RISC-V core optimization in 22nm FD-SOI technology

Max Doblas Font

Supervised by:

Francesc Moll Echeto

Miquel Moretó Planas

In partial fulfillment of the requirements for the master degree in:

Master's degree in Advanced Telecommunication Technologies (MATT)

October 2021

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

telecos
BCN

Abstract

The thesis aims to design and implement an in-order core named Sargantana using the open RISC-V Instruction Set Architecture (ISA). Sargantana targets to improve the in-order 5-stage Lagaro-Hun core used in the early iterations of the DRAC project. Sargantana has a more mature 7-stage pipeline with out-of-order write back, register renaming, and a non-blocking memory pipeline.

Also, this thesis applies microarchitectural design space exploration into Sargantana to balance the cycle performance and the clock frequency. 22nm FD-SOI commercial technology libraries are used to take physical effects into account in the register-transfer level (RTL) design. In this way, the processor's bottlenecks are analyzed to reach the maximum clock frequency using realistic technology. With this exploration, Sargantana achieves 2 CoreMark/MHz and 1 GHz in the worst corner (and up to 1,88 GHz in the faster) using 22nm FD-SOI commercial technology libraries. Sargantana obtains over its predecessor, the 5-stage Lagarto-Hun, an IPC speed-up of 1,37X.

Keywords: Microarchitecture, ASIC synthesis, In-order core, SRAM memories, Critical path, SystemVerilog, RISC-V.

Resum

La tesi té com a objectiu dissenyar i implementar un processador en ordre anomenat Sargantanal, el qual implementa el set d'instruccions (ISA) lliure RISC-V. Sargantana té com a objectiu millorar el processador Lagarto-Hun de cinc etapes, utilitzat en les primeres iteracions del projecte DRAC. Sargantana té una pipeline de set etapes més madura amb escriptura fora d'odre i una pipeline de memòria que no bloquejant.

A més, aquesta tesi aplica una exploració espacial de disseny microarquitectònic a Sargantana per equilibrar el IPC i la freqüència del rellotge. S'han utilitzat les llibreries tecnològiques comercials FD-SOI de 22 nm per tenir en compte els efectes físics en el disseny digital. D'aquesta manera, s'analitzen els punts limitants del processador per tal d'assolir una freqüència de rellotge màxima mitjançant una tecnologia realista. Amb aquesta exploració, Sargantana aconsegueix 2 CoreMark/MHz i 1 GHz en les pitjors condicions (i fins a 1,88 GHz en les millors) mitjançant les llibreries tecnològiques comercials FD-SOI de 22 nm. Sargantana obté sobre el seu predecessor, el Lagarto-Hun de cinc etapes, una millora en IPC de 1,37X.

Paraules clau: Microarquitectura, síntesi ASIC, processador en ordre, Memòries SRAM, Camí crític, SystemVerilog, RISC-V.

Acknowledgements

I would like to express my gratitude to Miquel Moretó for allowing me to join his research at BSC for nearly three years now. He has taken an active interest in my growth both on academic and personal levels.

Moreover, I would like to thank Francesc Moll. He has supervised my work directly from UPC. I have learned a lot of physical design concepts from him. Moreover, he is a lovely person to work with that also makes you always feel welcome.

I also want to thank the help of Víctor Soria, Guillem López, Narcís Roda, Xavier Carril, Neiel Leyva and Gerard Candón. They always have been there to work with me, facing the difficulties of this project. They also have done a spectacular job in the Sargantana core. Thanks to Victor for all the architectural ideas that he has brought to this stunning Sargantana design. I want to thank Gerard for sharing his excellent knowledge of vector processing on RISC-V, and his excellent debug skills. I want to thank Guillem for his expertise on the RTL environment tools and his always appreciating SystemVerilog structs. I want to thank Narcís for his versatility on the project. He was able to manage smoothly all the different things that he had faced. Thanks, Neiel, for its instruction cache design. Also, I would like to thank all the people involved in the Lagarto designs in Barcelona and Mexico. I want to thank Xavier for reviewing the documentation in order to improve the quality of the document. Finally, I want to thank all the other members from the synthesis and physical design group and the verification group that worked on the project for helping with the physical design and verification environments.

Moreover, I want to thank the help of Nehir Sonmez and Roger Figueres. They always have been there to help me to face the difficulties of this project. Nehir has followed all my progression, and he has given me advice and confidence to keep going. Roger has helped me with the technical part, which I have learnt a lot.

Finally, I would like to thank Professor Arvind and his research group for the knowledge obtained during the MIT fellowship. This knowledge has been advantageous during the Sargantana implementation in order to achieve higher performance in the end.

It has been a pleasure to work with all of them. With this group of people, everything was easier. I hope to work with them in the future.

Contents

List of Figures

List of Tables

Acronyms

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

BHT Branch History Table

BSV Bluespec SystemVerilog

BTB Branch Target Buffer

CMD Composable Modular Design

CPU Central Processing Units

CPP Contacted Poly Pitch

CRC Cyclic Redundancy Check

CSR Control and Status Register

DRAM Dynamic Random Access Memory

EDL Event Driven Language

EHR Ephemeral History Register

FD-SOI Fully Depleted Silicon On Insulator

FPGA Field-Programmable Gate Array

FPU Floating Point Unit

IP Intellectual Property core

IPC Instructions Per Cycle

ISA Instruction Set Architecture

I/O Input/Output

LUT LookUp Table

MCU Mcicrocontrollers

MSHR Miss Status Holding Registers

MIT Massachusetts Institute of Technology

MMU Memory Management unit

MOSFET Metal Oxide Semiconductor Field Effect Transistor

NoC Network on Chip

PC Program Counter

PnR Place And Route

PIPT Physically Indexed, Physically Tagged

PIVT Physically Indexed, Virtually Tagged

PVT Process, Voltage, Temperature

RAS Return Address Stack

RAW Read After Write

ROB Re-Order Buffer

ROCC Rocket Custom Co-processor Interface

RTL Register-Transfer Level

SIMD Single Instruction, Multiple Data

SoC System on Chip

SRAM Static Random Access Memory

TLB Translation Lookaside Buffer

VIPT Virtually Indexed, Physically Tagged

VIVT Virtually Indexed, Virtually Tagged

WAR Write After Read

WAW Write After Write

1 Introduction and Motivation

Every day, nanotechnology is advancing in a faster way. These technology advances allow the manufacturers to design circuits using smaller, faster, and cheaper transistors [5]. The field of computer architecture has taken advantage of the transistors improvements to evolve the processor designs continuously. Every day, computer architects can design more complex circuits making processors faster, cheaper, and more efficient for targeted applications. This, in turn, opens the door to other fields of study outside the computer architectural world, like autonomous driving, machine learning, or Internet-of-Things (IoT), to exploit this computational power to real-world use.

Apart from the technological improvements, open Instruction Set Architectures (ISA) have been proposed very recently. They bring the promise of a similar revolution in the hardware design that open-source software has represented in computing. Even achieving the spectacular revolution that Linux had represented on the operating system side. Since the release of the open RISC-V ISA, many implementations of processors have risen both from industry and academia. Many projects have been created to push the RISC-V ISA to the industrial level, designing and implementing capable cores and a reliable software stack to show its potential. One of this projects is DRAC (**D**esigning **R**ISC-V-based **A**ccelerators for next-generation **C**omputers). The project's objective is to design and successfully tape-out in 22nm FD-SOI technology an out-of-order processor along with accelerators for applications as genomics or computer vision.

This thesis aims to design and implement an in-order core named Sargantana using the open RISC-V ISA. Sargantana aims to improve the in-order processor 5-stage Lagarto-Hun used in the early stages of the DRAC project. Lagarto-Hun is a 5-stage core developed in a collaboration between the Instituto Politécnico Nacional de Mexico and the Barcelona Supercomputing Center (BSC). Two versions of the 5-stage Lagarto-Hun have been taped-out using a 65 nm process node from TSMC.

Although the Lagarto-Hun core was successfully taped-out with a frequency of 600 MHz, the Instructions Per Cycle (IPC) performance of the design could be improved. The first objective of Sargantana is to improve the IPC performance to be competitive with the other state-of-the-art academic cores. Some complex optimizations like out-of-order write-back are implemented in Sargantana to achieve that needed performance.

In this project, a second objective is set. In order to achieve good performance with an Application-Specific Integrated Circuit (ASIC) target, not only the IPC is important. The clock frequency plays a significant role in the performance of a processor. Sargantana has as an objective to achieve a high clock frequency of 1 GHz with a modern 22nm technology. This

frequency of 1 GHz is high enough to force us to make some architectural changes, but it is not as demanding as requiring some fine-tune modifications.

The entire design and verification process is described in the thesis. It describes the new Sargantana pipeline with all the IPC optimizations implemented and its impact on the maximum frequency. The thesis also contains all the environments implemented to validate the performance of the design. It describes the Register-Transfer Level (RTL) simulation/emulation environment for validating the IPC and the physical design flow to validate the maximum frequency that the design can achieve. Also, it includes all the verification followed to prove the correct behavior of the design.

A team of about 10 people has been involved in the implementation of the Sargantana core design, the SoC, and all the verification and validation environment. In particular, I was involved in nearly all the Sargantana core development parts, and I took part in the Sargantana design decisions. I have designed and implemented the new front-end of the core and the modification in the data cache to execute memory operations of 128-bit. I have implemented all the modules and modifications related to the privilege ISA, such as the Control and Status Register (CSR) that enable the boot of Linux distribution, which I have modified for the project. I have developed part of the validation environment, such as the Konata trace visualizer. I worked with other people on several modules of the core back-end as the memory pipeline, the integration of the Floating Point Unit (FPU) and Single Instruction, Multiple Data (SIMD) unit. Also, I have made all the evaluations and performance discussions in this thesis, having to adapt all the benchmarks used for all the cores evaluated.

Finally, and no less important, I am responsible for achieving the clock frequency and area targets of the design. I have done the initial synthesis flow to get feedback on the ASIC synthesis to continuously improve the design, achieving the right balance of frequency and IPC to get the maximum performance. All this work is described in this thesis.

1.1 Document Structure

The rest of this thesis is structured as follows:

- Chapter 2 provides a basic background on computer architecture and physical design implementation concepts to make the document self-contained.
- Chapter 3 introduces the RISC-V open-source ISA and presents the most relevant academic processor designs. It explains the specific design and shows IPC and clock frequency achieved by each core.
- Chapter 4 introduces all the tools and benchmarks used in RTL design and also the technology libraries used for ASIC synthesis.
- Chapter 5 does an in-depth analysis of the 5-stage Lagarto-Hun and introduces all the improvements that have been carried out on the Sargantana processor to achieve the expected frequency of 1 GHz with a good IPC.
- Chapter 6 evaluates the final design and compares it with the other academic designs.

- Finally, Chapter 7 discusses the results and concludes this work.

2 Background

First of all, it is important to introduce some concepts on computer architecture that will be used during this project. Not only purely architectural design concepts need to be presented, but the introduction of the physical design process and technology used for digital designs are necessary.

First, the basics about computer architecture design like pipeline hazards and how to mitigate them with basic optimizations are introduced. Some basic optimizations as forwarding or branch prediction are described. After that, there is the introduction of some advanced IPC optimization, some of which will be used in this project.

Finally, it is crucial to explain how a processor design is physically implemented, introducing the concept of standard cell libraries and hard Intellectual Property core (IP). At this point, the concept of the critical path is also needed. It is important to emphasize why it is so essential in the computer architecture field.

2.1 Microarchitecture Background

As it is known, a processor performs basic arithmetic operations, logic operations, memory operations or Input/Output (I/O) operations described by the instructions in a program. These instructions must modify the state of the processor as the register file, memory system, or special registers in the order specified by the program. The Instruction Set Architecture (ISA) specifies the processor behavior for each particular instruction.

The processor used to be divided into three main parts: the core (or cores in a multiprocessor), the memory hierarchy, and the I/O. In particular, this thesis focus only on the core part and some details of the memory hierarchy.

As it is explained in the physical design section, a processor design will be physically implemented using standard cells, which are implemented by transistors. Those transistors are not ideal and have some associated propagation delay of the signal. Then, a critical path can be defined as the logical path (all the gates between two registers) with more propagation delay on the design. Let us assume a processor that does all the defined steps (fetch, decode, and execution) in one clock cycle. The logic path in a processor like this will be enormous, limiting the maximum frequency of the design. To solve that situation and to be able to achieve higher clock frequencies, the concept of *pipeling* is used.

The instruction's execution can be divided into different parts, also known as stages, in which

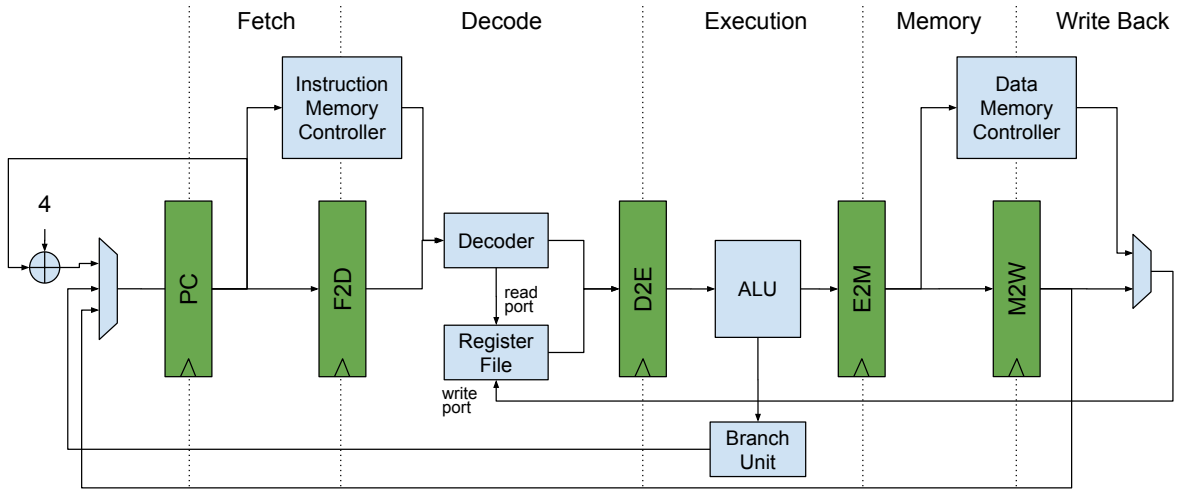


Figure 2.1: Pipelining a simple in-order core design

each stage is executed in a clock cycle and executed concurrently. Figure 2.1 shows an example of a processor with five distinctive stages. The processor is composed of these stages:

- Fetch (F): This module asks for the next instruction to memory and updates the Program Counter (PC).
- Decode and Read Register (D): This module receives the instruction from memory, decodes it, and reads the register needed for each instruction.
- Execution (E): It executes the instruction decoded using the values read from the register file.
- Memory (M): A memory request is performed if the instruction is a load or a store.
- Write Back (W): If the instruction is a load, it receives the data from memory. It writes the result of the instruction to the register file if it is needed.

Thus, the clock frequency of this particular design can be up to five times faster than a one-stage core. However, applying the pipeline concept also has some drawbacks that are important to take into account. These drawbacks are known as pipeline hazards that are explained in depth in Subsection 2.1.1. Basically, the pipeline hazards make the core unable to finish or commit one instruction per cycle (assuming that the one cycle version of the core commits at most one instruction per cycle), reducing the final throughput of the core.

2.1.1 Pipeline Hazards

An ideal processor that executes a complete instruction in one cycle has an IPC of one. However, this ideal IPC is impossible to achieve in a pipelined processor. Different factors introduce hazards that can force the next instruction to not execute in the following clock cycle. These hazards will result in not finishing an instruction every cycle. There are three primary hazards in a pipeline: structural hazard, data hazard, and control hazards.

Structural hazard The structural hazard is produced when two (or more) operations processed in the pipeline need the same resource. A simple example of a structural hazard can be found in a small core with a unified memory used to store instructions and data. The processors with a unified memory (the instructions and the data are located on the same physical memory) can experiment structural hazards when a memory instruction is on the execution stage (in the memory stage or pipeline for those processors that implement it). When a memory instruction is on the execution stage, the core makes a petition to the unified memory blocking it during that cycle. Since the core is pipelined, the fetch stage will also try to access the unified memory. However, the unified memory is locked by the older memory instruction. At this point, the fetch stage will not be able to fetch a new instruction. Thus, the core will have one empty cycle without committing an instruction. An easy solution to this hazard is duplicating the resources to allow both instructions to access the memory hierarchy. Two different memories, one for instructions and another for data, can be used in this case.

To sum up, the structural hazards are produced by a lack of physical hardware and can be solved by adding more hardware. However, adding more hardware increments the area and power consumption. Thus, it is a trade-off between area, power cost, and performance.

Data hazard Data hazards can be encountered when two or more instructions exhibit dependencies on a particular data during the execution of different stages of the pipeline. The potential data hazards have to be solved to avoid possible race conditions that can modify the expected execution result. Thus, if there is a data race condition, the result of the program will not follow the definition that the programmer expects. Three situations can produce a data hazard depending on each instruction's action (reading or writing) implicated in the data hazard. However, not all situations can be found on any processor because some combinations of instructions will not happen to a particular architecture. The three different data hazards are:

- **Read After Write (RAW), a true dependency**: It happens when one instruction needs a result that will be written by an older instruction. In the example in Figure 2.2a, instruction i_2 needs to read from register $r2$ that is written by instruction i_1. Maybe i_1 is not completed when i_2 reads the register. Thus, $r2$ has not computed the result yet and it still has the old value. The solution is to stall the pipeline and wait for i_1 to finish before i_2 reads register $r2$.

- **Write After Read (WAR), an anti-dependency**: refers to a situation where a younger instruction writes to a register that has to be read by an older instruction. It can only happen when the instructions are executed concurrently or out-of-order, and the younger instruction has been reordered before an older instruction for any reason. An example is shown in Figure 2.2b. The instruction i_3 needs to be read from the register $r1$ before the instruction i_4 overwrites the data in that register.

- **Write After Write (WAW), an output dependency**: It happens when two instructions have to write in the same register. The younger instruction's write-back needs to wait until the older one does it. As well as the WAR hazard, the WAW hazard can only

$i_1: \quad r2 \leftarrow r1 + r3$

$i_2: \quad r4 \leftarrow r5 + r2$

(a) RAW

$i_3: \quad r2 \leftarrow r1 + r3$

$i_4: \quad r1 \leftarrow r5 + r4$

(b) WAR

$i_5: \quad r2 \leftarrow r4 + r1$

$i_6: \quad r2 \leftarrow r3 + r5$

(c) WAW

Figure 2.2: Different types of data hazards

	Execution Cycles											
Program Instructions	1	2	3	4	5	6	7	8	9	10	11	12
I1: bne r1, r2, I8 (it jumps)	F	D	E	M	W							
I2: li r3, 7		F	x									
I3: addi r4, r4, 4			x									
I8: li r1, 0			F	D	E	M	W					
...												

Figure 2.3: Pipeline execution with a control hazard. A branch instruction redirects the PC, and the pipeline has to be flushed

be found in concurrent environments. However, in this case, not only an out-of-order issue architecture has this problem. With an architecture able to write back out-of-order is enough to encounter WAW dependencies. An example is shown in Figure 2.2c. The instructions i_5 and i_6 need to write to the same register.

Control Hazard A control hazard occurs when there is a control instruction or an exception in the pipeline that modifies the sequential order of the program. In a pipeline, several instructions are being executed in different stages. Thus, when a control instruction or an exception is detected and computed, the corresponding stage knows the new order of the program. Then, the younger instructions inside the pipeline that do not follow the new order on the program have to be discarded. In Figure 2.3, we show an example of a control hazard. In the example, a 5-stage pipeline design knows the result of a branch instruction in the execution stage. There is a branch instruction inside the execution stage. This branch instruction is executed as taken; thus, the next instruction will not be sequential. The instructions inside the fetch and decode stages have to be flushed because they are not in the program path anymore. As shown, the control hazards have a big impact on IPC since every time the program's sequential order is broken, at least two instructions will be flushed. In a core with more stages between fetch and the stage in which the control instruction is detected, the penalty will even be bigger.

2.1.2 IPC Optimizations

As we can see in Section 2.1.1, several factors introduce hazards. The only solution for these hazards is stalling the pipeline. A lot of lost cycles can be introduced in the pipeline, which can

degenerate the IPC and the performance of a core.

It is necessary to apply some optimizations to the pipeline in order to reduce or eliminate the hazard effects and achieve a proper IPC. In this section, several optimizations, some more simple and some more advanced, are exposed that can be implemented in a core pipeline to boost the IPC of a core. Apart from analyzing the impact on the IPC for each optimization, it is important to look at the impact on the maximum frequency that each optimization introduces.

2.1.2.1 Basic Optimizations: Bypassing, Branch Prediction, and Caches

The basic optimizations are introduced before jumping to more advanced ones. The basic optimizations are:

Bypassing (also known as forwarding). It is the action of routing a value from a source (the stage that generates the data) to a user (a stage that uses the data), bypassing a designated destination register.

Primarily for static pipelines, this allows the value produced to be used earlier in the pipeline without waiting for the write-back to the register file. Bypasses can nearly eliminate all the data hazards inside a pipeline. Each pipeline will implement the bypasses differently depending on the architectural design.

