Microarchitectural Design-Space Exploration of an In-Order RISC-V Processor in a 22nm CMOS Technology

Max Doblas Font

Supervised by:

Arvind

Miquel Moretó Planas

In partial fulfillment of the requirements for the degree in:

Telecommunications Technologies and Services Engineering

Bachelor's degree in Informatics Engineering

July 2020

Massachusetts Institute of Technology

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH
Centre de Formació Interdisciplinària Superior

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

telecos BCN

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Abstract

The purpose of this thesis is to apply microarchitectural design space exploration into an in-order processor to achieve a balance between cycle performance and maximum clock frequency. The work shows the impact on cycle performance, and maximum clock frequency for different pipeline optimizations applied to the processor. We target ASIC implementation using advanced synthesis tool flow with modern technology libraries to better analyze the processor's bottlenecks in terms of the maximum clock frequency in a real environment.

Usually, the ASIC target of a processor is not taken into account for academic computer architects since the process is large and expensive. Hence, the designs not always consider all the peculiarities of the ASIC implementation. This peculiarity can dramatically reduce the design's performance or make it not viable since the resource consumption could be enormous. We have analyzed and modified *Riscy*, an in-order core, to take into account these ASIC peculiarities enabling the ASIC target of the processor.

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

Resum

El propòsit d'aquesta tesi és aplicar l'exploració d'espai de disseny microarquitectònic en un processador en ordre per aconseguir un equilibri entre el rendiment per cicle i la freqüència màxima del rellotge. El treball mostra l'impacte sobre el rendiment per cicle i la freqüència màxima de rellotge per a diferents optimitzacions aplicades al processador. Utilitzem una implementació enfocada a ASIC usant unes eines de síntesis avançades amb biblioteques de tecnologia de fabricació modernes per analitzar millor els punts crítics del processador en termes de la freqüència de rellotge màxima en un entorn real.

En general, els arquitectes de computadors no tenen en compte la implementació ASIC d'un processador, ja que el procés és llarg i costós. Per tant, els dissenys no sempre consideren totes les peculiaritats de la implementació en tecnologia ASIC. Aquestes peculiaritats poden reduir dràsticament el rendiment del disseny o fer que no sigui viable, ja que el consum de recursos podria ser enorme. Hem analitzat i modificat emph Riscy, un nucli en ordre, per tenir en compte aquestes peculiaritats del procés ASIC i permetre el desenvolupament del processador en aquesta tecnologia.

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

Resumen

El propósito de esta tesis es aplicar la exploración de espacio de diseño microarquitectònic en un procesador en orden para conseguir un equilibrio entre el rendimiento por ciclo y la frecuencia máxima del reloj. El trabajo muestra el impacto sobre el rendimiento por ciclo y la frecuencia máxima de reloj para diferentes optimizaciones aplicadas al procesador. Utilizamos una implementación enfocada a ASIC usando unas herramientas de síntesis avanzadas con bibliotecas de tecnología de fabricación modernas para analizar mejor los puntos críticos del procesador en términos de la frecuencia de reloj máxima en un entorno real.

Por lo general, los arquitectos de computadores no tienen en cuenta la implementacion ASIC de un procesador, ya que el proceso es largo y costoso. Por lo tanto, los diseños no siempre consideran todas las peculiaridades de la implementación en techgnologia ASIC. Estas peculiaridades pueden reducir drásticamente el rendimiento del diseño o hacer que no sea viable ya que el consumo de recursos podría ser enorme. Hemos analizado y modificado *Riscy*, un nucleo en orden, para tener en cuenta estas peculiaridades del proceso ASIC y permitir el desarollo del procesador en esta technologia.

Palabras clave: Microarquitectura, síntesis ASIC, Processador en orden, Memorias SRAM, Camino crítico, Bluespec SystemVerilog, RISC-V.

Acknowledgements

I would like to express my gratitude to Professor Arvind for giving me the opportunity to be a member of his research group during my stay at CSAIL, MIT, and to actively collaborate after my return to Barcelona.

Moreover, I would like to thank Miquel Moretó. He established the first contact with professor Arvind and also supervised my work directly from UPC. He has taken an active interest in my growth both on academic and personal levels.

I would also like to thank Andrew Wright and his fantastic work done in the Riscy in-order core, which is the baseline processor used in this thesis. It is a very friendly core to work with thanks to the clear organization and the quality of the code. Thanks to Andy for his help during the thesis and all the knowledge I have learned from him.

Finally, I want to thanks 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 learned a lot.

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

This visiting experience could not have taken place without the support from CFIS. Special thanks also to Fundació Privada Cellex, the Barcelona Supercomputing Center, and Generalitat de Catalunya for supporting my stay financially.

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	semiconductor Intellectual Property core
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
I/O	Input/Output
MCU	Mcicrocontrollers
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
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

The research field of computer architecture has evolved continuously because of the advances in nanotechnology that allows manufacturing of circuits with smaller, faster, and cheaper transistors. These improvements enable computer architects to design complex circuits to make processors faster, cheaper and more efficient. The development of faster and more efficient processors opens the door to significant advances in many other areas, including medicine, autonomous driving and Internet-of-Things (IOT).

1.1 Motivation

Computer architecture research in the academic community has generally focused on designing processors to reduce the number of clock cycles it takes to execute a program. They use instructions-per-cycle (IPC) as a performance metric but *ignore* the length of the clock cycle, itself. In the end, the real performance of a processor is measured in terms of instructions-per-second and thus, both the IPC and the clock frequency matter. This thesis is about designing and evaluating microprocessors by taking into account both the IPC and clock frequency.

