dissipation. As demands for more powerful and energy-conscious processors intensify, the pursuit of computational efficiency emerges as a driving force shaping the trajectory of contemporary processor design. The significance of computational efficiency is underscored by its multifaceted impact on diverse facets of computing. Energy consumption, a critical concern in an age of environmental consciousness and escalating power costs is intrinsically linked to the efficiency of computational processes. A processor that executes instructions with optimal efficiency not only contributes to reduced energy consumption but also mitigates challenges associated with heat dissipation, a factor that has become increasingly challenging with the escalating transistor density and clock frequencies. Furthermore, computational efficiency extends its influence to the realm of mobile and embedded systems, where power constraints and battery life are at a premium. In these contexts, processors optimized for computational efficiency become enablers of prolonged device operation, fostering advancements in fields ranging from portable electronics to the In-

ternet of Things (IoT) [23]. By combining the streamlined instruction sets characteristic of RISC architectures with the parallel execution capabilities of superscalar designs, this thesis seeks not only to enhance raw computational power but to do so in a manner that is inherently mindful of resource utilization. As the industry pivots towards sustainable computing practices, the computational efficiency of processors becomes not merely a desirable attribute but an imperative for meeting the demands of contemporary computing landscapes.

The paramount importance of verification in processor design stems from the increasing complexity and sophistication of contemporary architectures, necessitating robust methodologies to ensure correctness. As the industry navigates between Moore's Law and Dennard Scaling, the sheer volume of transistors packed into a limited space amplifies the risk of design errors, underscoring the criticality of verification processes. These processes are not merely safeguards against functional bugs; they are indispensable mechanisms for validating that the processor adheres to its intended specifications. In an era where processors drive applications that range from safety-critical systems to cloud computing infrastructures, the consequences of undetected errors can be profound, demanding a meticulous and exhaustive verification approach. The design of a RISC-V superscalar CPU introduces additional layers of complexity, as it involves intricate interactions between concurrent execution units and advanced pipelining. Furthermore, the verification process is not confined to the design phase alone. As updates, optimizations, and new features are introduced, rigorous verification practices are a staple to avoid unintended consequences, such as performance regressions. In this thesis, verification is treated as an integral aspect of the design philosophy. As we delve into the nuances of parallel execution, pipelining, and the unique challenges posed by the RISC-V architecture, the thesis recognizes verification not just as a means of avoiding failure but as a tool to assist a processor design while maintaining or expanding the base functionalities.

This thesis aims to harness the potential of RISC-V in the development of a superscalar CPU, exploring its applicability in pushing the boundaries of computational efficiency.

In the subsequent chapters, this master thesis will delve into the design principles of superscalar processors, the challenges and opportunities in their integration, and the methodologies employed in the verification process. Through this exploration, we aim to contribute to the ongoing dialogue in the field of processor design, offering insights and innovations that propel computational efficiency into the next frontier of technological advancement.

1.1. Contributions

The present thesis proposes an extensive description of ViVit, a RISC-V superscalar in-order dual-issue CPU, and its corresponding verification method, which exploits a customized version of an open-source ISA simulator. The study aims to provide a comprehensive understanding of the CPU architecture and its verification mechanism, which is crucial for the development of high-performance computing systems. ViVit is a cutting-edge CPU architecture that leverages the RISC-V instruction set architecture (ISA) and implements a superscalar in-order dual-issue design. This design allows for the CPU to execute two instructions in parallel, which significantly enhances the overall performance of the processor. The verification mechanism employed in this study utilizes a customized version of an open-source ISA simulator. The simulator provides a means to test the CPU's functionality and performance under various conditions, including different workloads and environmental factors. The verification process is crucial in ensuring that the CPU operates as intended. The 4 main contributions are:

- The design of the CPU's microarchitecture. The present study centers on the microarchitecture design of the superscalar CPU, with a particular emphasis on each pipeline stage. Our analysis is supported by graphs and pseudocode, which aim to reinforce our findings. Additionally, we scrutinize the logic networks that could potentially emerge as the critical path of each stage. By examining the pipeline stages and their associated logic networks, we seek to better understand the microarchitecture of the superscalar CPU. Ultimately, this effort will help us to improve the functionality and performance of the CPU.

- A comparison between the performance of scalar and superscalar CPUs.The present study aims to compare the performance of scalar and superscalar CPUs, with a particular focus on the architectural distinctions between dual-issue and single-issue configurations. This endeavor is undertaken with the objective of highlighting the benefits of the superscalar design paradigm. Through the use of practical examples, we intend to demonstrate how the superscalar approach leads to improved performance and efficiency, and how it is superior to its scalar counterpart.

- The verification infrastructure comprises a functional Instruction Set Architecture (ISA) simulator and a communication socket that facilitates the exchange of data between the functional ISA simulator terminal and the SystemVerilog testbench for the superscalar CPU. This system is endowed with bespoke features that enable the exposure of either the register file or memory cells as required. The utility of these features is discussed comprehensively, and an in-depth methodology for the verification process is presented. One of the notable features of the verification infrastructure is the ability to expose either the register file or memory cells. This feature enables users to verify the contents of the processor's register file or memory cells at any point during the verification process. The verification methodology presented is a comprehensive approach to verifying a processor's design. It entails setting up a simulation environment, running verification tests, and analyzing the results to identify and resolve any issues that arise.

- Experimental evaluation. In order to evaluate the efficacy of a CPU, it is imperative to conduct a comprehensive analysis of its performance, power consumption, and area occupation across all the data extracted. This analysis provides a holistic view of the quality of the CPU, with a particular emphasis on identifying the modules that have a significant impact on both power consumption and area occupation. By conducting such an evaluation, we can gain an in-depth understanding of the CPU's overall quality and identify areas for potential optimization.

1.2. Structure of the thesis

The rest of this thesis is organized in 5 chapters:

- Chapter 2 provides the basic theoretical knowledge needed for a complete comprehension of the subsequent topics.
- Chapter 3 analyzes multiple scalar, superscalar and Out-of-Order CPUs which are already available and confronts them.
- Chapter 4 describes the whole microarchitecture module by module, compares it with a similar scalar one through practical examples, and then describes the verification infrastructure and method.
- Chapter 5 evaluates the performance results obtained through a set of benchmarks, as well as the post-implementation area occupation and power consumption estimations.
- Chapter 6 concludes the proposed work, and describes the possible future improvements.

2 | Background

Starting with the simplest architectures and moving towards the most complex ones, each stage and improvement is analyzed to build a foundation in CPU architecture. This allows for a clear understanding and justification of every choice made in the architectural design process.

This chapter aims to explain the theoretical concepts involved in CPU architectures. It starts with the concept of pipelining and the fundamentals of pipelined processors, along with the related hazards due to dependencies in the code. It then provides a high-level view of a 5-stage pipelined scalar CPU, highlighting the role of each pipeline stage, followed by a focus on superscalar architectures and the modifications required to obtain it from a scalar one. The concept of Out-of-Order (OoO) is explained, focusing on the differences between OoO dispatch and OoO completion. A comparison between these concepts is also made, underlining the impact of the type of architecture on the quality metrics of a design such as performance, area, and scalability, concluding by indicating which applications each CPU targets. Finally, the chapter concludes with an analysis of the RISC-V Instruction Set Architecture and an explanation of the ISA extensions adopted by the state-of-the-art processors.

2.1. Fundamentals of pipelined processors

When an application is written, it needs to be executed by a CPU. To do that, the application is compiled from the source code into a list of instructions that the processor can understand. Normally, the architecture would have to finish executing one instruction before it starts working on the next one. However, to speed up the process, most computer architectures are designed using a technique called pipelining.
Pipelining is a way to execute multiple instructions at the same time, by breaking down the execution process into different phases. This technique is called Instruction Level Parallelism (**ILP**). In each phase of the execution process, the CPU performs a specific task, and this determines the logic that must be implemented in the corresponding pipeline stage. By doing this, the architecture can execute instructions much faster, since it requires only a fraction of the time needed to complete the entire process.

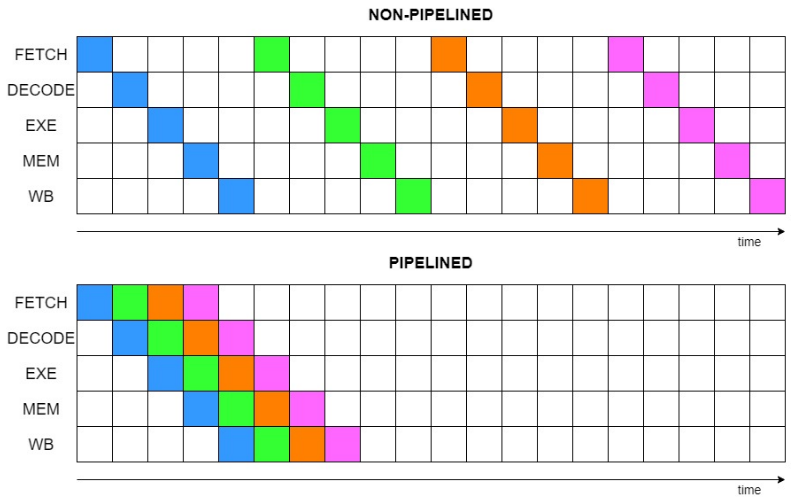


Figure 2.1: Comparison between non-pipelined and pipeline execution, with different colours indicating different instruction

Pipelined designs are evaluated based on their throughput, which refers to the number of instructions executed within a specific time frame, rather than on their latency. As pipelined architectures provide ILP, it is crucial to consider the different types of dependencies among subsequent instructions. These dependencies are called Read After Write (**RAW**), Write After Read (**WAR**), and Write After Write (**WAW**), and are a property of the compiled code. Failing to respect these dependencies can cause execution errors, known as hazards, which limit the amount of parallelism that can be exploited.
A RAW hazard occurs when an instruction produces a value that is required by a subsequent instruction before the first one has written it. If the second instruction attempts

to read the operand before it has been written, the read value will be incorrect.
A WAR hazard is the opposite of a RAW hazard, and occurs when an instruction overwrites a register value before a prior instruction has had a chance to read it. This type of hazard is only problematic during Out-of-Order dispatch, which will be discussed in section 2.4.
Finally, a WAW hazard occurs when two instructions target the same destination register, but do not follow the program order. In this case, the execution's accuracy is compromised, and the final register value will be incorrect. Similarly to the WAR hazard, this problem arises only during Out-of-Order dispatch.

2.2. Scalar CPU

In computer architecture, a basic pipeline consists of five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEM), and Write Back (WB). Each stage has a specific function that will be explained in the following subsections.

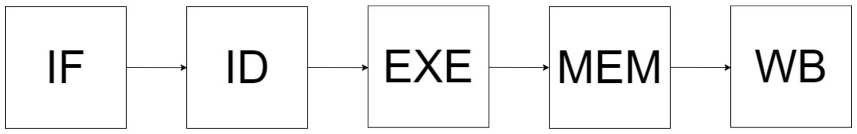


Figure 2.2: Abstract view of a scalar pipeline

2.2.1. Instruction Fetch (IF)

During the first stage of the front end, the CPU performs the Instruction Fetch. In this stage, the CPU reads instructions from a memory address determined by the next PC logic. This logic selects between the Program Counter (PC) register and a branch result. As each new instruction is fetched, the value of the PC register increases by 4.

2.2.2. Instruction Decode (ID)

During the Instruction Decode stage, the instruction from the IF stage is decoded and the register file is accessed to retrieve the values for the operands. In case these values have not been produced, the instruction is stalled, and a stall request is propagated to the IF stage as well. The Instruction Decode stage also drives auxiliary outputs that control the operation to be performed in the EXE stage and the result destination.
To optimize this stage, an early evaluation logic can be added to anticipate the calculation of the branch and target address by one stage. This requires the addition of an ALU and a multiplexer to the stage. A branch predictor can be used to avoid stalling every time a branch operand is needed. A branch predictor, which is essentially a memory containing the predicted branch outcome and target address, updates the predicted outcome for future uses whenever a prediction turns out to be wrong, depending on the predictor structure.
There are different types of predictors, each with varying prediction performance. The most complex ones are useful when the code has nested loop or other intricate branch structures, while the simpler ones have better performance in terms of miss rate for simple loops with lots of iterations.

2.2.3. Execute Stage (EXE)

During this stage, various types of operations are carried out. In each architecture, we can add all the required functional units as long as we ensure that the desired extension is also guaranteed in the Instruction Decode (ID) stage, and we prevent the Out-of-Order completion of operations that could cause incorrect execution.
One possible optimization in the Execution (EXE) stage is called "forwarding." If an Arithmetic Logic Unit (ALU) instruction produces a result that the next instruction requires, it can back-propagate it to one of the ALU inputs. This eliminates the need for stalls in case of Read After Write (RAW) between subsequent instructions.
If the architecture doesn't use the early evaluation method (discussed in subsection 2.2.2), the branch condition and target address are calculated in this stage and then propagated to the next Program Counter (PC) logic.

2.2.4. Memory (MEM)

After the EXE stage, the Memory stage receives the outputs. It can perform three different actions based on the type of operation. For a store operation, the Memory stage saves the propagated value in memory at the address calculated by the EXE, and doesn't send anything to the WB stage. For a load operation, the Memory stage forwards the value read from the data memory at the address calculated by the EXE to the WB stage. For a normal operation, the result is simply buffered to the next stage.
Similar to the previous stage, the Memory stage can perform an optimization called "bypassing" to back-propagate a value towards the EXE stage. This optimization works like the forwarding but from the Memory stage. It can be especially helpful in reducing the impact of RAWs between a load instruction and a subsequent operation that requires the loaded operand.

2.2.5. Write Back (WB)

This stage of the scalar CPU stores results from the previous stage in the Register File (RF).

2.3. Superscalar CPU

Superscalarity is a property of a processor architecture that enables it to issue more than one instruction per clock cycle. This number is known as the issue window width, which indicates the maximum number of instructions that can be dispatched together as a bundle. To prevent a bottleneck in terms of superscalarity, it is crucial to ensure that the same Window Width is granted to all the other stages of the pipeline, such as Instruction Fetch and Instruction Decode.
For instance, an ideal architecture can propagate an instruction to the next stage and complete its execution every clock cycle without any dependencies. In this case, with a single instruction fetched and decoded every cycle, the Issue will only have one operation at a time to dispatch towards the Execute stage, which would lead to every commit being a single commit, effectively avoiding the exploitation of the superscalarity and bottlenecking the theoretical performances. However, increasing the issue window width can exponentially increase the complexity of the architecture as every new instruction in the bundle would require comparison for dependencies and issue compatibility with others. This leads to an increase in hardware resources and a decrease in the maximum frequency. Compatibility to compose a bundle is a concept that will be detailed in subsection 2.3.1. The following subsections will highlight the modifications required to adapt the scalar stages to obtain a superscalar architecture, as well as the newly introduced logic.

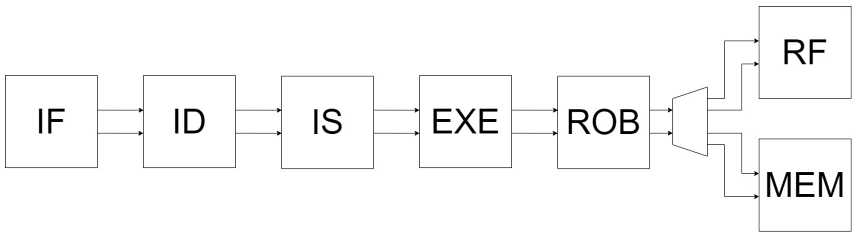


Figure 2.3: Abstract view of a superscalar dual-issue pipeline

2.3.1. Instruction Fetch (IF)

In a superscalar Instruction Fetch (IF) stage, multiple instructions are fetched from the Instruction Memory and sent to the outputs, following the issue compatibility rules of the superscalar architecture. However, the IF stage still behaves like a scalar one, with one exception: the need for a pre-decode logic. This logic can be integrated in the IF or added as a separate pipeline stage before the Instruction Decode (ID) stage.
The role of the pre-decode logic is to create a bundle of instructions that are compatible with each other. For example, it avoids creating a bundle if one of the instructions is a conditional or unconditional jump, because those instructions are usually propagated alone. Some rule sets even prohibit the concurrent propagation of RAW-dependent instructions.
To solve WAW and WAR hazards, which are commit order-related problems, the instructions are reordered after the Execute stage using a Reorder Buffer.

2.3.2. Instruction Decode (ID)

In terms of the scalar ID stage, the only significant alterations are the replication of input and output ports, as well as the decode logic. When it comes to the Early Evaluation logic, no modifications are necessary. This is because the pre-decode stage applies compatibility rules that allow jump or branch instructions to reach the ID alone, which is similar to the scalar architecture.

2.3.3. Issue (IS)

The Issue (IS) stage is a crucial component of a superscalar processor. It is a new set of logic that provides essential functionalities supporting out-of-order completion and, as a result, superscalarity. It is divided into two parts: the Issue Queue (IQ) and the Read Operands logic.
The IQ is responsible for getting the decoded instruction fields from the ID and adding them to a first-in-first-out (FIFO) queue. By communicating with the ROB, it assigns the index of the first available ROB cell (called destination ROBid) to the first free cell of the IQ and writes to that destination the current instruction Program Counter and destination address. The destination ROBid is a critical piece of information that informs the EXE stage where its result needs to be committed. The Read Operands logic checks the availability of the required values, starting from the RF, then the ROB, and lastly the Common Data Bus. As soon as the required information is available, the instruction is dispatched towards the EXE stage with the same additional flags described in the scalar

ID subsection 2.2.2. Based on the compatibility ruleset adopted, every instruction after the first one needs to be checked for RAWs to avoid the concurrent dispatch of data-dependent operations.
The Read Operands phase also takes care of renaming the destination register, either in an independent table or as an extension of the RF. Memory accesses for load and store operations are regulated thanks to a Disambiguation Buffer. Whenever a store is dispatched, its destination memory address is stored in this structure. As a result, a subsequent load operation is stalled as long as it sees the address it wants to read from inside the Disambiguation Buffer, avoiding a RAW hazard. The eviction of an address from this structure is described in subsection 2.3.6.

2.3.4. Execute (EXE) and Common Data Bus (CDB)

In a superscalar architecture, the base EXE stage is similar to the scalar one. However, the functional units (FUs) have to propagate the destination ROBid in addition to the result. Each FU can carry out an operation concurrently with the others. The newly produced results are propagated via the Common Data Bus (CDB) along with their destination ROBid to the ROB and the IS stage. This helps to speed up the Read Operands process if the values are immediately required.
It is important to note that Out-of-Order completion must always be taken care of in a superscalar architecture. This is because different FUs have different latencies, and multiple results could be ready at the same clock cycle. Therefore, either the CDB must be capable of committing the maximum number of possible concurrent results, or a priority and buffer logic must be implemented. An example of the latter is discussed in subsection 4.1.5.

2.3.5. Reorder Buffer (ROB)

The Reorder buffer (ROB) is a vital component in the construction of a superscalar processor. It is used along with the Issue to ensure that all the instructions are committed in order into the Register File (RF) and Memory (MEM) by reordering them based on their ROBid. The ROB receives the results from the EXE stage and commits them only when they are ready.
It functions like a FIFO structure, where the index is referred to as the ROBid, and it can be composed of various entries depending on the information required in the architecture infrastructure. In its basic form, it includes entries such as *result_ready*, *result_value*, *PC*, *destination_register*, *destination_memory*, *exception_flag* and *exception_code*.
The ROB also has the capability of ensuring a precise-exception behavior by having two

fields, *exception_flag* and *exception_code*: when an instruction that triggers an exception reaches the commit point, the ROB ensures that the exception is not propagated towards the RF and MEM. Instead, the entire pipeline is flushed, preserving the machine state (the contents of the RF and the MEM), and the execution restarts from the instruction after the one that triggered the exception.

2.3.6. Memory (MEM)

In-order memory commits are guaranteed by the ROB, similar to RF commits. However, store instructions must evict their destination address from the disambiguation buffer, a memory used to resolve address disambiguation problems in the IS.

2.3.7. Register File (RF)

To prevent any bottlenecking of the superscalar property of the architecture, the superscalar RF must provide a number of write ports that is equal to the issue window width. This is explained in section 2.3.

The internal structure of the superscalar RF has some additional fields that enable it to support renaming, which is necessary for Out-of-Order completion. These fields include a *ROBid* field, which stores the destination ROBid of the last issued instruction producing that register's value, and a *busy* flag that tells the IS stage whether that register value is ready or not. During a commit, the *busy* flag associated with the destination register is de-asserted only if the committing instruction ROBid matches the *ROBid* field. If it doesn't match, it means that there is currently another instruction inside the pipeline that targets that register, and the renaming must not be undone.