However, not all the effects of the bypasses are beneficial for the design. Bypasses introduce mainly two types of penalties. Firstly, bypasses augment the resources consumed (i.e., area in an ASIC target or number of LookUp Table (LUT) in an Field-Programmable Gate Array (FPGA) implementation). Forwarding needs extra wires to propagate the values and multiplexers to select from where the operands are coming from. Take into account that on a 64bit pipeline, the buses and multiplexers need to be of 64bit width. Also, it adds extra logic on the control side. It is worth mentioning that the increase in resources is not high on a simple single-issue processor, and it is not problematic. However, in a superscalar processor, the number of possible bypasses tends to grow in a quadratic way since there will be one bypass for each pair of issue ports and functional unit.

Secondly, and usually more critical, bypassing can add new, very long paths to the design. These paths can, in the worst-case, notably increase the critical path. To avoid that effect, the bypasses need to be appropriately selected. It is recommended to place all bypasses always at the start of stages or all at the end. Doing that, we ensure that the delay of an entire pipeline is not propagated to the start point of another pipeline doubling the critical path of the design.

Branch Prediction and Early PC Redirection As it is mentioned in the pipeline hazards subsection, control hazards have a notorious impact on pipeline performance, even more on the more complex pipelines with a considerable number of stages. This fact is more worrying, taking into account that control instruction in a program can be more than 20% of a normal workload [6]. There are two main optimizations to reduce the impact of the control hazards: the branch prediction and the early PC redirection.

Early PC redirection is based on redirecting the PC as soon as the instruction that breaks

the sequential execution of the program is detected and the new PC is obtained. For example, if a branch instruction is completely executed on the execution stage, it is unnecessary to wait until the commit stage for redirection. Alternatively, another example can be an unconditional jump. This type of instruction has the destination codified inside the instruction. Thus, it can be redirected at the decoder stage.

Consequently, it is possible to encounter other PC redirections in different stages at the same time. It is essential to maintain the priority of the redirection. The predominant redirection must be the redirection produced by the older instruction in the pipeline.

Nevertheless, early PC redirection has similar disadvantages as bypassing. Extra logic as multiplexers is placed. This new logic increases the power and area consumed. However, in this optimization, the number of points that need these new paths is much less than the bypassing. On the other hand, these new redirections can introduce new broad paths generating a new extensive critical path.

The second way to reduce the control hazards is doing *branch prediction*. A branch predictor is a hardware mechanism that predicts the address of the instruction following the branch. The predictors that are the most commonly used in simple processors are Branch Target Buffer (BTB), Branch History Table (BHT), and Return Address Stack (RAS).

- **Branch Target Buffer** remembers recent target PCs of the control instructions. It has a cache-like structure that stores the target address for a set of instructions. Like the caches, the set is usually obtained from the last bits of the control instruction PC. Then, when a new control instruction accesses the table, the table returns a prediction depending on the set mapped by the particular instruction. The BTB has to be updated at every PC misprediction with the new target PC.

- **Branch History Table** remembers how the branch was resolved previously, allowing to predict if the branch is taken or not before the execution of the condition. There are many implementations of this predictor. Some of them are simple as the Bimodal predictor, which is a simple array of two-bit saturated counters that crease or decrease depending if the branch is correctly predicted or not. If the upper bit is set, the predictor predicts a taken. However, more sophisticated versions of BTB predictors have been published like Gshare [7] or TAGE [8] which also adds a historical component to the prediction that enables the predictor to detect patterns.

- **Return Address Stack** is a predictor that is specified to predict return instructions destinations. BTB does not predict return instructions well because the return address does not always have the same direction (different parts of the program can call the same function). Return instructions always return to the last procedure call instruction and return address stacks are highly accurate. The implementation is relatively simple. Every time a CALL instruction is executed, its return address is pushed onto the stack. Every time a RETURN instruction enters the pipeline, the following address is popped off the stack, and the processor continues fetching from it.

All the different branch predictors have more or less the same type of penalties. All of them need new memories to store the information in order to make the prediction. These new

memories add extra area and power consumption to the design. Also, since these vast memories used to be implemented with SRAMs in an ASIC design, the Branch predictors tend to have substantial critical paths that can affect the maximum frequency of the design.

2.1.2.2 Superscalar Pipeline

With the basic optimizations, there is a limit of performance that cannot be overpassed easily. The main limitation is that the maximum IPC for a simple single-issue processor is 1 since the maximum number of instructions issued per cycle is only 1. However, during the execution, we will encounter pipeline hazards that will always decrease this optimum IPC.

To optimize the core further, one possible optimization is a superscalar pipeline. A superscalar processor is a single processor that implements a form of parallelism called instruction-level parallelism. Compared with a scalar processor that can execute at most one instruction per clock cycle, a superscalar processor can execute multiple instructions by dispatching multiple instructions to different execution units on the processor simultaneously in one clock cycle. Therefore, it allows more throughput (the number of instructions that can be executed per unit of time) than is possible at a given clock rate.

In order to implement this optimization, a considerable modification of the pipeline is needed. Nearly all the different parts of the pipeline have to be replicated in order to be able to execute multiple instructions in parallel. First of all, the fetch unit needs to be able to fetch at least the number of instructions that will execute in parallel. Also, a new issue logic has to be implemented to detect conflicts between the instructions issued in the same cycle. Also, it needs to track the availability of the resources in the execution stage to know if a particular type of instructions can be executed at that moment or not. This logic tends to be complex and very critical in terms of maximum frequency. Next, the core needs to implement several functional units in the execution stage in order to perform several operations in parallel. Usually, not all the functional units are replicated since some of them, like the FPU, are very expensive in terms of area. Finally, the commit stage also has to be replicated in order to finish several instructions per cycle.

To sum up, this optimization multiplies the maximum limit of the IPC by N (being N the number of parallel instructions on each stage). However, this maximum IPC is difficult to achieve, and the real IPC will be much lower since, with more instructions in parallel, the number of hazards will increase notoriously. Now multiple instructions can have data dependencies, structural dependencies on the same cycle. For example, imagine that there are two instructions, and the second one needs the result of the first one as an operand. Thus, the second instruction will not be issued in the same cycle as the first one. Another example can be on a dual-issue processor with only an FPU. If two floating point instructions must be issued at the same cycle, the second one will need to wait for one cycle since we have encountered a structural hazard. The Intel Pentium P5 [1] was the first modern processor to implement the superscalar concept. In Figure 2.4, there is the diagram of the P5 core. The core has two separate integer pipelines (U and Y) that can be executed in parallel.

Finally, it is essential to see the downsides of these optimizations. As mentioned before, the increment in area is directly proportional to the number of parallel instructions in the core.

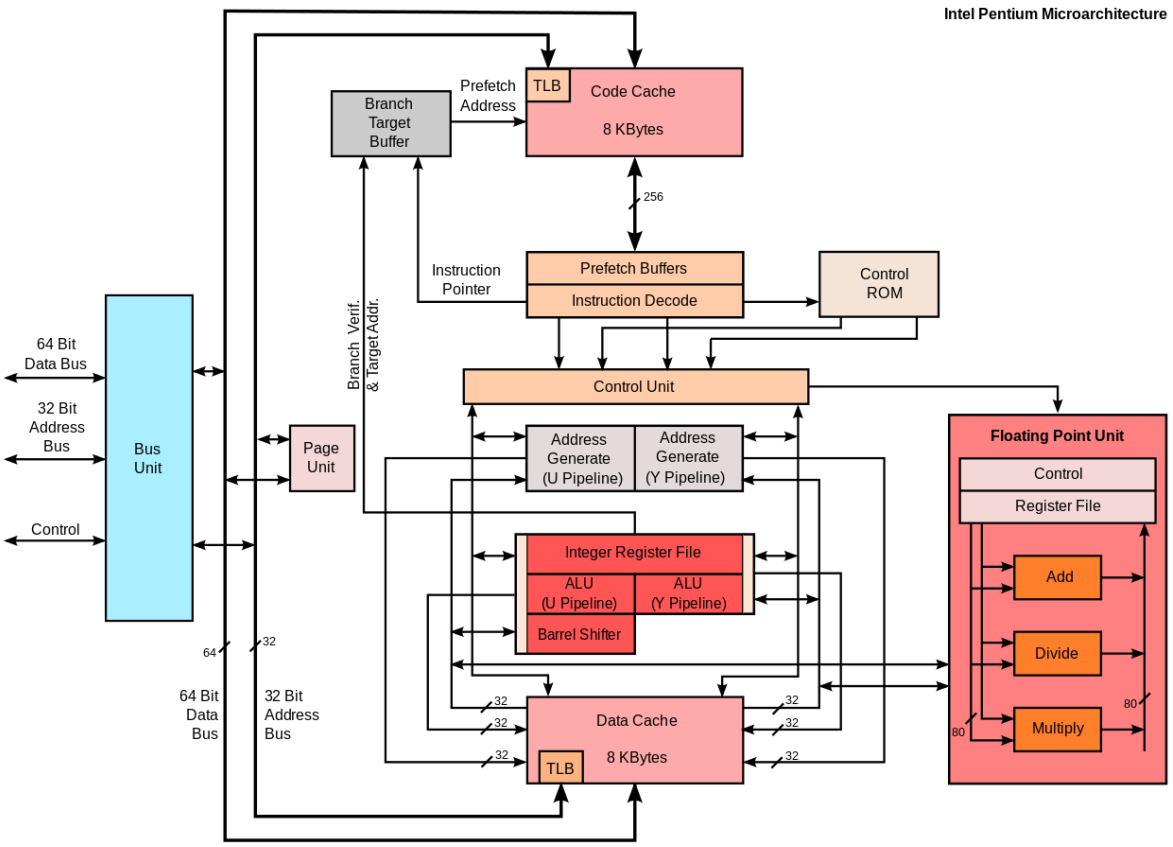


Figure 2.4: Intel Pentium P5 pipeline diagram. Source: [1]

Thus, it is a costly optimization. Also, the critical path on the issue stage can be problematic due to the increment of the issue logic complexity.

2.1.2.3 Out-Of-Order Issue and Execution

As shown in the superscalar section, although adding optimizations, there are still some hazards that are hard to solve, like the back-to-back dependency of two instructions, which prevents issuing both instructions at the same time. At this point, on an in-order processor, the second instruction will be blocked until all the hazards are solved. This blocked instruction stalls all the younger instructions since the execution of the instructions should respect the order of the program. In order to remove this strict blocking situation, the OoO (out-of-order) concept is introduced.

The key concept of OoO processing is to allow the processor to avoid stalls by filling these "slots" during execution time with other instructions that are ready to be executed (that do not have any hazard this cycle). However, the result of the instructions must be committed (modify the state of the processor) with the order described on the program (program order). Thus, the instructions must be reordered at the end to appear that the instructions were processed as usual. A reasonably complex circuitry is needed to maintain that program order.

The complex point of an out-of-order design is the issue stage. The issue stage needs to

track which instructions are ready to be executed (the operands are already computed). This optimization is not in the scope of this thesis. However, some mechanisms and ideas of out-of-order processors can be implemented on an in-order design, increasing the performance without making the design that complex. In this thesis, there is a concept that will be used, which is the out-of-order write-back.

Having an out-of-order write-back basically allows two instructions to finish in a different order than the program order. This characteristic is beneficial in terms of performance because a short latency operation can now be executed after a very long latency operation without waiting for the longest to finish. For example, imagine a memory load instruction that misses on the cache. If there is an arithmetic operation with no data dependencies with this load, the arithmetic instruction can be executed before the load gets the data.

Out-of-order write-back is usually implemented alongside the register renaming mechanism. Register renaming is a feature that is consistently implemented on out-of-order designs. Register renaming removes the WAW dependencies (and WAR dependencies on cores in which the access to the register file does not follow the program order) by having more physical registers than architectural registers and maintaining a map of those for each instruction. with register renaming, an instruction can write its result to an architectural register before an older instruction reads the old value of that register. It can be done since the younger instruction will write on a different physical register that later will be mapped as the conflicting architectural register.

Finally, in the out-of-order write-back is essential to reorder the instruction before committing it to follow the program order. This reordering is usually done by a Re-Order Buffer (ROB). The ROB tracks the state of all in-flight instructions in the pipeline. The role of the ROB is to provide the illusion to the programmer that his program executes in order. Once the instructions are decoded and renamed, they are dispatched to the ROB. As instructions finish execution, they inform the ROB and are marked not busy. Once the head of the ROB is no longer busy, the instruction is committed, and its architectural state becomes visible.

2.1.2.4 Single Instruction Multiple Data (SIMD) and Vector Processor

Single instruction, multiple data (SIMD) is a type of parallel processing. As the name suggests, it takes an operation specified in one instruction and applies it to more than one set of data elements simultaneously. For example, a simple arithmetic operation in a traditional scalar processor would operate over two data operands and store the result in the destination register. In SIMD processing, several independent operand pairs operate with the same operation specified on the instruction to produce the same number of independent result data. In Figure 2.5, there is an example of a SIMD addition of a vector of 4 elements.

A processor that implements SIMD exploits data-level parallelism, but not concurrency: at the end, it can process data simultaneously (parallel). However, each computing unit performs the same instruction with different data. SIMD is particularly applicable to common tasks such as matrix multiplication, vector operations used in multimedia, machine learning, and other applications. Most modern CPU designs include SIMD instructions to improve the performance of this kind of application. The first widely deployed desktop SIMD was with Intel's MMX extensions to the x86 architecture [9].

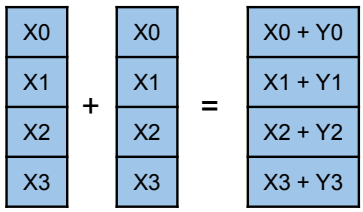


Figure 2.5: SIMD addition of a vector of 4 elements

2.2 Physical Implementation

All the processor parts and optimizations explained until now are a collection of combinational functions and memories chosen precisely to achieve maximum performance and efficiency. However, these elements have to be somehow implemented to be used in the real world. These logic circuits and registers are physically implemented using a large assembly of logic gates. Logic gates are simple circuits that generate a digital signal as output that depends on one or more digital signals inputs. The final physical implementation of those logic gates depends on the technology used. For example, we can map all the gates of one design to one FPGA.

FPGAs contain many programmable logic blocks and a hierarchy of reconfigurable interconnects that connect the different blocks. Logic blocks can be configured to calculate complex logic functions or act as simple logic gates like AND and XOR. Also, logic blocks include memory elements. In this case, synthesis and placing tools only have to map the design to the already fabricated logic gates and memories and do routing using the already fabricated interconnection layers.

In the case of an ASIC design, tools do not have a restriction in terms of the logic blocks available. In this case, the tools have more freedom to implement the design. The limits are set by the silicon area available for the design and the maximum power consumption that the design will consume. In ASIC design, the logic will be implemented with transistors. However, the complexity of the tools would be very high if the minimum element were the transistor. The synthesis and place and route tools use already designed essential elements known as standard cells to avoid this complexity.

2.2.1 Standard Cell Libraries

A standard cell is a group of interconnected transistors with its own layout that implements a boolean logic function (e.g., AND, OR, NAND, XOR, NOT) or a storage function (flip-flop or latch). Inside the boolean logic functions, there are simple operations like NAND, NOR, and XOR and more complex functions such as 2-bit full-adder or muxed D-input flip-flop. Many vendors are designing and selling standard cell libraries as Synopsys, Globalfoundries, or TSMC. It is worth mentioning that a particular standard cell can only be used for the particular technology fabrication node associated with it because the essential part of the standard cells is the layout. This layout is made expressly for a particular technology.

Standard cells are designed to optimize the design for a particular purpose. There are standard cells optimized for power, area, or performance. Thus, there are several types of

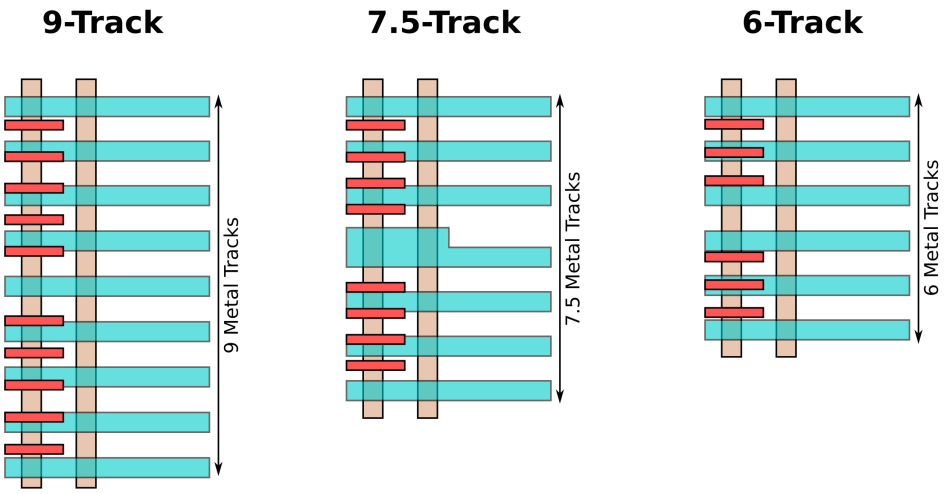


Figure 2.6: Comparation between diferent standard cells with diferent tracks [2]

standard cells, depending on those parameters. There are many parameters on a technology node that can be configured, such as the number of tracks, gate pitch, minimum metal pitch, cell ratio, possible PMOS width, and NMOS width [10]. Selecting those parameters is done by the vendor. The vendor provides several libraries with specific configurations.

The user has to select a specific library depending on the particular design. Usually, the parameters which the user can select are the technology node (65nm, 22 nm, 7 nm), transistor gate pitch, which is also referred to as Contacted Poly Pitch (CPP) (104CPP, 116CPP), the number of tracks (7T, 8T, 12T), or the threshold voltage (regular-Vt, high-Vt f, low-Vt).

The number of tracks is generally used as a unit to define the height of the std cell. It can be related to metal lanes that can be horizontally routed inside the standard cell. For example, a 9 track library implies that 9 routing tracks are available for routing 9 parallel wires with minimum pitch inside a standard cell. In Figure 2.6

If we compare, for example, 6T and 9T libraries, 9T will be faster because it will have more area, and it can have higher drive strength transistors or more transistors to make a more complex operation. Thus, it will give better performance. However, using the 9T library will consume more area and power for the same circuit.

- 9T library is used for higher performance.
- 6T library is used for higher density and low power.

2.2.2 Hard IP Blocks

Standard cell libraries give enough modularity to implement any digital logic. However, sometimes the efficiency or performance achievable with those cells is not enough. Maybe the design has an analog part that is not implementable with the standard cells. For these situations, Hard IP blocks are used. Hard IP blocks are defined as IP cores that cannot be modified. Because they have a low-level representation, hard IP offers better predictability of chip performance in terms of timing and area. Thus, analog IP (SerDes, PLLs, DAC, ADC, PHYs) are provided

to chip makers in transistor-layout format. Also, some digital circuits like SRAMs (very dense memories) are given in a hard IP format.

2.2.3 Performance of a Digital Circuit. Process, Voltage, Temperature Variations

The performance of a single-core processor is the direct multiplication of the IPC and the clock frequency, and therefore it is essential to give the same importance to both parameters. Sometimes the most challenging part of the design is to achieve the right balance of these two parameters to get the maximum performance. The computer architects use instructions-per-cycle (IPC) as a performance metric, but often *ignore* the length of the clock cycle itself.

The achievable clock frequency of a circuit depends basically on two different parts. The digital design itself will determine how many standard cells will have the critical path. The other factor is the delay associated with each standard cell. This delay is associated with the underlying technology used to implement them. Nowadays, processors mainly use Metal Oxide Semiconductor Field Effect Transistor (MOSFET) as the base technology, where all transistors and the interconnection paths have some associated delay. This delay is cumulative and applies to all signals that flow through a given path.

This standard cell delay is not equal for all the chips of the same technology. It can also depend on the chip's conditions. A fundamental concept for synthesis and physical design are Process, Voltage, Temperature (PVT) variations. Since a chip has to work into very different conditions correctly, some characterization of the technology is needed for a wide range of conditions. To ensure that the design is capable of performing correctly in the target frequency, it is necessary to simulate it at different conditions of the process, voltage, and temperature, which the integrated circuit may face after fabrication. These conditions are called corners.

Process variation is the deviation in attributes of the transistor during the fabrication. Not all the dies in a wafer are manufactured with the same characteristics. These differences come from process variations like a bad centering of the tools or the not uniform distribution of metal particles. This variation is dominant in lower node technologies (<65nm).

Voltage variation is significant nowadays since the operation voltages of modern technology nodes are near or below 1 V. Thus, a minimal variation in the power supply can produce a significant variation of the operating point of the transistors. We need to consider voltage variation in the design process. Multiple factors can produce voltage variation, such as the parasitic resistance of the power grid. Another factor is the parasitic inductance of the circuit, which is a rapid increment of the current flowing in the circuit that can cause voltage spikes. However, this voltage variation can be used intentionally to reduce the power consumption or be augmented to increase performance.