Clock frequency depends primarily upon the underlying technology used to implement the circuits. Nowadays, processors 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 additive and applies to all signals that flow through a given path. The majority of the techniques used to increase the IPC also adds extra logic to the processor design, increasing the number of transistors and interconnection paths as a result. This extra logic invariably increases the latency of the signals that pass through them. The clock period has to be large enough to let the slowest signal arrive at the destination. Thus, even an infrequently used logic path can have a huge negative effect on the clock frequency.

The clock frequency and area of a processor design cannot be estimated without hardware synthesis of the design using technology-specific Application-Specific Integrated Circuit (ASIC) gate libraries. Often researchers focus on IPC of an architecture because it can be determined by building cycle-accurate software simulators. Unfortunately, estimating the clock frequency and the area of a circuit accurately is quite difficult because hardware synthesis requires a whole set of different and expensive tools, and access to ever changing gate libraries and memory compilers. Gate libraries are designed by chip manufacturers for each CMOS technology family they support and are almost never available openly or freely. Not surprisingly, architecture literature is full of techniques to improve the IPC but severely lacking in the quantitative impact of these techniques on clock frequency or area (extra circuitry).

IPC can be improved by reducing the number of stalled cycles of the processor. For example, it is possible to make some predictions to *speculatively execute* future instructions before we are sure that they are the correct instructions to execute, and squash the execution of the instruction if the prediction turns out to be wrong. Also, bypassing the data directly from a stage where an older instruction is executing to a stage where a younger instruction is executing, without storing it in an intermediate register, can shave off a clock cycle from execution. One can replicate computational unites to execute instructions concurrently to increase the throughput of the processor. The impact of all these and many other microarchitectural techniques on reducing IPC is well understood but not their quantitative impact on clock frequency and area.

Since the performance of a single-core processor is the direct multiplication of the IPC and the clock frequency, it is essential to give the same importance to both parameters. The right balance of these two parameters gives the maximum possible performance. Sometimes this is the most challenging part. To achieve the right balance, first measuring the two different metrics is required. On the one hand, the IPC is simple to compute, as we only need to simulate, emulate, or run a benchmark on a processor and determine the number of cycles and executed instructions. Also, adding some counters to the design allows architects to quickly know where the processor is spending most cycles.

On the other hand, it is challenging to know the actual clock frequency that a processor design is capable of achieving. Since this frequency is highly dependent on the technology chosen and the low level optimizations applied by opaque tools, access to the tools and libraries needed to do synthesis are essential. In thesis, we study several design tradeoffs in processor design using a rather modern 22nm CMOS technology, which is available to us.

1.2 Objectives

This project aims to improve an in-order processor called *Riscy*, a 5-stage in-order core developed at the Massachusetts Institute of Technology (MIT), by taking ASIC hardware synthesis into account. Many versions of *Riscy* have been synthesized for FPGAs and have been used in teaching. A simplified different versions of *Riscy* was prototyped using carbon nanotube field-effect transistors [14]. Our first objective was to enable the synthesis of the processor using an ASIC synthesis tool flow. Some specific parts of the processor, such as the memories located on the caches, needed special treatment to be synthesizable. These memories consume too much area if they make use of the same type of cells as a register. Memory specific circuit designs can substantially reduce their area but these special memories need to be generated by a special *memory compiler*.

The second objective is to improve the design of the in-order processor to achieve good performance for an ASIC target. We want to achieve a high clock frequency of 1 GHz. This frequency of 1 GHz is high enough to force us to make some changes in the design, but it is not so demanding as to require a complete redesign of the processor. We pay special attention to not decreasing the IPC, while increasing the clock frequency. Also, as the Riscy design is quite simple, we analyze and implement extra features to increase the IPC without reducing the clock frequency.

1.3 Document Structure

The rest of this thesis is structured as follows:

- Chapter 2 provides an overview of basic computer architecture and physical design implementation concepts to make the document self-contained.
- Chapter 3 analyzes some academic processor designs explaining their structure and properties, such as IPC and clock frequency.
- Chapter 4 introduces the tool flow we use for Register-Transfer Level (RTL) design and also for ASIC synthesis.
- Chapter 5 describes all the in-depth analysis and improvements that have been carried out on the Riscy processor to achieve the expected frequency of 1 GHz with a good IPC.
- Chapter 6 evaluates the final design.
- Finally, Chapter 7 discusses the results and concludes this work.

2 Background

As we want a self-contained document, we have to introduce some basic ideas on computer architecture. First, we explain how a processor is physically implemented and the concept of gate delays, which will lead to the critical path concept. Also, we introduce some basic concepts of ASIC manufacturing. Then, we introduce the basics about the structures in a processor. In particular, we introduce the concept of an in-order processor design, which is used throughout this thesis. Finally, we describe some pipeline optimizations to increase the IPC. It is essential to understand the penalties on clock frequency that can be introduced with these IPC optimizations.

2.1 Physical Implementation

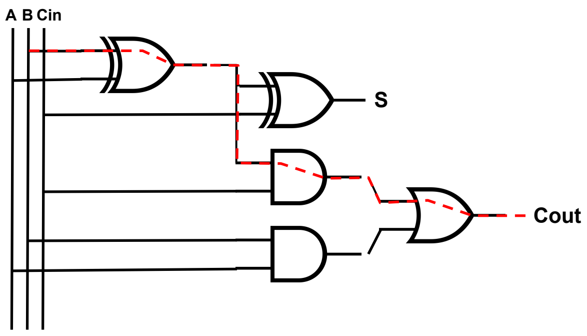
Processors run a set of functionalities implemented with a collection of combinational circuits and registers that are chosen precisely to achieve maximum performance and efficiency. These circuits and registers are physically implemented using a large assembly of logic gates. These logic gates are a combination of transistors to implement a variety of Boolean functions. Logic gates generate a digital signal as output depending on one or more digital signals that arrive at them as inputs.