2.4. Out-of-Order CPU

An out-of-order (OoO) CPU is a type of processor architecture that can execute instructions without following the program order. When an instruction cannot be dispatched in its program order, but a successive one can, the latter gets executed first.
It is important to understand that there is a difference between Out-of-Order dispatch and completion. Dispatch refers to the order in which the Execute stage is fed, while completion refers to the order in which the functional units (FUs) produce results. The presence of multiple FUs with different latencies implies Out-of-Order completion, but not necessarily Out-of-Order dispatch, which requires additional support logic in the Instruction Stage (IS) to determine when and how to exploit this functionality.
The Reorder Buffer (ROB) plays a crucial role in Out-of-Order CPUs. Together with the In-Order nature of the fill queue logic, it resolves not only Read After Write (RAW) hazards but also Write After Read (WAR) and Write After Write (WAW) hazards, which could otherwise invalidate the correctness while working Out-of-Order.
The main complexity introduced by Out-of-Order dispatch is the logic required to resolve any data and name dependencies, as well as to understand the availability of the operands before deciding which instruction(s) to dispatch. This leads to an increase in area and complexity, which grows exponentially with the number of potentially dispatchable instructions.
The rules for Out-of-Order dispatch must be coherent with the base in-order architecture and can vary depending on the design. To achieve similar overall performance increases, compile-time code reordering techniques can be used, which unburden the hardware from implementing the logic nets described above.

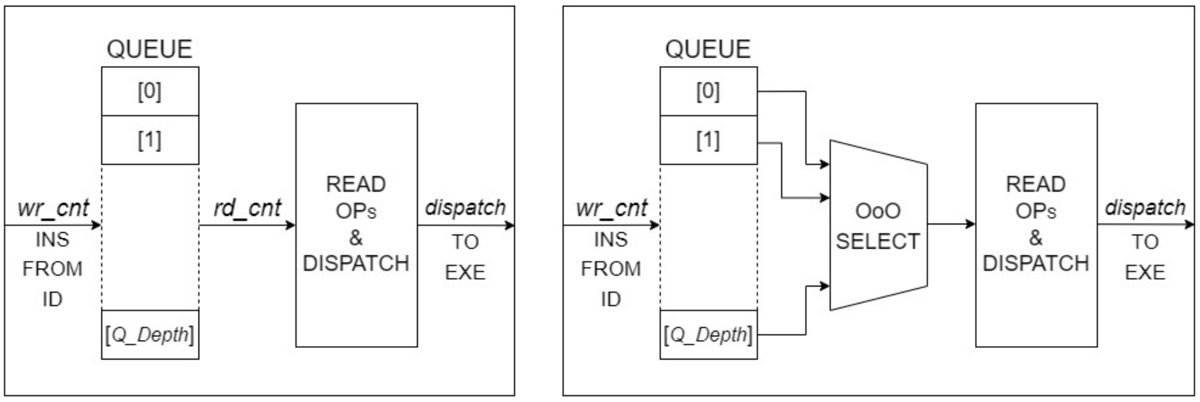


Figure 2.4: A high-level view of an In-Order (left) vs Out-of-Order (right) Issue stage

2.5. Comparison of CPU architectures

When considering different processor designs, a scalar CPU is the most basic option. Although it may not offer the highest performance in terms of Instructions per Cycle (IPC), it is a good choice for low-end applications that require limited power consumption due to its low complexity and small size.
On the other hand, a superscalar CPU is a more advanced solution for high-end applications that require high computational power and efficiency, while still being reasonably resource-efficient. Compared to an Out-of-Order (OoO) architecture, a superscalar CPU is simpler and offers a good foundation for modifications. However, when choosing the issue window width, the tradeoff between area and complexity must be taken into account. In conclusion, a superscalar architecture represents a smart balance between performance and scalability, making it a solid choice for a range of applications.

Table 2.1: Architectures comparison summary

	Performance	**Area**	**Scalability**	**Target applications**
Scalar	Medium-Low	Low	High	Low-end, Low power
Superscalar	Medium-High	Medium	Medium	High-end, Medium power
Out-of-Order	High	High	Very-Low	Leading-Edge

2.6. RISC-V Instruction Set Architecture

In computer science, an Instruction Set Architecture (**ISA**) is an abstract model of a computer and it defines all the supported instructions, data types, registers and hardware support for managing the main memory.
Is it possible to distinguish between two main types of ISA, in particular: CISC, Complex Instruction Set Computer, which is characterized by a large set of complex instructions available and offers different addressing modes, on the other hand, RISC, Reduced Instruction Set Instruction Computer, can count on simpler instructions.
RISC-V is an open-source RISC ISA and it's getting more popular day by day, not only thanks to its free availability but also thanks to its modularity and extensibility. In fact in RISC-V is possible to either add an extension to the existing ISA, exploiting all the free opcodes, or removing the existing extensions that are not needed for the current application.
The base extension is the I-extension and it provides instructions for load/store operations and integer calculations, while the others such as M- and F-extension offer a set of instructions needed for multiplications or divisions and floating-point operations.

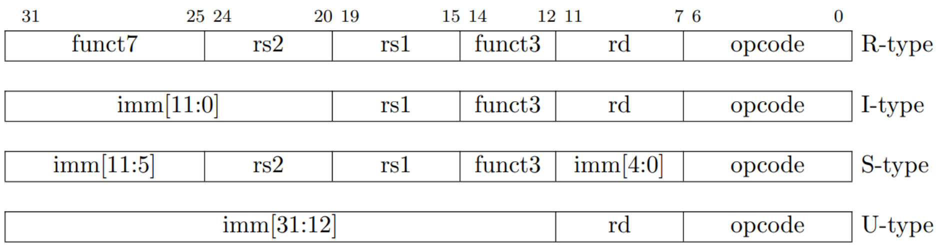


Figure 2.5: Types of instruction encodings as described in the RISC-V specification [24]

In the base RV32I ISA [24], there are four core instruction formats (R/I/S/U). All are fixed 32 bits in length and must be aligned on a four-byte boundary in memory. The source (rs1 and rs2) and destination (rd) registers are kept at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.
The RV32F extension adds 32 floating-point registers, f0–f31, each 32 bits wide, and a floating-point control and status register fcsr, which contains the operating mode and exception status of the floating-point unit. Most floating-point instructions operate on

values in the floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory. Instructions to transfer values to and from the integer register file are also provided.
The RV32M extension introduces a set of instructions to directly perform different types of multiplications and divisions. MUL performs a 32-bit $\times$ 32-bit multiplication of rs1 by rs2 and places the lower 32 bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper 32 bits of the full 2 $\times$ 32-bit product, for signed $\times$ signed, unsigned $\times$ unsigned, and signed rs1 $\times$ unsigned rs2 multiplication, respectively. DIV and DIVU perform a 32 bits by 32 bits signed and unsigned integer division of rs1 by rs2, rounding towards zero. REM and REMU provide the remainder of the corresponding division operation. For REM, the sign of the result equals the sign of the dividend.
The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions. The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but a few opcodes are used for different purposes depending on base ISA width. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the standard floating-point extension (F and D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.
The standard atomic-instruction extension, named “A”, contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model. The base RISC-V ISA has a relaxed memory model, with the FENCE instruction used to impose additional ordering constraints. The address space is divided by the execution environ-

ment into memory and I/O domains, and the FENCE instruction provides options to order accesses to one or both of these two address domains. To provide more efficient support for release consistency, each atomic instruction has two bits, aq and rl, used to specify additional memory ordering constraints. The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing. No ordering constraint is implied to accesses to the other domain, and a FENCE instruction should be used to order across both domains. If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation. If only the aq bit is set, the atomic memory operation is treated as an acquire access, i.e., no following memory operations on this RISC-V hart can be observed to take place before the acquire memory operation. If only the rl bit is set, the atomic memory operation is treated as a release access, i.e., the release memory operation cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the aq and rl bits are set, the atomic memory operation is sequentially consistent and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V hart and to the same address domain.

3 | State of the Art

The RISC-V Instruction Set Architecture (ISA) owes its increasing popularity to its open-source nature, which has made it a popular choice not only within academic settings but also in the commercial sector. Its availability has become essential for small to medium-sized companies, who face a financial barrier when it comes to using other Instruction Set Architectures, such as ARM [1], making RISC-V the go-to solution. Additionally, a RISC-V ecosystem [8] has been developed, which provides useful tools like software compilers, SoC development frameworks, and functional simulators for verification. This has simplified the process of generating FPGA or ASIC solutions based on the RISC-V ISA. This article provides an overview of the most significant RISC-V cores currently in use, with a particular emphasis on superscalar ones. Each core is discussed in detail, including its supported ISA, number of functional units (FUs) and pipeline stages, the hardware description language (HDL) used to design it, the frameworks it is supported by, and whether it is capable of executing instructions out-of-order (OoO). Table 3.1 is also provided to highlight the differences between the cores before analyzing them in detail.
Finally, a comparison will be made between the cores in terms of their supported ISA, number of pipeline stages, FUs, execution order, superscalarity, HDL used, and frameworks for RTL generation.

Table 3.1: Comparison between State of the Art processors. The column "Stages" represents the pipeline stages of the architecture.

	Architecture	OoO	ISA	Stages	FUs	HDL
Rocket [18]	scalar	no	RV32/64 IMAFDC	5	4	Chisel
CVA6 [4]	scalar	no	RV64 IMAFDC	6	6	SV
Shakti-C [11]	scalar	no	RV32/64 IMAFDC	5	3	BSV
LAMP [22]	scalar	no	RV32 IMF	5	4	SV
PicoRV [23]	scalar	no	RV32 IMC	5	?	Verilog
MicroRV [23]	scalar	no	RV32 IMC	5	?	SpinalHDL
RI5CY [23]	scalar	no	RV32 IM	4	?	SV
Noel-V[5]	scalar superscalar	no	RV32/64 IMAFDBCH	7	6	VHDL
Dual Pipeline [19]	superscalar	no	RV32 IMAFD	5	3	Verilog
BOOM [17]	superscalar	yes	RV64 IMAFDC	10	8	Chisel

Legend: SV SystemVerilog, BSV Bluespec SystemVerilog.

3.1. In-Order scalar CPUs

3.1.1. Rocket

Rocket [18] is an in-order RISC-V scalar processor developed at University of California, Berkeley. It is written in Chisel HDL, an object-oriented hardware description language based on Scala.

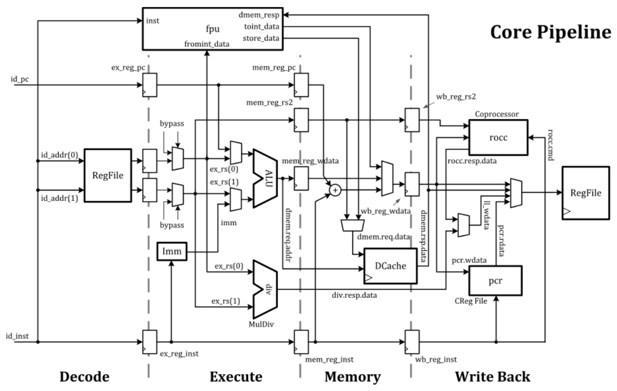


Figure 3.1: Rocket Core Pipeline

The processor under discussion features a 5-stage pipeline, namely: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Writeback (WB). It has the capability to support RV32/64G ISA, which can be conveniently extended with subsets such as M, A, F, and D, thanks to the high abstraction of the Chisel HDL. Furthermore, the branch predictor can be customized with a configurable Branch Target Buffer (BTB), Branch History Table (BHT), and Return Address Stack (RAS). This feature allows for the precise tuning of the processor's performance. To integrate Rocket cores, caches, and interconnects into a cohesive System-on-Chip (SoC), users can employ the open-source SoC design generator Rocket Chip Generator [15]. This tool is integrated within the open-source Chipyard [3] framework, which facilitates the expansion of the execute stage with accelerators and the attachment of a co-processor. The processor has been used as a starting point for development and has seen numerous tape-outs since 2012. SiFive U54, a quad-core processor capable of reaching frequencies up to 1.5 GHz, is among the latest of these tape-outs.

3.1.2. CVA6

CVA6 [4] (formerly named Ariane) is an in-order, single-issue, 64-bit processor. The core is written in SystemVerilog and its micro-architecture is designed to reduce critical path length while keeping IPC losses moderate. The purpose of the core is to run a full OS at reasonable speed and IPC.

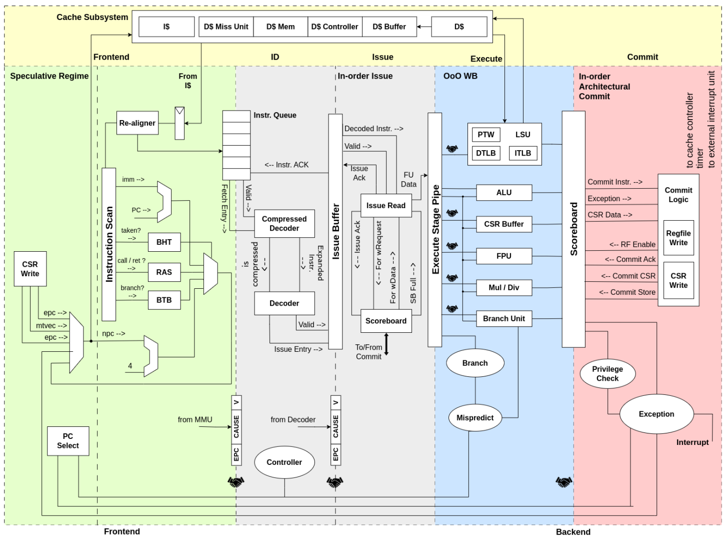


Figure 3.2: CVA6 Core Pipeline. Source: [4]

In order to achieve optimal speed and performance, the core is equipped with a six-stage pipeline. This is similar to the five-stage pipeline of the Rocket architecture, which is discussed in subsection 3.1.1, but includes a dedicated stage for Program Counter (PC) calculation. The Instruction Set Architecture (ISA) that is supported is RV64GC, which includes the MAFD extensions. The front-end is equipped with a branch prediction unit consisting of a Branch Target Buffer (BTB), Branch History Table (BHT), and Return Address Stack (RAS). This unit can be selected based on the requirements of the application. To increase the Instructions Per Cycle (IPC), the CPU is equipped with a scoreboard that issues data-independent instructions to hide the latency to the data RAM (cache). The core consists of six functional units in the execute stage, including an Arithmetic Logic Unit (ALU), a dedicated multiplier/divider, an optional Floating-Point Unit (FPU), which aims to be IEEE 754-2008 compliant, a CSR buffer, branch unit, and a load/store unit (LSU). The core has been integrated into both Chipyard [3] and the OpenPiton [6] projects, making it simpler to generate, simulate, and customize a CVA6-based System on Chip (SoC).

3.1.3. Shakti C-Class

The SHAKTI C-Class, a member of the base family of SHAKTI Processor Program [11], initiated by the IIT Madras in 2014, is a controller-grade processor designed for IoT-, industrial-,and automotive segments.
The core is designed for a frequency range from 500 MHz to 1.5G Hz and is capable of booting and running Linux and RTOS. The processor is written in Bluespec SystemVerilog (BSV), which can be synthesized in Verilog code with an open-source compiler.

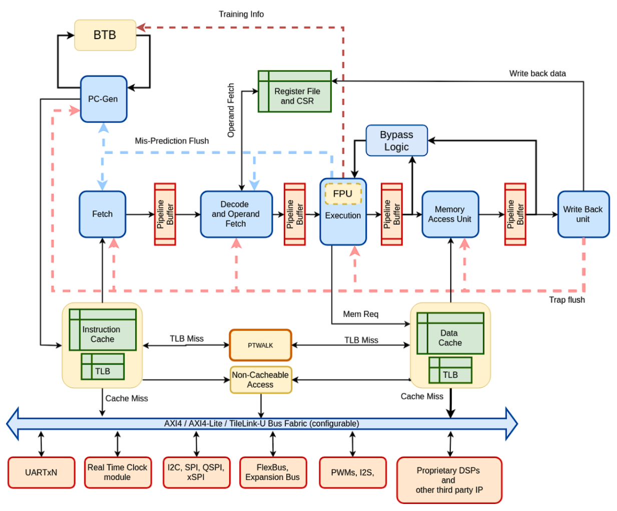


Figure 3.3: Schematic view of a SHAKTI C-Class. Source: [11]

The SHAKTI processor features a 5-stage in-order pipeline and supports both RV32I and RV64I instruction sets, including MAFDC extensions. The processor is highly customizable and permits selective activation of the S and M extensions. The core's front-end incorporates a two-level GShare branch predictor, while the execution stage comprises three distinct functional units: M-Box, F-Box, and ALU. It is important to note that the SHAKTI C-Class processor is not currently integrated into the Chipyard framework. However, the SHAKTI project offers an independent framework for creating System-on-Chips (SoCs), known as "shakti-soc," as well as an SDK called "shakti-sdk" and a verification framework called "RISC-V Trace Analyzer." Overall, the SHAKTI processor's impressive features and high degree of configurability make it a valuable tool for a variety of applications in the business and academic worlds.

3.1.4. LAMP

The proposed CPU [22] features a single-issue, in-order, 5-stage pipeline implementing the standard Integer (I), Multiply (M), and Floating Point (F) extensions of the RISC-V ISA.

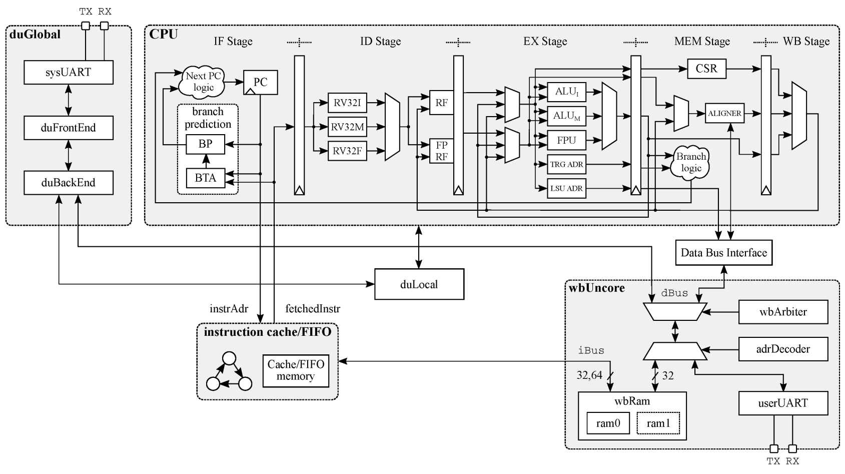


Figure 3.4: Schematic view of the SoC and CPU. Source: [22]

The Instruction Fetch (IF) stage of the CPU can fetch up to a single, 32-bit, fixed-length instruction per clock cycle. Moreover, the program counter is updated in the same clock cycle following the standard RISC-V ISA architectural specification manual. The Instruction Decode (ID) stage extracts from the fetched instruction the information required to drive the Register File (RF), the Floating Point Register File (FP-RF), and the immediate operand logic to set up the operands for the execution stage. The ID stage also forwards the signals derived from the instruction operation code (opcode) to the five functional units (FUs) implemented in the EX stage, i.e., the Arithmetic-Logic Unit, the Integer Multiply and Divide, the Load-Store Adder Unit (LSU-ADR), the Floating Point Unit (FPU) and the jump/branch ALU (TRG-ADR). In the EX stage, the operands are multiplexed with the results from the memory (M) and write-back (WB) stages to implement the EX-EX and M-EX forwarding paths. Moreover, the M-M forwarding path is also implemented in the M stage to optimize the load-store instruction patterns. Both the address and the condition of the branch instructions are computed in the EX stage while the PC update due to a control-flow instruction is delayed up to the M stage as in the standard RISC pipeline.

3.1.5. PicoRV and RavenSoC

PicoRV is a RISC-V Instruction Set Architecture (ISA)-based central processing unit (CPU) that incorporates the RV32I, RV32M, and RV32C extensions, as well as being configurable to work with RV32E, RV32IM, RV32IC, and other combinations. As a co-processor to a field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC), PicoRV is designed to offer high performance and an efficient computing solution. It can also function as a standalone CPU core. PicoRV's primary feature is its customizable bus, which can be tailored to different applications and protocols. This versatility makes the PicoRV32 highly adaptable, and a preferred choice for implementing custom CPUs. Notably, it serves as a base for a significant portion of RISC-V-based processors. PicoRV has been implemented on various FPGAs, including Xilinx Kintex and Virtex series. The CPU utilizes a maximum of 2019 look-up tables (LUTs) for the CPU and 88 LUTs for memory. The processor can operate at clock speeds of up to 769 MHz SOCs, making it a high-performance computing solution for various applications.