Temperature variation is not only related to the ambient temperature. In the end, the temperature on the chip is the sum of the ambient temperature, and a large amount of power

is dissipated in a tiny chip area that increases the temperature significantly. The temperature inside the chip can vary within a big range depending mainly on the power dissipation of the chip. That is why temperature variation needs to be considered. In the end, temperature variation has to be considered because the delay of the cells varies with the temperature. On all technologies, the delay increases with temperature. However, this is not true for modern technology nodes. For deep sub-micron technologies, exists a phenomenon called temperature inversion [11].

2.3 RTL Simulation, FPGA Emulation, and Gate Level Simulation

In processor design, a huge part of the development efforts is spent on verifying the design. It is essential to test the design constantly to find errors and evaluate the applied modifications' performance implications. Testing regularly the design makes it easier to find which step of the implementation has produced some error and fix it early. The most common thing is testing the design at different levels.

In the early stages, when the design is even in the specification stage, the computer architects use high-level simulators to evaluate the early specifications of the design. There are accurate cycle simulators for single or small multi-core processors [12], simulators for large multi-core processors [13, 14], and simulators for massive multi-core HPC systems with more than 2048 cores [15]. However, these simulators only give the computer architects some estimation of the performance they will get to the final design.

When the design is implemented or even partially implemented, the most commonly used strategy is the RTL simulation. RTL simulation is the cheapest way for complexity and infrastructure development to simulate a design since the only necessary thing is the RTL design. The simulator makes a model of the completed digital design, which models the logic function. Also, the simulators maintain the naming of each variable in the HDL design, and it can give the cycle state of each variable for debugging purposes. RTL simulations are cycle-accurate. However, it is impossible to compute the sub-cycle delays since it does not have a physical technology model. In terms of simulation time, RTL simulations are slow since the computer has to compute the result of all the registers and wires one by one for each cycle. This peculiarity makes the simulation very suitable for debugging purposes but not suitable for benchmarking very large programs. New ideas like [16] allow connecting small RTL modules to a high-level full-system simulator allowing the execution of larger benchmarks taking advantage of the fast execution of these simulators.

For benchmarking of very large programs, it is necessary to emulate the processor design on an FPGA. FPGAs give the developer a much easier physical implementation of the design than ASIC. In the end, the physical design process is similar, but since the FPGA is reconfigurable nearly in real-time, the loop from having a design to having that design implemented and running on an FPGA is much faster than having a custom ASIC running. The FPGA performance can be ten times lower than ASIC. However, it is more than enough to be used as a benchmarking platform.

FPGA prototyping has the necessity of using hard IPs for external communication and to obtain resulting data. Apart from this extra circuitry needed, FPGAs have a big downs side compared to the RTL simulation that is the fact of not having the possibility to know the value of all the wires or registers cycle by cycle. This happens because the FPGA does not have enough outputs to give that amount of information. Therefore, possible errors are hard to find and solve. Another downside of the FPGA prototyping compared with the RTL simulation is the compilation/synthesis time. The design synthesis is quite slow, and for small runs, the gain on emulation speed of FPGA over the RTL simulation is not worth it.

Finally, with the two ways of simulation/emulation described, it is not possible to ensure that the design is bug-free. Also, it is impossible to know the performance expected in the final ASIC implementation since it is not possible to obtain the frequency of the design. For that purpose, gate-level simulations are used. Mainly, the gate-level simulations are used to ensure the correct operation of the circuit in the post-synthesis and post-P&R states. This type of simulation is really accurate because it includes all the information of the technology node that will be used for fabrication. It uses the model of the technology standard cells and hard IPs. With this information, it is possible to find timing constraints violations, find glitches, and find possible propagation of X values (e.i., uninitialized registers). In Table 2.1, there is a summary of all the different simulations and emulations approaches with their properties.

Table 2.1: Different types of simulation and emulation approaches together with their properties

	Speed	**Information**	**Time Stamp**
High level simulator	Fast	High level information	Cycle
RTL simulation	Slow	Register and wires state	Cycle
FPGA emulation	Fast	Limited by the I/O	Limited by the I/O speed
Gate level simulation	Super slow	Post-synthesis gate state	Sub-cycle

2.4 Processor Benchmarking

It is not easy to compare two processor designs in terms of performance. Even having very similar specifications, the performance can vary a lot depending on the implementation of the system. Similarly, it is challenging to predict how a modification will affect the overall performance of a processor. Moreover, a processor will not perform equally on all the different workloads.

For this reason, it is essential to have a methodology to measure the design performance after each microarchitectural decision. Also, it will be needed for comparing different designs. This methodology is known as benchmarking. Benchmarks are small programs designed to mimic a particular type of workload on the compact program's design. Benchmarks extract the critical algorithms of an application, which contains the performance-sensitive aspects of that application. Exists two types of benchmarks. The first type is the synthetic benchmarks. These benchmarks are specially created programs that stress some particular components of the architecture. Secondly, the application benchmarks run real-world programs on the design to evaluate the performance of a real-world application in a very accurate manner. The application benchmarks have an accurate representation of real-world problems. Synthetic benchmarks help

test individual processor parts in a specific way.

During this project, two benchmarks have been used. Embedded Microprocessors Benchmarks Consortium (EEMBC) CoreMark has been selected, a compact benchmark, to quickly evaluate the performance impact of the different modifications made to the processor during this thesis. Also, it is useful for a quick comparison of our design to other competitors since it returns a simple score output. The second benchmark is the EEMBC AutoBench suite. EEMBC AutoBench gives a more robust performance evaluation since it has several benchmarks emulating different applications. This way is easier to see in which application the processor is still not achieving the target results.

2.4.1 CoreMark

EEMBC's CoreMark [17] is a benchmark that measures the performance of CPUs used in embedded systems. It is intended to become an industry standard, replacing the widely used Dhrystone benchmark [18]. CoreMark returns a performance indicator generated by the execution of a simple code (CoreMark and CoreMark/MHz) replacing the DMIPS/MHz obtained in the Dhrystone. However, this simple code is not entirely arbitrary and synthetic. The code for the benchmark uses basic data structures and algorithms that are common in practically any application. CoreMark executes the following list of basic algorithms: list processing (find and sort), state machine (determine if an input stream contains valid numbers), matrix manipulation (common matrix operations), and Cyclic Redundancy Check (CRC). CoreMark also sets specific rules about how to run the code and report results, thereby eliminating inconsistencies. To avoid aggressive optimizations from the compiler side, every algorithm in the benchmark derives a value that is not available at compile time.

Finally, it is worth mentioning that the CRC algorithm serves another function, a part of the computational complexity. It works as a linked list's hash function to ensure the CoreMark benchmark's correct operation, essentially providing a self-checking mechanism.

CoreMark is commonly used to compare performance between processors quickly. Indeed, it can not be completely reliable, but it gives an initial idea to rank different processors quickly and with some level of trust.

2.4.2 EEMBC AutoBench Performance Benchmark Suite

AutoBench 1.1 [19] is a suite of benchmarks created by EEMBC focused on the performance evaluation for microprocessors in automotive, industrial, and general-purpose applications. This benchmark suite has 16 kernels divided into two groups depending on whether they have ,or not floating point operations. The two groups are:

- Integer kernels: Finite Impulse Response (FIR) (aifirf), Bit Manipulation (bitmnp), Cache "Buster" (cacheb), CAN Remote Data Request (canrdr), Pointer Chasing (pntrch), Pulse Width Modulation (PWM) (puwmod), Road Speed Calculation (rspeed), Tooth to Spark (ttspark).

- floating point benchmarks: Angle to Time Conversion (a2time), Fast Fourier Transform (FFT) (aifftr), Fourier Transform (iFFT) (aiifft), Basic Integer and Floating Point (basefp), Inverse Discrete Cosine Transform (iDCT) (idctrn), Inverse Fast Infinite Impulse Response (IIR) Filter (iirflt), Matrix Arithmetic (matrix), and Table Lookup and Interpolation (tblook).

3 Related Work

Nowadays, RISC-V, with its open-source nature, is pushing universities and research centers to start on the development of open-source processors and platforms. Many designs are being developed, from very simple in-order designs to superscalar out-of-order designs with huge complexity. This new opportunity that the open-source environment is giving to the academy is pushing computer architecture research into a promising future. This section presents the most well-known academic designs proposed by the different research centers and universities.

3.1 RISC-V ISA

RISC-V is an open instruction set architecture (ISA) that originated in 2010 from a research project at the University of California at Berkeley and is now supported by the RISC-V Foundation, counting over 100 international institutions, including universities and research centers, and companies worldwide. RISC-V is a new ISA initially designed to support computer architecture research and education, but it is sophisticated enough to become a standard in the industry.

The RISC-V ISA is defined as avoiding implementation details as much as possible (although a commentary is included in implementation-driven decisions). It should be read as the software-visible interface to a wide variety of implementations rather than designing a particular hardware artifact. The RISC-V manual is structured in two volumes. There is the unprivileged volume [20] that covers the design of the base unprivileged instructions, including optional unprivileged ISA extensions. Unprivileged instructions are generally used in all privilege modes in all privileged architectures, though behavior might not vary depending on privilege mode and privilege architecture. The second volume provides [21] the design of the privileged architecture.

The unprivileged ISA can be separated into two blocks, base integer ISA and extensions. It allows small designs to implement only the specific extensions that it needs in each case. In this way, the design can still be efficient in a specific field. The most commonly used extensions are listed in Table 3.1.

3.2 In-Order Designs

There are many in-order designs in academia since the complexity is lower than other designs, and it is easy to have a functional version implemented. It can be quickly implemented and

Name	Description
Base	
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers
RV64I	Base Integer Instruction Set, 64-bit
RV128I	Base Integer Instruction Set, 128-bit
Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision floating point
D	Standard Extension for Double-Precision floating point
G	Shorthand for the base and above extensions
C	Standard Extension for Compressed Instructions
V	Standard Extension for Vector Operations

Table 3.1: RISC-V: base Integer ISAs and extensions

verified thanks to its simple nature. In this section, there are a few examples of in-order designs with more relevant information.

3.2.1 Ariane Core

One of the most recognized academic in-order cores is Ariane (now known as CORE-V CVA6). Ariane is a single issue, 6-stage, in-order core developed on the ETH Zurich university. It implements the 64-bit RISC-V ISA. Apart from implementing the base ISA, it also implements the M and C extensions in its default configuration. In addition, Ariane can accept the F and D since it is possible to choose by some define constant to add a floating point unit compliant with the IEEE specification. The base ISA and the extensions are compliant with the User-Level ISA version 2.1 and the draft privilege extension 1.10. In terms of the privileged level specifications, it fully implements the three levels of privilege (machine mode, supervisor, and user). This privilege support allows the core to boot a Unix-like operating system successfully.

The purpose of the Ariane core is to run a full operating system with a reasonable target of IPC and power efficiency. In order to achieve the desired performance, the core needs to run into a moderate-high frequency. In order to achieve that frequency, the pipeline of Ariane is divided into 6 different stages. It has two stages of fetch, two decoding stages, one execution stage (however, the only functional unit that takes only one cycle is the arithmetic unit), and a commit stage. The first-level caches are very configurable. The instruction cache is configured as a 4-way associative cache in the default configuration and data cache as an 8-way associative. Instruction cache and data cache have 1-cycle, and 3-cycle hit latency, respectively.

Ariane has been fabricated several times in a considerable number of projects [22], [23]. The most relevant tape-out [3] was fabricated with the 22nm Fully Depleted Silicon On Insulator (FD-SOI) node from GlobalFoundries. In the physical design of this tape-out, the design was signed-off with a 902MHz worst case at 0.72 V, 125 °C and SSG. Nevertheless, when the design

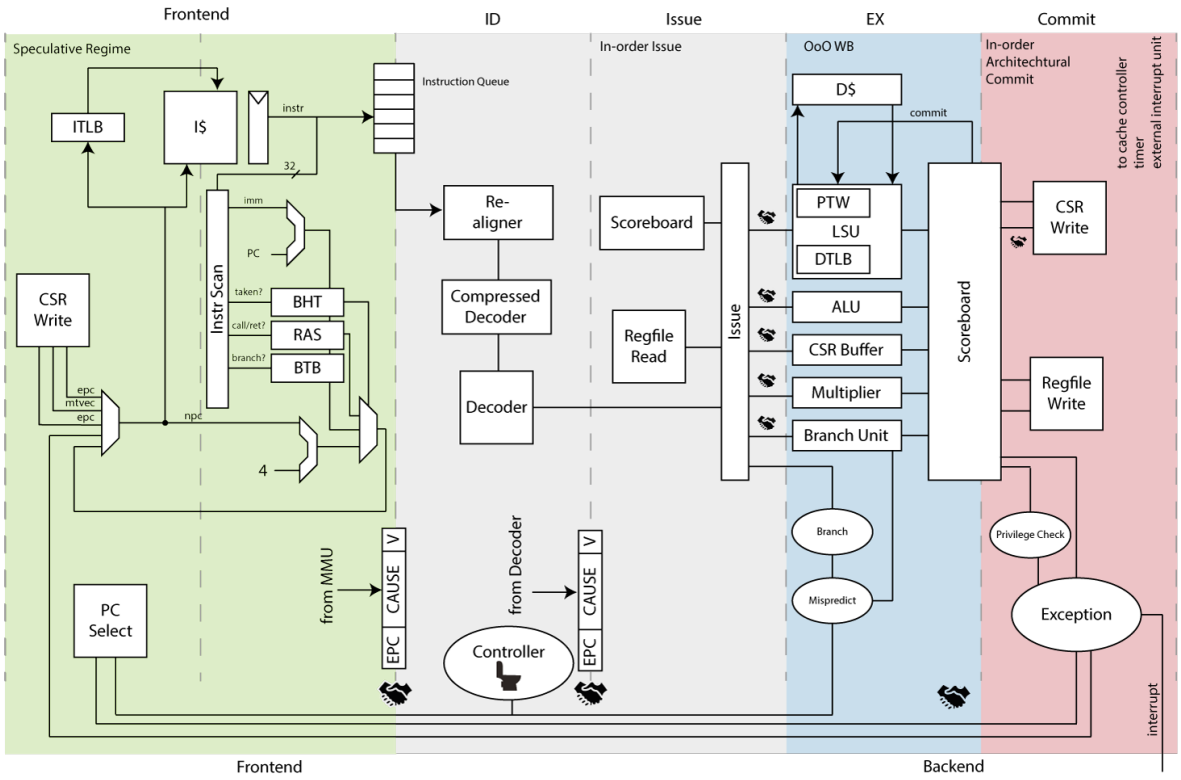


Figure 3.1: Blockdiagram of Ariane. Source: [3]

was tested after fabrication with more extreme conditions (operational voltage of 1.15V and using the body-biasing technique), it can achieve a clock frequency of 1.7 GHz. The Ariane core can obtain 1.61 Dhrystone MIPS per MHz (DMIPS/MHz).

In Figure 3.1 there is the pipeline diagram of Ariane. On the front end, we can encounter two fetch stages. The PC generation with the instruction cache access is located in the first stage. Since it uses a Virtually Indexed, Physically Tagged (VIPT) cache, the access on the instruction Translation Lookaside Buffer (TLB) is done in parallel. On the second fetch stage, the instruction arrives from the instruction cache. The branch predictors are located in this stage. This is because some instruction information is needed, and the second stage is the first one to have the instruction itself.

An instruction queue divides the decode stage from the front-end. On the decode stage, there is the re-alignment mechanism apart from the decoder. The next stage is the issue stage. In the issue stage, the register file is read. Also, the scoreboard is accessed in order to detect the data and structural dependencies. In the execution stage, there are multiple processing units. There is an Arithmetic Logic Unit (ALU), a multiplication and division unit, and a branch unit in charge of computing the branch/jump target, and it detects if there was a misprediction. Apart from those units, the memory pipeline is also located in the execution stage. Since Ariane can do the write-back out of order, there is no need to coordinate the different functional units to write in the correct order. Finally, the write-back in the register file and the exception/CSR mechanism is located on the commit stage.

Until late December of 2019, Ariane was developed and improved by the OpenHW group.

Now, the official name is CVA6. OpenHW adds a new quality step to push the design into the industry standard. CVA6 has a more advanced verification environment and some new microarchitectural optimizations like the register renaming to eliminate the WAW dependencies.

3.2.2 Rocket64 Core

Another well know academic RISC-V core is the Rocket core [24]. Rocket is a core pipeline developed by the Berkeley Architecture group and was one of the demonstrators of the potential of RISC-V ISA. Rocket is a 5-stage in-order scalar core generator. Rocket is a very configurable core and can implement the RV32G or RV64G ISAs. It has a simple front-end with the essential branch predictors, Memory Management unit (MMU) that supports page-based virtual memory, and a non-blocking data cache to improve performance. Rocket supports the three privilege levels (machine mode, supervisor mode, and user mode) to execute a Unix-like operating system. The Rocket chip generator is entirely written in Chisel [25], an HDL language based on Scala. The Chisel compiler produces a synthesizable Verilog RTL code.

Rocket generator is very configurable. There are a large number of parameters to select which ISA extension will be implemented or what will be the size of the microarchitectural structures for the resources/performance trade-off. The ISA extensions M, A, F, and D can be individually selected. The size of the branch predictors such as BTB, BHT, and RAS can be modified. Also, the caches and TLB sizes can be configured and the number of floating point pipeline stages. Thanks to all the very configurable modules and different extension implementations, Rocket can be considered a library of processor components.

The pipeline of Rocket core is divided into 5-stage. However, in some documents, it is described as a 6-stage pipeline [4] if the PC generation is considered as a stage itself. The structure of the front-end is shown in Figure 3.2. The front-end has the PC generation and the different branch predictors (BTB, BHT, and RAS). Besides, it has the instruction cache with its own TLB. The TLB and the instruction cache are accessed simultaneously since it follows a virtual indexed, physically tagged scheme. The output of the cache is registered to eliminate a possible critical path through the cache SRAMs.

On the back-end part of the pipeline, there are four different stages shown in figure 3.3. The decode stage is straightforward. In the same decoding stage, the register file is read alongside the scoreboard to detect data and structural dependencies. In the execution stage, there are multiple processing units. There is an ALU, a multiplication and division unit, and a branch unit in charge of computing the branch/jump target, and it detects if there was a misprediction. In order to increase the IPC, the bypasses are implemented in this stage too.

Rocket has a memory stage to mask the latency of the memory operations since it does not have an out-of-order write-back. This data cache has a load and store queue, allowing non-blocking cache execution. Finally, the data is written back to the register file in the commit stage. Apart from this, the commit stage has the Rocket Custom Co-processor Interface (ROCC). ROCC interface facilitates communication with coprocessors/accelerators such as crypto units (e.g., SHA3) and vector processing units. Any ROCC command will be executed by the coprocessor (barring exceptions thrown by the coprocessor); nothing speculative can be issued over ROCC.

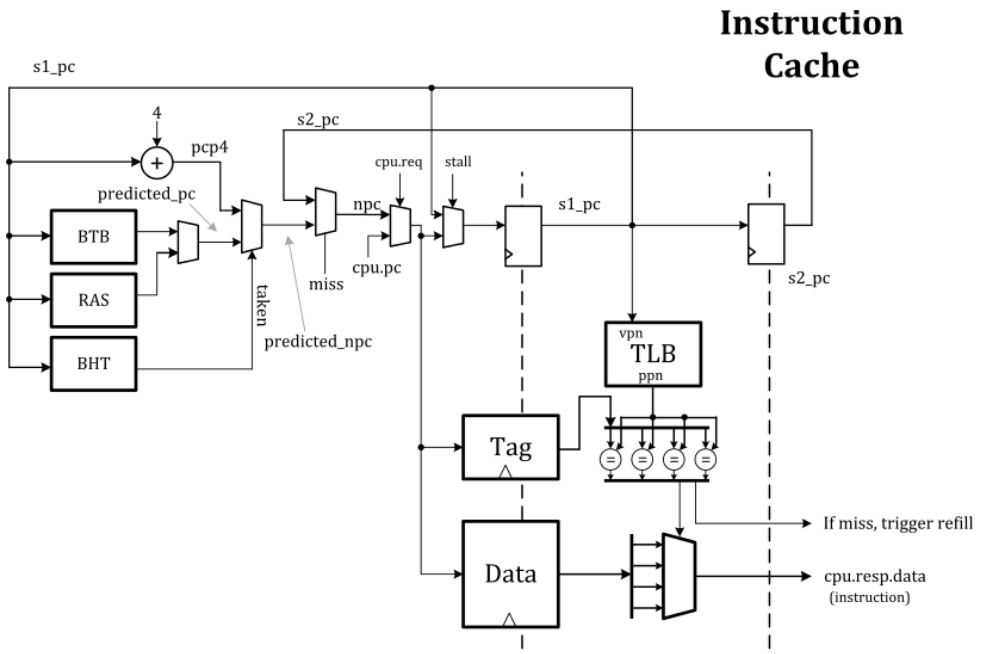


Figure 3.2: Blockdiagram of Rocked core frontkend. Source: [4]

The Rocket core has been taped out [26] along the 64-bit Hwacha vector accelerator connected by the ROCC. The tape-out was done using a TSMC40GPLUS process. The design achieved 1.72 DMIPS/MHz, a 2.3 CoreMark/MHz, and a maximum frequency of 1.3 GHz when it is running with a high operational voltage of 1.2V. In this operation point, the SRAM array on the caches becomes the speed-limiting factor.