An essential parameter of a circuit is the propagation delay, which is the time between the moment the input of a logic gate becomes stable and valid, and the moment the output of that logic gate becomes stable and valid. Often on the datasheets, it is specified as the time required for the output to reach half of its final output level when the input changes to half of its final input level. This delay can be modeled with different degrees of precision. A very simple model is to assign a delay to every type of logic gate, and compute the delay of the signal as the sum of all the gates delay it passes through. This simple model has proven quite useful in practice in gaining understanding of the delay properties of the design.

However, more complex models are needed to get more realistic results. For example, the resistance of conductive materials tends to increase with temperature, and therefore the operating temperature should be taken into account in computing the delay. The increases in output load capacitance, often from placing increased fan-out loads on a wire, also increase propagation delay. These factors influence each other through an RC time constant of the gate, which is a more realistic model. Even simple RC models are not enough for today's high-speed digital circuits [9], if we want very accurate results. Also, these delays depend not only on the design, but also on the fabrication process (as it is explained in detail in Section 2.1.3).

A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) Full adder truth table



(b) Full adder gate schematic

Figure 2.1: Full adder physiscal implementation. The critical path of the full adder circuit is indicated with a red dotted line in the subfigure b.

At this point, we can introduce the concept of a critical path. The critical path is the path of a circuit with the most significant signal delay between its start and ending point. As the delay is introduced directly from the gates and the interconnections in the path, this critical path depends also on the technology used. In the end, this critical path will be the one that is limiting the maximum clock frequency that the circuit is going to achieve. Thus, it is required to reduce it in order to achieve a higher frequency on the design. There are many techniques to reduce the critical path and to increase the clock frequency [1]. For example, the synthesis tools sometimes replace the circuit, which is responsible for the critical path, for another with the same functionality that has a shorter critical path.

We want to summarize these concepts using a full adder circuit as a simple example. The Boolean function of the one-bit adder is shown in Equation 2.1. Signals A and B are the input values that have to be added. C_{in} is the input carry from a previous adder. Output carry and sum are represented by the signals C_{out} and S, where the sum of the inputs equals $2 \cdot C_{out} + S$.

$$S = A \oplus B \oplus C_{in}; \quad C_{out} = ((A \oplus B) \cdot C_{in}) + (A \cdot B) \tag{2.1}$$

The truth table is shown in Figure 2.1a and the gate circuit in Figure 2.1b. In this example, we are using a simpler model for timing analysis. We are only taking into account a fixed delay for each type of gate. In this example, let us assume that the gate delays are: AND (10ns), OR (5ns), XOR (10ns). These are not realistic numbers but useful for illustrating the delay and critical paths. Using these values, we can distinguish four different paths depending on the delay. In this case, the critical path comes from A going to the XOR gate with the signal B. Then to the AND gate with C_{in} and finally to the OR gate to generate C_{out}. The delay of the critical path is the sum of the gate delays on the path (one XOR, one OR, and one AND gate), giving a total delay of 25ns. Thus, the maximum clock frequency of the circuit will be 40 MHz at most.

2.1.1 Standard Cell Libraries

In reality, at the synthesis step, the logic gates are known as standard cells. A standard cell is a group of transistors and interconnection structures that provides a Boolean logic function

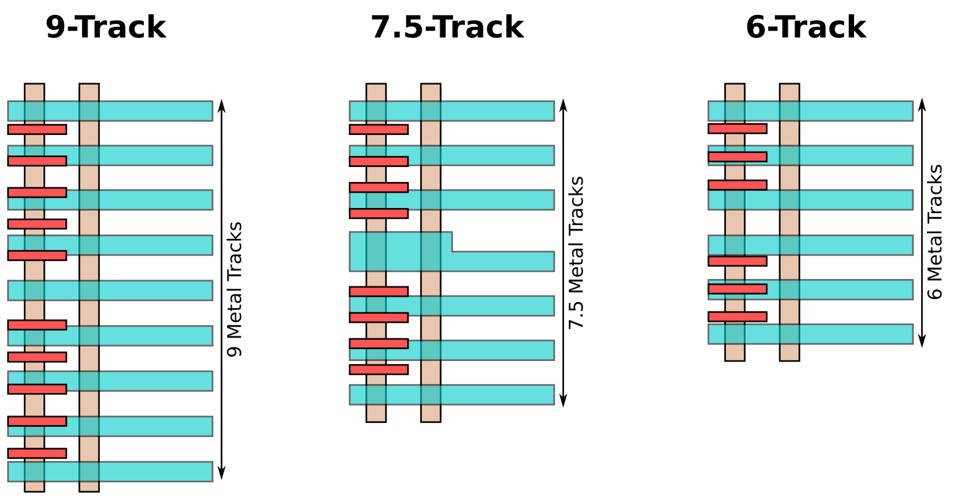


Figure 2.2: Comparation between diferent standard cells with diferent tracks [28]

(e.g., AND, OR, XOR, XNOR, inverters) or a storage function (flip-flop or latch). There are elementary cells that implement a NAND, NOR, and XOR Boolean function, although there are cells of much greater complexity that are more commonly used (such as a 2-bit full-adder, or muxed D-input flip-flop). Vendors like Synopsys, GlobalFoundries, or TSMC have libraries of standard cells of different technologies.