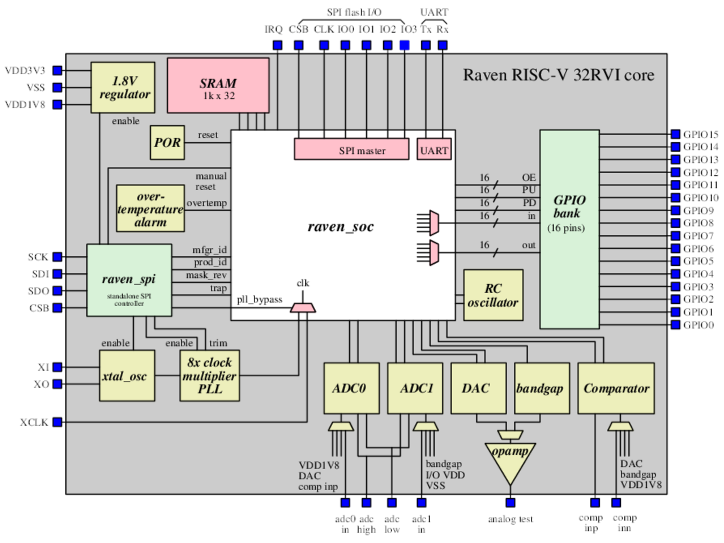


Figure 3.5: Block diagram of RavenSoC. Source: [7]

The RavenSoC, available at [7], leverages a 32-bit RISC-V core, specifically the PicoRV32. The core has undergone significant testing, having been previously implemented in an FPGA, and is incorporated within the RavenSoC as the first System on Chip (SoC) to utilize it. The RISC-V board implementation found in the RavenSoC is particularly well-suited for Internet of Things (IoT) applications, as it offers a range of features that are essential for such work. Furthermore, the code base is both robust and highly extendable, making it an excellent reference for similar designs.

3.1.6. MicroRV

MicroRV32 [23] also known as RV32 is an open-source RISC-V platform that integrates an RV32I-based RISC-V core to several peripherals via a generic bus system. It is implemented in SpinalHDL and can run the FREERTOS operating system. It is developed for educational and research purposes.

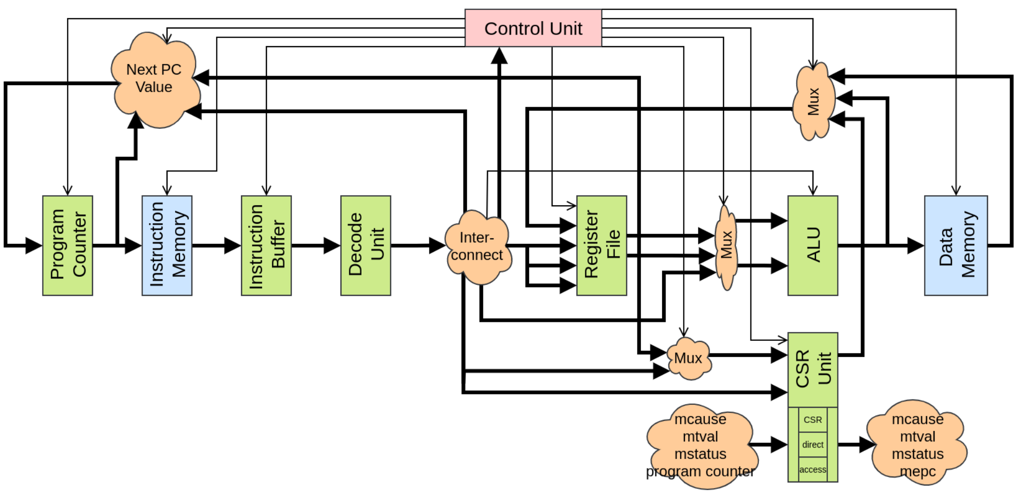


Figure 3.6: DataPath of MicroRV

The CPU core uses an interface consisting of an address, a command, and data to interact with other peripherals. A handshake bus interface is employed, wherein the bus master (CPU core) sends a valid signal to notify the bus slaves (other peripherals) of a payload on the bus. The peripherals are addressed and mapped at the top level, with the transaction packet routed to their respective peripheral based on the memory address. The peripherals respond to the transaction request in a single clock cycle.

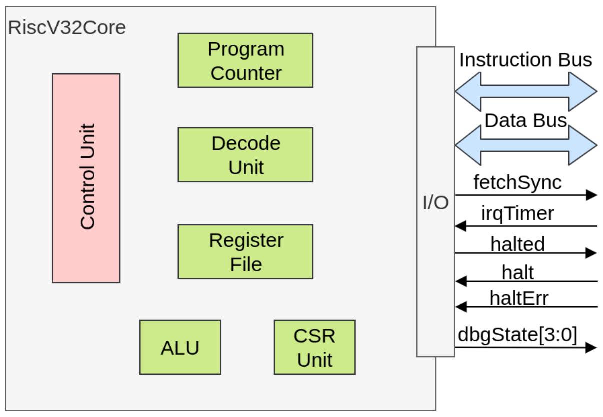


Figure 3.7: Block Scheme of MicroRV

3.1.7. RI5CY

RI5CY is a 32-bit RISC-V core that has been developed as an open-source platform for researchers and developers. It is written in SystemVerilog [14] and supports the RV32IMC instruction set. RI5CY utilizes a single-issue four-stage instruction pipeline and is designed for separate instruction and data memories. Notably, the core does not employ caches, thereby increasing its suitability for various applications. The use of SystemVerilog ensures that the core is both efficient and reliable, enabling researchers and developers to leverage RI5CY for a wide range of applications.

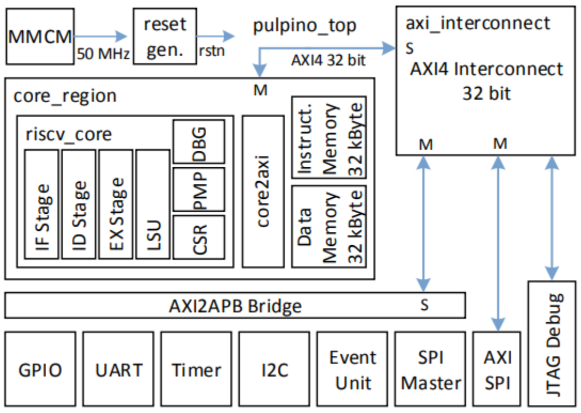


Figure 3.8: Block Scheme of RI5CY [20]

The RI5CY processor, based on the RISC-V architecture, is a highly efficient and reliable solution for researchers and developers seeking to design a wide range of applications. The processor's ability to execute code from separate instruction and data memories enhances its versatility, while its cache-less design ensures suitability for a variety of applications. Moreover, the use of SystemVerilog guarantees robustness and reliability, making RI5CY an excellent choice for businesses and academics alike. The design is synchronous using the rising clock edge of a single clock domain and implements a low active asynchronous reset for all storage elements. Some peripherals are attached to the processor core via a 32-bit AXI on-chip bus system to provide interfaces to GPIO, UART, I2C, SPI, etc.

3.2. In-Order superscalar CPUs

3.2.1. Noel-V

Developed by Frontgrade Gaisler, Noel-V [5] is a customizable RISC-V core capable of being synthesized as both scalar or superscalar, and designed in VHDL. It features multiple levels of configuration, ranging from secure, fault-tolerant, and high performance to area-optimization.

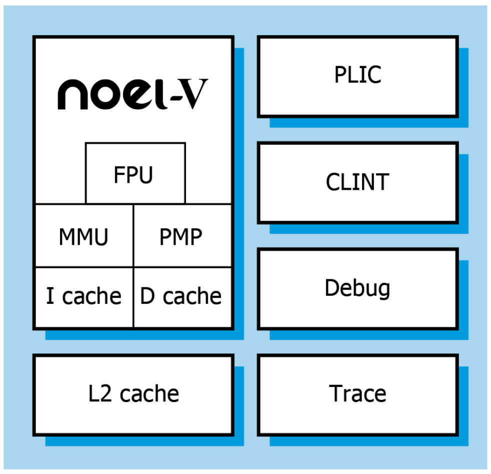


Figure 3.9: Features of the NOEL-V processor. Source: [5]

The processor can have either RV32I or RV64I as its base Instruction Set Architecture (ISA). It also supports different subsets, including MAFDBC. Its F-extension can use either a 32/64-bit non-pipelined, area-efficient Floating Point Unit (FPU) or a high-performance, fully-pipelined IEEE-754 FPU. If used as a processor, it can be configured as a dual or single-issue. However, if it is set up as a controller, it can only be used as a single-issue processor. The architecture includes an advanced branch predictor and cache controller that can maintain a one-cycle-per-store throughput. When set up as a high-performance or general-purpose processor, it can run Linux and supports supervisor and user privilege modes. The processor provides a solution to the problem of Single Event Upsets in critical aerospace applications. It employs a reconstruction scheme that can correct single-bit errors and detect up to 4-bit errors. It can be synthesized using common tools such as Xilinx Vivado, Synplify, and Synopsys DC.

3.2.2. Dual pipeline superscalar

A dual-issue, 32-bit superscalar processor [19] that was presented at the 2020 23rd Euromicro Conference on Digital System Design held in Kranj, Slovenia. It was designed in Verilog.

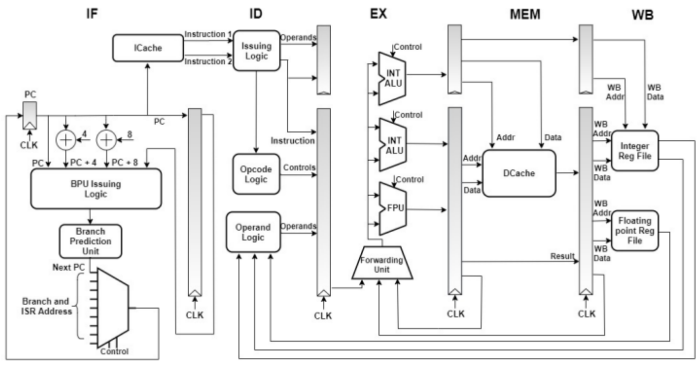


Figure 3.10: Scheme of the Dual Pipeline superscalar processor. Source: [19]

The processor is equipped with a 5-stage dual-pipeline and supports the RV32IMAFD ISA. It utilizes a dynamic branch prediction unit that operates parallelly with the Instruction Decode stage. In the issue stage, two instructions can be dispatched at the same clock cycle if the second instruction does not have any dependencies with the first. Otherwise, only one instruction, in program order, is propagated towards the Execute stage, while the other is held back in the Instruction Decode stage. While creating a bundle, a conditional or unconditional jump is always issued alone. Similarly, no bundle can be created with two memory access operations. Both the instruction cache and data cache are 8 KB, 2-way set associative, with a block size of 32 bytes. The data cache has two memory access ports, one of which is reserved for atomic reads only. This processor was verified through the standard RISC-V Toolchain, which was used to compile standard C code and well-known benchmarks. It was implemented on a Virtex-7 (xc7vx485tffg1761-2) FPGA board, achieving a maximum frequency of 40 MHz. It should be noted that the processor does not support Linux, which necessitates the implementation of a supervisor privilege level. Furthermore, it is noteworthy that the processor supports the use of virtual memory.

3.3. Out-of-Order CPUs

3.3.1. BOOM

BOOM is a processor that uses superscalar Out-of-Order technology and has been designed at the University of California, Berkeley. This highly parametric processor is deeply customizable and has been specifically crafted to act as a baseline for education and research in the field of Out-of-Order processors. Its robustness and versatile nature make it an ideal option for those seeking to explore the intricacies of processor technology. In addition, the BOOM processor has been integrated into the Chipyard framework, which further enhances its value proposition. The BOOM processor is a remarkable achievement in the field of processor technology and is expected to have a significant impact on the research and education community.

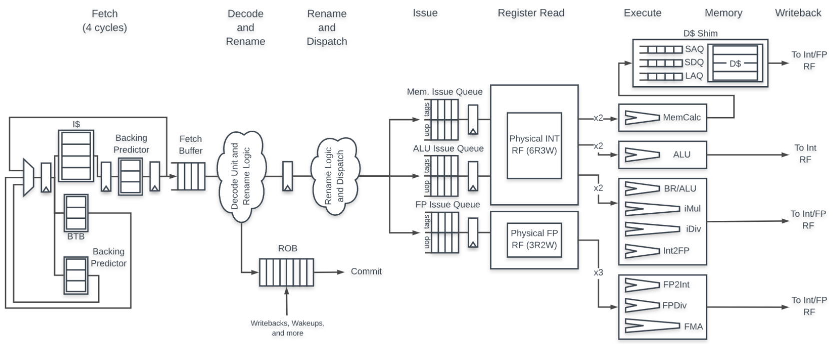


Figure 3.11: Simplified BOOM pipeline. Source: [2]

The BOOM core is a High-Level Description Language (HDL) written in Chisel. It supports the RV64I base Instruction Set Architecture (ISA), which can be extended with the subsets A, F, D, and M. The core was designed based on the Rocket, which is an In-Order, "classic" 5-stage pipeline that has been verified. This helped to simplify the design work for the BOOM core, which implements a 10-stage pipeline. The front end of the BOOM core features a highly configurable branch prediction unit, which is based on two-level predictors such as GShare or TAGE, a Next-Line Predictor (NLP), and 12 cycles of branch-mispredict penalty. The issue width is also configurable, with a micro-scheduler that assigns dispatched operations to an available functional unit, divided between specialized and mixed. Just like the Rocket core, the Rocket Chip Generator allows for easy expansion of the Execute stage, enabling the addition of ISA extensions and/or accelerators. The data cache consists of two dual-ported banks, which can be used to exploit the superscalar properties of a dual-issue architecture.

4 | Methodology

In this chapter, we will describe the microarchitecture of the superscalar CPU, reported in figure 4.1, and the corresponding verification method that was developed. A crucial metric to consider when selecting the target FPGA board is the system's resource utilization, which increases with the issue window width. It is imperative to note that transitioning from a scalar to a superscalar architecture leads to a considerable increase in area occupation. This is due to the need to replicate most of the ports and interconnections to support the propagation of multiple instructions through the pipeline. Furthermore, the logic networks play a significant role, as they must be extended, often doubling the initial complexity, to account for possible dependencies between multiple instructions. Section 4.1 exposes the methodology employed to design a synthesizable hardware architecture of a superscalar CPU and develop a verification tool in detail. Although the focus will be on a dual-issue version, the same reasoning can be conceptually extended to an N-issue one. The subsequent sections will portray the functional ISA simulator adopted, the customization required for verification, and the verification setup and process.

4.1. Microarchitecture of the superscalar CPU

In this section, we will analyze each block of the pipeline stages, explaining all the implemented functionalities and structures using high-level schemes and pseudocode. The description pertains to a dual-issue superscalar architecture, but it can be easily extended to a generic N-issue one. The width of each signal and structure is determined by parameters, and a particular setup is described in Chapter 5.

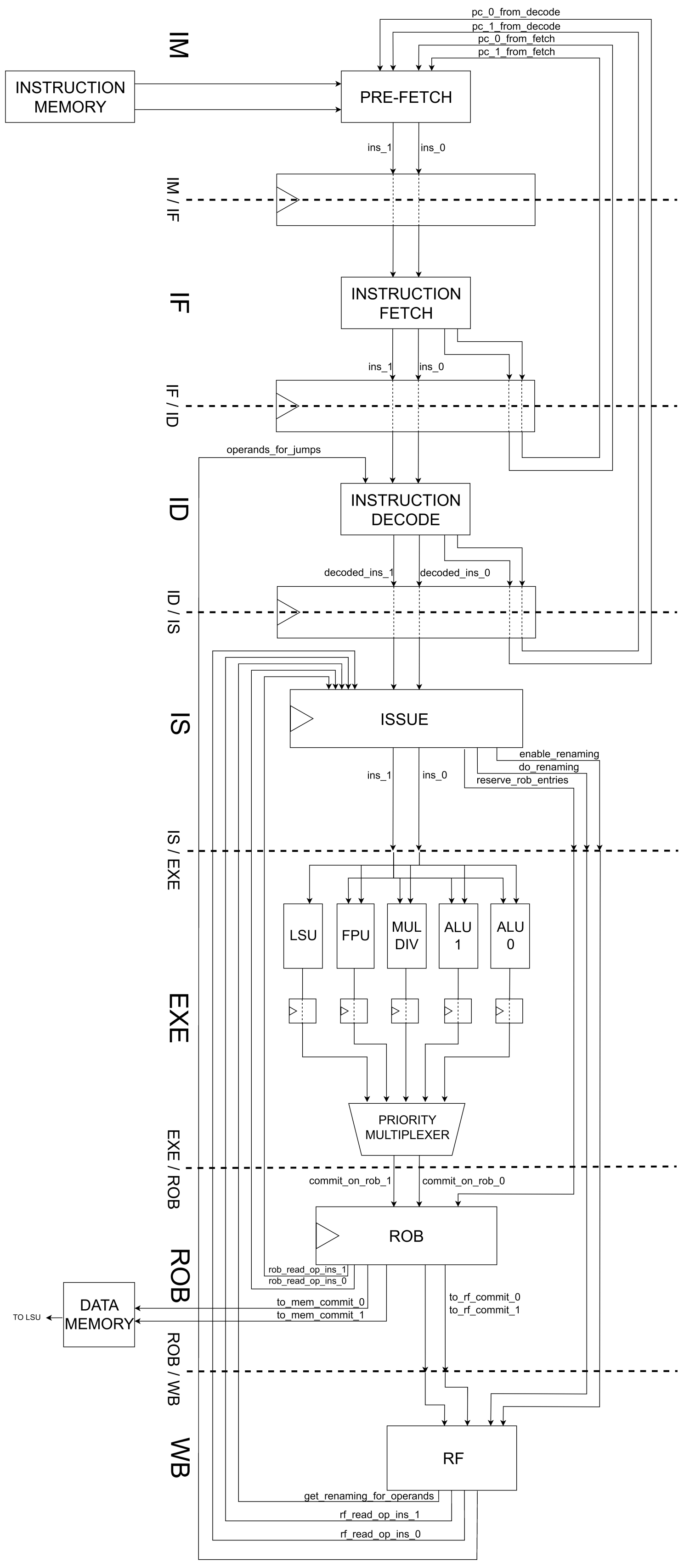


Figure 4.1: High level scheme of the superscalar CPU

4.1.1. Instruction Memory and pre-fetch

The architecture's Instruction Memory (**IM**) is implemented as a RAM that can be accessed from the Fetch stage through the desired Program Counter. The width of its entries depends on the program counter length parameter PC_LEN, which is typically equal to the architectural length. Additionally, its depth is also parametric and is dependent on the instruction memory depth parameter INS_MEM_DEPTH.

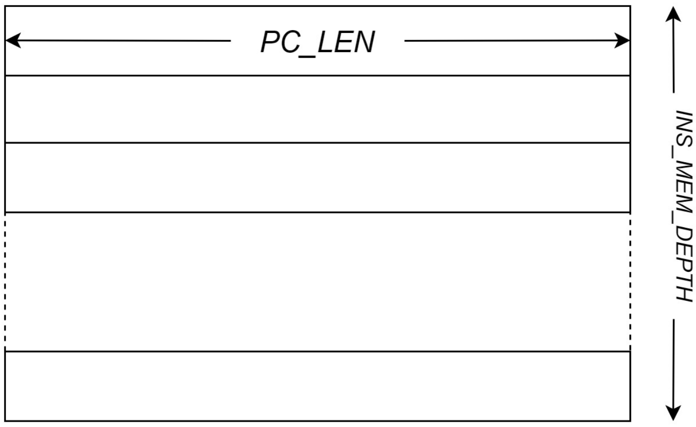


Figure 4.2: Parametric structure of the functional scalar Instruction Memory

A superscalar architecture requires a pre-fetch logic in the Instruction Memory stage, with sufficient input and output ports to support dual-issue properties.