3.2.3 Riscy In-Order Core

Riscy in-order is a single issue that has two configurations in terms of number stages: a simple 5-stage version and high-performance 7-stage configuration [27]. The Riscy core family was developed in the CSAIL laboratory of the MIT university. It implements the RV64I integer ISA and the G extension's complete implementation (grouping the extensions M, A, F, and D) and the C extension. It also implements the privileged ISA achieving to boot the Linux kernel successfully. Riscy in-order, like its bigger brother RiscyOO, is written entirely in Bluespec SystemVerilog [28], taking advantage of the modularity and atomicity given by the properties of this language. Bluespec SystemVerilog has a compiler, named Bluespec compiler (BSC), that emits standard Verilog for maximum compatibility with any synthesis toolchain. The BSC has an open source [29] version that includes all the necessary tools for simulation and synthesis.

The 5-stage pipeline is shown in Figure 3.4. The first stage in the pipeline is the Fetch stage, which allocates the PC generation and the instruction cache access. Riscy implements BHT and BTB branch predictors that improve the PC generation.

The Decode stage is the second stage of the pipeline. It is a complex stage because it performs the re-alignment of the instructions, the decoding, the read of the register file, redirecting a misprediction of a JAL instruction, and the access to the scoreboard to detect dependencies. In the Execution stage, the ALU and the branch misprediction module are located along with the

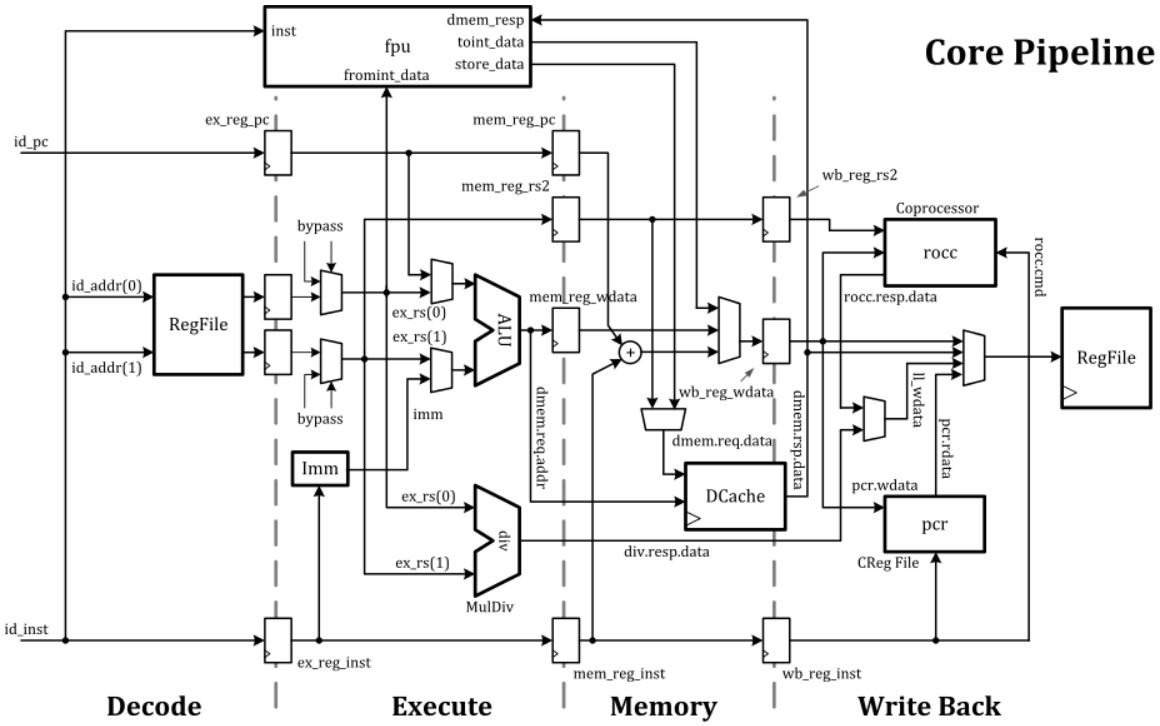


Figure 3.3: Blockdiagram of Rocked core backend. Source: [4]

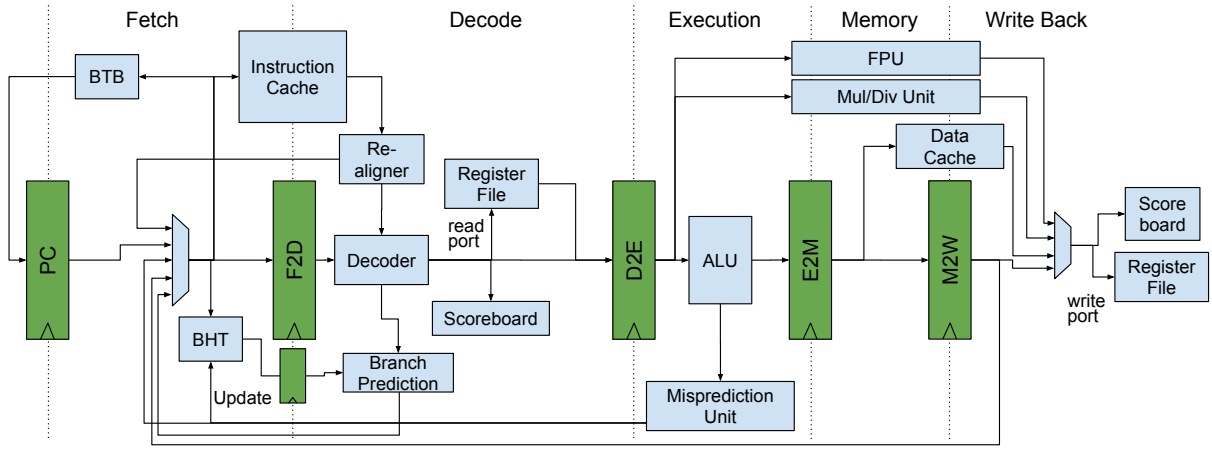


Figure 3.4: Diagram of the original Riscy 5-stage pipeline

request of the FPU and multiplication and division unit. The Memory stage only sends requests to the data cache. The direction is already computed on the Execution stage. Since the access latency is one cycle, the cache response with the data arrives on the next cycle. The last stage is the Write-Back. In Write-Back, all the results from the multicycle units are recollected. Also, the data cache response arrives at this stage. The Register File writes also are made in this stage since all the data is already computed. Finally, the exception mechanism and the CSR access are also located here.

The 7-stage pipeline introduces two different stages on the front-end and an extra cycle of latency on the data cache. This way reduces a lot the critical paths making a significant increment on the maximum frequency. However, this modification reduces the IPC of the design.

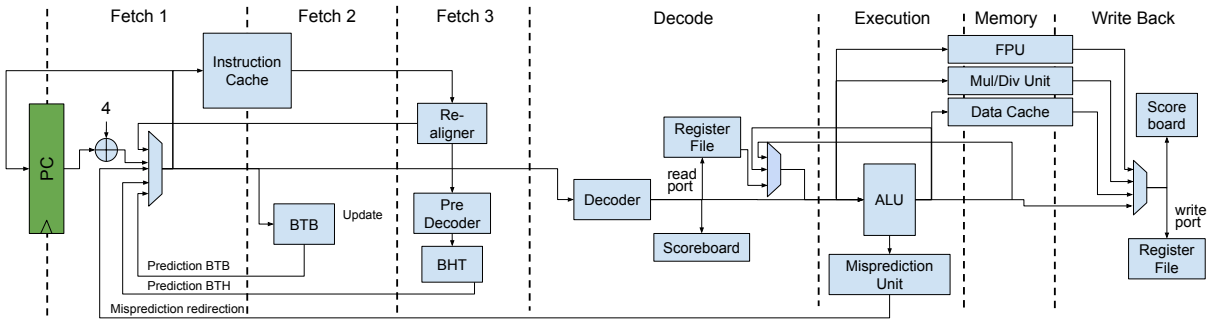


Figure 3.5: Diagram of the original Riscy 7-stage pipeline

In Figure 3.5, there is a diagram of the Riscy 7-stage pipeline.

Both versions achieve a good performance. The 5-stage pipeline can achieve a 2.23 CoreMark/MHz and a maximum frequency of near 1 GHz on the typical corner on the 22nm FD-SOI Globalfoundries technology node. The 7-stage is achieving a lower CoreMark/MHz of 2.01. However, it achieves a frequency of 1.44 GHz on the same conditions. This increment of frequency gives a speedup of 1.31X with respect to the 5-stage pipeline.

3.3 Out-of-Order Designs

Single-issue in-order cores have limited IPC performance and are not very suitable for HPC workloads. Thus, some universities and laboratories are researching more complex processors to target that type of application. Out-of-order designs are much more complex to implement and even harder to verify. However, there are outstanding designs in academia.

3.3.1 RiscyOO

RiscyOO [30] is a RISC-V superscalar out-of-order from the MIT Computer Science & Artificial Intelligence Lab. It has the peculiarity of using the framework called Composable Modular Design (CMD) that gives the design a very high level of abstraction to its modules. The RiscyOO is written in Bluespec SystemVerilog and is compiled into Verilog, which can be synthesized to FPGA or synthesized using standard ASIC design flows. The properties added from CMD ensure composability when modules inside the design are individually optimized or changed, maintaining the interface.

RiscyOO is capable of booting Linux, and it can be synthesized on FPGAs at a frequency up to 40 MHz. A diagram of the pipeline of the design is shown in Figure 3.6. The design can achieve a frequency of up to 1.1 GHz on an ASIC target in a 32 nm technology. RiscyOO has been evaluated using an FPGA environment. The performance evaluation shows excellent results in the SPEC 2017 benchmark suite that it beats in-order processors in terms of IPC but will require more architectural work to compete with wider superscalar commercial ARM processors. The strong point of the design is the CMD properties. The modules designed under this framework (e.g., ROB, reservation stations, and load-store unit) can be reused and refined by other implementations.

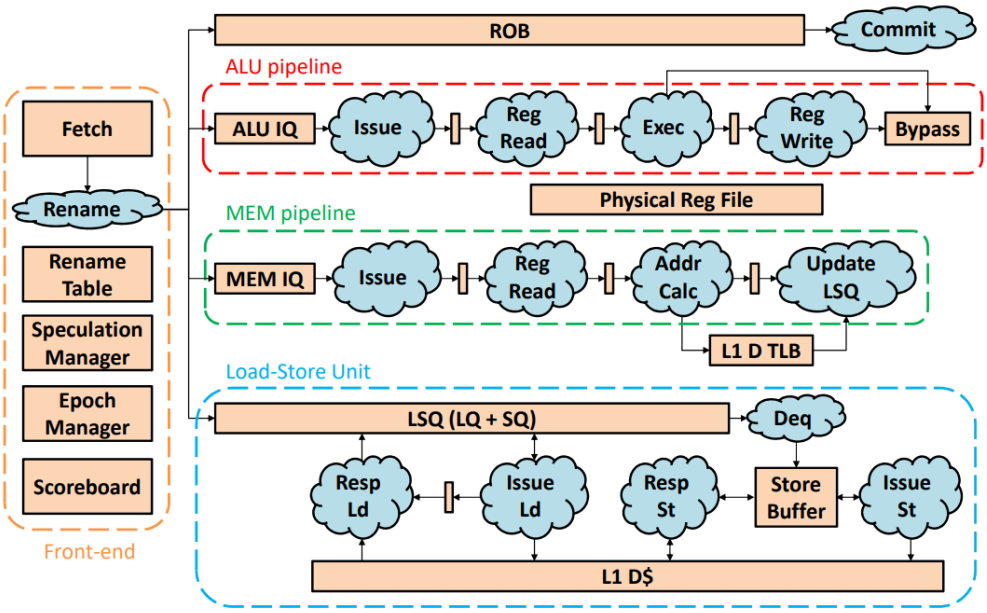


Figure 3.6: RiscyOO structure diagram

3.3.2 Berkeley Out-of-Order Machine (BOOM)

The Berkeley Out-of-Order Machine (BOOM) is an open-source superscalar out-of-order RISC-V core written in the Chisel. It implements the RV64GC ISA. BOOM is synthesizable with a very parameterizable component. BOOM was created at the University of California, Berkeley, in the Berkeley Architecture Research group. The target for this core was a high performance, to be synthesizable, and to be parameterizable for architecture research. Like the majority of the modern high-performance cores, BOOM is superscalar and out-of-order. Part of the modules used for BOOM is extracted from the Rocket generator as a library of components. The essential modules imported from Rocket are the cache hierarchy and uncore (controllers, I/O) to quickly bring up an entire multi-core processor system able to boot Linux.

Until now, there have been three revisions of the BOOM core. There is a diagram of the three versions in Figure 3.7. Furdermore, there is a resilient, wide-voltage-range implementation of BOOM tape-out on TSMC's 28-nm HPM process [31]. The oldest version is BOOMv1 [32]. BOOMv1, configured similarly to an ARM Cortex-A9, achieves 3.91 CoreMark/MHz with a core size of 0.47 mm2 in TSMC 45 nm excluding caches (and 1.1 mm2 with 32 kB L1 caches). BOOMv1 can achieve the same maximum frequency as the Rocket core in this technology (1.5 GHz). Both have the same critical path on the SRAM memories.

The second revision of the core was named BOOMv2 [33]. This generation is based on the information collected through the physical design using a commercial technology node, TSMC 28 nm, of the first generation. BOOMv2 reimplements the front-end with more stages in order to improve the critical path. Moreover, it integrates better branch predictors. On the back-end, it has a new custom register file to reduce the size and access time. Finally, it adds a more advanced distributed issue queue to reduce the critical path in this module. This new issue queue increases the number of instructions that can be scheduled to respect the original unified issue queue. The new revision reduces the total fanout-of-four by 24%, but it reduces the IPC

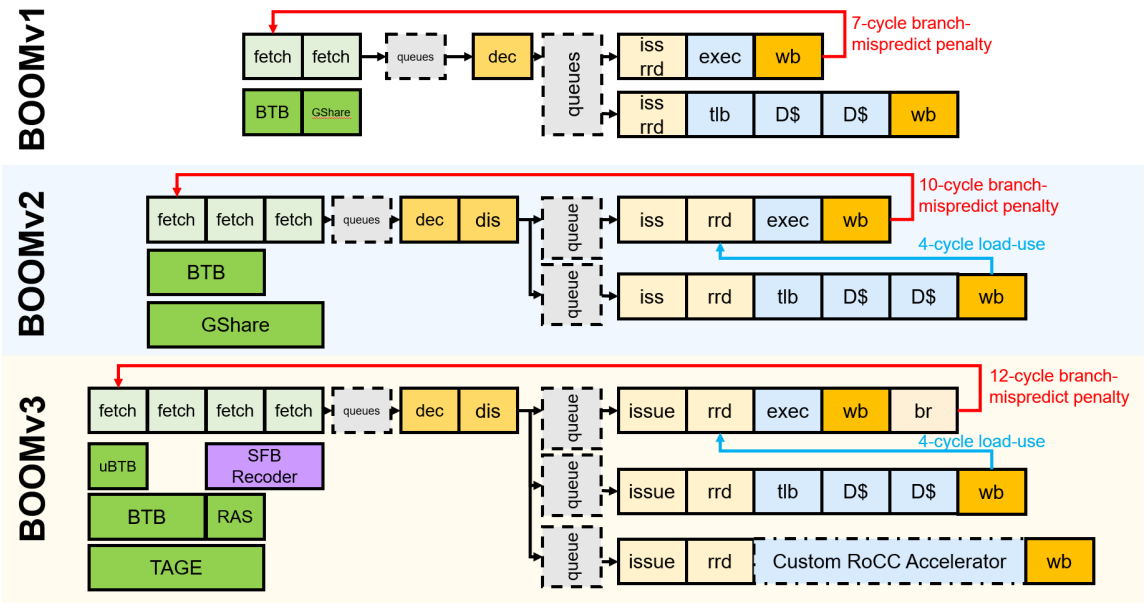


Figure 3.7: Evolution of the BOOM pipeline

by 20%.

The current version of the BOOM microarchitecture (SonicBOOM, or BOOMv3) [34] has a considerable performance improvement and can be compared with commercial high-performance out-of-order cores. SonicBOOM can achieve up to 6.2 CoreMark/MHz. These results give it a 2X gain in performance compared to BOOMv2. SonicBOOM supports a broader set of software stacks and addresses the main performance bottlenecks of the core while maintaining physical realizability. It supports the RISC-V C extension, modifying all the front-end and the branch predictors' structures. The most significant contribution to overall core performance is the inclusion of a high-performance TAGE [8] branch predictor. It is synthesized at 1GHz with the same TSMC 28 nm process as BOOMv2.

4 Experimental Environment

During this thesis, several tools and programs have been used to verify the behavior and validate the performance of the design. This chapter describes the environment used for the RTL design and verification and the benchmarks selected for the IPC performance analysis. Finally, there is a section in which all the physical design tools are described.

4.1 RTL Environment

The core has been written in SystemVerilog. SystemVerilog [35], standardized as IEEE 1800, aims to provide a well-defined and official IEEE unified hardware design, specification, and verification standard language. The selection of this Hardware Description Language (HDL) is related to the massive support of the industry. The industry support gives us a wide range of tools and libraries that simplify simulation, verification, and physical design (FPGA and ASIC).

In this project, the tool Verilator [36] (an open-source C++-based RTL simulator) is used to perform the cycle-accurate simulation. Also, Verilator provides extra features like code coverage that are very useful for verification. For FPGA emulation, we have used the Kintex 7 FPGA boards for Xilinx. Some Hard IP blocks for FPGA synthesis have been generated with the Xilinx Vivado proprietary tool.

4.2 ASIC Tool-Flow Environment

The setup used for the ASIC synthesis and Place & Route phases is described in this section. To do the ASIC synthesis, we use the Genus [37] tool from the Cadence. For the Place & Route phases, we use the Innovus [38] also from Cadence.

4.2.1 Standard Cell Libraries

We make use of 22FDX technology for this project. These libraries are designed for a 22nm Fully-Depleted Silicon-On-Insulator (FD-SOI) technology node from Globalfoundries. Synopsis provides several standard cell libraries for this technology. For this project, only core cell libraries are used with these three corners: typical, best-case, and worst-case. Since the target is energy efficient, we will use the following configuration:

- Number of tracks: 8 tracks for high density.

Table 4.1: List of Standard Cell libraries used in the design for Globalfoundries 22FDX.

Track	Vt	Corner	Conditions
8T	Low	Typical	0.8 V and 25°C
8T	Low	Fast	0.88 V and -40°C
8T	Low	Slow	0.72 V and 125°C
8T	SuperLow	Typical	0.8 V and 25°C
8T	SuperLow	Fast	0.88 V and -40°C
8T	SuperLow	Slow	0.72 V and 125°C

- CPP: 104
- Threshold voltage: a combination of low-Vt and ultralow-Vt for high speed.

4.2.2 Hard IP Blocks

The design uses hard IP blocks for the SRAM arrays inside the instruction, data, and second-level caches. Memory compilers provided by Synopsis have been used in order to generate the SRAM blocks. The compilers generate the IP blocks with all the physical information needed for the physical design tool flow. Table 4.2 lists the memory blocks generated for the processor design. In this design, we use the high-performance SRAMs since that is a critical part of the design. It uses the same corners as the Standard Cell libraries used for synthesis (Typical, Fast, and Slow).

Table 4.2: List of SRAM cells used in the design for Globalfoundries 22FDX.

SRAM block name	Words	Width	Description
IN22FDX_R1PH_NFHN_W00064B080M02C256	64	80	1RW IL1 cache tag array
IN22FDX_R1PH_NFHN_W00064B088M02C256	64	88	1RW DL1 cache tag array
IN22FDX_R1PH_NFHN_W00256B128M02C256	256	128	1RW Data L1 array
IN22FDX_R1PH_NFHN_W00512B128M02C256	512	128	1RW Data L2 array
IN22FDX_R1PH_NFHN_W00128B128M02C256	128	128	1RW L2 cache tag array
IN22FDX_R1PH_NFHN_W00128B048M02C256	128	48	1RW L2 cache tag array

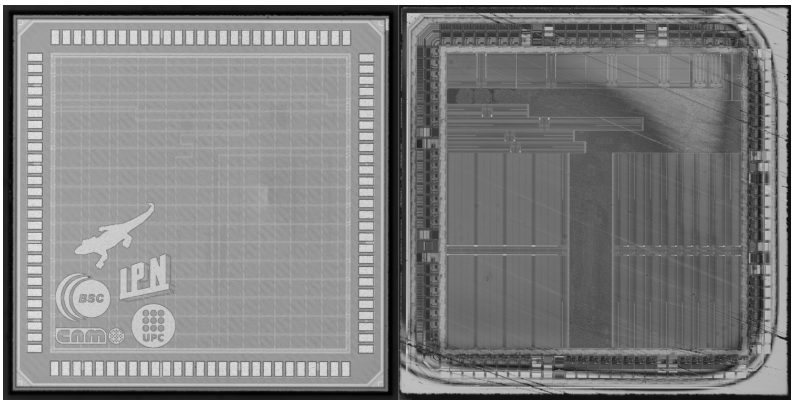
5 Microarchitectural Design of Sargantana

Sargantana is an in-order design that targets some HPC applications being an embedded core. Also, it is essential to consider that the core is designed in the academy and has to be very friendly to work as a base implementation for other projects. With those restrictions, the core should not be a super complex design like a super-scalar out-of-order design for the following reasons: an embedded core needs to be power-efficient, and a super-scalar out-of-order design is very complex to develop, verify and can be very difficult to modify for testing new modifications for academic research. However, it should have a high-performance component. Thus, it should achieve high IPC with a relatively high clock frequency. It should also be modular and easy to modify to enable the research of new custom instructions, accelerators, or optimizations for different HPC kernels (machine learning, autonomous driving, or genomics).