Standard cells are designed based on power, area, and performance. Thus, there are several types of standard cells, depending on those parameters. Standard Cell architecture is all about deciding cell height based on the pitch (minimum distance metal lines) and library requirements. 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 [13]. In the end, the vendor provides several libraries where these parameters are already chosen. Finally, the user has to select a library depending on the technology node (65nm, 22 nm, 7 nm), transistor gate pitch, that is also referred as Contacted Poly Pitch (CPP) (104CPP, 116CPP), the number of tracks (7T, 8T, 12T), or the Vt 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 lanes inside the standard cell. For example, a 9 track library implies that 9 routing tracks are available for routing 9 wires in parallel with minimum pitch inside a standard cell. In Figure 2.2

If we compare, for example, 6T and 9T libraries, 9T is faster and will give better performance because the area for 9T is more so that we can place higher drive strength transistors in it. 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.

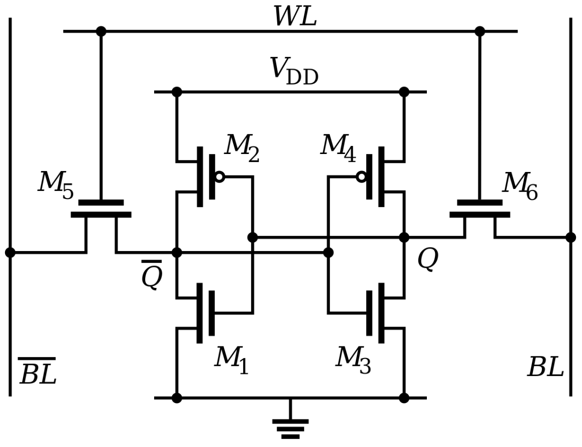


Figure 2.3: A six-transistor CMOS SRAM cell

2.1.2 Hard IP Blocks

Hard IP blocks are defined as IP cores that cannot be modified. Because it has a low-level representation, hard cores offer better predictability of chip performance in terms of timing performance and area. Usually, analog and mixed-signal logic are defined as a lower-level. Are descrived in the physical level. Thus, analog IP (SerDes, PLLs, DAC, ADC, PHYs) are provided to chip makers in transistor-layout format.

Whether analog or digital, such modules are called ”hard cores” (or hard macros) because chip designers cannot meaningfully modify the core’s application function. Transistor layouts must follow the target foundry’s process design rules. Thus, hard cores delivered for one foundry’s process cannot be easily ported to a different processor foundry. Foundry operator companies (such as IBM, Fujitsu, Samsung, TI) offer a variety of hard-macro IP functions built for their foundry process, helping to ensure compatibility.

The most commonly used hard IP in a core design is the Static Random Access Memory (SRAM) memories. SRAM memories are needed for implementing large memory arrays. SRAM is a variety of semiconductor memory commonly used in electronics, microprocessors, and general computing applications. This variety of semiconductor memory is static because it holds data as long as power is applied, and does not need to be dynamically refreshed as in the case of Dynamic Random Access Memory (DRAM) memory. However, this type of memory is volatile, which means it can not hold data if power is removed. It is commonly used for big memories inside the chip because the footprint and power consumption are much smaller than regular flip-flop registers. The basic SRAM is designed with six transistors (known as 6T). The schematic is shown in figure 2.3.

2.1.3 Process, Voltage, Temperature Variations

A fundamental concepts for synthesis and physical design are Process, Voltage, Temperature (PVT) variations. A physically implemented design has to work under many possible conditions, from the Siachen Glacier at a temperature as low as -40°C up until the Sahara Desert at 60°C. To ensure that the design is capable of running in the chosen frequency, it is necessary to simulate

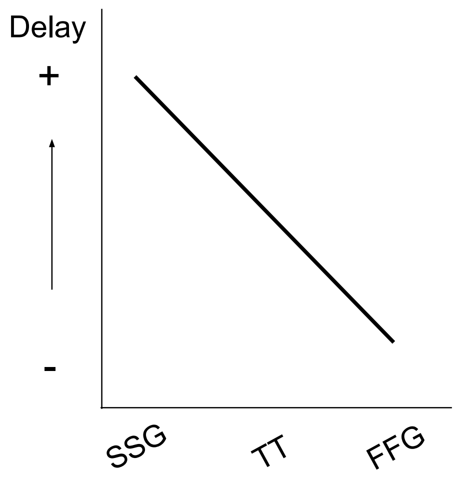


Figure 2.4: Cirduit delay depending on the process variation

it at different conditions of the process, voltage, and temperature, which the integrated circuit may face after fabrication. These conditions are called corners. All of these parameters affect the delay of the cell.

Process variation is the deviation in attributes of the transistor during the fabrication. When manufacturing a die, the area at the center and the boundary will have different process variations. This happens because layers that will be getting fabricated can not be uniform all over the die. As we go away from the center of the die, layers can differ in their sizes.

Process variation is different in different technologies, but is dominant in lower node technologies (<65nm). Important factors that can cause process variation are the wavelength of the ultra-violet light and the manufacturing defects. It can cause oxide thickness variation, dopant and mobility fluctuation, variation of transistor size, and other effects.