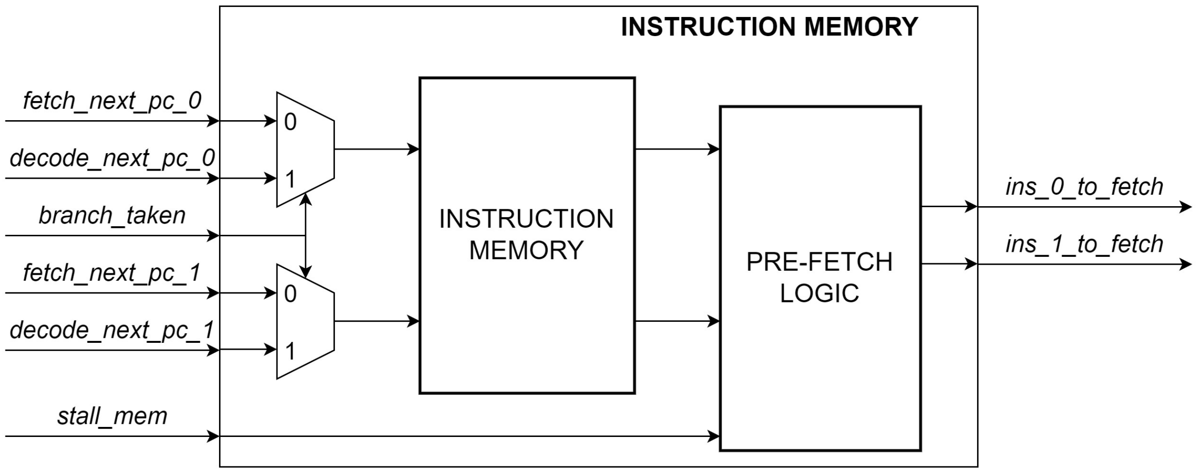


Figure 4.3: High-level view of the Instruction Memory & Pre-Fetch stage of the superscalar CPU

The Instruction Memory contains two pairs of Program Counters that are available for selection. The first pair, denoted as *fetch_next_pc_0* and *fetch_next_pc_1*, originates

from the Instruction Fetch, while the second pair, referred to as *decode_next_pc_0* and *decode_next_pc_1*, is derived from the Instruction Decode's jump logic.
During the Pre-Fetch stage, the last propagated instruction *ins_1_to_fetch* is preserved in an internal variable. On each propagation of a bundle, the currently propagating first instruction of the bundle, *ins_0_to_fetch*, is compared to the stored one. If both are either a conditional or unconditional jump, then the outputs of the stage, *fetched_ins_0* and *fetched_ins_1* with *fetched_pc_0* and *fetched_pc_1*, are not modified. This is because, during the post-jump synchronization period, the one-cycle latency required to assert the *stall_mem* would cause a change in the IF inputs for a single cycle.

4.1.2. Fetch and pre-decode

The Instruction Fetch (**IF**) is the second stage of the pipeline and datapath. It is responsible for extracting two instructions from the memory in a single clock cycle. One of its main features is the ability to perform a "pre-decode" of the input instructions, which allows for synchronization between front end blocks. This is particularly important in the case of subsequent bundles of instructions containing branches.
It must be noted that a CPU does not require to support multi-instruction Fetch stage to be defined as superscalar. However, implementing multi-instruction Fetch stage can prevent potential bottlenecks in the case of an ideal architecture or a simple executable with few data dependencies and 1-cycle operations, as explained in Section 2.3.

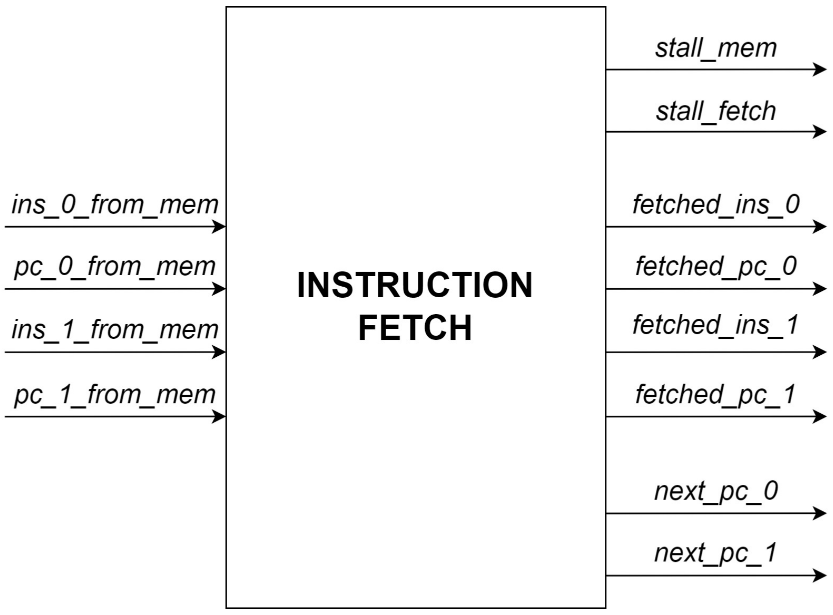


Figure 4.4: High-level view of the Fetch stage of the superscalar CPU

The Instruction Fetch and pre-decode stage in our architecture is designed to operate in a fully-sequential and positive edge-sensitive manner. This stage includes auxiliary signals such as *stall_mem*, *stall_fetch*, and counters that support the synchronization logic of the pre-decode block.

At each clock cycle, two instructions are extracted from the Instruction Memory and sent to the pre-decode logic, as described in Algorithm 4.1, through the block's input ports: *ins_0_from_mem* and *ins_1_from_mem*. The pre-decode logic analyzes the bundle of instructions and distinguishes between three cases: the first instruction is a conditional or unconditional jump, the second one is a jump, or neither are jump instructions.

In the first case, the second instruction of the bundle is flushed, and the jump is propagated as *fetched_ins_0*. This is because our architecture always incurs a cost when encountering a conditional or unconditional jump. After propagating the instruction in this case, the *stall_mem* is asserted to keep its inputs steady, and a number of clock cycles are counted to ensure synchronization with the rest of the front end. After propagation, the *stall_fetch* flag is also asserted to avoid decoding the same instruction multiple times. This is particularly useful in the case of JAL and JALR operations, which must be propagated to the Issue stage, unlike branches, which are treated as NOPs as they are entirely resolved in the Decode stage.

In the second case, the pre-decode logic propagates the first instruction of the bundle as *fetched_ins_0* and stores the second one, the jump, in an internal variable. After one clock cycle, this stored jump instruction is fed to the Decode stage also through *fetched_ins_0*, and the behavior is the same as the first case.

Lastly, in the third case, the bundle of instructions is simply buffered to the Decode stage, and the *next_pc_0* and *next_pc_1* are driven based on the input Program Counters. It should be noted that in each of the cases above, each instruction (*fetched_ins_0* and *fetched_ins_1*) is always propagated together with its corresponding PC (*fetched_pc_0* and *fetched_pc_1*, respectively, coming from the input ports *pc_0_from_mem* and *pc_1_from_mem*).

It is crucial to take note of a critical synchronization point in the process. Whenever the code encounters a conditional or unconditional jump, it is imperative to assert the *stall_mem* flag in order to avoid any changes to the input values of the Fetch stage. This step allows the pre-decode logic to follow the correct logic path during the required synchronization period. This period is guaranteed through the counter mentioned earlier, and therefore it is important to ensure that the *stall_fetch* flag behaves similarly. It is used to prevent multiple issues of valid jump instructions such as Jump And Link (JAL) and Jump And Link Register (JALR).

Algorithm 4.1 ***Pre-decode logic***

```
if instruction[0].isbranch then
    fetch(instruction[0]);
    synch_frontend;
else if instruction[1].isbranch then
    fetch(instruction[0]);
    store(instruction[1]);
    fetch_stored_instruction;
    synch_frontend;
else
    fetch(instruction[0], instruction[1]);
end if

function synch_frontend
    for i = 0, i < synch_time, i++ do
        stall_mem, stall_fetch = 1;
    end for
    stall_mem, stall_fetch = 0;
endfunction
```

The synchronization period represents the minimum number of clock cycles required for time coherence of the front end, and is automatically extended when necessary jump operands are not available right away. Once this period has passed, both stall flags are de-asserted, and the next instructions are fetched in the same clock cycle.
This implies that the preliminary check of the pre-decode must also be performed inside every pre-decode case themselves. This accounts for the presence of conditional or unconditional jumps inside the correct next bundle, and ensures that the process is operating in a synchronized manner.

4.1.3. Decode and jump logic

After the instruction fetch stage, the next step in the pipeline is the superscalar Instruction Decode (**ID**) stage. This hardware stage is responsible for decoding 32-bit RISC-V instructions, which can come from the I-, M-, and F-extensions. The ID stage has two primary functions: decoding the instructions and resolving and calculating jumps.

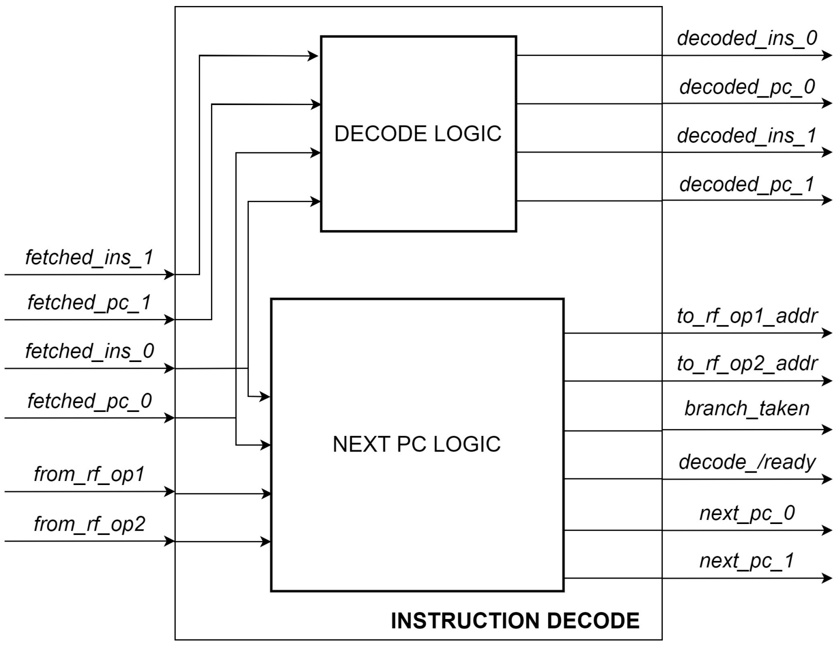


Figure 4.5: High-level view of the Decode stage of the superscalar CPU

The decode logic is responsible for processing the input fetched instructions and their corresponding Program Counters through input ports *fetched_ ins_ 0*, *fetched_ ins_ 1*, *fetched_ pc_ 0*, and *fetched_ pc_ 1*. The logic uses the 7 lower bits of the encoding, known as "opcode," to determine the type of operation being represented. Additionally, it may also inspect other parts of the encoding, such as the *func3* and *func7* fields, to accurately identify the encoded operations.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

Figure 4.6: Types of instruction encodings as described in the RISC-V specification [24]

Following instruction fetching, the system extracts useful information and generates a *decoded instruction (decoded_ ins)*.

Table 4.1: Structure of a decoded instruction (*decoded_ins*).

Entry name	Logic Length (bits)	Explanation
which_fu	WHICH_FU_LEN	Choose the FU
ctl_fu	CTL_FU_LEN	FU operation setup
op1_addr	REG_IND_LEN	Address of op1
op1_f_noti	1	'0' if op2 is integer
op2_addr	REG_IND_LEN	Address of op2
op2_f_noti	1	'0' if op2 is integer
imm	ARCH_LEN	Immediate value for op2
op2_i_notr	1	'1' if op2 is immediate
dest_addr	REG_IND_LEN	Address of the destination
dest_rob_id	ROB_IND_LEN	ROBid of the instruction
dest_f_noti	1	'0' if destination is integer
is_store	1	'1' if instruction is a store
is_load	1	'1' if instruction is a load
st_amt	2	bytes to be stored
exc	1	'1' if exception triggered
exc_code	EXC_CODE_LEN	Type of exception
valid	1	'1' if instruction is valid

Legend: WHICH_FU_LEN length of FU selector, CTL_FU_LEN length of the FU setup vector, REG_IND_LEN length of the RF addresses, ARCH_LEN architectural length, ROB_IND_LEN length of the ROBid, EXC_CODE_LEN length of the exception code.

A decoded instruction consists of the addresses of the destination and operands, the extended immediate value (if required), the functional unit responsible for performing the operation, the type of calculation to be performed, and flags that indicate whether the instruction is a load or a store. Additionally, the offset immediate and byte amount associated with store instructions and any exception flags are included in the decoded instruction.
Once the instruction is decoded, the resulting *decoded_ins* is transmitted to either of the output ports, *decoded_ins_0* or *decoded_ins_1*, along with the corresponding Program Counter, *decoded_pc_0* or *decoded_pc_1*.
The jump logic is another crucial component of the system, which resolves and calculates

jump operations. This logic first verifies the opcode and *func3* fields of the *fetched_ins_0* to distinguish between conditional and unconditional jump instructions. It then performs different operations for each type.
For unconditional jumps, such as Jump and Link (JAL) instructions, the immediate value is extracted, and a signed addition with the instruction PC is performed. On the other hand, Jump and Link Register (JALR) instructions, which require a base address, undergo an additional step. The jump logic first checks the RF for the availability of the operand. A request is then propagated to the output *to_rf_op1_addr*. Once the operand is ready, the immediate value is extracted, and a signed addition is performed.
For branch instructions, two operands are requested from the RF through *to_rf_op1_addr* and *to_rf_op2_addr*, and their values are obtained from *from_rf_op1* and *from_rf_op2*. These values are then used to calculate the jump condition. If true, the next PC is calculated as the sum of instruction PC and jump immediate. Otherwise, the value of PC + 4 and PC + 8 is propagated. The calculated next program counters are communicated to the Instruction Memory through the output ports *next_pc_0* and *next_pc_1*, along with the *branch_taken* flag. The *decode_ready* signal is propagated towards the Instruction Fetch stage.
The jump logic must ensure synchronization with the front end. Whenever a conditional or unconditional jump instruction reaches the Decode stage, the front end must be stalled to keep the inputs of the Decode fixed. This ensures that the jump logic is given enough time to re-enter the same logic path if the jump resolution takes more than one cycle, as in the case where it waits for a non-ready operand. This synchronization results in a minimum of one clock cycle latency whenever a jump is resolved.
The decode logic is implemented as a sequential positive edge-sensitive net. The jump logic, on the other hand, requires the creation of a set of ports to propagate a read address to the RF and receive from it the status of the requested register and its value. This is done through combinatorial logic, which implements an asynchronous inter-module communication that resolves the read request in a single clock cycle. The downside of this solution is the high propagation delay of the logic, resulting in a limitation on the maximum achievable frequency.
The *branch_taken* signal is asserted every time a conditional or unconditional jump instruction reaches the Decode stage, changing the behavior of the Instruction Memory as described in Subsection 4.1.1.

4.1.4. Issue

Once a bundle has been decoded, it is then sent to the Issue (**IS**) stage. This stage is responsible for dispatching instructions to the Execute stage and renaming operands to ensure that operations are executed and completed correctly, in an Out-of-Order manner. The fill queue logic is the first step for each instruction, after which the instruction is dispatched through the double read operands logic.

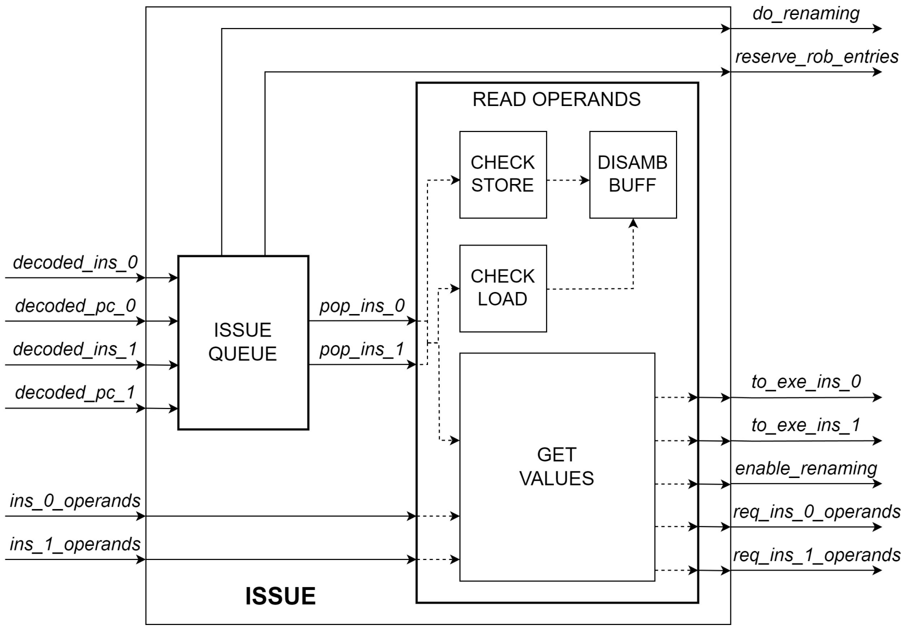


Figure 4.7: High-level view of the hardware Issue stage of the superscalar CPU

A bundle of decoded instructions is first processed by the *fill queue* logic. This logic checks if the incoming instructions, through the input ports *decoded_ ins_ 0* and *decoded_ ins_ 1*, are valid. If they are valid, they are queued in a first-in, first-out structure called the *Issue Queue* (IQ). The IQ has a configurable depth called *ISSUE_ DEPTH*. Each entry in the IQ has multiple fields, most of which mirror the composition of a decoded instruction. These fields include the functional unit selector, operand 1 and 2 addresses, immediate value (if expected by the decoded instruction), destination address, exception signals, memory access-related signals, and a valid signal for the handshake.
The Issue Queue also includes two additional fields: the destination *ROBid*, which points to the first free cell of the ROB, and the operand renamings, which are obtained from the extended Register File.

Table 4.2: Structure of an Issue Queue entry.

Entry field name	Logic Length (bits)	Explanation
which_fu	WHICH_FU_LEN	Choose the FU
ctl_fu	CTL_FU_LEN	FU operation setup
op1_addr	REG_IND_LEN	Address of op1
op1_f_noti	1	'0' if op2 is integer
op1_renamed	ROB_IND_LEN	Renaming of op1
op2_addr	REG_IND_LEN	Address of op2
op2_f_noti	1	'0' if op2 is integer
op2_renamed	ROB_IND_LEN	Renaming of op2
imm	ARCH_LEN	Immediate value for op2
op2_i_notr	1	'1' if op2 is immediate
dest_addr	REG_IND_LEN	Address of the destination
dest_rob_id	ROB_IND_LEN	ROBid of the instruction
dest_f_noti	1	'0' if destination is integer
is_store	1	'1' if instruction is a store
is_load	1	'1' if instruction is a load
st_amt	2	bytes to be stored
exc	1	'1' if exception triggered
exc_code	EXC_CODE_LEN	Type of exception
valid	1	'1' if instruction is valid

Legend: WHICH_FU_LEN length of FU selector, CTL_FU_LEN length of the FU setup vector, REG_IND_LEN length of the RF addresses, ARCH_LEN architectural length, ROB_IND_LEN length of the ROBid, EXC_CODE_LEN length of the exception code.

The logic in question serves multiple purposes in addition to filling the queue. It is responsible for writing the instruction PC, destination address, and the number of stored bytes (in case of a store instruction) to the destination cell of the Reorder Buffer (ROB). This ensures that all the necessary information is propagated through the output ports, specifically the *reserve_rob_entries* port. Furthermore, it writes a request on the *do_renaming* port and updates the renaming field of the Register File (RF) entry. This is achieved by writing the destination ROBid to the renaming field, which is pointed to by the destination address of the decoded instruction. The corresponding busy flag in the RF is not

asserted at this point, to prevent an instruction from writing a register that is also its operand, thus avoiding stalling indefinitely.
Finally, the busy flag for branches of the destination register in the RF is asserted to make a subsequent jump instruction in the Decode stage stall if it requires the value produced by the current instruction. To simplify the explanation of this complex process, the pseudocode provided below can be referred to.