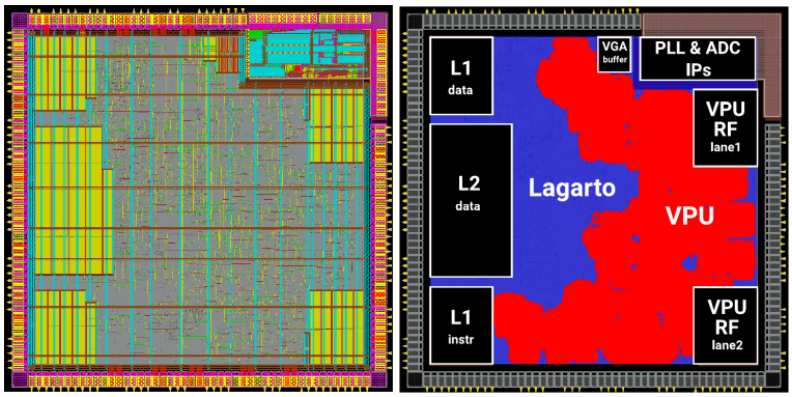
5.1 5-stage Lagarto-Hun Analysis

The new core Sargantana has the 5-stage Lagarto-Hun [39] core as a predecessor. 5-stage Lagarto-Hun is a single-issue in-order 5-stage core developed by CIC-IPN, BSC, IMB-CNM (CSIC), and UPC, which implements the 64bit RISC-V ISA with the M and A extensions. It implements the privilege ISA 1.7 with the machine, supervisor, and user modes. It can boot Linux successfully. The design has been taped-out two times. Also, It has been integrated with the Open-Piton network on chip (NoC) [40]. In the first tape-out named preDRAC [41], the 5-stage Lagarto-Hun targets a frequency of 200 MHz in a 65nm TSMC process node. The total chip area is 3.57 mm^2. In Figure 5.1a, there are two images of the fabricated chip.

The second tape-out is named DVINO (DRAC Vector IN-Order). This tape-out integrates a new revision of the scalar 5-stage 5-stage Lagarto-hun core, which implements a frozen specification of the RISC-V ISA (2.1 integer ISA and 1.11 for the privileged ISA). This new core revision achieves a 15% higher IPC and can target a frequency of 600 MHz using the same 65 nm TSMC technology. In order to generate this high-frequency clock, the design has a PLL integrated into the chip. In addition, it integrates the Hydra vector processor unit (VPU). The version of Hydra integrated into the design is a 2-lane VPU with a vector length of 4096 bits implementing the RISC-V vector ISA 0.7.1. The area of the new design is 8.6 mm2. In Figure 5.1b, there is the image with the layout of the design.



(a) Microscope photograph of the preDRAC chip (left) and at the polySi level (right).



(b) Layout of the DVINO chip (left) and area for the different modules of the chip (right).

Figure 5.1: Layout of the different 5-stage Lagarto-Hun tape-outs.

5.1.1 Pipeline Description

5-stage Lagarto-Hun has a very simple pipeline. It has a fetch stage, a decoder stage, a read register stage, an execution stage, and a write-back/commit stage. It also integrates a one-cycle latency instruction cache and a three-cycle latency data cache. In Figure 5.2, there is a representation of the pipeline's block diagram. The pipeline stages are:

Fetch Stage is the stage where the next instruction is obtained from memory. It contains the PC and the PC generation with a BTB and BHT of 128 entries. Also, it contains the request and response interface with the instruction cache. This stage can take more than one cycle since the latency of the instruction cache is 1. However, a buffer to store the last block of data from the cache acceded (128 bits) is implemented. This buffer has 0 latency, and if the access is a hit on the buffer, the fetch stage takes only one cycle. Thus, only one in every four sequential accesses needs to take a one-cycle penalty for accessing the instruction cache.

Decoder Stage reads the 32-bit instructions received from fetch and obtains from the different fields of the instruction: the type, operands, and operations needed to be executed. 5-stage

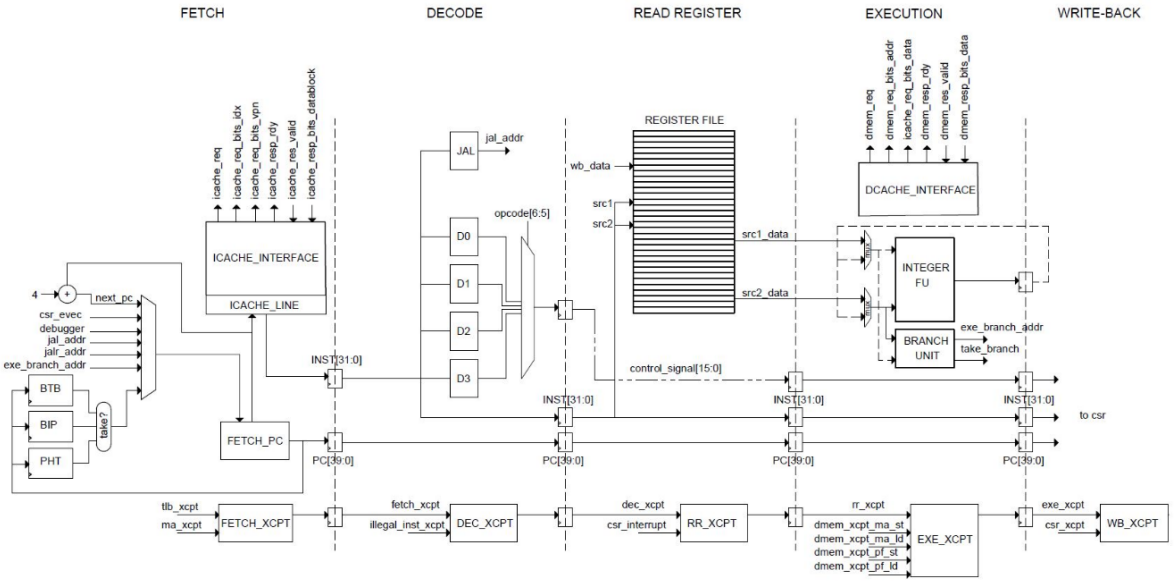


Figure 5.2: 5-stage Lagarto-Hun pipeline

Lagarto-Hun can decode the following subsets:

- RV64I base integer instruction set, version 2.1
- "M" standard extension for integer Multiplication and Division, version 2.0
- "A" standard extension for Atomic Instructions, version 2.0
- "V" standard extension for Vector Instructions, version 0.7.1
- Privileged Instruction Set, version 1.11

In addition, the Decode stage computes the immediate operand that can be codified in the instruction and resolves the Jump and Link instructions.

Read Register Stage performs two different tasks. First, the source operands are read from the integer register file. In this particular design, there are no data dependencies because all the possible bypasses are implemented. In the case needing the result of a memory operation, there will be a structural dependency because the execution stage will not be ready. Thus, it will not be a data dependency itself. Second, the Jump and Link Register(JALR) instructions are computed, and the PC is redirected in this stage.

Execution Stage performs all the operations specified by the instructions. The execution stage contains the following functional units:

- Arithmetic Logical Unit (ALU) performs all kinds of basic integer operations in one cycle
- Integer Multiplication Unit (IMU) solves 32-bit multiplications in one cycle and 64-bit multiplications in two cycles

- Integer Division Unit (IDU) solves 32-bit division in 18 cycles and 64-bit division in 34 cycles
- Branch Unit (BU) resolves branch instructions in one cycle and detects a misprediction. If a misprediction is detected, the BU sends a redirection of the PC to the fetch stage. Moreover, is the module in charge of sending the update to the branch predictors every branch.
- Load Store Unit (LSU) sends and receives the load and stores accesses to and from the L1D. It takes two latency cycles on a hit.

For those operations that take more than one cycle to execute, the pipeline is stalled until finishing.

Write-Back Stage writes the data, computed in the execution stage or coming from memory, in the integer register file if needed. The bypass network uses this same information to forward data to consumer instructions at the execution stage. Furthermore, this stage notifies the exception handler if there are any exceptions/interruptions, and it communicates with the CSRs (Control and Status Registers).

5.1.2 IPC Analysis

The CoreMark score for Lagarto Hun is 1.46 CoreMark/MHz, while similar 5-stage cores like Rocket or Riscy 5-stages achieve 2.3 and 2.21, respectively. This IPC difference is a massive performance gap, although having a maximum frequency of 600 MHz. There are mainly two points where 5-stage Lagarto-Hun can be improved to solve that lack of IPC.

The first point is the front-end part. As explained, the front end is not totally pipelined, and it can take more than one cycle if an instruction cache access is needed. There is an instruction buffer that potentially can reduce the number of cache accesses to one every four instructions. However, this only happens in perfect sequential accesses. This sequential pattern is not usually on a regular code. The percentage of control instructions can be higher than a 10% [6]. This non-sequential pattern forces new access to the instruction cache produced by a potential miss on the instruction buffer even if there is a branch prediction hit. Thus, the front-end is far from fetching one instruction per cycle on average.

The second point is the execution stage. On Rocket and Riscy 5-stage cores, there are much fewer stalls for structural hazards. This is because those pipelines do not block the execution stage when a multi-cycle instruction is executed. Each functional unit has an independent pipeline. Also, both cores implement a memory stage to mask the latency of the data cache in a hit and unlock the execution stage for other operations. These mechanisms are not implemented on the 5-stage Lagarto-Hun. Thus, after every memory, multiplication, or division operation, there are multiple stalled cycles.

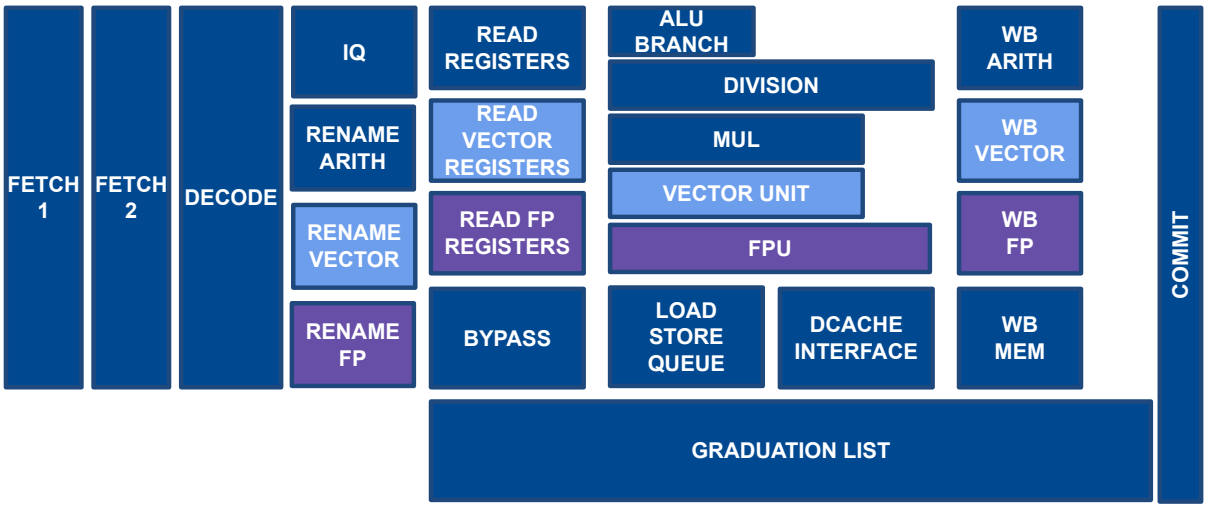


Figure 5.3: Simplified block diagram of Sargantana pipeline

5.1.3 Critical Path Analysis

The critical analysis is only made in the DVINO tape-out since it is the only tape-out that the design was pushed to the frequency limit. Also, the VPU is not taken into account since we only consider the core itself. In this tape-out, there are three main points where it can limit the frequency. The first and second points were located in the multiplication and division unit. The multiplication was a one-cycle module on the original design, and the division was 64 cycles module. The multiplication unit is now a three-cycle module to avoid the enormous critical path there. The division was improved, reducing the number of cycles. The maximum bits computed in a cycle to not become the critical path at the end are 2. Thus, half of the latency on the original design.

Apart from these two critical points, the frequency is limited by the SRAMs of the caches. In the end, this was the limiting factor in this design. To improve the frequency, the SRAMs of the cache memories were split and converted to single-ported SRAMs. These modifications need some changes on the control side inside the caches.

The analysis of 5-stage Lagarto-Hun shows very usual critical paths on in-order designs such as the SRAMs inside the caches or the multiplication unit. It is important to consider that this analysis was done on a simple core with very simple control logic and hazard detection. This analysis can not be interpolated with more complex designs.

5.2 Sargantana Pipeline

The Sargantana pipeline tries to address all the performance problems of the 5-stage Lagarto-Hun while making it more modular and suitable for adding different functional units. In Figure 5.3, there is a very simplified block diagram of the Sargantana pipeline. Sargantana is a much more complex design having 7-stages to write-back plus a commit stage. Also, it adds a floating point and a SIMD pipeline to make it more complete.

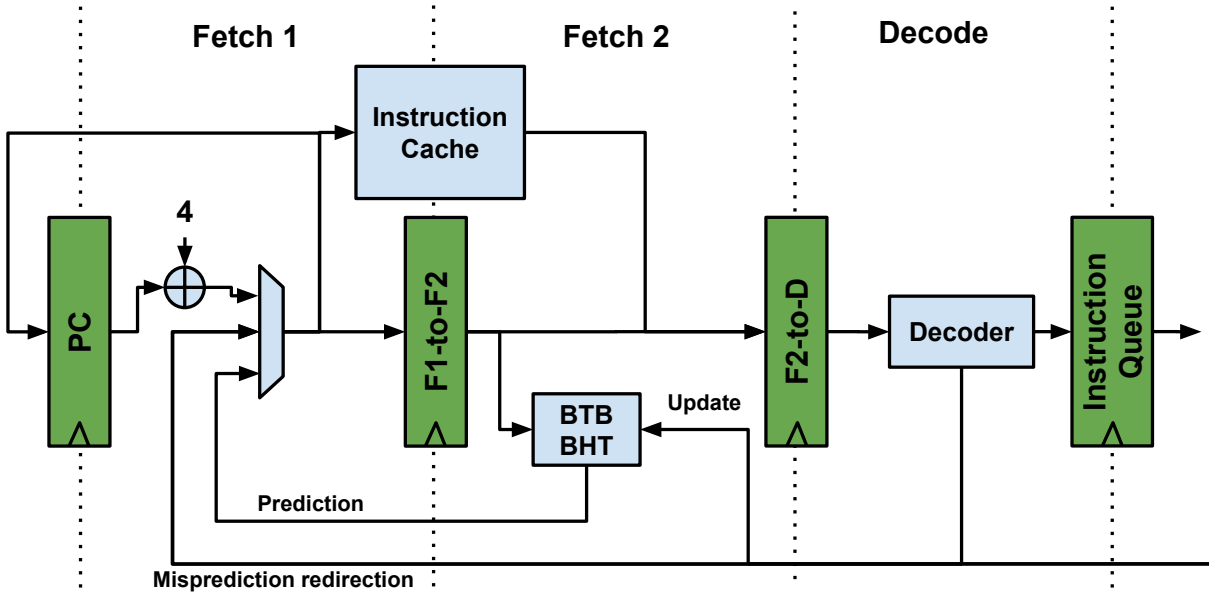


Figure 5.4: Block diagram of Sargantana front-end.

5.2.1 Frond-End Implementation

The front-end is divided into three different stages in Sargantana. There are two fetch stages and a decoding stage. An instruction queue decouples the front-end and the back-end.

The instruction cache is the same as the 5-stage Lagarto-Hun. However, the request and the response of the caches in this processor are split into two different stages. This modification allows the pipelining of the access. This modification removes the penalty of the instruction cache access in the 5-stage Lagarto-Hun, allowing the front-end to fetch one instruction per cycle if there is no miss in the instruction cache. The instruction cache only has one cycle access time since it follows a VIPT structure, removing a cycle of latency by doing the TLB access in parallel with the tag array. Then, in the Figure 5.4 there is the block diagram of the front-end. The operations done in each stage are:

- **Fetch Stage 1:** a request with the actual PC address is sent to the first-level cache.
- **Fetch Stage 2:** the response of the first level cache is received. This response can also be an instruction address or page exception. In case of having an instruction cache miss, this stage will stall until the instruction cache responds. It is worth mentioning that the Decoding logic is not in this particular stage. The reason is not to generate a critical path at this point. This stage contains access to tag array SRAMs and the tag comparison. Thus, the logic of decoding after these two costly operations will become a huge critical path. Also, the BTB and BHT predictors are accessed to predict the next PC.
- **Decode Stage:** The decoding of the particular instruction is done in this stage. Apart, JAL instruction redirects the PC since the target is already known. Finally, the resulting decoded instruction is pushed to the instruction queue.

5.2.2 Back-End Implementation

In the Sargantana core, the complexity of the back-end increases substantially compared with the 5-stage Lagarto-Hun. Now, the back end is capable of doing out-of-order write-back. As it has been explained, Sargantana is not intended to be an out-of-order core because of the complexity and the embedded target. However, adding the ability to do only out-of-order write-backs does not increase the complexity and the resources as the out-of-order issue does. In addition, out-of-order write-back has the increment on performance as one of the main advantages. Mainly, out-of-order write-back enables the ability to execute short-latency operation after a very long latency operation without waiting for the longest to finish and thus increasing the final performance.

The other effect of that out-of-order write-back is the higher modularity of the core. With this feature, the control logic of the issue stages decreases notoriously because the logic does not need to check if the current operation will finish before an old instruction on the fly. Thus, adding new functional units of different latency is easier without modifying the issue logic.

Another new feature that helps, in the same way, is the register renaming. Register renaming is a feature that is consistently implemented on out-of-order designs. Also, it has beneficial effects in some in-order designs like Sargantana. In the end, register renaming removes the WAW dependencies (and WAR dependencies on cores that the access to the register file does not follow the program order). Then, register renaming has a very similar effect to the out-of-order write-back.

Thanks to these two new features, Sargantana integrates a SIMD pipeline, a floating point pipeline, a more complex memory pipeline with multiple in-fly accesses, a new pipelined multiplication unit, and a new parallel computing division unit. In Figure 5.5, there is a detailed block diagram of the back-end of the processor.

5.2.2.1 Renaming and Register Files

Sargantana needs three different Register files for the different types of operands: integer, floating point, and SIMD. The register files have 64 registers each to implement the register renaming mechanism (there are 32 architectural registers). The integer and floating point registers are 64bit since the design implements the RV64 ISA with the D extension. The SIMD register size is set to 128bit. The integer register file has two ports for reading the two possible integer operands. The floating point register file has three different reading ports since the floating point ISA extensions require them. Finally, the SIMD register file has four different outputs. In this case, it has two ports for the operand, one port for the old value of the destination register needed for the mask operations, and the mask itself located on the register *v0*.

In this processor, the register renaming will be checkpointed to make a fast one-cycle recovery when an exception or a misprediction is detected. In the Sargantana design, there are three identical register renaming mechanisms for the three different register files. The register renaming mechanism is based on two blocks: the rename table and the free list.