These variations will produce some changes in the parameters like threshold voltage, distancing its value from expected, and modifying the transistor bandwidth, thus, the delay. The foundries characterize these variations and generate different timing models for synthesis tools, depending on the fabrication's quality. The quality can go from slow to fast, as represented in the next curve shown in Figure 2.4.

Voltage variation is significant nowadays since the operation voltages of modern chips are near or below 1 V. A small variation in the power supply can cause a significant change in the nominal operating voltage. Thus, we need to consider voltage variation in the design process. There are multiple reasons for voltage variation, such as the parasitic resistance of the power grid or parasitic inductance, which the current flowing through can cause voltage glitches. The voltage can also be modified consciously to reduce the power consumption or, alternatively, can be augmented to increase the performance.

Because of all these factors, it is necessary to consider the voltage variation in the design simulation. Figure 2.5 shows the relation between supply voltage and delay.

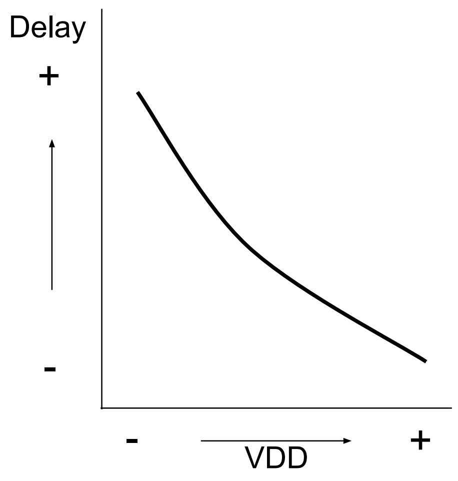


Figure 2.5: Cirduit delay depending on the voltage variation

Temperature variation is not only related to the ambient temperature; we also have to consider that the chip can dissipate a large amount of power in a tiny area and increasing the temperature significantly. The temperature inside the chip can vary within a big range, and that is why temperature variation needs to be considered. Figure 2.6 shows the variation of delay concerning temperature. Delay of a cell increases with temperature. However, this is not true for all technology nodes. For deep sub-micron technologies, this behavior is different. In these technologies, the phenomenon called temperature inversion [39] appears. To understand that, we will use the MOSFET drain current (I_D) equation shown in 2.2. The delay is directly proportional to I_D and inversely proportional to the output capacitance. Output capacitance does not depend on temperature. However, as temperature increases, mobility (μ) and threshold voltage (V_t) start decreasing. The I_D is proportional to mobility and decreases with Vt quadratically.

$$I_D = \frac{1}{2}\mu_n C_{ox}\frac{W}{L}(V_{GS}–V_t)^2 \tag{2.2}$$

In higher technology nodes, where the supply voltage is very high, the effect of V_t is as low as $(V_{GS} - V_t)$ value is significant. Hence mobility plays a significant role in deciding the current level. So at higher technology nodes, when the temperature increases, mobility decreases, and as a result, the delay will increase. Nevertheless, in lower technology nodes, especially in technologies smaller than 65nm, the supply voltage is low. Thus, $(V_{GS} - V_t)$ is very small, and as it is squared, a small variation can be very significant in the I_D. Consequently, an increase in V_t (increasing temperature) can decrease the delay, contrary to larger technology nodes.

2.2 Microarchitecture Background

A processor performs basic arithmetic, logic, controlling, Input/Output (I/O) operations specified by the instructions in a program that is stored in memory. The processor is divided into three main parts. The core (or cores in a multiprocessor), the memory, and the I/O. In this

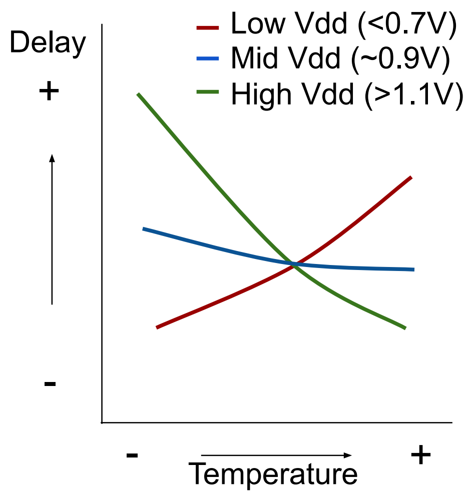


Figure 2.6: Circuit delay depending on the temperature variation. It depends on the Power supply voltage

project, we focus only on the core part. The core is the element of the processor in charge of asking for an instruction from memory, processing it, and storing the results to internal registers or memory. A classical processor core performs three main actions: fetch, decode, and execute.

During the fetch, the core sends a request to the memory hierarchy, usually the instruction cache, for the next instruction that has to be executed. This instruction is selected using the Program Counter (PC). The PC is a register that stores the address of the next instruction to be executed. After an instruction is fetched, the PC is incremented by the length of the instruction, so that it will contain the address of the next instruction in the sequence. However, the PC can be modified with control instructions to follow a different sequencing than the implicit sequencing.

The decoder is the next step. In this stage, the core interprets the instruction obtained in the fetch state to determine the actions that have to perform during execution. These actions are codified in the instruction using a particular Instruction Set Architecture (ISA).

Finally, in the execution stage, the core executes the actions decoded from the instruction and changes the state of the core accordingly (storing the results to the register file, making a request to data memory, modifying the PC, etc.).

2.2.1 Pipeline Design

As explained earlier, a processor design will be physically implemented using transistors. Let us assume a processor that does all the defined steps (fetch, decode, and execution) in one cycle. The critical 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 frequencies, we introduce the concept of *pipeling*.