Algorithm 4.2 ***Fill queue***

```
if decodedins.valid then
    issueq[wr_cnt] = decodedins;
    get_renaming_fromRF(decodedins.op1, decodedins.op2);
    reserve_rob_entry(decodedins.PC,decodedins.dest_addr,decodedins.store_amt);
    if !decodedins.isstore then
        rename_destination(decodedins.destaddr);
    end if
end if
```

This routine is executed for both instructions in the bundle, but there are some differences between instruction[0] and instruction[1], the two input instructions of the stage. Firstly, the write counter used to fill the IQ is duplicated so that each instruction in the bundle has its own counter. For example, for instruction[1], the counter is simply one position ahead of the one for instruction[0], making it easier to fill the queue. The logic for incrementing the write counters becomes more complex for the bundle than for a single instruction since they must be incremented based on whether there is a single valid instruction or an entire valid bundle. Another difference is in the steps to obtain the operand renamings. If an operand of instruction[1] is produced by instruction[0], its renaming ROBid becomes the value of the destination ROBid of instruction[0]. Otherwise, due to the one-cycle delay in updating the contents of the extended RF, the second instruction would get the wrong ROBid for one of its operands. The busy assertion, which corresponds to the enabling of the renaming through the *enable_renaming* output port, is performed when an instruction is dispatched. The double read operands function retrieves one or two instructions from the Instruction Queue (IQ) in FIFO order. It checks if the required operands are available in the Register File (RF) and the Re-Order Buffer (ROB), dispatching them to the Execute stage. It also signals which functional unit (FU) needs the operands, which operation it should perform, the destination ROBid, and whether the operands are ready or not. The *get_operand()* function is executed only when both *dispatch_op1_ready* and *dispatch_op2_ready* are asserted.

Algorithm 4.3 ***Double read operands***

```
if readop_en & pop_en then
    get_value(issueins[0].op1_addr)
    if decodedins.op2isreg then
        get_value(issueins[0].op2_addr)
    else if decodedins.op2isimm then
        dispatchins[0].op2 = decodedins[0].imm;
    end if
    check_store(issueins[0]);
    check_load(issueins[0]);
    dispatch_instruction(issueins[0]);
    enable_renaming;
    //The same applies for instruction[1], with the differences highlighted in Algorithm
    4.4
end if

function get_value(issueins.op_addr)
if rf_ready then
    dispatchins.op = rf.data[issueins.op_addr];
else if rob_ready then
    dispatchins.op = rob.data[issueins.op_addr];
else
    dispatchins.ready = 0;
end if
endfunction

function check_store(issueins)
if issueins.is_store then
    dispatch_st_imm(issueins.op1,issueins.op2);
    update_disambbuff;
end if
endfunction

function check_load(issueins)
if issueins.is_load then
    if issue.mem_addr == disambbuff then
        stall_instruction(issueins.op1,issueins.op2);
    end if
end if
endfunction
```

Before dispatching the instruction, the function checks whether it is a store or a load. If it is a store, it writes the memory destination address to a cell in the Disambiguation Buffer. This cell is emptied when the corresponding store instruction is committed. If it is a load, the function checks if the memory address targeted by the load instruction is currently being written by a store. In that case, it stalls the load until the needed memory cell is written. These two functionalities are necessary to avoid Read-After-Write (RAW) hazards on memory cells. The double read operands function treats instruction[0] and instruction[1] differently. When instruction[0] produces an operand, an additional check is performed while trying to read the operand from the RF. Since the renaming of a register is enabled during the dispatch of an instruction targeting it, the Issue stage needs to obtain the operand after instruction[0] exits from the Execute stage. Otherwise, it may read an incorrect value. Once the double read operands function completes, the dispatched instructions described in Algorithm 4.3 are transmitted to the Execute stage through the *to_exe_ins_0* and *to_exe_ins_1* output ports.

Algorithm 4.4 Differences during *double read operands*

```
instr[0]:
if rf_ready then
    ...
end if
instr[1]:
if (rf_ready) & (instr[0].dest != instr[1].op_addr) then
    ...
end if
```

The double read operands is implemented as a combinatorial network because it requires multiple subsequent checks to ensure availability of the required values. Similar to the jump logic data request net mentioned in Subsection 4.1.3, there are two sets of asynchronous interconnections for both the dispatched instructions. One set checks for the availability of values inside the Register File, while the other looks inside the Reorder Buffer. These networks impose an upper limit on the maximum achievable frequency, just as it happens in the ID stage.

4.1.5. Execute

The Execute (**EXE**) stage is responsible for producing the final result values. It consists of a variable number of functional units (FUs), which can be selected during the design phase based on the specific requirements of the application.

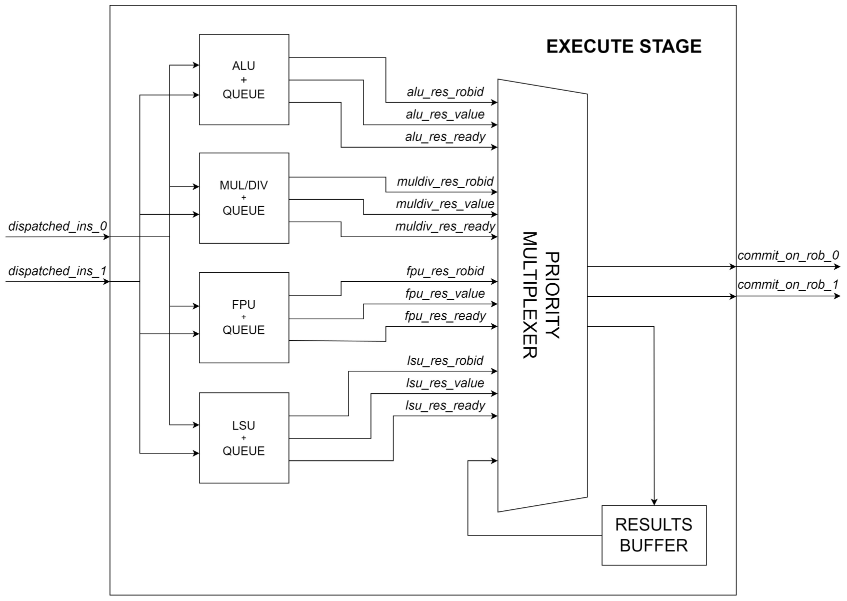


Figure 4.8: High-level view of the Execute stage inside the hardware superscalar CPU

The supported types of FUs are:

- **ALU**: performs integer arithmetic-logic operations. It can work on 32-bit words performing sum, subtraction, bit shifts, and bitwise logical operations such as NOT, AND, NAND, OR, NOR, XOR, XNOR, and CSR directives.
- **MULDIV**: executes different types of multiplications, divisions, and remainder, such as MUL, DIV, REM, and their corresponding signed/unsigned variations.
- **LSU**: performs all the memory-related operations. Is capable of executing Load and Store operations SW and LW, including the subset for Byte and Half-Word instruction, like SB, LB, SH, and LH.
- **FPU**: it can execute floating-point operation. In addition, it is also used for particular instructions that require the single-precision property of Floating-Point numbers.

The needed FU can be selected through the WHICH_FU signal, and the operation to perform through the CTL_FU, described in Subsection Decode and jump logic 4.1.3 and propagated by the Issue stage together with the operands and their status.

The first method implemented in the Execute stage is the *get_ operands*, which is performed by every FU and checks, in order: whether or not the instruction has already been dispatched (to avoid unnecessary multiple iterations of the same operation), if the operands are ready and if the target FU (WHICH_FU) corresponds to the functional unit ID, an attribute unique to every type of unit. The proposed algorithm represents the routine to get each of the dispatched instructions from the Issue stage.

Algorithm 4.5 ***Get Operands***

```
if !instr_already_dispatched then
    if instr_ready & (FU_id == dispatchins.which_fu) then
        get_values(dispatchins);
    end if
    instr_already_dispatched = 1;
end if

function get_values(dispatchins)
    fu_op1 = dispatchins.op1;
    fu_op2 = dispatchins.op2;
    fu_dest_addr = dispatchins.dest_addr;
    fu_dest_rob_id = dispatchins.dest_rob_id;
    fu_res_f_noti = dispatchins.res_f_noti;
    fu_st_imm = dispatchins.st_imm;
    fu_ctl = dispatchins.ctl_fu;
endfunction
```

To ensure proper execution of bundles requiring the same functional unit, a double dispatch mechanism is employed. This means that every type of functional unit must have a queue. Whenever a bundle is created, a function named *fill_ exe_ queue* is called. This function uses the values obtained through the *get_ operands* function to fill the queue of the FU. Otherwise, if a bundle contains instructions that go to the same FU, it would be impossible to execute them. Each of these two methods is called twice by every FU, with a different flag indicating the targeted instruction.

Algorithm 4.6 ***Execute stage logic***

```
functionalUnits = FU0, FU1, FU2, ...;

get_dispatched_ins(0);
get_dispatched_ins(1);
for FU in functionalUnits do
    FU.exe();
end for

function get_dispatched_ins(bundle_position);
    for FU in functionalUnits do
        FU.get_operands(bundle_position));
        FU.fill_exe_queue(bundle_position));
    end for
endfunction
```

When the process reaches its final step, the method *Exe* is invoked. It is in charge of identifying the type of operation that needs to be executed based on the CTL_FU signal.

Algorithm 4.7 ***Exe*** *(Single-Cycle non-pipelined)*

```
case fu_ctl:
    00000 : exe_add(fu_op1, fu_op2, fu_dest_rob_id);
    00001 : exe_sub(fu_op1, fu_op2, fu_dest_rob_id);
    ...
endcase;

function exe_add(op1, op2, dest_rob_id)
    res_value = op1 + op2;
    res_rob_id = dest_rob_id;
    res_ready = 1;
endfunction

function exe_sub(op1, op2, dest_rob_id)
    res_value = op1 - op2;
    res_rob_id = dest_rob_id;
    res_ready = 1;
endfunction
```

In the case of a single-cycle operation, the method generates the output result, whereas, for multi-cycle operations, the pipeline progresses, writing the corresponding output onto the FUs output registers. Once the operation is finished, the corresponding FUs' *result_ready* flags are asserted by the method.
What is reported above in Algorithm 4.7 is an example of the pseudocode for a simplified *Exe* method. This structure can be extended to cover other cases as needed. To achieve the Multi-Cycle pipelined FUs, FIFO-like buffers are used. Algorithm 4.8, reported below, is used to mimic the progress of the pipeline.

Algorithm 4.8 ***Exe*** *(Multi-Cycle pipelined)*

```
case fu_ctl:
    00000 : exe_add(fu_op1, fu_op2, fu_dest_rob_id);
    ...
    res_value = value_fifo[read_cnt];
    res_rob_id = dest_rob_id_fifo[read_cnt];
    res_ready = 1;
    read_cnt = read_cnt + 1;
endcase;

function exe_add(op1, op2, dest_rob_id)
    value_fifo[write_cnt] = op1 + op2;
    dest_rob_id_fifo[write_cnt] = dest_rob_id;
endfunction
```

Once the Execute stage produces a result, the commit process is managed by a Priority Multiplexer, which verifies every FU that has the *res_ready* signal asserted and writes the result of two operations on the Reorder Buffer (ROB). If more than two operations are finished in the same clock cycle, the remaining ones are stored in a small buffer and become the next ones to be committed. When more than two operations are ready to be committed, the priority is given to load and store instructions to speed up the availability of results. These results are often required by subsequent instructions. The algorithm outlines the high-level description of what happens for each of the two committing operations.

Algorithm 4.9 ***Priority Multiplexing***

```
if LSU.res_ready then
    rob.data[LSU.res_rob_id] = LSU.res_value;
    rob.res_ready[LSU.res_rob_id] = LSU.res_ready;
    buffer = other_FUs_ready_results;
else if !buffer.empty then
    rob.data[LSU.res_rob_id] = buffer.res_value;
    rob.res_ready[LSU.res_rob_id] = buffer.res_ready;
    buffer = other_FUs_ready_results;
else if FU0.res_ready then
    rob.data[FU0.res_rob_id] = FU0.res_value;
    rob.res_ready[FU0.res_rob_id] = FU0.res_ready;
    buffer = other_FUs_ready_results;
else if FU1.res_ready then
    rob.data[FU1.res_rob_id] = FU1.res_value;
    rob.res_ready[FU1.res_rob_id] = FU1.res_ready;
    buffer = other_FUs_ready_results;
else if FU2.res_ready then
    rob.data[FU2.res_rob_id] = FU2.res_value;
    rob.res_ready[FU2.res_rob_id] = FU2.res_ready;
    buffer = other_FUs_ready_results;
else if FU3.res_ready then
    rob.data[FU3.res_rob_id] = FU3.res_value;
    rob.res_ready[FU3.res_rob_id] = FU3.res_ready;
    buffer = other_FUs_ready_results;
end if
```

4.1.6. Re-order Buffer and Commit Router

The Reorder Buffer (**ROB**) is a FIFO-like structure with a parametric depth (ROB_DEPTH). Entries are composed of multiple fields, described in table 4.3. Its main function is to commit instructions towards either the Register File or the Data Memory.

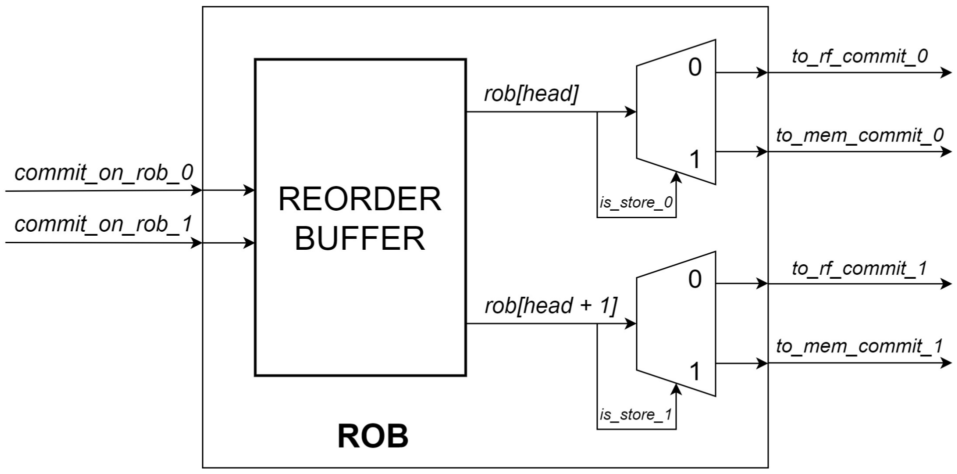


Figure 4.9: High-level view of the hardware Reorder Buffer stage of the superscalar CPU

Table 4.3: Structure of a Reorder Buffer entry.

ROB entry field name	Logic Length (bits)	Explanation
ins_pc	PC_LEN	Instruction PC
res_ready	1	'1' if the result is ready
res_value	ARCH_LEN	Value of the result
res_f_noti	1	'0' if result is integer
res_addr	REG_IND_LEN	Address of the destination register
is_store	1	'1' if instruction is a store
store_amt	2	Amount of stored bytes
mem_dest	MEM_IND_LEN	Data Memory address
exc	1	'1' if exception triggered
exc_code	EXC_CODE_LEN	Type of exception

Legend: PC_LEN length of the PC, ARCH_LEN architectural length, REG_IND_LEN length of the RF addresses, MEM_IND_LEN length of the Data Memory addresses, EXC_CODE_LEN length of the exception code.

First of all, the values produced by the Execute stage are committed on the ROB thanks to a sequential logic which checks for the validity of the result and writes it in the *dest_rob_id* cell, asserting the corresponding *res_ready* flag. All the values and accessory signals necessary for the commit on ROB are included in the *commit_on_rob_0* and *commit_on_rob_1* input ports for simplicity of representation.

Algorithm 4.10 ***Commit on ROB***

```
if commit_enable[0] then
    rob.res_value[dest_robid_0] <= commit_on_rob_0_data;
    rob.res_ready[dest_robid_0] <= 1'b1;
end if
if commit_enable[1] then
    rob.res_value[dest_robid_1] <= commit_on_rob_1_data;
    rob.res_ready[dest_robid_1] <= 1'b1;
end if
```

The Reorder Buffer (ROB) plays a pivotal role in maintaining program order in Out-of-Order completion architectures. Its primary role is to commit the results of completed instructions by using signals such as *rob.data[head]* and *rob.data[head+1]*. Consequently, the ROB must generate the result and additional information for the head instructions inside it. This is accomplished through a sequential logic that verifies if the *res_ready* flags of the head cells are asserted and propagates everything needed for the commit towards a module called the Commit Router. Each Commit Router directs one of the committing instructions towards the Register File or the Data Memory, depending on the *is_store* flag of the ROB entry. It drives the output signals *to_rf_commit_0*, *to_rf_commit_1*, *to_mem_commit_0*, and *to_mem_commit_1*.
To simplify the explanation, Algorithm 4.11 models the production of the committing instructions from the ROB and the routing process. Additionally, there is a combinatorial logic used to send the value requested by the double read operands in the Issue stage. This logic must be realized in a combinatorial way to obtain the exact value at the same clock cycle the data is required, which results in a higher area occupation.

Algorithm 4.11 ***Commit from ROB***

```
if rob.res_ready[head] then
  if !is_store_0 then
    to_rf_commit_0 <= rob.data[head];
  else
    to_mem_commit_0 <= rob.data[head];
  end if
end if
if rob.res_ready[head + 1] then
  if !is_store_1 then
    to_rf_commit_1 <= rob.data[head+1];
  else
    to_mem_commit_1 <= rob.data[head+1];
  end if
end if
```

4.1.7. Register File

The last pipeline stage of the superscalar CPU is the Register File (**RF**), which stores the values of all the registers required by the ISA.

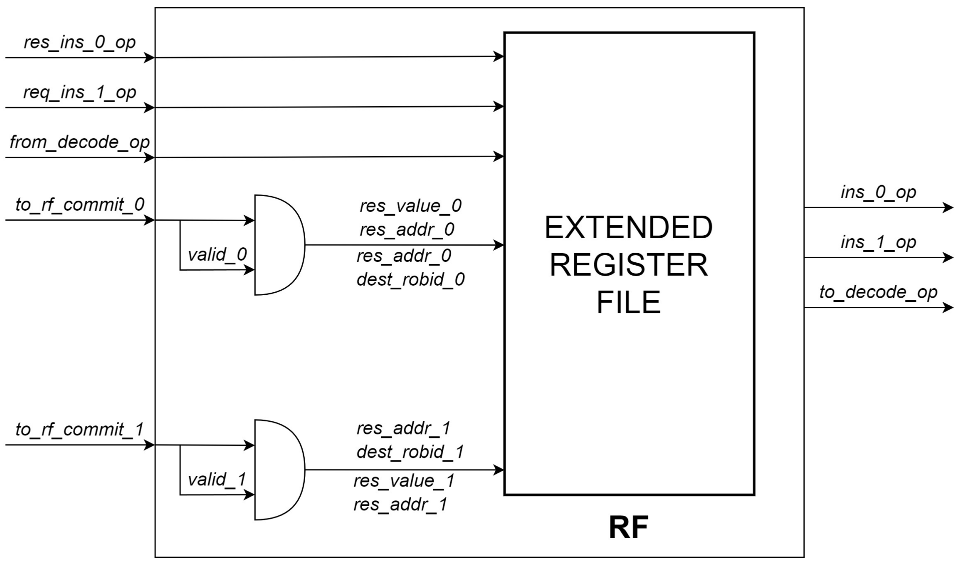


Figure 4.10: High-level view of the hardware Register File stage of the superscalar CPU

The dimensions of the RF (register file) are defined by different parameters, in particular: *REG_IND_LEN* defines the RF depth, and *ARCH_LEN* its data field width. In addition to this basic field of the RF, an extension is added to allow renaming. For this purpose, $ROB\_IND\_LEN + 2$ bits are added to the initial ones, creating the $BUSY$ and B_BUSY fields. These are needed to store the ROBid of the operation targeting that register and to indicate whether the register value is updated or not.