The rename table is a structure that contains the mapping between the architectural registers (32 ISA visible registers) to the physical registers (64 architecture visible registers in Sargantana design). This table has 32 entries, one for each architectural register. Each entry contains the

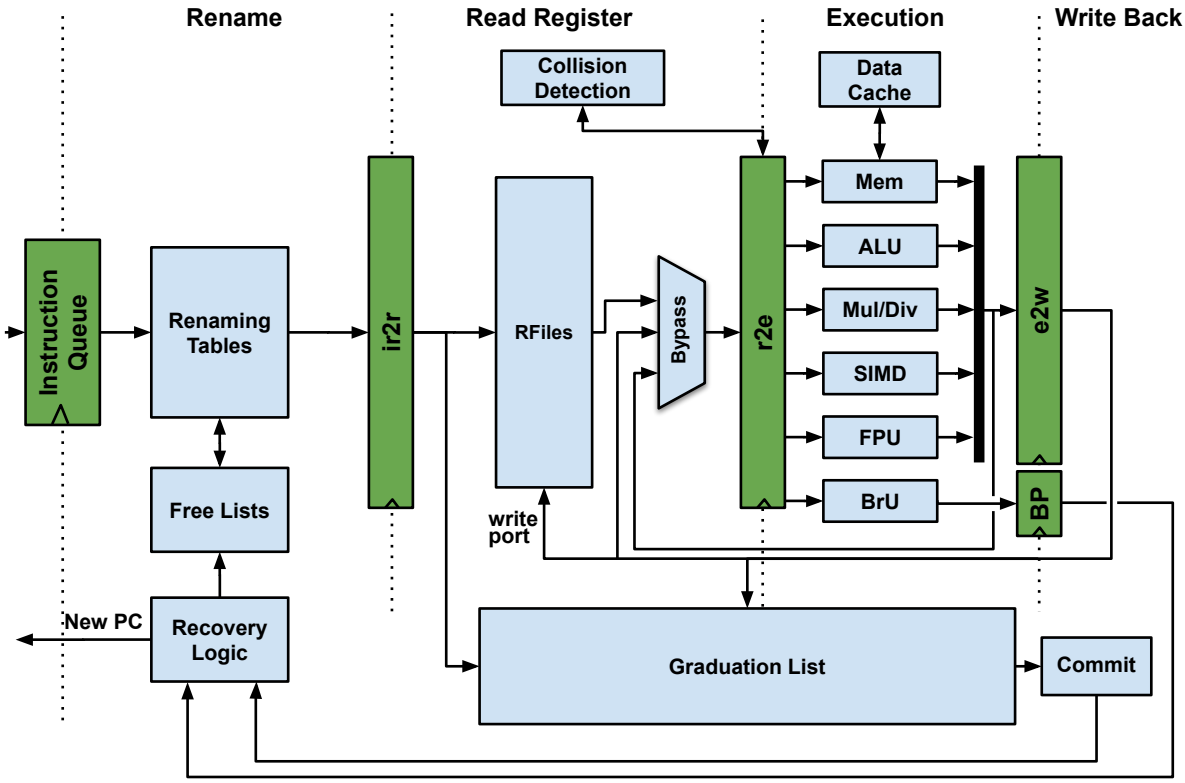


Figure 5.5: Block diagram of Sargantana back-end.

actual physical register that contains the updated value. Additionally, for each entry, there is a ready bit that indicates whether the value of the register is in the register file or must be caught on the fly using the bypass logic.

The rename mechanism is updated on each cycle. Each time the core renames a new instruction checks where the architectural registers are mapped. If this instruction produces a result that must be written on the register file, it writes a new mapping for that particular destination register. Also, three instructions can write back each cycle, setting the destination register's ready bit to true. This means that the data produced by that instruction can be found in the register file.

The instructions in Sargantana are executed speculatively. A roll-back mechanism is needed to avoid the corruption of the structure. One option is to recover all the speculation translations done by taking the values from the reorder buffer. However, this type of solution needs many cycles to recover since the instructions on the reorder buffer have to be processed. Moreover, only one instruction per cycle can be processed. Therefore, a checkpoint mechanism is implemented in which each time a branch instruction is encountered, a checkpoint is done.

Additionally, we will have an extra copy of the mapping table that keeps the correct mapping of the already committed instructions. In case of exception, this commit table is copied to the mapping table. In case of a branch misprediction, the copy done when the branch instruction was renamed is recovered.

In Figure 5.6 there is the renaming stage diagram with N speculation checkpoints. The

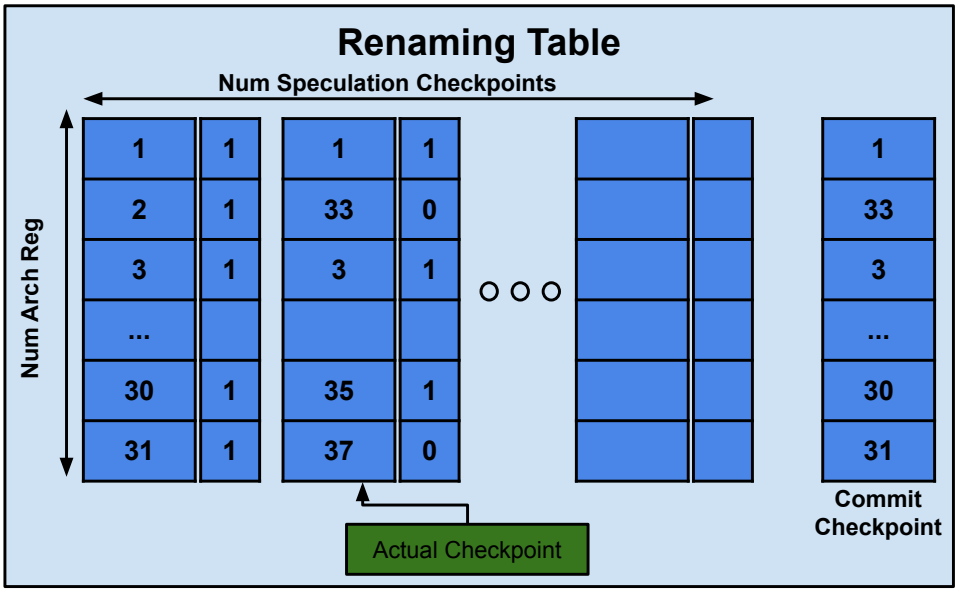


Figure 5.6: Block diagram of the renaming table.

actual checkpoint is a pointer to the version of the rename table that is used in this cycle. When a new control instruction comes to the rename stage, the *actual checkpoint* is incremented, and the new checkpoint pointer is associated with the control instruction. Also, the old checkpoint mapping is copied to the new checkpoint table. When a control instruction is mispredicted, the actual checkpoint is set to the one associated with that particular instruction.

The second module in the renaming mechanism is the free list. A new free physical register is needed for every new instruction with a destination register to be renamed. Those free registers are the ones that are neither mapped on the commit checkpoint or speculatively checkpoint. Since checking all the different mappings each cycle to detect which physical registers are free is very expensive, the free list is used. The free list is a FIFO-like structure that contains the physical registers that are free to use for a new instruction as a destination register. This structure also needs to be able to recover the state in case of a misprediction or an exception. For doing that, it also implements checkpoints like the rename table. In Figure 5.7, there is a diagram of the free list implemented on Sargantana. The functionality is straightforward:

- **Reset state:** all the free registers are in the list (32:63). The dequeue pointer checkpoints, the dequeue pointer (deqP), and enqueue pointer (enqP) are set to 0.
- **Rename a non-control instruction:** If the instruction needs a destination register, the free list returns the value in the deqP, and the deqP is incremented.
- **Rename a control instruction:** If the instruction needs a destination register (only JALR will need that), the free list returns the value in the deqP, and the deqP is incremented. Then the new deqP is stored at a new checkpoint. The value of the checkpoint is returned to the control instruction.
- **Commit of an instruction:** when an instruction is committed, the instructions send the old physical register. This register is set on the enqP, and the enqP is incremented

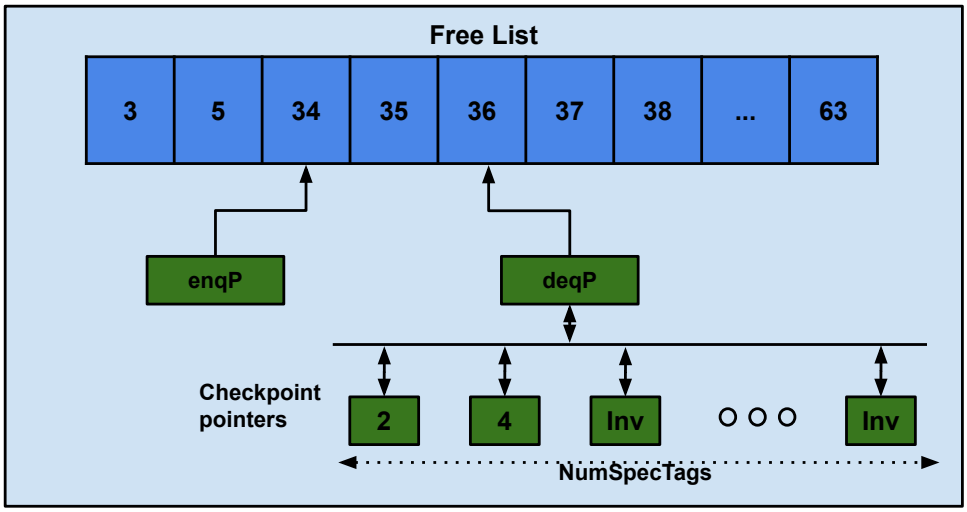


Figure 5.7: Block diagram of the free list.

- **Recover mechanism of a misprediction:** the instruction sends the dequeue pointer checkpoint to recover. The dequeue checkpoint is copied to the deqP
- **Recover mechanism of an exception:** The deqP is set to the enqP

5.2.2.2 Issue, Write-Back and Forwarding Mechanisms

Although Sargantana has an out-of-order write-back, not all the structural hazards can be eliminated in this design. As Sargantana has many functional units, the number of writing ports on the integer register file is significant. All the different functional units can potentially write this register file, even the FPU and SIMD units. To reduce the number of read ports, Some FUs are sharing it. In this design, the ALU and the Mul/Div unit share one port. The FPU and SIMD also share a port. Moreover, the memory pipeline has a port of its own. The ALU and Mul/Div unit share a port because the latency on these modules are fixed, and it is easy to detect collisions. The FPU and SIMD unit can share a port since the FPU has an output queue, and the results can be dequeued when the SIMD is not outputting any scalar result. The memory pipeline has its port since it has variable latency.

The logic used for detecting collisions on the issue of the ALU and Mul/Div operations is shown in Figure 5.8. The mechanism is a shift register. The register size equals the longest instruction latency (in this case, 64bit division, 38 cycle latency). When a new instruction is going to issue, put a 1 in the bit that equals the instruction latency minus one. For example, a 64-bit multiplication instruction will put a 1 in the second bit since the latency is three cycles. This shift register shifts every cycle. Thus, in the last example, the one put by the multiplication instruction will be removed to the register in three cycles. The only thing needed to check if there is a collision is to check if there is a one in the bit related to the latency of the new instruction. If there is a 1, the new instruction will write back at the same time as an older instruction and must not be issued.

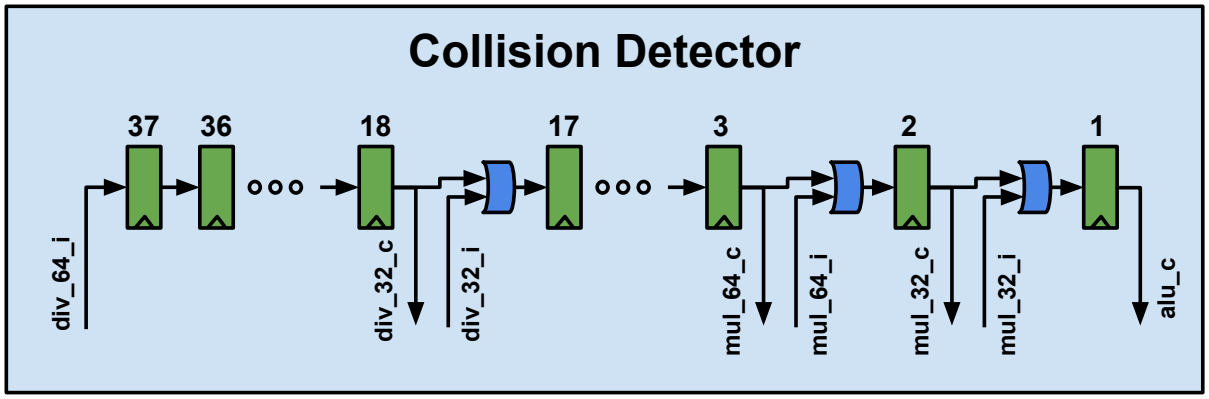


Figure 5.8: Write-back collision detection in the ALU and Mul/Div pipeline. The suffix *_i* indicates a new issue of the particular type of instruction. The suffix *_c* indicates a collision of the particular type of instruction.

5.2.2.3 Graduation List and Commit Stage

In order to enable the out-of-order write-back, the core needs a reordering structure to commit the instructions in order. This ordering module used to be the Re-Order Buffer (ROB). However, in the Sargantana design, we have implemented register renaming, and we can write the speculative results to the register file. Thus, the ROB does not need to keep the results. This type of ROB without speculated results is named graduation list.

The implementation of the graduation list in Sargantana is a circular buffer with two pointers to keep track of the first and the last instruction valid inside the list. Each time an instruction is introduced on the graduation list, it returns an instruction ID. This ID is used to mark instructions that have finished. Since there can be up to four instructions finishing at a given cycle, one for each functional unit, the graduation list has four ports to mark as finished the instructions.

When there is an exception at the commit pointer (the dequeue pointer), the entire graduation list is flushed, setting the enqueue pointer to the commit pointer. When there is a branch misprediction, the instruction that produces the misprediction sends the ID to the graduation list. The enqueue pointer is set to that instruction on the graduation list. Thus, all the younger instructions are flushed.

5.2.2.4 ALU, FPU, Multiplication Unit and Division Unit

Sargantana has several functional units. This subsection describes the implementation of the ALU, FPU, multiplication unit, and division unit. The ALU on the Sargantana is a one-cycle simple design. It is based on parallel computing of all the operations. Then, the result is selected with the instruction operation using a multiplexer.

The multiplication unit is a pipeline unit optimized for achieving high-frequency and high-throughput. For this reason, the multiplication unit takes three cycles for a 64bit multiplication (2 cycles for a 32bit multiplication). In the first cycle, the sign is modified for a signed operation, and the operands are negative. In the second cycle, two 64 by 32-bit multiplications are

performed. Also, the result is inverted in the case of a 32bit signed operation and if one of the operands is negative. Furthermore, the 32bit result is outputted. In the third cycle, the sum of the two intermediate results generates a 64 by 64-bit multiplication. Also, the result is inverted in the case of a 64bit signed operation, and one of the operands is negative.

The division unit is optimized in terms of consumed resources. Making a high throughput division unit capable of computing one division per cycle does not make sense for the few divisions used on the most common programs. In Sargantana there are two non pipelined division modules that take 38 cycles for a 64bit division using the long division algorithm. This version of the divisor can do two steps of the algorithm every cycle. It is not possible to do more steps per cycle because the critical path becomes too long. There is an arbiter to select which divider will be used for each issued division instruction. If the two dividers are busy, a new division instruction will be blocked.

Finally, the floating point unit used in this design is provided and maintained by the University of Zagreb (initially based in ETHZ's Ariane FPU). The FPU is a single 64-bit unit partitioned to operate on two independent 32-bit values when needed. All the operations are fully pipelined except the division and square root. Latency is defined here:

- Non-pipelined operations (div/sqrt):
  - 64bit: maximum latency is 25 and minimum is 6.
  - 32bit: maximum latency is 15 and minimum is 6.
- Pipelined operations (other): Latency is always 5 cycles.

Maximum latency for non-pipelined operations is for each non-special case, while minimum latency is for special cases: zero as an operand, infinity as an operand, and sqrt of a negative number.

5.2.2.5 Memory Pipeline

The memory pipeline is the most complex pipeline in the design. The cache hierarchy used in this design is taken from the lowRISC project [42]. In Figure 5.9, there is a block diagram of the lowRISC data cache.

This cache implements a MESI coherence protocol with the second-level cache over TileLink [43] communication protocol. This cache is VIPT, it translates the virtual page address in parallel with the access of the tag and data arrays. This cache is non-blocking. It uses a bank of Miss Status Holding Registers (MSHR) with a store buffer to implement the non-blocking feature.

The latency of a hit memory operation depends on the type of access. A hit on memory access in this cache has four different stages for a store and three stages for a load. The stages are the following:

- **Stage 1:** In this stage, the request from the core is received. Here, the modules of metadata and data are accessed. The TLB also accessed.

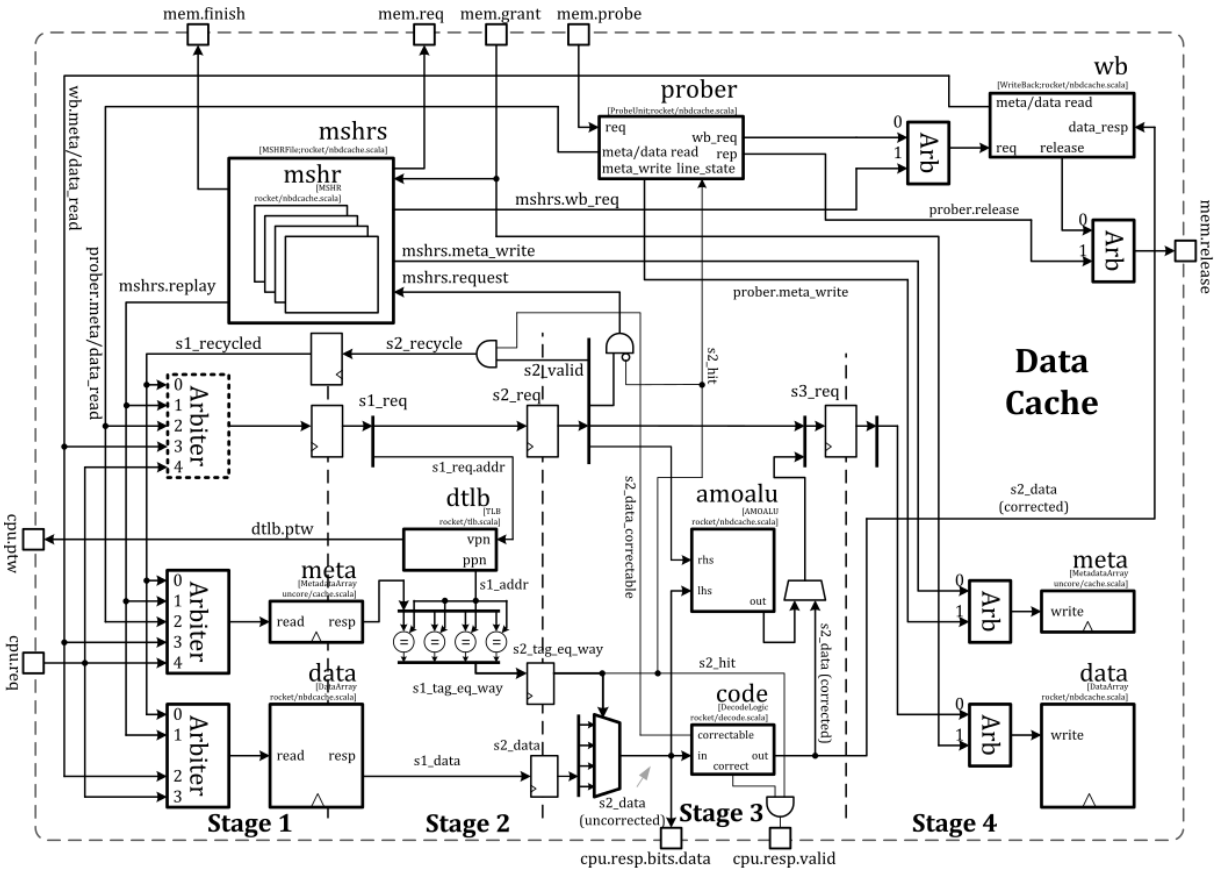


Figure 5.9: Block diagram of the first level data cache. Source: [4]

- **Stage 2:** Here, the responses of the arrays and the TLB are recieved and all the tags of the different ways are compared with the access address.
- **Stage 3:** If a miss occurs, the MSHR File receives a new request. If there is a hit and the access is a load, the cache makes a response to the core with the valid data. If it is a store, the data continues to the next stage through an ALU that implements the atomic operations.
- **Stage 4:** Finally, in this stage, the data and the metadata are written on the banks.

The configuration of the data cache used on this design is a 16 entry dTLB, 4-way associativity, 64 sets of 512-bit size, and 2 MSHR. It is worth mentioning that the internal data arrays are split to reduce the size to be implemented with high-performance single-ported SRAMs for high-frequencies.

Sargantana is able to send multiple memory operations at the data cache to take profit from the non-blocking cache. To do so, the memory pipeline integrates two buffers, the load/store queue (LSQ) and the pending memory request queue (PMRQ). In Figure 5.10, there is a block diagram of the memory pipeline. In the first stage, the access address is computed, and the instruction is enqueued on the LSQ. In the second stage, if the data cache can accept a new request (it can be blocked if the MSHR is full or if the instruction is a non-committed store), an instruction from the LSQ is dequeued, and the data cache is accessed. Since the latency on a

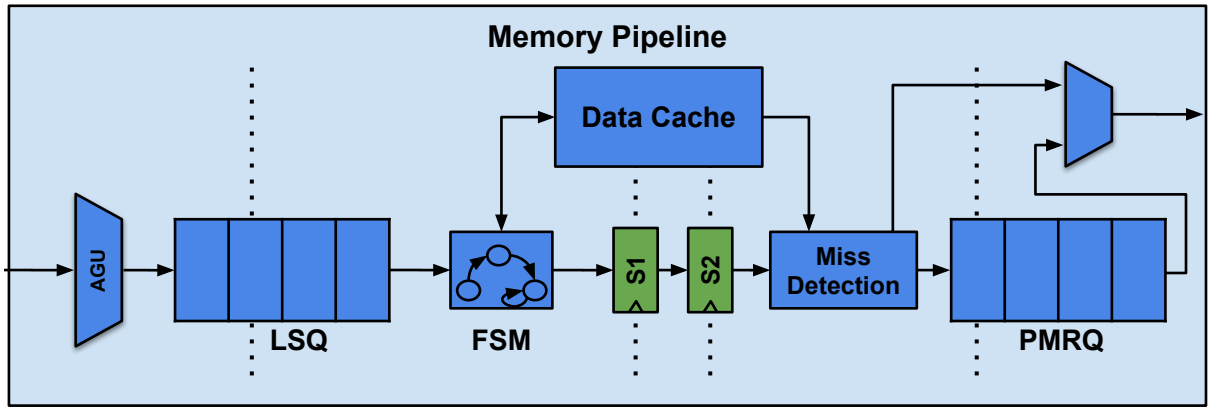


Figure 5.10: Block diagram of the memory pipeline.

data cache hit is three cycles, the instruction information is pipelined parallel to the cache access on the third cycle. In the fourth cycle, the response of the cache is received in case of a hit. Then, the data is sent to the pipeline. If the access is a cache miss, the instruction information is pushed to the PMRQ, waiting for the result. When the missed result comes from the data cache, the result is merged with the instruction information in the PMRQ thanks to a tag ID that relates the cache access and the instruction information. In the next cycle, the instruction is dequeued of the PMRQ, and it is returned to the core pipeline to write back.