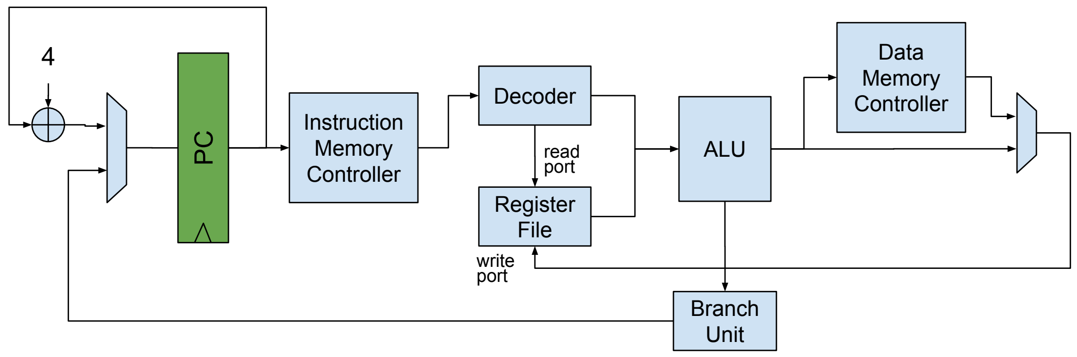
The pipeline concept is used in a variety of areas, for example, in factories. Let us describe a simple example to better understand this concept. Imagine a factory that makes bread. The action of making bread can be divided into three steps: making the dough, cooking it, and distribute (this step includes all the packaging as well). Suppose that in the factory, there is only one employee that is an expert in these three areas. Also suppose that making the dough takes 40 minutes, baking it 50, and the distribution takes 35 minutes. This employee can make and send one batch of bread every 125 minutes.

If we apply the concept of pipelining, we can do the following. Now we have three employees that each one is an expert on only one step of the bread fabrication. Now each one only does one step. With this distribution, the employees can work in parallel if all of them have the materials needed for it. In the first minutes, the only employee that is doing something is the one that elaborates dough. In a few minutes, this employee gives the dough to the second one. At this moment, these two employees can perform their actions at the same time. When the second employee finishes, the third one will start his process allowing to have the three employees to work in parallel. Then, using the pipeline structure, we can make a batch of bread every time that the slower worker finishes. This slower step, the baking step in this example, is the critical path of the pipeline. Thus, the factory can produce one batch of bread every 50 min. Thus, using a pipelined structure, we have more bandwidth, but the time needed to do a completed from the start to the end is not reduced, even it can increase. In this example, the production on a single batch takes three times 50 minutes, a total of 150 minutes, that is bigger than the 125 minutes taken from a single employee.

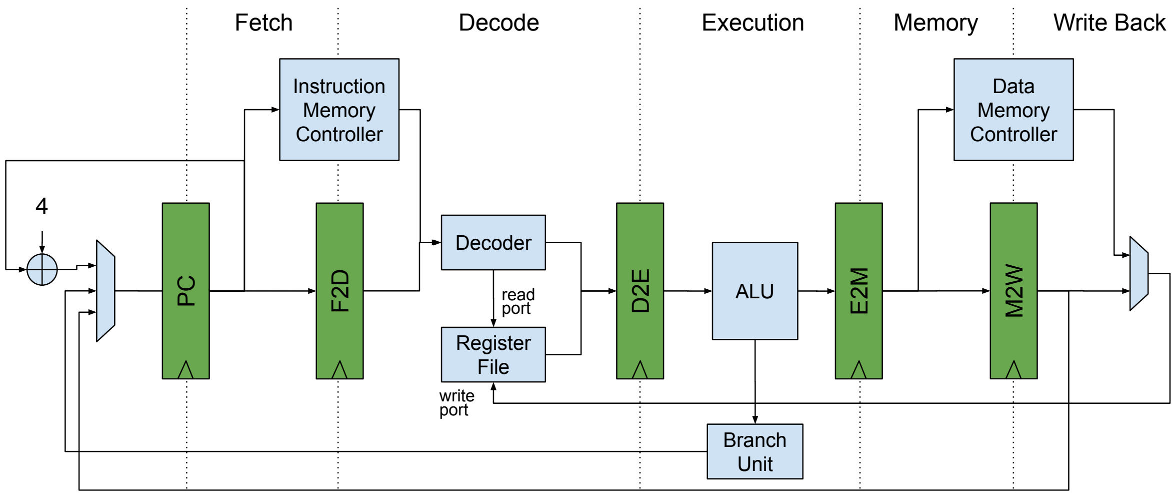
This simple pipeline concept can also be applied to computer architecture. The instruction's execution can be divided into different parts, stages, which can be executed in parallel. Figure 2.7 shows an example of a processor with five distinctive stages. The processor is composed of these steps:

- Fetch (F): This module asks for the next instruction to memory and updates the 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): If the instruction is a load or a store, a memory request is performed.
- 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.

On the top side of the figure, the core executes the whole instruction in one cycle while on the bottom side, the core is divided into five pipeline stages. With the last configuration, the delay on each stage is up to five times smaller than the one-stage core. Thus, the clock frequency can be up to five times faster. However, applying the pipeline concept has also some drawbacks that are important to take into account.



(a) Single cycle core design.



(b) 5-stage pipelined core design.