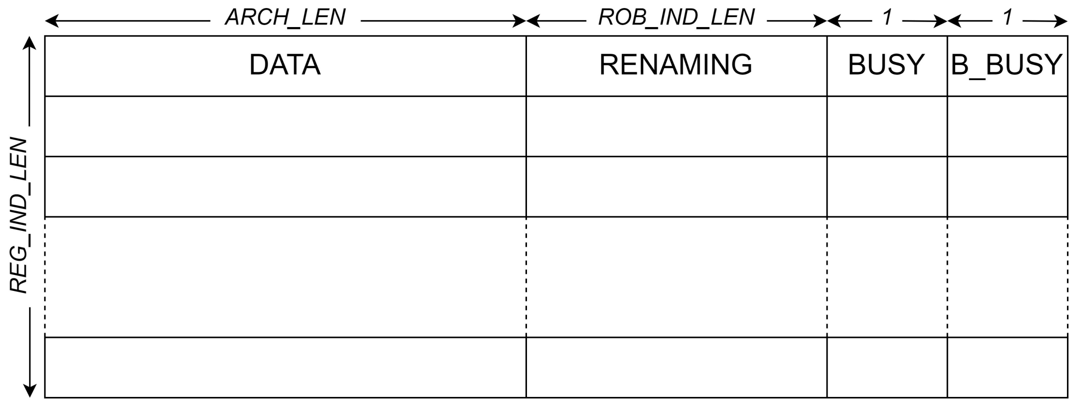


Figure 4.11: Structure of the extended Register File

Upon the ROB committing an instruction on the RF inputs denoted by *to_rf_commit_0* and *to_rf_commit_1* and subsequently receiving a valid assertion, a routine is initiated. The first step of this routine involves the data being committed onto the RF cell, following which the renaming pertaining to the targeted address is reversed. This process effectively clears all RF extensions associated with the aforementioned renaming.

Algorithm 4.12 ***Commit on RF***

```
if valid_0 then
    data[dest_addr_0] <= res_value_0;
end if
if valid_1 then
    data[dest_addr_1] <= res_value_1;
end if
```

To ensure a responsive and accurate renaming mechanism, a sequential logic has been implemented to manage both the execution and reversal of register renaming. This is necessary because a committed instruction may attempt to undo the renaming of a cell that has been targeted by the fill queue during the same clock cycle. Additionally, two asynchronous combinatorial networks have been added to the Register File to fulfill operand requests from the double read operands logic in the Issue stage (*req_ins_0_op* and *req_ins_1_op*) and the jump logic of the Decode stage (*from_decode_op*). These networks transmit the busy status and value of requested registers and can be found at the bottom-left part of Figure 4.1. The busy status and register values are communicated to the Issue stage through the *ins_0_op*, *ins_1_op*, and *to_decode_op* outputs. While this allows for the resolution of a read request in a single clock cycle, it also becomes the primary constraint on the maximum attainable frequency.

4.1.8. Data Memory

Regarding the Data Memory, the only detail worth discussing is the eviction of the Disambiguation Buffer cell during the commit of a store instruction. The Algorithm that follows is a high-level representation of the routine performed by the testbench to commit a store instruction on the Data Memory.

Algorithm 4.13 ***Commit on Mem***

```
function commit_on_mem(res_value, mem_dest, st_amt)
    case st_amt
    1 :
    mem[mem_dest] = res_value[7:0];
    2 :
    mem[mem_dest] = res_value[7:0];
    mem[mem_dest + 1] = res_value[15:8];
    4 :
    mem[mem_dest] = res_value[7:0];
    mem[mem_dest + 1] = res_value[15:8];
    mem[mem_dest + 2] = res_value[23:16];
    mem[mem_dest + 3] = res_value[31:24];
    endcase
    clear_disambbuff(mem_dest);
endfunction

function clear_disambbuff(mem_dest)
    for i = 0, i < DISAMB_LEN, i++ do
        if mem_dest == ISSUE.disambbuff[i] then
                ISSUE.disambbuff[i] = 0;
        end if
    end for
endfunction
```

4.2. Examples of execution

This section presents examples of execution to better illustrate the features and capabilities of the superscalar CPU, as well as its limitations. The examples are divided into three subsections, based on the presence of data dependencies and jump instructions, whether conditional or unconditional. Multiple graphs are presented to describe the pipeline's temporal advancement in terms of clock cycles (cc). The "Ideal" graphs show a theoretical behavior where RAWs (Read After Write) are not considered a problem, while the "Real" graphs account for the delay created to resolve these dependencies.

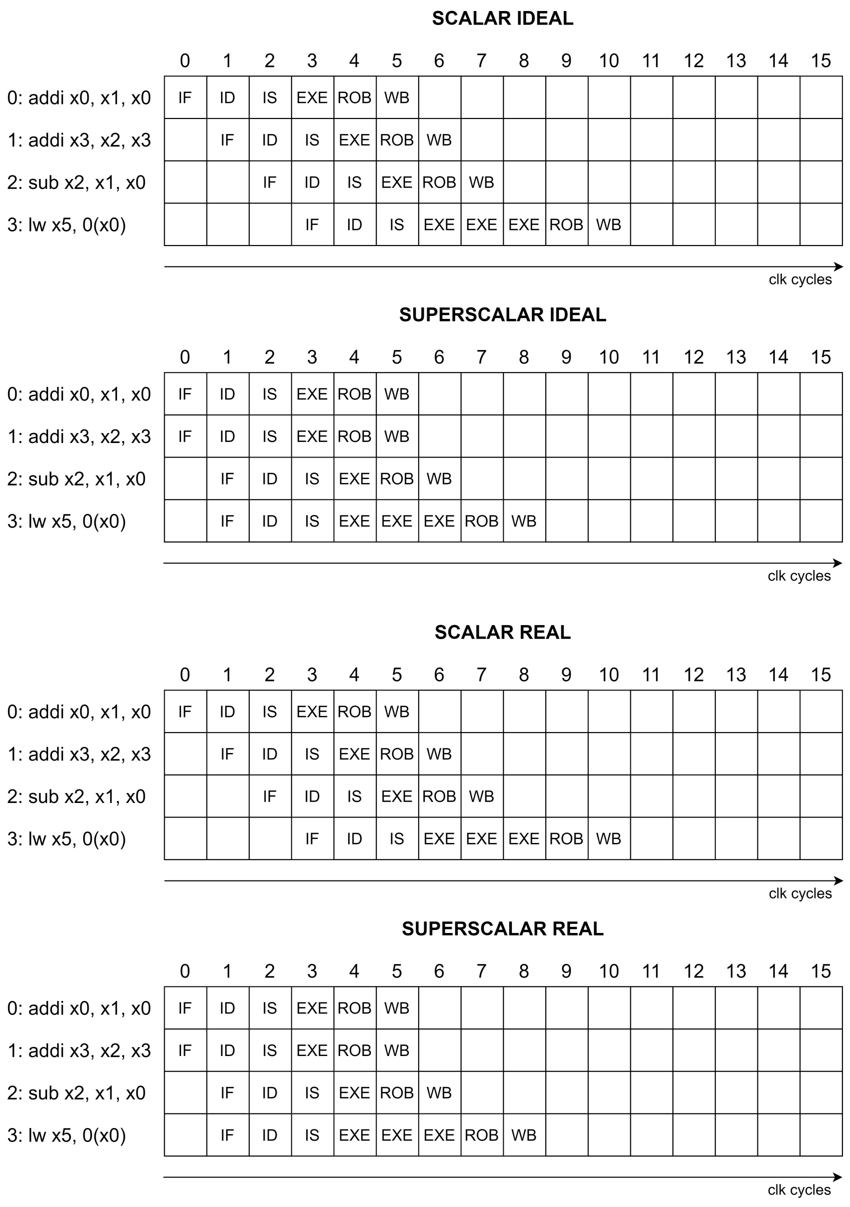
Two CPUs are compared: a single-issue scalar and a dual-issue superscalar. Both have two single-cycle ALUs and one three-cycle pipelined LSU functional units. The results of a FU become available after the end of the Execute stage. The only difference between the architectures is the size of the instruction bundle, which is one for the scalar and two for the superscalar. This allows for a coherent comparison, highlighting the possible benefits gained by going superscalar.

Regarding the examples featuring conditional or unconditional jump instructions, it must be noted that every branch is considered to have its operands ready immediately, in every case where the values are not produced by any reported instruction, leading to a penalty of only 2 cc. In reality, this is only the base value of the penalty and is extended by the number of clock cycles needed to make the required operands become available. This assumption is made for simplicity of discussion and doesn't change the behavior of the architectures in any way. Also, every register involved in a branch condition is considered to have a value such that every branch condition is true. This simplification, while it doesn't faithfully represent every possible scenario, is useful to show particular behaviors in the following examples.

Finally, arrows between instructions are used to highlight the data dependencies, while stalls are indicated by "(s)" following the name of the stage where the instruction is stalled.

4.2.1. Examples with no data dependencies

1. In this example, no dependencies among instructions are taken into account. As a result, the superscalar architecture gains an advantage from dual-issue and the immediate dispatch of additional instructions to the EXE stage. This translates into a two-clock cycle execution improvement for the superscalar architecture.



SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x3		IF	ID	IS	EXE	ROB	WB									
2: sub x2, x1, x0			IF	ID	IS	EXE	ROB	WB								
3: lw x5, 0(x0)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x3	IF	ID	IS	EXE	ROB	WB										
2: sub x2, x1, x0		IF	ID	IS	EXE	ROB	WB									
3: lw x5, 0(x0)		IF	ID	IS	EXE	EXE	EXE	ROB	WB							

clk cycles

SCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x3		IF	ID	IS	EXE	ROB	WB									
2: sub x2, x1, x0			IF	ID	IS	EXE	ROB	WB								
3: lw x5, 0(x0)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

SUPERSCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x3	IF	ID	IS	EXE	ROB	WB										
2: sub x2, x1, x0		IF	ID	IS	EXE	ROB	WB									
3: lw x5, 0(x0)		IF	ID	IS	EXE	EXE	EXE	ROB	WB							

clk cycles

Figure 4.12: Example 1.1 - Independent single-cycle instructions

2. In this particular case, the code contains an instruction that takes multiple cycles to execute, but there are no data dependencies. In the "SUPERSCALAR" execution, the instruction 1 writes its result to the ROB at cc 4 before instruction 0, which writes to the ROB at cc 6. However, the Reorder Buffer ensures that the correct program order is maintained, resulting in both instructions committing together to the RF at cc 7. Thanks to the dual-issue property and lack of data dependencies in the code, the superscalar execution finishes two clock cycles earlier than the scalar one.

SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x5, x4, x1		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
2: sub x2, x1, x3			IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB						
3: lw x3, 0(x6)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x5, x4, x1	IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB								
2: sub x2, x1, x3		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
3: lw x3, 0(x6)		IF	ID	IS	EXE	EXE	EXE	ROB	WB							

clk cycles

SCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x5, x4, x1		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
2: sub x2, x1, x3			IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB						
3: lw x3, 0(x6)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

SUPERSCALAR REAL

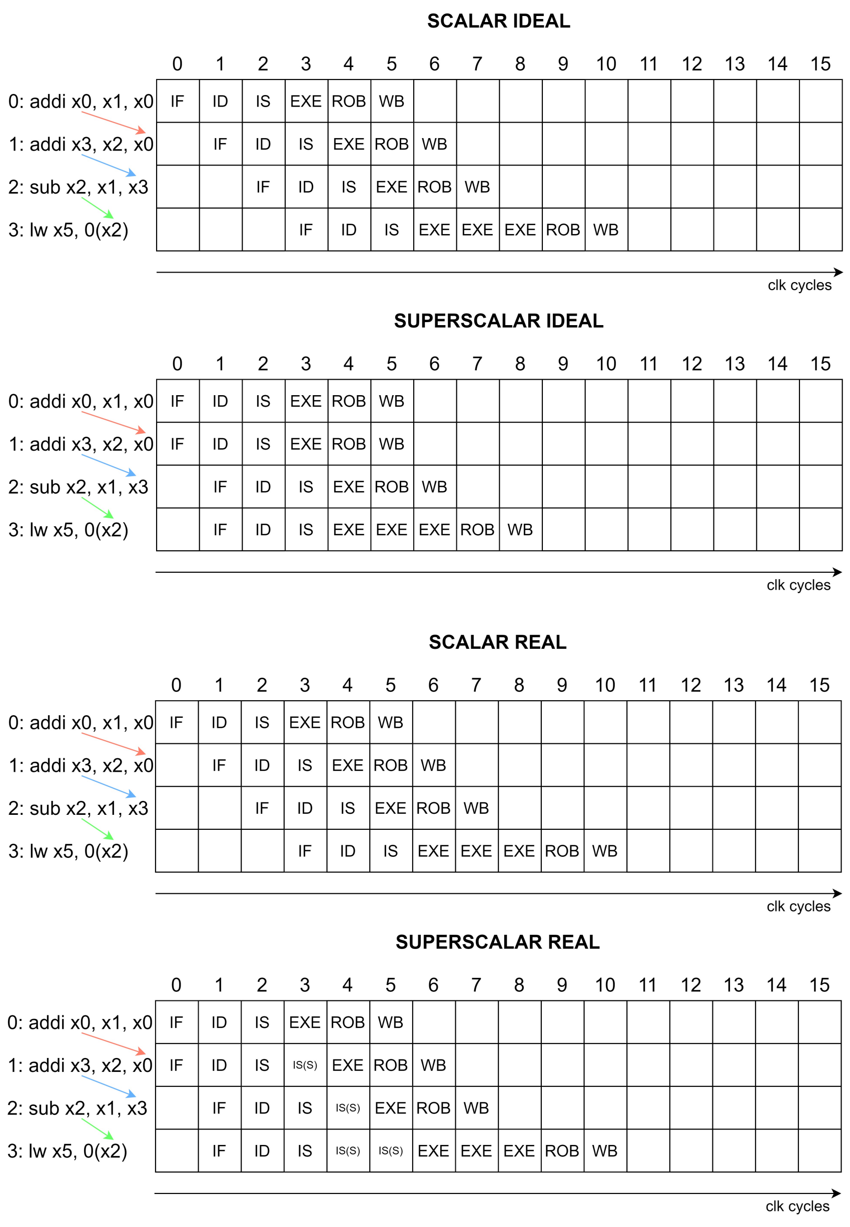
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x5, x4, x1	IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB								
2: sub x2, x1, x3		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
3: lw x3, 0(x6)		IF	ID	IS	EXE	EXE	EXE	ROB	WB							

clk cycles

Figure 4.13: Example 1.2 - Independent single-cycle and multi-cycle instructions

4.2.2. Examples with data dependencies

1. This example is an extension of Example 1.1 (refer to Figure 4.12). In this case, each instruction has a Data Dependence with the previous one. This scenario demonstrates the limitation of the superscalar architecture when dealing with multiple consecutive RAW dependencies. Consequently, the improvement obtained through the dual-issue capability is nullified, as shown in the "SUPERSCALAR REAL" graph.



SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x0		IF	ID	IS	EXE	ROB	WB									
2: sub x2, x1, x3			IF	ID	IS	EXE	ROB	WB								
3: lw x5, 0(x2)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x0	IF	ID	IS	EXE	ROB	WB										
2: sub x2, x1, x3		IF	ID	IS	EXE	ROB	WB									
3: lw x5, 0(x2)		IF	ID	IS	EXE	EXE	EXE	ROB	WB							

SCALAR REAL

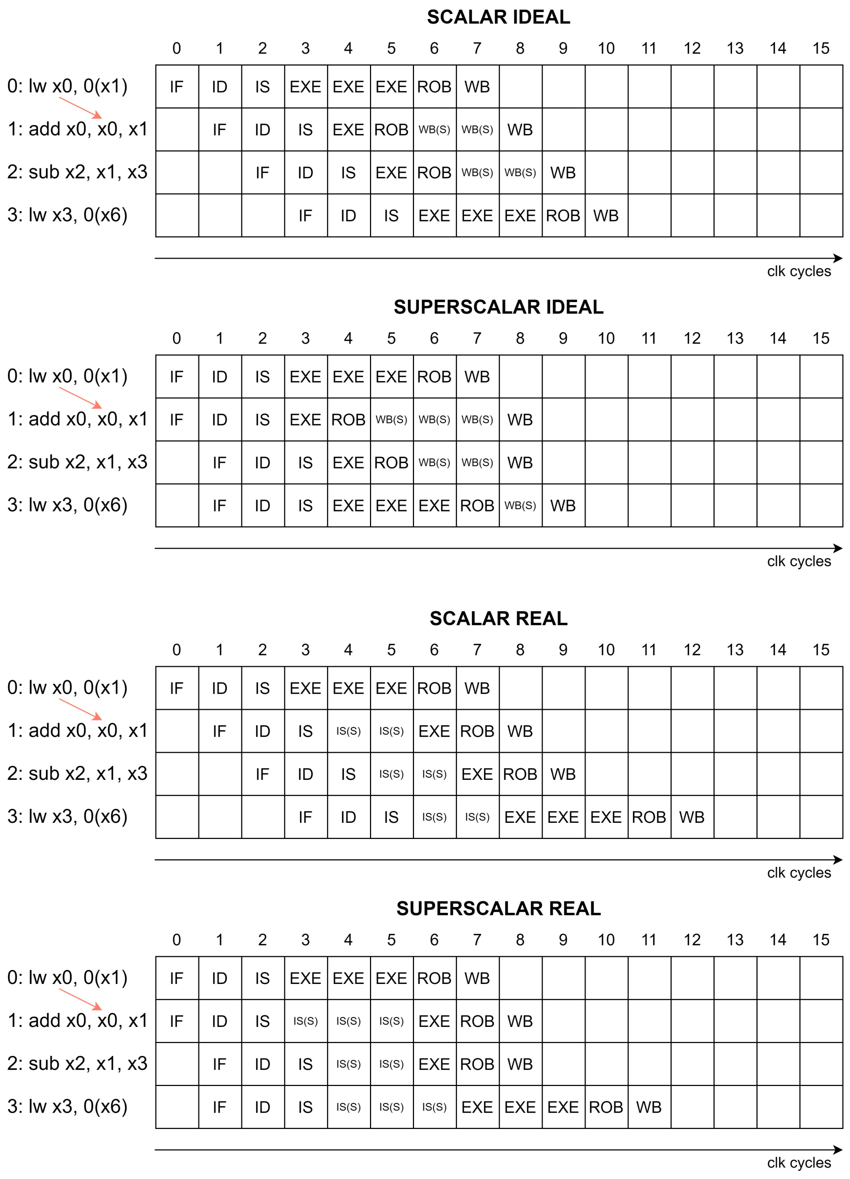
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x0		IF	ID	IS	EXE	ROB	WB									
2: sub x2, x1, x3			IF	ID	IS	EXE	ROB	WB								
3: lw x5, 0(x2)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

SUPERSCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x1, x0	IF	ID	IS	EXE	ROB	WB										
1: addi x3, x2, x0	IF	ID	IS	IS(S)	EXE	ROB	WB									
2: sub x2, x1, x3		IF	ID	IS	IS(S)	EXE	ROB	WB								
3: lw x5, 0(x2)		IF	ID	IS	IS(S)	IS(S)	EXE	EXE	EXE	ROB	WB					

Figure 4.14: Example 2.1 - Single-cycle instructions with data dependencies

2. In this example, there is a data dependence between instruction 0 and instruction 1 in the code. As a result of the dependence, both scalar and superscalar executions experience a decrease in performance. However, the superscalar architecture has a dual dispatch capability that allows it to dispatch both instruction 1 and 2 at cc 6, as shown in the "SUPERSCALAR REAL" graph. This leads to a 1-clock cycle improvement compared to the "SCALAR REAL".



SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x0, x0, x1		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
2: sub x2, x1, x3			IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB						
3: lw x3, 0(x6)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x0, x0, x1	IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB(S)	WB							
2: sub x2, x1, x3		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
3: lw x3, 0(x6)		IF	ID	IS	EXE	EXE	EXE	ROB	WB(S)	WB						

clk cycles

SCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x0, x0, x1		IF	ID	IS	IS(S)	IS(S)	EXE	ROB	WB							
2: sub x2, x1, x3			IF	ID	IS	IS(S)	IS(S)	EXE	ROB	WB						
3: lw x3, 0(x6)				IF	ID	IS	IS(S)	IS(S)	EXE	EXE	EXE	ROB	WB			

clk cycles

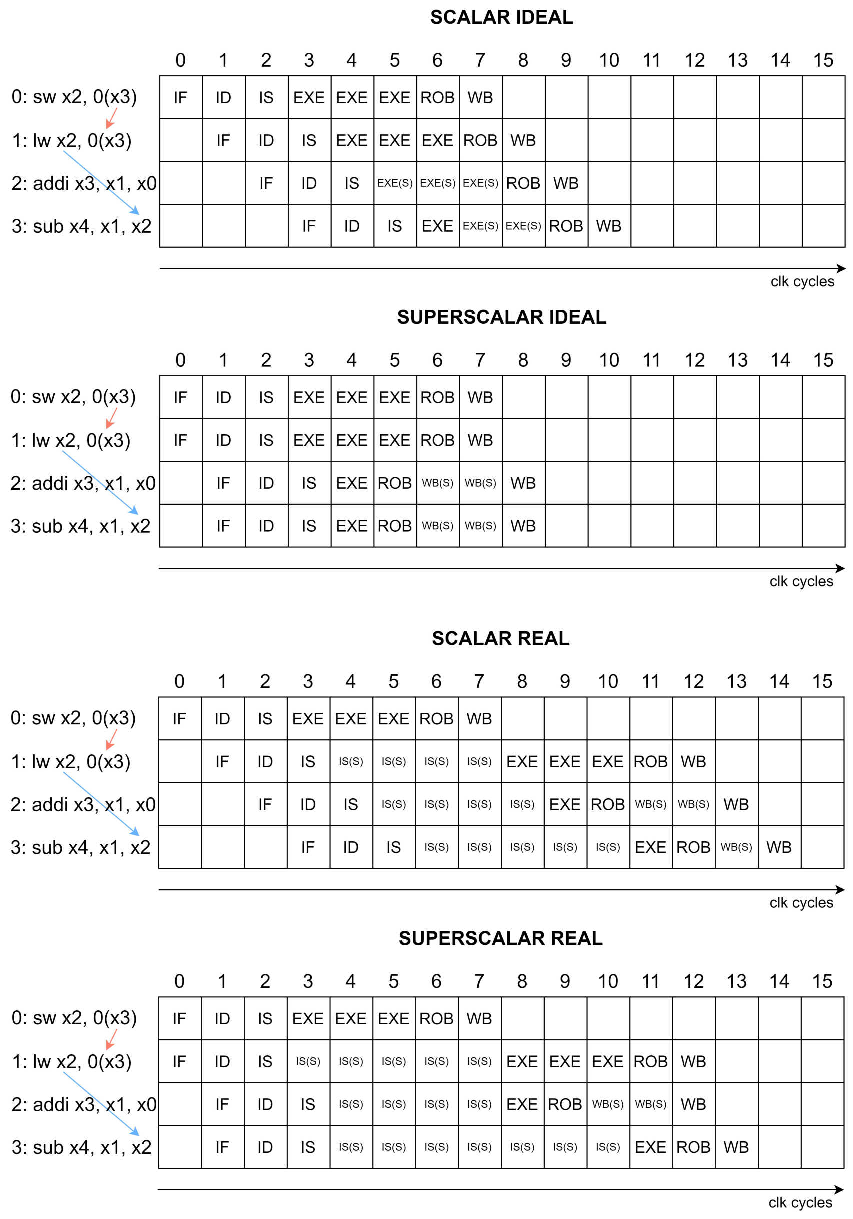
SUPERSCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: lw x0, 0(x1)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: add x0, x0, x1	IF	ID	IS	IS(S)	IS(S)	IS(S)	EXE	ROB	WB							
2: sub x2, x1, x3		IF	ID	IS	IS(S)	IS(S)	EXE	ROB	WB							
3: lw x3, 0(x6)		IF	ID	IS	IS(S)	IS(S)	IS(S)	EXE	EXE	EXE	ROB	WB				

clk cycles

Figure 4.15: Example 2.2 - Single-cycle and multi-cycle instructions with data dependencies

3. This example explains the behavior of the Disambiguation Buffer. In an ideal scenario, instruction 1, which uses the destination address of the previous store as a base for the "lw" instruction, can be issued and dispatched without considering any memory disambiguation problems. However, in reality, it needs to be stalled until the WB (Write Back) of the store is completed. As a result, in the "SUPERSCALAR REAL" graph, instruction 1 is stalled for 5 cc (clock cycles) in the IS (Instruction Scheduler), waiting for the completion of instruction 0. Although this substantial reduction in performance is not ideal, it is necessary to ensure the correct execution of the instructions.



SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: sw x2, 0(x3)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: lw x2, 0(x3)		IF	ID	IS	EXE	EXE	EXE	ROB	WB							
2: addi x3, x1, x0			IF	ID	IS	EXE(S)	EXE(S)	EXE(S)	ROB	WB						
3: sub x4, x1, x2				IF	ID	IS	EXE	EXE(S)	EXE(S)	ROB	WB					

clk cycles

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: sw x2, 0(x3)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: lw x2, 0(x3)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
2: addi x3, x1, x0		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							
3: sub x4, x1, x2		IF	ID	IS	EXE	ROB	WB(S)	WB(S)	WB							

clk cycles

SCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: sw x2, 0(x3)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: lw x2, 0(x3)		IF	ID	IS	IS(S)	IS(S)	IS(S)	IS(S)	EXE	EXE	EXE	ROB	WB			
2: addi x3, x1, x0			IF	ID	IS	IS(S)	IS(S)	IS(S)	IS(S)	EXE	ROB	WB(S)	WB(S)	WB		
3: sub x4, x1, x2				IF	ID	IS	IS(S)	IS(S)	IS(S)	IS(S)	IS(S)	EXE	ROB	WB(S)	WB	

clk cycles

SUPERSCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: sw x2, 0(x3)	IF	ID	IS	EXE	EXE	EXE	ROB	WB								
1: lw x2, 0(x3)	IF	ID	IS	IS(S)	IS(S)	IS(S)	IS(S)	IS(S)	EXE	EXE	EXE	ROB	WB			
2: addi x3, x1, x0		IF	ID	IS	IS(S)	IS(S)	IS(S)	IS(S)	EXE	ROB	WB(S)	WB(S)	WB			
3: sub x4, x1, x2		IF	ID	IS	IS(S)	IS(S)	IS(S)	IS(S)	IS(S)	IS(S)	IS(S)	EXE	ROB	WB		

clk cycles

Figure 4.16: Example 2.3 - Store and load instructions targeting the same memory address

4.2.3. Examples with branches

1. In this example, we observe the behavior of the front end when the first instruction of a fetched bundle is a branch. Figure 4.17 illustrates that when instruction 0 is a branch, instruction 1 is flushed when the jump instruction is decoded at cc 1 of the "SUPERSCALAR" graphs. Once the time needed for the jump to be solved (2 cc) has elapsed, the pipeline restarts with the IF of the instruction determined by the PC calculated by the "NEXT PC LOGIC" present in the ID.

SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: beq x0, x1, 2	IF	ID	ID													
1: addi x3, x2, x3		IF	flush													
2: sub x2, x1, x0				IF	ID	IS	EXE	ROB	WB							
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	WB				

clk cycles

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: beq x0, x1, 2	IF	ID	ID													
1: addi x3, x2, x3	IF	flush														
2: sub x2, x1, x0				IF	ID	IS	EXE	ROB	WB							
3: lw x5, 0(x0)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

SCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: beq x0, x1, 2	IF	ID	ID													
1: addi x3, x2, x3		IF	flush													
2: sub x2, x1, x0				IF	ID	IS	EXE	ROB	WB							
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	WB				

clk cycles

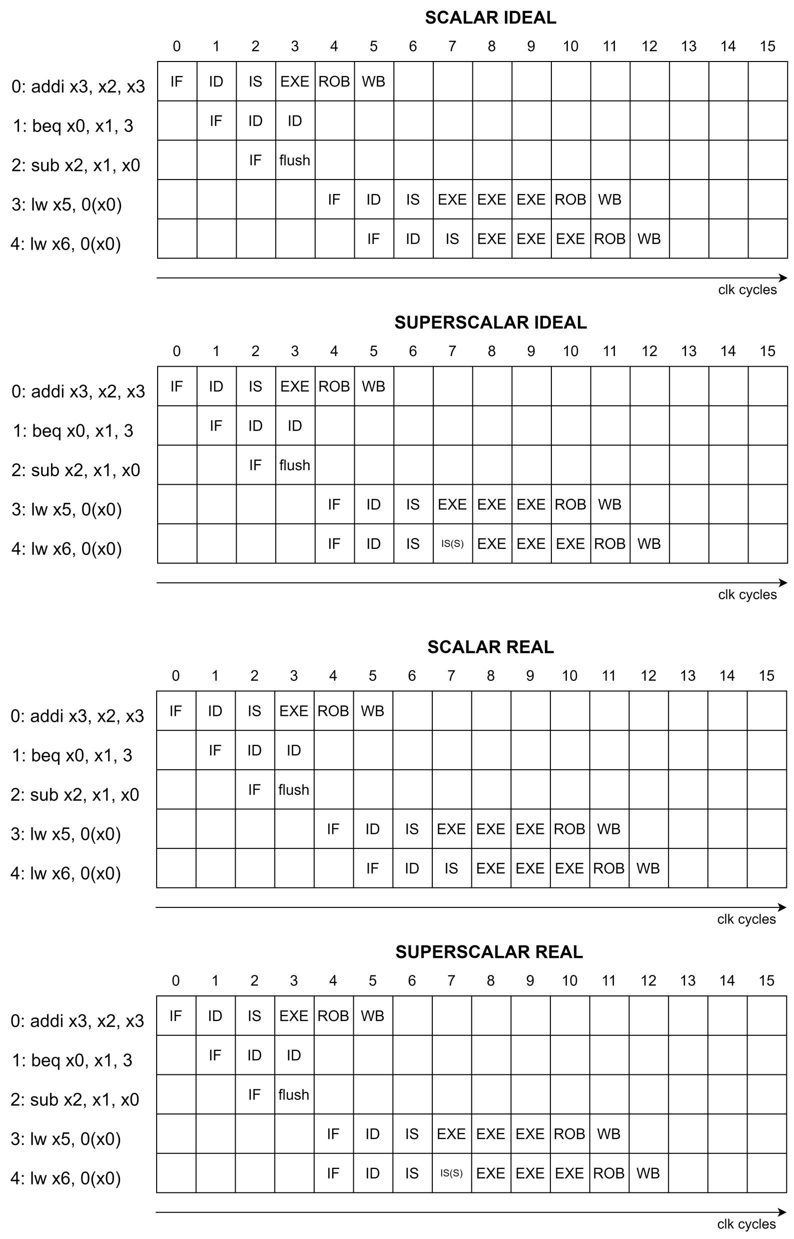
SUPERSCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: beq x0, x1, 2	IF	ID	ID													
1: addi x3, x2, x3	IF	flush														
2: sub x2, x1, x0				IF	ID	IS	EXE	ROB	WB							
3: lw x5, 0(x0)				IF	ID	IS	EXE	EXE	EXE	ROB	WB					

clk cycles

Figure 4.17: Example 3.1 - Branch as first instruction of the bundle

2. In the current scenario, the branch instruction comes second in the fetched bundle, which is made up of instructions 0 and 1. As depicted in Figure 4.18, the first instruction 0 is sent to the ID stage, which stores instruction 1 because it is a branch. After 1 cycle, the stored instruction is processed alone, and the Jump Calculation begins, resulting in instruction 2 being flushed at cycle 3. The pipeline restarts at cycle 4 with the correct PC, but as shown in the "SUPERSCALAR REAL" graph, there is no advantage to using a superscalar architecture over a scalar one since there is only one LSU responsible for handling load and store operations.


clk cycles
clk cycles
clk cycles
clk cycles

SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x3, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3		IF	ID	ID												
2: sub x2, x1, x0			IF	flush												
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	WB				
4: lw x6, 0(x0)						IF	ID	IS	EXE	EXE	EXE	ROB	WB			

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x3, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3		IF	ID	ID												
2: sub x2, x1, x0			IF	flush												
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	WB				
4: lw x6, 0(x0)					IF	ID	IS	IS(S)	EXE	EXE	EXE	ROB	WB			

SCALAR REAL

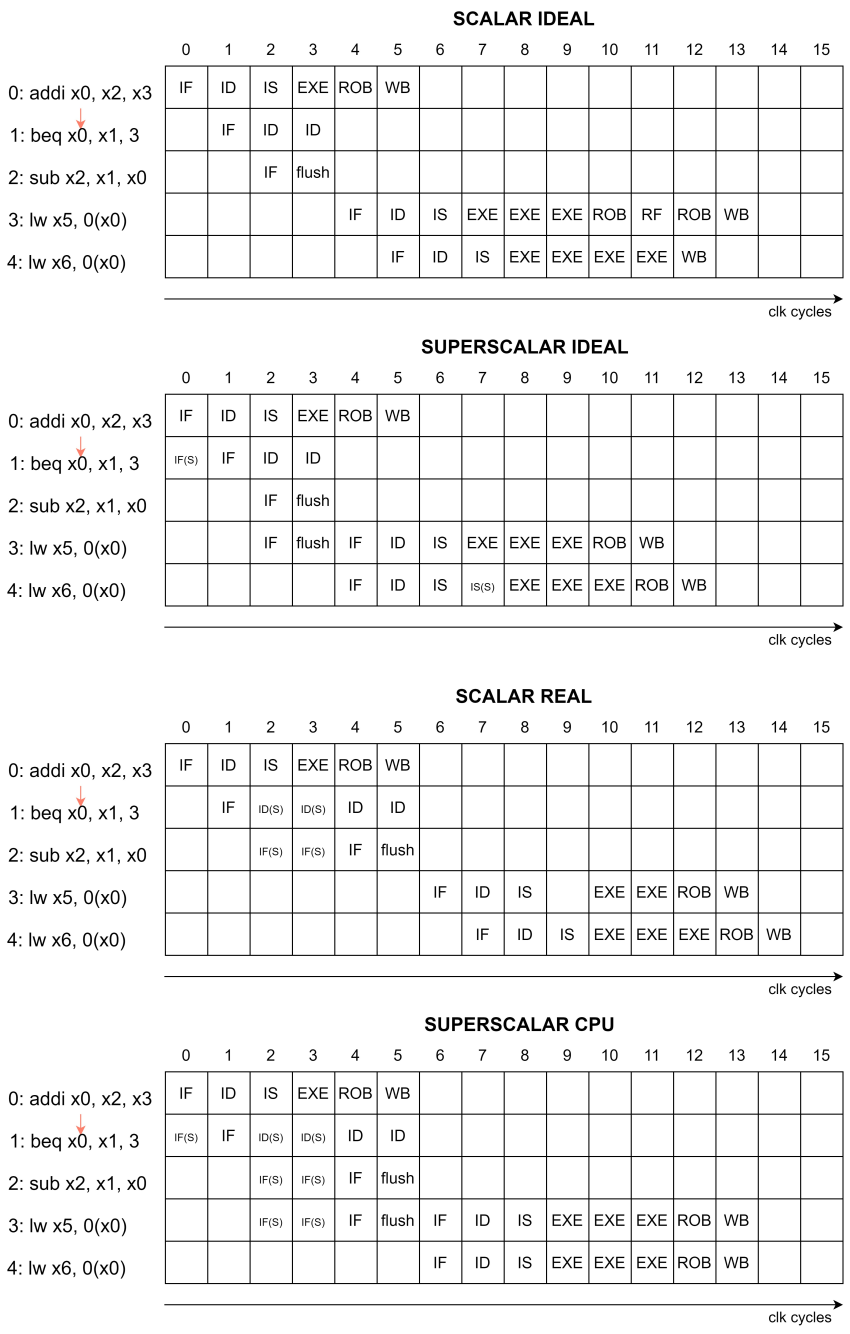
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x3, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3		IF	ID	ID												
2: sub x2, x1, x0			IF	flush												
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	WB				
4: lw x6, 0(x0)						IF	ID	IS	EXE	EXE	EXE	ROB	WB			

SUPERSCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x3, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3		IF	ID	ID												
2: sub x2, x1, x0			IF	flush												
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	WB				
4: lw x6, 0(x0)					IF	ID	IS	IS(S)	EXE	EXE	EXE	ROB	WB			

Figure 4.18: Example 3.2 - Branch as second instruction of the bundle, with no data dependencies

3. In this example, the first set of instructions performs a task that generates a result required by the subsequent branch instruction. As shown in Figure 4.19, the branch instruction has to pause in the ID stage because it can't be executed until the previous instruction produces the necessary result. The penalty for this jump is more than the base 2 clock cycles since it takes an additional 2 cc to produce the result required for the branch instruction.



SCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3		IF	ID	ID												
2: sub x2, x1, x0			IF	flush												
3: lw x5, 0(x0)					IF	ID	IS	EXE	EXE	EXE	ROB	RF	ROB	WB		
4: lw x6, 0(x0)						IF	ID	IS	EXE	EXE	EXE	EXE	WB			

SUPERSCALAR IDEAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3	IF(S)	IF	ID	ID												
2: sub x2, x1, x0			IF	flush												
3: lw x5, 0(x0)			IF	flush	IF	ID	IS	EXE	EXE	EXE	ROB	WB				
4: lw x6, 0(x0)					IF	ID	IS	IS(S)	EXE	EXE	EXE	ROB	WB			

SCALAR REAL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3		IF	ID(S)	ID(S)	ID	ID										
2: sub x2, x1, x0			IF(S)	IF(S)	IF	flush										
3: lw x5, 0(x0)							IF	ID	IS		EXE	EXE	ROB	WB		
4: lw x6, 0(x0)								IF	ID	IS	EXE	EXE	EXE	ROB	WB	

SUPERSCALAR CPU

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0: addi x0, x2, x3	IF	ID	IS	EXE	ROB	WB										
1: beq x0, x1, 3	IF(S)	IF	ID(S)	ID(S)	ID	ID										
2: sub x2, x1, x0			IF(S)	IF(S)	IF	flush										
3: lw x5, 0(x0)			IF(S)	IF(S)	IF	flush	IF	ID	IS	EXE	EXE	EXE	ROB	WB		
4: lw x6, 0(x0)							IF	ID	IS	EXE	EXE	EXE	ROB	WB		

Figure 4.19: Example 3.3 - Branch as second instruction of the bundle, with data dependence

4.3. Spike-based Verification Infrastructure

A functional ISA simulator is a software application capable of executing a compiled program and simulating the corresponding machine state step-by-step. It doesn't simulate the whole pipeline behavior at every clock cycle, but only the commits. Every simulator step exposes a new instruction result, following the program order of the application.
The objective is to verify the correctness of execution of the architecture described in section 4.1. To achieve this objective, three approaches were considered: checking the whole machine state at the end of execution, checking it at periodic checkpoints, or monitoring it at every instruction commit.
To obtain automatic verification during a simulation, we had to establish communication between the ISA simulator and the testbench. This communication was established using a socket between the testbench and the ISA simulator. This made it possible to send commands from the testbench and receive useful data back to perform the verification routine. The details about the verification process will be described in section 4.3.3, while the implemented functionalities will be discussed in subsection 4.3.2.

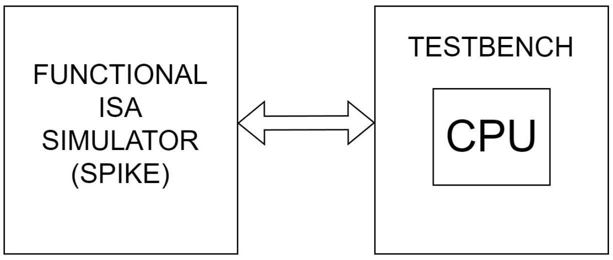


Figure 4.20: High-level view of the communication used for the verification functionalities

4.3.1. Choice of the ISA simulator

We opted to use Spike, a simulator for RISC-V ISA [9], as it is considered to be the golden model for this Instruction Set Architecture. Spike is an open-source functional simulator and is based on the GNU Toolchain for RISC-V [10]. It is written in C++ and can be modified as per our requirements. Spike can run programs via a Proxy Kernel or as bare-metal applications. By default, it executes the input executable from beginning to end. With the *enable_commitlog* directive, it is possible to expose the details about the committed instruction step-by-step. Spike also supports a *debug mode*, in which the simulator accepts commands such as the number of steps it needs to run or which values to expose.

4.3.2. Customization of the ISA simulator

In order to enable interactivity, the Spike debug mode had to undergo some modifications to allow it to receive commands from a client and send back necessary data. The simulation is launched from the terminal and Spike sets up a socket communication on the server side to wait for the client connection. The native *interactive()* function, already present in the base version of the simulator, now waits for commands communicated by the client instead of listening to the terminal. When a *run* command is received, the execution proceeds by the specified number of steps, and after each step, Spike sends back the disassembly of the committed instruction and the commitlog, which includes the PC, instruction encoding, destination register, and result.
Additionally, the simulator has other native utility functionalities such as the *reg* and *mem* commands. These commands respectively provide the content of a targeted register and memory location. These values are sent to the client in a single communication, without the need for disassembly, as they do not imply a committing instruction.
We also added a new functionality called *expose_rf*, which exposes the whole RF by printing it on the terminal and sending the register values to the client one-by-one. The *client.cpp* file contains a set of functions used to send the commands described above and control the simulation.