Load instructions can be issued to the data cache at the moment when the operands are ready. However, this is not true with the store and atomic operations (AMO) operations. In those cases, the memory pipeline must ensure that the store accesses do not speculatively access the data cache. Imagine that a store accesses the cache and writes the data speculatively, and after that, an older instruction raises an exception, and the store must be flushed. Since the value is already written to the cache, the execution will not be consistent. To protect the execution, the Sargantana memory pipeline implementation can not dequeue a store instruction from the LSQ until the store instruction achieves the commit pointer on the graduation list. At that point, the store is no longer speculative. To do that, the memory pipeline communicates with the graduation list to detect if the particular store instruction is in the commit point using the graduation list tags.

5.2.2.6 SIMD Pipeline

In order to improve the performance of the core in specific workloads, Sargantana implements an elementary SIMD unit. In this case, the unit has 128bit to maximize the data cache bandwidth, which can deliver a maximum data packet of 128bits.

Sargantana's SIMD unit is a module designed to run a smaller sub-set of the RISC-V Vector extension ISA. This set of instructions include basic arithmetic operations, shifts, comparisons, and vector memory operations. The length of the vector registers is fixed at 128 bits, and they can be configured to contain 8, 16, 32, or 64 bit-wide elements, using the *vsetvl* instruction. The module also offers support for masked operations, with the mask value permanently residing in vector register v0.

The SIMD unit only supports 128 bit unit stride memory operations. The effective address of the operation must also be aligned to 128 bits. The alignment limitation ensures that only a single memory request must be issued per memory operation.

The computation of a vector instruction gets split into two Functional Units. These units contain a 64-bit wide ALU, each capable of performing operations interpreting the 64-bit source operands as eight 8 bit wide elements, four 16 bit wide elements, two 32 bit wide elements, or a single 64-bit wide element. The functional units only take one cycle to compute the result.

In the case of an instruction that uses a scalar or immediate operand, the value is replicated to build a 128-bit wide operand to feed into the FUs. The outputs of the FUs are concatenated, and if the operation is masked, each element of the result is chosen to be the old physical destination registers contents or the result produced by the FUs.

At this moment, this SIMD unit implements a custom instruction for accelerating the Wavefront algorithm [44], a sequence alignment algorithm for genomics. The name of the instruction is "Vector count equals" - vcnt. This instruction counts, starting at element 0, how many elements in a row are equal between two vector registers, and saves the result into a scalar register. The pseudo-code 5.1 shows a high-level implementation of the instruction functionality (In the example, SEW=8).

```
void vcnt(int8_t* vs2, int8_t* vs1, int32_t* rs1) {
  int i, acc = 0;
  for (i=0;i<16;++i) {
    if (vs1[i] != vs2[i]) break;
    ++acc;
  }
  *rs1 = acc;
}
```

Listing 5.1: Pseudo-code of the vcnt instruction behavior with a element size of 8 bits

5.2.3 Verification and Performance Evaluation Infrastructure

A crucial part of processor design is the verification and performance evaluation strategy. In this project, there are several levels of verification and performance evaluations.

5.2.3.1 RTL Simulation Environment: Waveforms, Commit Traces and Konata

The first level of design validation is the RTL simulation. The tool used for RTL simulation is Verilator since it gives us the versatility to make new simulation features in C and incorporate it into the test bench in a simple way. Verilator is capable of generating waveforms of all the wires and registers of the design in a *vcd* format for debugging purposes.

Taking advantage of the versatility of Verilator, we have implemented a Direct Programming Interface (DPI) in C to generate a commit trace of the core simulation. This trace has the PC of the instruction committed by the core, along with the instruction's encoding, the destination register, and the result value. With this information, it is possible to compare this commit trace with the commit trace of an ISA simulator-like spike [45]. With this trace comparison, it is

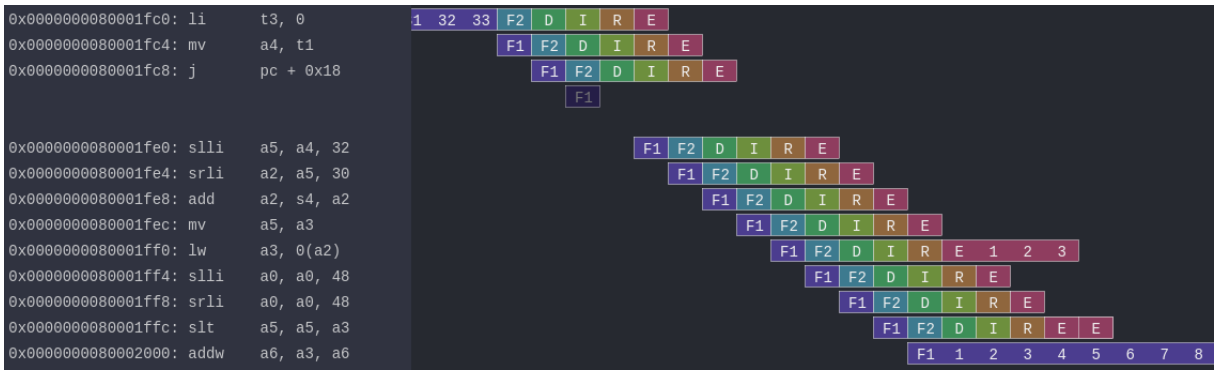


Figure 5.11: Konata visualization of a Sargantana execution.

possible to verify very large programs instruction per instruction. It is also possible to verify random verification tests from random generators like riscv-dv [46] or torture-tests [47].

Another feature implemented on this design is the use of the Konata pipeline visualizer [48]. Konata is a tool that enables the graphic visualization of the pipeline execution on each stage over the cycles. It is beneficial for performance validation purposes. In order to implement this feature, the only modifications done on the design is the integration of a new ID field on the Instruction information propagated on the entire pipeline. The ID of each instruction inside the pipeline is collected on a single module. This module uses a custom DPI of Sargantana to generate a trace file with Konata's format to be processed adequately by the tool.

In Figure 5.11, there is an example of visualization of a Sargantana execution using Konata. The figure shows how the first and last instructions are an instruction cache miss since they take more than one cycle to execute the first fetch stage. Also, the jump instruction on the third position produces a branch misprediction since the pipeline is flushed. Thus, two execution cycles are lost. Finally, it is possible to see that the load (8th instruction) takes four cycles of latency. This extra latency produces that the shift left (11th instruction) needs to wait one cycle in the execution stage since there is a data dependency. Finally, in order to facilitate the performance analysis, Sargantana implements a Performance Monitoring Unit (PMU) with different performance counters of architectural events, such as different pipeline stalls, branch mispredictions, and instruction and data cache metrics.

5.2.3.2 FPGA Environment and Linux Boot

RTL simulation is slow. In order to run more extensive benchmarks and kernels, the Sargantana design has been ported to an FPGA implementation. The FPGA used is a Kintex KC705 from Xilinx. This board contains 1 GByte of high-speed memory (DDR3). The design can run at 50 MHz on this FPGA. This frequency is high enough to run big kernels in a reasonable execution time. In the end, FPGA emulation is more than 100X faster than RTL simulation on Verilator.

A Linux distribution has been compiled for the Sargantana core to prove the functionality of the core (a part of the extensive RTL verification). A relatively new version of the Linux kernel has been used., in particular, version 5.8 [49]. Builtroot [50] tool is used to make more accessible the acquisition process of the Linux image.

Sargantana has a first boot loader located on a boot ram inside the FPGA. This boot ram acts as a Read-Only Memory (ROM) programmed when the bit-stream is generated. The Linux image and the second boot loader are located on the SD card attached with a custom Serial Peripheral Interface (SPI) controller in the SoC. The first boot loader copies all the SD card data to the DDR3 memory on the FPGA and then jumps to this address space to start executing the second boot loader.

The OpenSBI [51] tool is used as a second boot loader since it simplifies a lot the driver configuration. OpenSBI implements a pseudo machine hypervisor that takes care of all the timer, console, and interruption routines. Then, we have modified the drivers inside the OpenSBI code, which are much simpler than the drivers inside Linux, to enable the console terminal of Linux using a UART interface.

5.2.3.3 Physical Design and Gate-Level Simulations

The design is ASIC-ready, either using the 65nm TSMC or the 22nm FD-SOI Globalfoundries technology node. The synthesis flow has been made using the *Genus* tool from Cadence. The 22nm FD-SOI Globalfoundries technology libraries used are enumerated in Secton 4.2.1 and the SRAMs for the caches on Secton 4.2.2.

Furthermore, a gate-level simulation environment has been developed to verify the physical design flow. The gate-level simulation has been done with the Xcelium program, also from Cadence. Gate-level simulation verifies that the design still works as expected after the synthesis or the place and route. It checks the timing violations, the glitches, and the X propagation thanks to the technology libraries models.

6 Evaluation

In order to evaluate the design, the EEMBC AutoBench and CoreMark are used to obtain the IPC performance. However, not only the Sargantana design has been evaluated. Also, the 5-stage Lagarto-Hun, Rocket, Ariane (and the new version, CVA6), 5-stage and 7-stage Riscy have been evaluated with identical benchmark suites for comparison proposes. Sargantana also has been evaluated in terms of maximum frequency achievable in 22nm FD-SOI technology.

6.1 Synthesis Experiments of Sargantana

Sargantana has successfully synthesized with 22nm FD-SOI GlobalFoundries technology. With this technology and using the libraries mentioned in Secton 4.2.1 and the SRAMs for the caches described in Secton 4.2.2, the core achieves the frequencies shown in Table 6.1. Also, the post-synthesis frequencies for the 5-stage Lagart-Hun, 5-stage Riscy and 7-stage Riscy cores are shown. These are the only cores that we can directly compare because both use the same synthesis scripts and conditions. As we can see, Sargantana achieves less frequency than its predecessor or the 7-stages Riscy because the limiting factor is the FPU module in this design. Thus, we also include the synthesis results of a Sargantana core without the FPU to show the maximum potential of the core. Without the FPU, Sargantana is the core that achieves the highest frequency.

Doing a deeper analysis of the critical paths of the design, the synthesis reports show that the 100 first critical paths in the design are internal of the FPU. The critical path is difficult to solve since the design is from another institution and it relies on aggressive retiming optimizations done by the synthesis tools that are not under our control. Without the FPU, the reports show a critical path on the front-end. In this case, the path goes from the tag array SRAMs through the hit control logic (tags comparisons), arriving at the fetch stage. Inside the fetch stage, the path affects the instruction request logic and is propagated to the instruction TLB. This critical path can be solved by increasing the number of fetch stages.

Apart from the frequency results, the synthesis reports can help us make an area estimation of the design. The estimated area post-synthesis of the core with the two first-level caches is 0.5 mm^2, while the area of the 5-stage Lagarto-Hun is 0.2 mm^2. Figure 6.1 contains the area distribution inside the Sargantana core (first-level caches included). The biggest part of the pipeline is the execution stage. The execution stage incorporates very large modules. It includes the FPU, the SIMD unit, and the memory pipeline (including large memories for the LSQ and PMRQ).

Table 6.1: Maximum frequency for the 5-stage Lagarto-hun, Riscy cores and Sargantana in the slow corner (0.72V@125ºC), the typical corner (0.8V@25ºC) and the fast corner (0.88@-40ºC)

	Slow Corner (GHz)	Typical Corner (GHz)	Fast Corner (GHz)
5-stage Lagarto-Hun	1,05	1,46	1,69
7-stage Sargantana	1,03	1,39	1,66
7-stage Sargantana no FPU	1,08	1,58	1,88
5-stage Riscy	0,73	0,99	1,36
7-stage Riscy	1,06	1,44	1,67

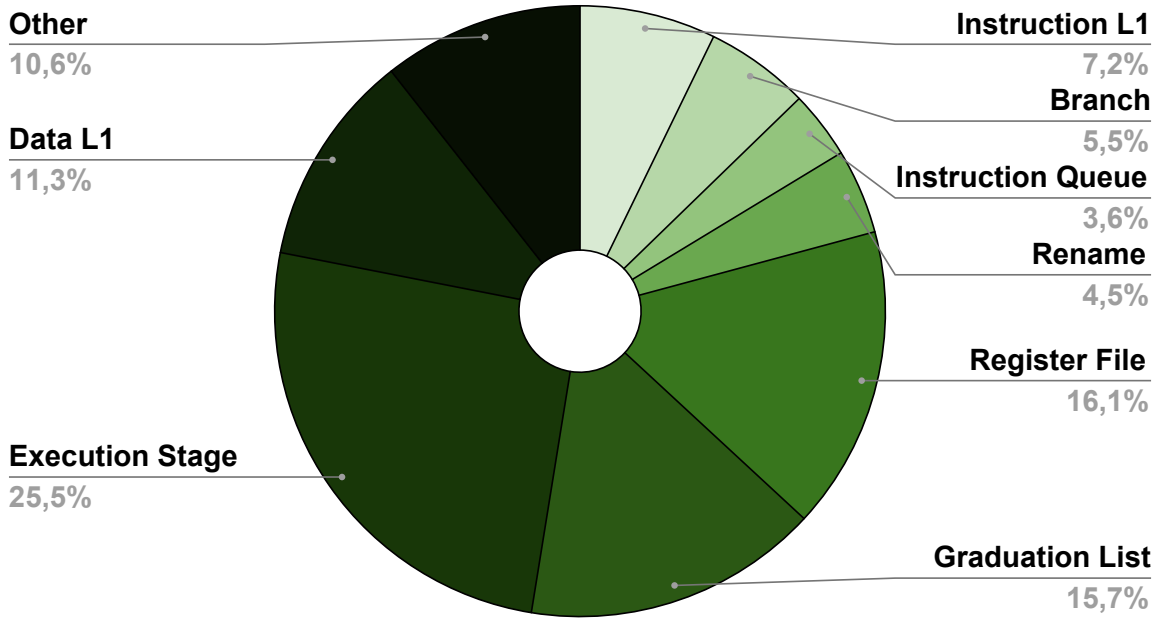


Figure 6.1: Area distribution inside the Sargantana core (first-level caches included)

The second biggest part is the Register Files. Considering the integer, floating point, and vector register in this processor, the total memory size is 16Kbits. Also, the register files have multiple reading and writing ports. The third biggest module is the graduation list. The graduation list also needs a lot of memory per element, and it has 64 entries. However, this module can be further optimized because it has some fields like the exception code or the exception information that can be removed or reduced to only store the oldest instruction. We leave it as future work since it does not have any impact on the performance.

6.2 IPC Evaluation Using the EEMBC AutoBench

In this section, the IPC performance of Sargantana is evaluated using the EEMBC AutoBench benchmark suite. First, it is compared with the Sargantana predecessor, the 5-stage Lagarto-Hun. Then, the Sargantana core is compared with other academic in-order processors. Finally,

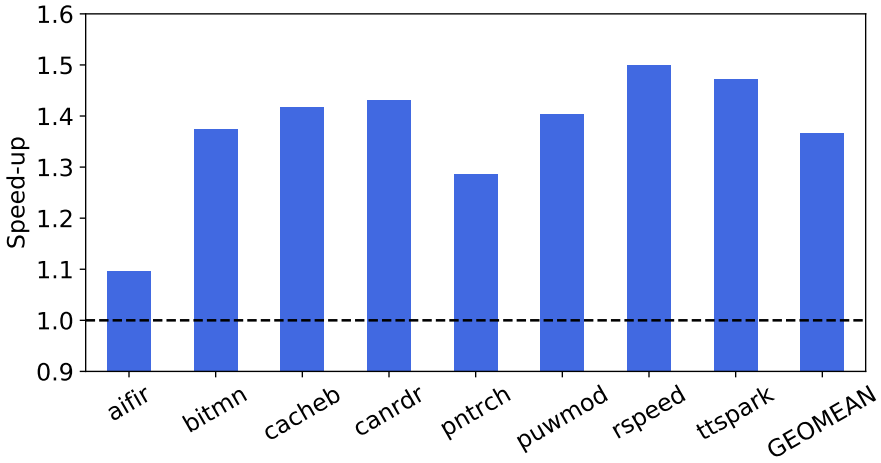


Figure 6.2: Speed-up of the integer benchmarks of the AutoBench suite of Sargantana against the 5-stage Lagarto-Hun. The geometric mean of all the benchmarks is shown on the right bar.

there is a discussion on some particular bottlenecks of the Sargantana design in terms of IPC.

6.2.1 IPC Evaluation Against the 5-stage Lagarto-Hun

As it can be seen in Figure 6.2, Sargantana core outperforms the IPC performance of its predecessor, the 5-stage Lagarto-Hun. Sargantana achieves an average IPC of 0,595 while the 5-stage Lagarto-Hun achieves 0,436. Thus, Sargantana has an average speed-up in terms of IPC of 1,37X. This increment of IPC comes mainly from two particular points. In comparison with the 5-stage Lagarto-Hun front-end, the new front end is capable of fetching one instruction per cycle from the instruction cache. The second big update is the out-of-order write-back scheme with the non-blocking memory pipeline. The speed-up is more or less equal in all the benchmarks except the aifir benchmark because it has many branch mispredictions that negatively impact the 7-stage pipeline of Sargantana. Since the 5-stage Lagarto-Hun does not implement an FPU, we do not include benchmarks with floating-point operations in this performance comparison.

6.2.2 IPC Evaluation Against Other Designs

In Figure 6.3, there is the IPC of each core for each integer EEMBC Autobench benchmark. There is the geometric mean of all the IPCs of each core at the right. In Figure 6.4, there is the exact figure using only the floating point benchmarks. It is divided into these two figures since the floating point benchmarks are not representative to compare the pipeline structure since the impact of the different FPU is vast.

The IPC comparison on the integer benchmarks shows that Sargantana only beats the 6-stage Ariane by 1.19X apart from the 5-stage Lagarto-Hun core. The two 5-stage cores, 5-stage Riscy and Rocket, have a higher IPC than the 6-stage and 7-stage pipelines. The reason why this is happening is the extra branch misprediction penalty. However, these extra pipeline stages decrease the critical paths and allow the core to run at a higher frequency. Rocket has a higher IPC than 5-stage Riscy because it has a non-blocking memory pipeline.

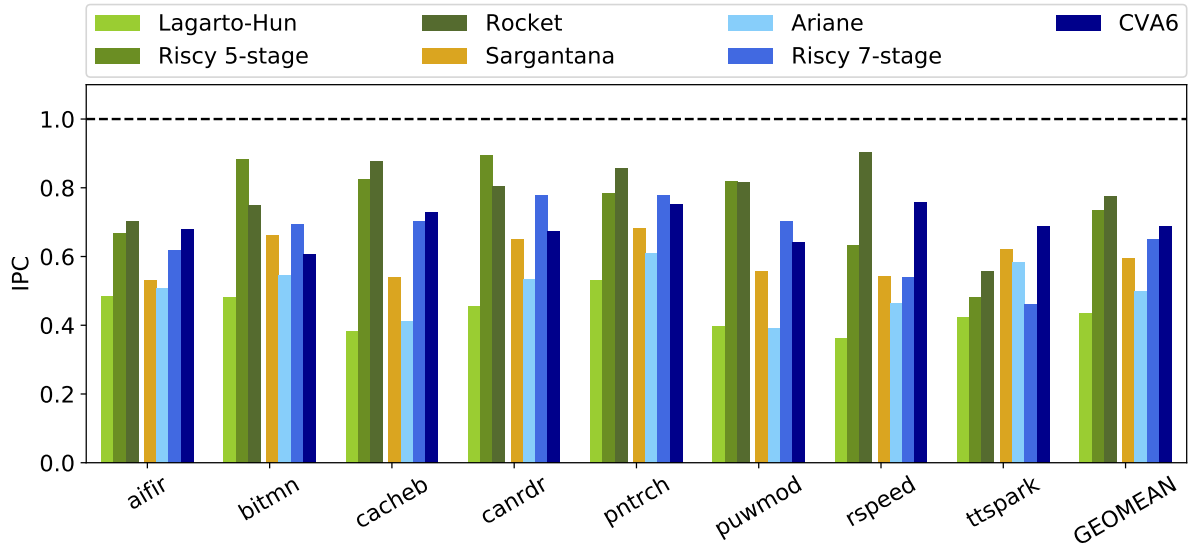


Figure 6.3: IPC of the integer benchmarks of the AutoBench suite for different cores grouped by the number of stages. The geometric mean of all the benchmarks is shown on the right bar.