Figure 2.7: Pipelining a simple in-order core design

2.2.2 Pipeline Hazards

Until now, a processor that executes a completed instruction in one cycle has an IPC of one (i.e. the ideal IPC in the case of a processor than only issues one instruction per cycle). In contrast, this ideal IPC is impossible to achieve in a pipelined single-issue 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. The main hazards are:

Structural hazard A structural hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. In the example in Figure 2.8, we are showing the execution of a small code on a processor with a unified memory that is used to store instructions and data. A structural hazard can be found when there is an instruction in the fetch state and, at the same cycle, a memory instruction in the memory stage. In this situation, both instructions need to access the same memory. As a result, the newest instruction (the one in the fetch state) has to be stalled. An easy solution is to duplicate the resources to allow both instructions to access the memory hierarchy. In this case, two different memories, one for instructions and another for data, can be used as a solution.

	Execution Cycles											
Program Instructions	1	2	3	4	5	6	7	8	9	10	11	12
I1: lw r1, 0(r4)	F	D	E	M	W							
I2: li r3, 7		F	D	E	M	W						
I3: addi r4, r4, 4			F	D	E	M	W					
I4: and r1, r1, r3				-	F	D	E	M	W			
...												

Figure 2.8: Pipeline execution of a processor with a unified instruction and data memory showing a structural hazard.

$$\begin{aligned} i_1 &: \quad r2 \leftarrow r1 + r3 \\ i_2 &: \quad r4 \leftarrow r5 + r2 \end{aligned}$$

(a) RAW

$$\begin{aligned} i_3 &: \quad r2 \leftarrow r1 + r3 \\ i_4 &: \quad r1 \leftarrow r5 + r4 \end{aligned}$$

(b) WAR

$$\begin{aligned} i_5 &: \quad r2 \leftarrow r4 + r1 \\ i_6 &: \quad r2 \leftarrow r3 + r5 \end{aligned}$$

(c) WAW

Figure 2.9: Different types of data hazards

Data hazard Data hazards can happen when different instructions exhibit data dependencies while reading or writing a particular data in different stages of the pipeline. If the potential data hazards are ignored, it can result in race conditions and the wrong operation of the processor. There are three situations that can produce a data hazard:

- **Read After Write (RAW), a true dependency**: refers to a situation where one instruction needs a result that has not yet been calculated for an older instruction. In the example in Figure 2.9a, the instruction i_2 needs to read from the register $r2$ that is calculated in the instruction i_1. Maybe the i_1 is not completed when the i_2 is issued, and $r2$ does not have the data ready. The solution is to stall the pipeline and wait for the i_1 to finish before the i_2 reads the register $r2$.

- **Write After Read (WAR), an anti-dependency**: It happens when a younger instruction writes to a register that has to be read for an older instruction. It can only happen when the instructions are executed in concurrency or out of order. An example is shown in Figure 2.9b. The instruction i_3 needs to 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 be found in concurrent environments. An example is shown in Figure 2.9c. The instructions i_5 and i_6 need to write to the same register.

To show an example, we make use of a RAW dependency as it is the only data hazard that can be found in the 5-stage processor presented before. In this processor, the registers are

	Execution Cycles											
Program Instructions	1	2	3	4	5	6	7	8	9	10	11	12
I1: addi r2, r1, 1	F	D	E	M	W							
I2: and r3, r2, r1		F	-	-	D	E	M	W				
...												

Figure 2.10: Pipeline execution with a RAW hazard. The isntruction i2 is stalled until the reister r2 is wirten back

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

read on the decoding stage and are written in the last stage, write back. As we can be seen in Figure 2.10, there must be at least two cycles between the producer and consumer instructions, if the register file does the write before the actual read.

Control Hazards A control hazard occurs when there is a control instruction or an exception on the pipeline that modifies the sequential order of the program. Therefore, there are instructions inside the pipeline that are not following the program order and must subsequently be discarded. In Figure 2.11, we show an example of a control hazard with the same 5-stage pipeline design as before. In this case, we assume that we know the result of a branch instruction in the execution stage. At this moment, the redirection of the PC is done if the branch results in a jump. Thus, the instructions inside the fetch and decode stages have to be flushed because they are not in the program path anymore. Thus, a taken branch or jump instructions are penalized more in larger pipelines as the number of flushed instruction increases, and the latency to fill the pipeline is more significant.

2.3 IPC Optimizations

As we show in Section 2.2.2, several factors introduce some lost cycles. If the only solution to these hazards is stalling the pipeline when an instruction can not be executed, the performance