4.3.3. Verification

After successfully establishing communication on Spike's side, we needed to cross-compile the *client.cpp* file using the cross-compiler in Vivado TCL console, as described in the reference user guide [13]. This allowed us to use the functions of the C++ client in the testbench, enabling control of Spike during the simulation. The functions we used were '*connect_client*' to establish communication with the server, *client_send* to send specific commands, and *client_disassembly* and *client_read* to receive data from the server.
To keep the testbench organized, we created a package that defined some SystemVerilog (SV) functions. These functions were based on the ones in the C++ client, but also took care of manipulating the sent and received data. After receiving the commit information, we converted the '*char**' arrays into strings and then into logic values, making them easily comparable to the architectural signals.

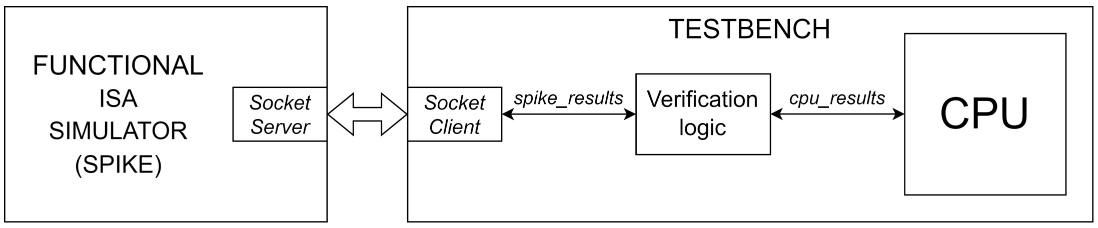


Figure 4.21: Scheme of the Verification process

To perform the verification, first of all, Spike must be started in interactive debug mode and told which application to run. It will then wait for a connection with the client.
Then, in the testbench, the architectural instruction and data memory are initialized with the content of the application objdump, then the client-side connection is established with the DPI function *connect_client*. After the architecture and the functional Execute have simulated one clock cycle, the TB checks if one or two new commits occurred. If they did, it tells Spike to simulate one step through the *spike_step* SV function, and then compares the PC, destination and result coming from the golden model with the ones produced by the architecture, interrupting the simulation in case of a mismatch in any of them. In case of a store instruction, the TB also calls the *spike_mem* SV function, which tells Spike to expose the content of the memory address that was just written. This must be done since the commitlog for stores is not automatically produced.
A final check is for equality between the committed PC and a target final PC, a parameter that must be set according to the dump of the executed application: if they match, the simulation ends and the verification is successful.

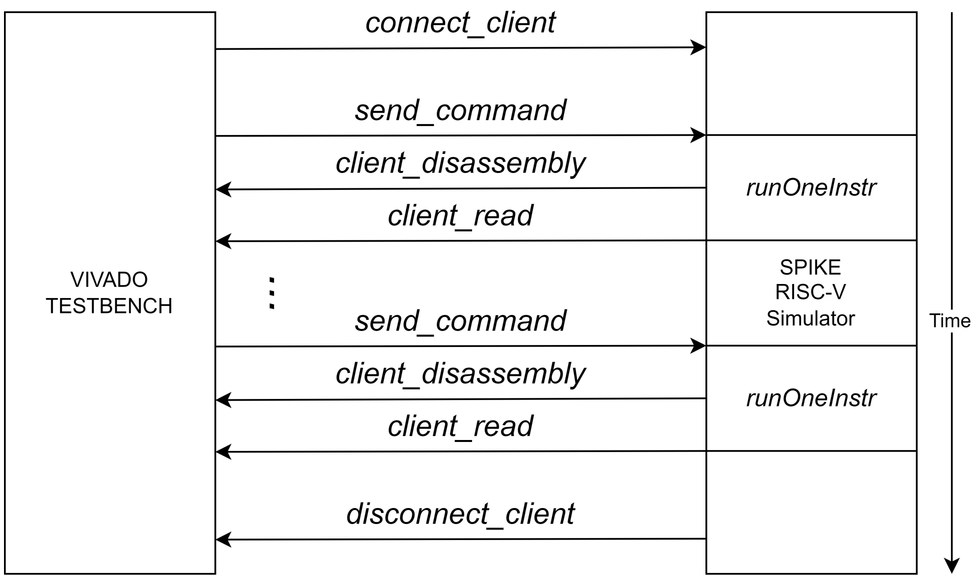


Figure 4.22: Temporal advance of the verification process

5 | Experimental Evaluation

The focus of this chapter is on the experimental results obtained using a specific setup as detailed in Section 5.1. Subsequently, these results are discussed and analyzed to derive a comprehensive view of the quality metrics attributed to the presented CPU. The analysis of these results provides crucial insights that are indispensable for a better understanding of the CPU's capabilities. The following sections provide a detailed account of the analysis, which aids in the examination of the CPU's performance, reliability, and overall quality.

5.1. Experimental setup

The design is written in SystemVerilog. Synthesis and implementation are carried out on Vivado 2022.2, targeting an AMD Xilinx Artix-7 75 FPGA (xc7a75tftg256-1), which provides 47200 Lookup Tables (LUTs), 94400 Flip-Flops (FFs), 105 BlockRAMs (BRAMs) and 180 Digital Signal Processing units (DSPs). The target frequency of the implementation is 77 MHz. For the verification, the used Spike version is 1.0.0. Everything is hosted by a machine running Linux OS distribution Ubuntu 22.04.
All the bare-metal *elf* executables were compiled using the *riscv32-unknown-elf-gcc* compiler based on the GNU RISC-V 32-bit Toolchain [10] and were provided by the official *riscv-test* [12] repository or WCET website [14]. The utilized benchmarks are *multiply*, *towers* and *median* from the *riscv-test* repository for verification, while for the performance evaluation *bsort*, *cnt*, *crc*, *fdct*, *jfdctint*, *prime*, *select*, *fac*, *janne_complex*, *lcdnum* and *matmult* from the WCET suite.
Regarding the interface between Vivado and Spike, DPI(Direct Programming Interface) is used and all the C functions are compiled through the *xsc* compiler. For the evaluation, the chosen configuration of the pipeline parameters is the following: RV32I base ISA with M and F extensions, 2 ALUs with one cycle of latency, 1 MULDIV with 5 cycles of latency, 1 FPU with 5 cycles of latency, 1 LSU with 3 cycles of latency, 16 entries deep Issue Queue, 64 entries deep ROB, 32 bit wide and 2^{10} entries deep Instruction Memory, 32 bit wide and 2^{16} entries deep functional Data Memory. For the rest of this Chapter, the configuration of the architecture described in Section 4.1 is referred to as ViVit.

To initialize the architectural Instruction Memory, the 32-bit encodings and Program Counters present in the *objdump* of the target executable are utilized. For the purpose of verification, the corresponding bare-metal application is executed by Spike in interactive debug mode, as described in Section 4.3.3. Several metrics are taken into consideration to evaluate the overall quality of the architecture.

- **Perfomance**
  The evaluation of performance is determined by the total execution time and the percentage of instructions that utilize the superscalar feature of the architecture. These values are generated within the testbench and are subsequently presented on the TCL console upon completion of the execution process.
- **Area**
  The process of area evaluation is conducted by Vivado during the implementation phase, where the primary focus is on the utilization of resources such as LUTs and FFs. This phase is critical in analyzing the overall resource consumption of a digital circuit and is an essential step in ensuring optimal performance. By providing comprehensive data on the utilization of resources, Vivado enables designers to make informed decisions and to optimize their designs for maximum efficiency.
- **Power**
  Upon completion of the implementation process, a power estimation report is generated, which not only highlights the total value but also provides detailed information on the power consumption of individual blocks. This report offers valuable insights into the performance of the system and helps in identifying areas that require optimization to achieve energy efficiency.

Two different sets of benchmarks are used for different purposes.
The verification of the correctness of the architecture, as described in section 4.3.3, is based on multiple applications part of the official *riscv-tests* [12] suite. On the other hand, the performance evaluation is done executing the WCET [14] benchmarks.

5.2. Experimental results

This section presents an analysis and explanation of the numerical results derived from simulation, synthesis, and implementation.

5.2.1. Performance results

To obtain performance results, Vivado simulation can be used. The simulation results for each of the benchmarks from the WCET suite are presented in the tables below. For Table 5.1, a method similar to the verification process is used, except that Spike is not used to verify every instruction. Instead, a counter on the testbench is incremented at every clock cycle.

Table 5.1: Performance of ViVit (Superscalar) in terms of number of clock cycles

WCET	ViVit
bsort100	5699
cnt	7975
crc	65796
fdct	2438
jfdctint	3761
prime	14303
select	2505
fac	538
janne_complex	487
lcdnum	379
matmult	43330

In order to examine the Worst Case Execution Time (WCET), several benchmarks were utilized as described in the [14]. The benchmarks provided information on the total number of commits, single and double commits, and the percentage of double commits in relation to the total. This information allowed for an assessment of the "superscalarity" of the system. As shown in Table 5.2, the majority of the benchmarks were able to exploit the superscalar features of the architecture, with the exception of the *prime* benchmark which was hindered by its nested data dependencies and jumps, thereby preventing the use of the dual-issue and dual-commit functionality. This emphasizes the importance of a significant amount of instruction level parallelism in the program code for a superscalar architecture to be beneficial. In the case of the *prime* benchmark, only 3x2 instructions were able to exploit the double commit feature.

Conversely, the *matmult* WCET benchmark had the highest percentage of instructions that exploited the double commit feature over the total, with a percentage of 72.7%. This was due to the code's suitability for a dual-issue architecture, allowing for the creation of bundles of independent instructions in the majority of cases. This benchmark created a scenario where a dual-issue CPU could exploit its superscalar feature, obtaining the best percentage between the reported applications.

For the remaining benchmarks, the percentage of double commits oscillated around an average value of 47%. This meant that the architecture was able to fully exploit the dual-issue and dual-commit features almost half of the time.

Table 5.2: Numbers of commits and percentage of exploitation of the superscalar property

WCET	Total	Single	Double	Double/Tot
bsort100	1237	617	310	50,1%
cnt	2006	872	567	56,5%
crc	21186	11986	4600	43,4%
fdct	1363	637	363	52,3%
jfdctint	1748	860	444	50,8%
prime	1754	1748	3	0,3%
select	845	533	156	36,9%
fac	124	76	24	38,7%
janne_complex	77	57	10	25,9%
lcdnum	101	79	11	21,7%
matmult	12156	3312	4422	72,7%

5.2.2. Power consumption

The post-implementation feature "Report Power Utilization" in Vivado offers a useful tool for analyzing power consumption of individual modules. By generating a hierarchical view, this feature provides a detailed breakdown of power consumption relative to the total CPU power budget. With percentages provided for each module, this report can help identify areas of high power consumption and inform power management decisions.

Table 5.3: Power estimation after implementation. The percentages are referred to the total CPU power budget.

DUT	Module	Total [mW]	Module/CPU [%]
CPU		**251**	
	Fetch	1	0,4%
	Decode	4	1,6%
	Issue	46	18,4%
	ROB	160	63,9%
	RF	40	15,7%

In order to identify the module that has the highest power consumption in the processor, each module's power consumption was analyzed. As per the results presented in Table 5.3, it is evident that the Reorder Buffer consumes the highest power between the stages, accounting for 63.9% of the total power budget with a power consumption of 160 mW.
The structure of the Reorder Buffer is the primary reason for the high power consumption of this module. Its entries are composed of various fields, each of which is accessible from different logic networks. For instance, the fill queue of the Issue stage writes the destination, program counter, and store amount, while the Execute stage commits a result. The Issue stage accesses these results for the double read operands logic. The Commit Router, along with the commit on ROB and commit from ROB, does not significantly influence the power metrics. Furthermore, the double read operands combinatorial network contributes significantly to this high power consumption.
With an average power consumption of 40 mW, the Register File is the second module with the highest power consumption. The power consumption of the Issue stage, which is 46 mW, is highly impacted by the double read operands logic and its correlated combinatorial networks. The Issue stage's renaming logic and combinatorial network contribute significantly to the power consumption of the Register File. The renaming logic must per-

form many nested checks to prevent problems in limit cases where an instruction commits to a register that is being renamed.
The Fetch and Decode stages have a combined total power consumption of 5 mW, which is significantly lower than the other modules in the processor. The simplicity of the logic nets implemented in these stages led to this low power consumption, despite their extension.
In summary, the Reorder Buffer and Register File are the modules with the highest power consumption in the processor, with the double read operands logic and combinatorial networks contributing significantly to their power consumption. On the other hand, the Fetch and Decode stages have a significantly lower power consumption due to the simplicity of their implemented logic nets.

5.2.3. Area and timing Results

While simulation has the ability to provide performance results, the quality metrics that are of utmost importance, such as resource utilization and timing, can only be extracted after synthesizing and implementing the design using Vivado. To obtain these metrics, we have created a table that displays all the results, with a particular emphasis on the percentage of target FPGA utilization. We began by extracting the number of overall CPU resources utilized. Subsequently, we conducted an *out_of_context* implementation for each module to collect the same set of data. The Worst Negative Slack (WNS) of each module was obtained using the same CPU clock frequency, which is reported in the *Fmax* column.

Table 5.4: Resource and Timing report after implementation. The percentages are referred to the total number of resources available on the target FPGA.

DUT	Module	LUT	FF	WNS	*Fmax*
CPU		**26298 (55,7%)**	**13593 (14,4%)**	0.594 ns	**77 MHz**
	Fetch	299 (0,6%)	322 (0,3%)	6.196 ns	
	Decode	973 (2,1%)	302 (0,3%)	5.940 ns	
	Issue	2524 (5,3%)	2134 (2.3%)	1.626 ns	
	ROB	16204 (34,3%)	9491 (9.6%)	2.729 ns	
	RF	6233 (13,2%)	1280 (1,4%)	3.962 ns	

The data presented in Table 5.4 indicates a significant utilization of LUTs by the ROB. This can be primarily attributed to the two complex combinatorial networks that facilitate communication with the Issue stage. One of these networks is responsible for retrieving values during the Read Operands, while the other inserts PC, destination, and store amount during the fill queue. It is noteworthy that these networks must access every cell of the ROB in a single clock cycle without a synchronous read port, which significantly increases their complexity.

Regarding the utilization of FFs, it is mainly due to the ROB's structure itself, which consists of different and sometimes wide fields. A portion of the FFs is also used to implement the Issue Queue in the Issue stage. The high number of LUTs is justified by the presence of three heavy combinatorial networks. These include one used to gather operands from the RF and ROB, one to perform and enable renaming in the extension fields of the Register File, and the last one to implement the double read operands logic itself.

In terms of the RF, FFs are mainly utilized to implement the extended Register File. The utilization of LUTs is not due to the logic used for the commit of the instructions but to the net used to perform enabling and disabling of the renaming. It is worth noting that the CPU does not use any DSP and BRAM resources.

The CPU can run up to a maximum frequency of 77 MHz, with a Worst Negative Slack (WNS) of 0.594 ns. The Issue module represents the critical path for the architecture, having a large combinatorial net that limits the timing performance of the stage. The total CPU WNS is much lower than that of the limiting stage since it considers the totality of the interconnections between blocks.

6 | Conclusions

The present thesis delves into the intricate design of a superscalar RISC-V dual-issue CPU and examines its potential for significant advancements in modern processor design. The architecture's ability to execute two instructions per clock cycle provides an unprecedented level of efficiency gains and underscores the significance of adopting the RISC-V instruction set architecture. The research offers insightful observations on optimizing parallelism and resource utilization, paving the way for enhanced performance in future computing systems. The ViVit CPU aims to address a specific market need for a high-performance processor with medium-low area occupation and power consumption, complemented by its custom verification tool and infrastructure. The CPU is the result of a well-crafted combination of hardware and software techniques, optimized for performance, power efficiency, and functionality. As a future improvement, it is suggested that the hardware implementation of the Execute stage and compiler technologies be refined for better exploitation of the dual-issue capabilities. Incorporating advanced branch prediction techniques and an enhanced instruction fetch mechanism can reduce pipeline stalls and improve overall execution efficiency. Further, employing more sophisticated Out-of-Order execution mechanisms and advanced scheduling algorithms can unlock greater parallelism within the processor, leading to improved performance in diverse computational workloads. In addition, exploring the possibility of specialized accelerators or co-processors for specific tasks can augment the overall capabilities of the superscalar RISC-V dual-issue CPU. This can enhance the CPU's performance in specific applications while minimizing power consumption. In future developments, power efficiency and scalability are crucial aspects that need to be taken into account. Techniques for dynamic voltage and frequency scaling, as well as advanced power gating strategies, can contribute to creating more energy-efficient processors without compromising performance. The continuous evolution of the superscalar CPUs necessitates ongoing research and development efforts. Future improvements should address both hardware and software aspects to unlock the full potential of this architecture and meet the ever-growing demands of modern computing. By considering all aspects of the CPU's design and development, it can be ensured that it remains a relevant and highly sought-after technology for years to come.

Bibliography

[1] Homepage of ARM Reference Site. URL https://www.arm.com/.

[2] Homepage of BOOM Reference Site. URL https://docs.boom-core.org/en/latest/sections/intro-overview/boom-pipeline.html.

[3] Homepage of Chipyard Reference Site. URL https://chipyard.readthedocs.io/en/stable/index.html.

[4] Homepage of CVA6 OpenHW Reference Site. URL https://docs.openhwgroup.org/projects/cva6-user-manual/03_cva6_design/intro.html.

[5] Homepage of NOEL-V Reference Site. URL https://www.gaisler.com/index.php/products/processors/noel-v.

[6] Homepage of OpenPiton Reference Site. URL http://parallel.princeton.edu/openpiton/.

[7] Homepage of RavenSoC GitHub repository. URL https://github.com/efabless/raven-picorv32.

[8] Homepage of RISC-V Ecosystem Reference Site, . URL https://wiki.riscv.org/display/HOME/RISC-V+Software+Ecosystem.

[9] Homepage of Spike RISC-V ISA Simulator Repository, . URL https://github.com/riscv-software-src/riscv-isa-sim.

[10] Homepage of GNU RISC-V Toolchain Repository, . URL https://github.com/riscv-collab/riscv-gnu-toolchain.

[11] Homepage of SHAKTI Processor Reference Site. URL https://shakti.org.in/index.html#intro.

[12] Homepage of riscv-tests Repository. URL https://github.com/riscv-software-src/riscv-tests.

[13] xsc Compiler - Vivado Design Suite User Guide: Logic Simulation (UG900). URL `https://docs.xilinx.com/r/en-US/ug900-vivado-logic-simulation/xsc-Compiler`.

[14] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015. doi: 10.1109/SIES.2015.7185039.

[15] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[16] M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007. doi: 10.1109/N-SSC.2007.4785534.

[17] C. Celio, D. A. Patterson, and K. Asanović. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html`.

[18] A. Dörflinger, M. Albers, B. Kleinbeck, Y. Guan, H. Michalik, R. Klink, C. Blochwitz, A. Nechi, and M. Berekovic. A comparative survey of open-source application-class risc-v processor implementations. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 12–20, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384049. doi: 10.1145/3457388.3458657. URL `https://doi.org/10.1145/3457388.3458657`.

[19] T. Gokulan, A. Muraleedharan, and K. Varghese. Design of a 32-bit, dual pipeline superscalar risc-v processor on fpga. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 340–343, 2020. doi: 10.1109/DSD51259.2020.00062.

[20] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer. Open-source risc-v processor ip cores for fpgas — overview and evaluation. In *2019*

8th Mediterranean Conference on Embedded Computing (MECO), pages 1–6, 2019. doi: 10.1109/MECO.2019.8760205.

[21] R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997. doi: 10.1109/6.591665.

[22] G. Scotti and D. Zoni. A fresh view on the microarchitectural design of fpga-based risc cpus in the iot era. *Journal of Low Power Electronics and Applications*, 9(1), 2019. ISSN 2079-9268. doi: 10.3390/jlpea9010009. URL `https://www.mdpi.com/2079-9268/9/1/9`.

[23] M. Sharma, E. Bhatnagar, K. Puri, A. Mitra, and J. Agarwal. A survey of risc-v cpu for iot applications. In *Proceedings of the International Conference on Innovative Computing & Communication (ICICC) 2022*, February 2022. Available at SSRN: `https://ssrn.com/abstract=4033491` or `http://dx.doi.org/10.2139/ssrn.4033491`.

[24] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html`.

List of Figures

List of Tables

Acknowledgements

We would like to thank our advisor, prof. Davide Zoni, and co-advisor, Andrea Galimberti, for the opportunity to work on this project and the support they provided.
A special acknowledgement goes to our parents and relatives, who supported us during these academic years, and all the friends, who we shared this journey with.