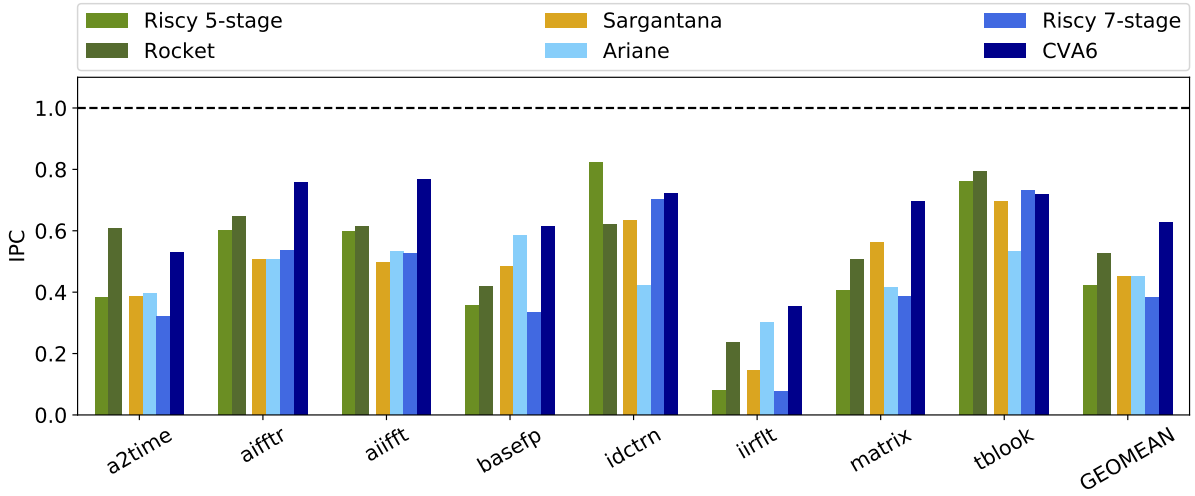


Figure 6.4: IPC of the floating point benchmarks of the AutoBench suite for different cores grouped by the number of stages. The geometric mean of all the benchmarks is shown on the right bar.

6.2.3 Deeper IPC Analysis of Sargantana's IPC Penalties

The comparison between Sargantana and the other 6-stage and 7-stage cores shows that Sargantana can still improve its IPC. In this section, the two main issues are discussed.

The first reason why the Sargantana core has less IPC than the other state-of-the-art cores is that the stores are blocked on the LSQ, waiting until the instruction reaches the commit pointer on the graduation list. These blocked stores are not only blocking other stores (since also they need to wait until the instruction is on the commit point). These stores are also blocking load operations since the LSQ is a FIFO structure. Other instructions usually use these load operation results as operands. This means that the blocked stores are generating

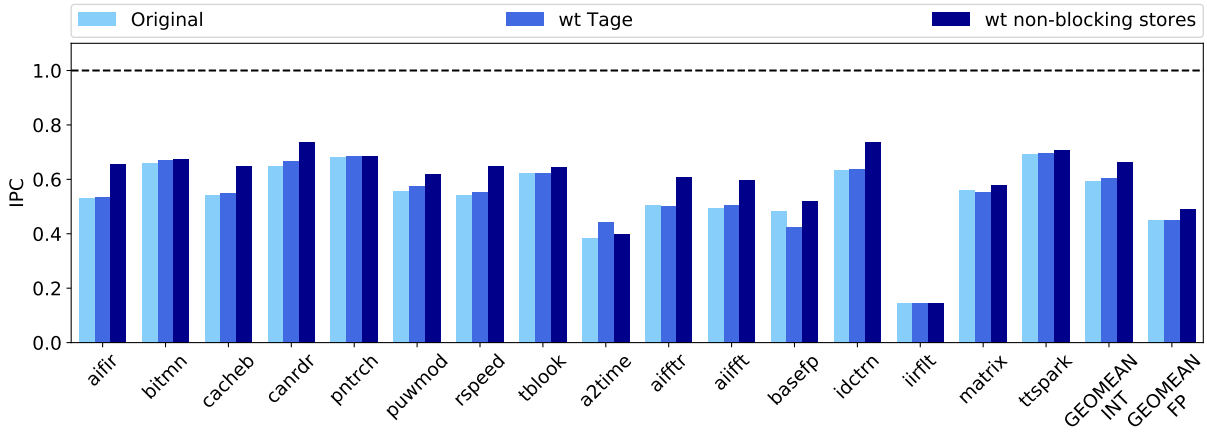


Figure 6.5: Speed up and IPC of the AutoBench suite for the different possible modifications on Sargantana.

potential data hazards on other instructions.

To analyze that specific case, Sargantana integrates a performance counter. It counts the number of cycles that a store blocks the memory pipeline with a load that potentially can be issued to the data cache. The results obtained show that the number of cycles that a store blocks a load can be up to 19% of the total execution cycles depending on the AutoBench benchmark selected.

In order to validate the results, a new version of the Sargantana memory pipeline has been implemented. The new version does not wait for the stores to be committed to sending the memory petition to the data cache. This modification is not compliant with the RISC-V ISA. However, for baremetal tests without exceptions, this solution will execute without errors and can estimate the performance of having a proper store buffer. The results of executing the AutoBench suite with this new modification are shown in Figure 6.5. Now, with the cache modification, the core is up to 1.23X faster in some benchmarks. The geometric mean speed-up is 1.11X. These simulations show that Sargantana could be faster than the 7-stage Riscy and closer to the CVA6 if a proper store buffer is implemented.

The second penalty on this design is the branch mispredictions. The misprediction penalty is five cycles since the redirection is done in the write back stage for critical path reasons. The branch predictors used in this design are very simple. These simple predictors produce a high number of mispredictions translated into a considerable number of lost cycles. In order to improve this point, the only solution is reducing the predictors. Some experiments have been done using a three-cycle, non-synthesizable 64Kbit TAGE. In Figure 6.5, there are the AutoBench benchmark results with the TAGE. The speed-up on IPC is low, 1.02X on average. It is not improving because the benchmarks have many branches that depend on random data without correlation. These branches are not predictable with any branch predictor.

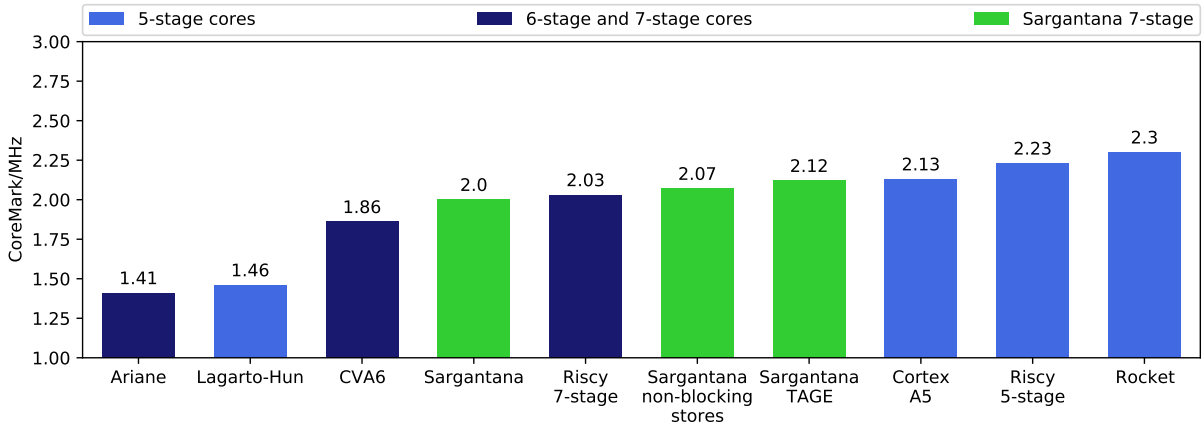


Figure 6.6: CoreMark/MHz score of the different cores. The cores are separated by number of stages.

6.3 IPC Evaluation Using the EEMBC AutoBench

Finally, to have a final comparison of the different cores, Figure 6.6 shows the CoreMark/MHz. CoreMark/MHz has a direct relation with the IPC. The figure separates the 5-stage cores from the 6-stage and 7-stage cores. The 5-stage cores (except the 5-stage Lagarto-Hun) have a higher score as expected. In this benchmark, Sargantana performs better than CVA6, and it is very close to the 7-stage Riscy performance. In this figure, there is also the Cortex-A5 form Arm[52] as an industrial reference of a single issue, low-power, in-order, 8-stage core.

Also, the two possible Sargantana optimizations (TAGE branch predictor and non-blocking stores) have been evaluated. Both optimizations make that Sargantana obtains a higher score than the 7-stage Riscy. In this case, the non-blocking store optimization gives a 3.5% of speed-up. In the case of the TAGE, the gain on performance is more significant. It gains a 6%. The misses per kilo-instruction (MPKI) on branch prediction is reduced a 2.85X, from 22,3 to 7,7.

7 Conclusions

In this thesis, the Sargantana core design is presented and evaluated. Sargantana is an advanced 7-stage in-order core design that significantly improves the performance of its predecessor, the 5-stage Lagarto-Hun. The final design has an IPC improvement of around 1.37X with respect to the 5-stage Lagarto-Hun design executing the Autobench suite. The improvements come from a faster front-end and a more complex back-end. The advanced back-end can do write-backs out-of-order, and with the renaming mechanism, most of the structural hazards have been removed. The last relevant optimization is the non-blocking memory pipeline, which can have multiple memory instructions in flight.

Apart from the IPC improvement, Sargantana has a more modular design that allows easy integration of new functional units thanks to its out-of-order write-back mechanism. Taking advantage of this feature, Sargantana incorporates an FPU and a SIMD pipeline.

The ASIC physical design of Sargantana has been made for a modern 22nm ASIC technology target. The core achieves 1,02 GHz in the slow corner, 1,39 GHz in the typical corner, and 1,66 GHz in the fast corner, having the critical path inside the FPU module. Without the FPU pipeline, the design can achieve 1,08 GHz on the slow corner, 1,58 GHz on the typical corner, and 1,88 GHz on the fast corner. This maximum frequency is higher than the 7-stage Riscy core and very similar to the frequencies achieved by the 6-stage Ariane and CVA6 cores. However, a direct comparison of the architectural design's frequency of Ariane and CVA6 can not be made since the results are not obtained with the same tool flow.

Comparing the IPC performance of Sargantana with the other cores, Sargantana achieves a slightly lower IPC compared to the 6-stage and 7-stage academic cores in the AutoBench benchmarks for embedded cores. However, with the presented optimizations, Sargantana achieves a very similar IPC on average. On the CoreMark benchmark, Sargantana gets 2 CoreMark/MHz, a better score than CVA6 and nearly the same as Riscy 7-stage. With these optimizations, it achieves better scores than the 6-stage and 7-stage cores.

7.1 Future Work

Even though Sargantana achieved significant performance improvements compared to its predecessor, some implementation aspects can be further improved in terms of IPC and maximum frequency.

In Sargantana, the limiting module in terms of frequency (apart from the FPU) is a 3-stage

front-end. The control logic of the hit detection inside the instruction cache propagates the SRAMs delays and the delay generated for the tag comparison to the TLB. It is necessary to fine-tune this logic to not propagate this delay to the TLB. This modification will increase the maximum frequency even further.

The two different IPC optimizations proposed in the evaluation section must be fully implemented and verified. TAGE is a complex branch predictor that needs some work to achieve higher frequency because it needs massive SRAM memories. Thus, the TAGE access needs to be multicycle. On the other side, a non-blocking pipeline that can advance loads over the stores needs the implementation of a store buffer. This optimization can be done in the memory pipeline of Sargantana without significant changes.

It would also be interesting to implement a superscalar pipeline in Sargantana. The core is already prepared to execute and write back several instructions in the same clock cycle. The only parts missing are the superscalar front-end, which is not a considerable modification, and a superscalar issue. This second part significantly increases the complexity of the issue logic. The issue logic has to consider the instructions already issued and the other instructions that will issue in this cycle. Moreover, the complexity of the issue stage could create significant critical paths that could affect the maximum frequency.

A final goal will be to fabricate Sargantana and measure the actual performance of the real silicon. There are several steps needed to make a successful tape-out. Although all the steps needed to make a successful tape-out are not finalized yet, the design and the verification are mature enough for fabricating Sargantana in the next months.

Bibliography

[1] Case, B. 'intel reveals pentium implementation details. *Microprocessor Report*, 5(23):9–17, 1993.

[2] Schor, D. IEDM 2017 + ISSCC 2018: Intel's 10nm, switching to cobalt interconnects. URL `https://fuse.wikichip.org/news/525/iedm-2017-isscc-2018-intels-10nm-switching-to-cobalt-interconnects/6/`.

[3] Zaruba, F. and Benini, L. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019. ISSN 1557-9999. doi: 10.1109/TVLSI.2019.2926114.

[4] LowRISC Inc. Rocket core overview. URL `https://www.cl.cam.ac.uk/~jrrk2/docs/tagged-memory-v0.1/rocket-core/`.

[5] Hennessy, J. and Jouppi, N. Computer technology and architecture: an evolving interaction. *Computer*, 24(9):18–29, 1991. doi: 10.1109/2.84896.

[6] Poovey, J., Conte, T., Levy, M., and Gal-On, S. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, 29:18 – 29, 11 2009. doi: 10.1109/MM.2009.74.

[7] McFarling, S. Combining branch predictors, 2017. WRL Technical Note TN-36, Jun. 1993.

[8] Seznec, A. A new case for the tage branch predictor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 117–127, 2011.

[9] Lempel, O., Peleg, A., and Weiser, U. Intel's mmx/sup tm/ technology-a new instruction set extension. In *Proceedings IEEE Computer Society International Conference (COMPCON) 97. Digest of Papers*, pages 255–259, 1997. doi: 10.1109/CMPCON.1997.584723.

[10] George, A. and Niska, K.P. Standard Cell Library. `http://www.signoffsemi.com/standard-cell-library-2/`, 2017. [Online; accessed 11-July-2020].

[11] Zu, Y., Huang, W., Paul, I., and Reddi, V.J. Ti-states: Processor power management in the temperature inversion region. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[12] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N.,

Hill, M.D., and Wood, D.A. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL `https://doi.org/10.1145/2024716.2024718`.

[13] Sanchez, D. and Kozyrakis, C. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th annual International Symposium in Computer Architecture (ISCA-40)*, June 2013.

[14] Carlson, T.E., Heirman, W., and Eeckhout, L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, November 2011.

[15] Grass, T., Allande, C., Armejach, A., Rico, A., Ayguadé, E., Labarta, J., Valero, M., Casas, M., and Moreto, M. Musa: A multi-level simulation approach for next-generation hpc machines. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 526–537, 2016. doi: 10.1109/SC.2016.44.

[16] López-Paradís, G., Armejach, A., and Moretó, M. Gem5 + rtl: A framework to enable rtl models inside a full-system simulator. In *50th International Conference on Parallel Processing*, ICPP 2021, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390682. doi: 10.1145/3472456.3472461. URL `https://doi.org/10.1145/3472456.3472461`.

[17] EEMBC. EEMBC CoreMark. URL `https://www.eembc.org/autobench/`.

[18] Weiss, A.R. Dhrystone benchmark: History, analysis, scores and recommendations.

[19] EEMBC. EEMBC AutoBench Performance Benchmark Suite. URL `https://www.eembc.org/autobench/`.

[20] Waterman, A. and Asanović, K. The risc-v instruction set manualvolume i: Unprivileged isa. Technical report, SiFive Inc. and EECS Department, University of California, Berkeley, Dec 2019. URL `https://riscv.org/specifications/isa-spec-pdf/`.

[21] Waterman, A. and Asanović, K. The risc-v instruction set manualvolume ii: Privileged architecture. Technical report, SiFive Inc. and EECS Department, University of California, Berkeley, Jun 2019. URL `https://riscv.org/specifications/privileged-isa/`.

[22] Mauro, A.D., Conti, F., Schiavone, P.D., Rossi, D., and Benini, L. Always-on 674u 4gop/s error resilient binary neural networks with aggressive sram voltage scaling on a 22-nm iot end-node. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(11):3905–3918, 2020. doi: 10.1109/TCSI.2020.3012576.

[23] Zaruba, F., Schuiki, F., Mach, S., and Benini, L. The floating point trinity: A multi-modal approach to extreme energy-efficiency and performance. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 767–770, 2019. doi: 10.1109/ICECS46596.2019.8964820.

[24] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D.A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. 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`.

[25] Chisel 3. Chisel/FIRRTL Hardware Compiler Framework. URL `https://www.chisel-lang.org/`.

[26] Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanović, V., and Asanović, K. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202, 2014.

[27] Doblas Font, M. Microarchitectural design-space exploration of an in-order risc-v processor in a 22nm cmos technology. B.S. thesis, Universitat Politècnica de Catalunya, 2020.

[28] Nikhil, R. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, 2004.

[29] Bluespec Inc. Bluespec Compiler (BSC). URL `https://github.com/B-Lang-org/bsc`.

[30] Zhang, S., Wright, A., Bourgeat, T., and Arvind, A. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 68–81, 2018.

[31] Celio, C., Chiu, P.F., Asanović, K., Nikolić, B., and Patterson, D. Broom: An open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. *IEEE Micro*, 39 (2):52–60, 2019. doi: 10.1109/MM.2019.2897782.

[32] Celio, C., Patterson, D.A., and Asanović, K. 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`.

[33] Celio, C., Chiu, P.F., Nikolic, B., Patterson, D.A., and Asanović, K. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html`.

[34] Zhao, J., Korpan, B., Gonzalez, A., and Asanovic, K. Sonicboom: The 3rd generation berkeley out-of-order machine. *Fourth Workshop on Computer Architecture Research with RISC-V*.

[35] SystemVerilog. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[36] Veripool. Verilator Verilog/SystemVerilog simulator. URL `https://www.veripool.org/wiki/verilator`.

[37] Cadence Design Systems, Inc. Genus Synthesis Solution, . URL `https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html`.

[38] Cadence Design Systems, Inc. Innovus Implementation System, . URL `https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html`.

[39] Ramírez, C., Hernández, C., Morales, C.R., García, G.M., Villa, L.A., and Ramírez, M.A. "Lagarto I–Una plataforma hardware/software de arquitectura de computadoras para la academia e investigación". In *Research in Computing Science*, volume 137, pages 19–28, 2017.

[40] Leyva-Santes, N., Pérez, I., Hernández Calderón, C., Vallejo, E., Moretó, M., Beivide, R., Ramirez, M.A., and Villa-Vargas, L. *Lagarto I RISC-V Multi-core: Research Challenges to Build and Integrate a Network-on-Chip*, pages 237–248. 12 2019. ISBN 978-3-030-38042-7. doi: 10.1007/978-3-030-38043-4_20.

[41] Abella, J., Bulla, C., Cabo, G., Cazorla, F.J., Cristal, A., Doblas, M., Figueras, R., González, A., Hernández, C., Hernández, C., Jiménez, V., Kosmidis, L., Kostalabros, V., Langarita, R., Leyva, N., López-Paradís, G., Marimon, J., Martínez, R., Mendoza, J., Moll, F., Moretó, M., Pavón, J., Ramírez, C., Ramírez, M.A., Rojas, C., Rubio, A., Ruiz, A., Sonmez, N., Soria, V., Terés, L., Unsal, O., Valero, M., Vargas, I., Villa, L., and Ramííez, C. An academic risc-v silicon implementation based on open-source components. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, 2020. doi: 10.1109/DCIS51330.2020.9268664.

[42] lowRISC. lowRISC 64-bit SoC. URL `https://www.lowrisc.org/`.

[43] U. C. Berkeley. The tilelink specification, version 0.3.3. `https://docs.google.com/document/d/1Iczcjigc-LUi8QmDPwnAu1kH4Rrt6Kqi1_EUaCrfrk8`. [Online; accessed 23-July-2019].

[44] Marco-Sola, S., Moreto, M., Espinosa, A., and Moure Lopez, J. Fast gap-affine pairwise alignment using the wavefront algorithm. pages 1–8, September 2020. doi: 10.1093/bioinformatics/btaa777. © The Author(s) 2020. Published by Oxford University Press.

[45] RISC-V Foundation. Spike, a risc-v isa simulator. `https://github.com/riscv-software-src/riscv-isa-sim`, 2019. [Online; accessed 1-October-2021].

[46] Google, Inc. Random instruction generator for RISC-V processor verification. URL `https://github.com/google/riscv-dv`.

[47] Lee, Y. and Cook, H. RISC-V Torture Test Generator. URL `https://github.com/ucb-bar/riscv-torture`.

[48] Shioya, R. Konata, an instruction pipeline visualizer. `https://github.com/shioyadan/Konata`, 2019. [Online; accessed 1-October-2021].

[49] Linux Kernel Organization, Inc. The Linux Kernel Archives. URL `https://www.kernel.org/`.

[50] Buildroot Association. Buildroot: Making Embedded Linux Easy. URL `https://buildroot.org/`.

[51] Western Digital Corporation. RISC-V Open Source Supervisor Binary Interface (OpenSBI). URL `https://github.com/riscv-software-src/opensbi`.

[52] Arm. Cortex-A5 Technical Reference Manual. URL `https://developer.arm.com/documentation/ddi0433/latest`.