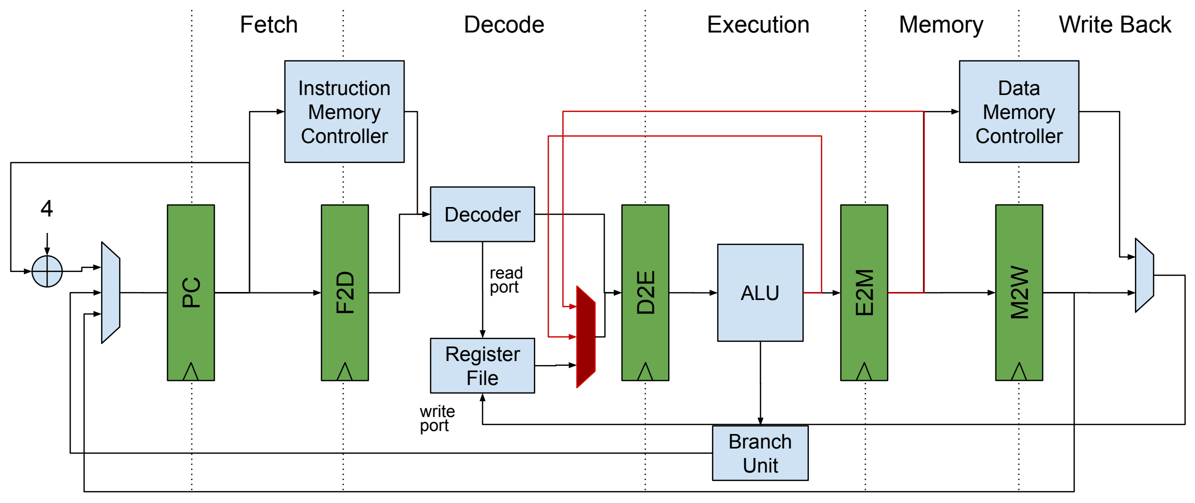


Figure 2.12: Pipeline 5-stage with bypasses

of the core can be far from the ideal IPC. To achieve a good IPC, it is necessary to apply some optimizations to the pipeline to reduce or eliminate the hazard effects. In this section, we introduce simple optimizations that can be done in an in-order core. We will not introduce out-of-order execution, because it will not be applied in this project.

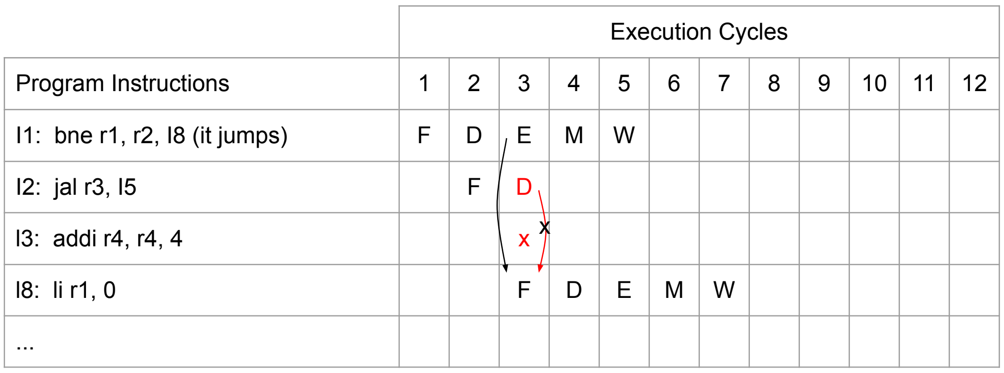
2.3.1 Bypassing

Bypassing (also known as forwarding) 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 at an earlier stage in the pipeline without waiting for the write-back. Bypasses can nearly eliminate all the data hazards inside a pipeline. The possible bypasses that can be implemented on our example 5-stage core are shown in Figure 2.12 in red.

The bypasses **1** and **2** are used to forward the data generated in the execution stage. Bypass **1** is used when the RAW dependency is back-to-back. Instead, bypass **2** is used when there is another instruction between the producer and the consumer instruction. It does not make sense to forward data generated by a memory instruction in this processor, as the result of memory instructions is on the write-back stage. There is no data hazard when the producer is an instruction computed in the execution stage if we implement these bypasses. However, the data hazards with memory instructions producers still need to stall the pipeline to wait for the result.

However, bypasses introduce some penalties on the design. On the one hand, they augment the resources consumed (i.e. area in an ASIC target). Forwarding needs extra multiplexers for selecting from where the operands are coming from. Also, it adds extra logic on the control side. It is worth mentioning that the increment on resources is not high, and usually, it is not problematic.

On the other hand, and more importantly, bypassing can add new paths to the pipeline. If these bypasses are not well-selected can interconnect extensive paths and, in the worst-case,



	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: jal r3, I5		F	D									
I3: addi r4, r4, 4			x									
I8: li r1, 0			F	D	E	M	W					
...												

Figure 2.13: Pipeline execution with PC redirection from different stages in the same cycle. The PC is redirected with the ¡destination of the older instruction

notably increase the critical path. It is recommended to place all bypasses always at the start of stages or all at the end. If it is not done like that, the delay introduced by the bypass can be up to the sum of the two stages that are connected, resulting in a significant negative impact on the maximum frequency.

2.3.2 Branch Prediction and PC Redirection

As we know, control hazards have a notorious impact on pipeline performance. It is more worrying if we know that the control instruction in a program can be more than 20% of the total instructions [25]. There are two main ideas to reduce the loss of performance due to PC redirection, the early PC redirection, and the branch prediction.

Early PC redirection consists of redirecting the PC when it is possible to precisely know which will be the next PC for the next instruction. For example, an unconditional jump, that has the destination codified inside the instruction, can be redirected at the decoder stage. However, for a conditional jump, it is necessary to wait until the condition is computed on the execution stage to redirect the PC. With this technique, it is possible to reduce the penalty for a branch or another PC redirection.

Consequently, it is possible to implement PC redirections in different stages at the same time. It is crucial to correctly implement the control logic to select the new PC in simultaneous redirections. The predominant redirection must be the redirection produced by the older instruction in the pipeline. In the example in Figure 2.13, there is a conditional branch instruction (i1) that is mispredicted on the execution stage, which has to redirect the PC. On the decoding stage, there is an unconditional jump instruction (i2) that also wants to redirect the PC. Since the instruction i2 should not be executed, the redirection of this instruction must not affect the execution.

Nevertheless, early PC redirection has similar disadvantages as bypassing. New multiplexers and control logic are placed, increasing the resources consumed. Also, these new redirections can introduce new broad paths generating a new extensive critical path. It is essential to choose carefully new redirections to avoid reducing the maximum clock frequency of the core, resulting

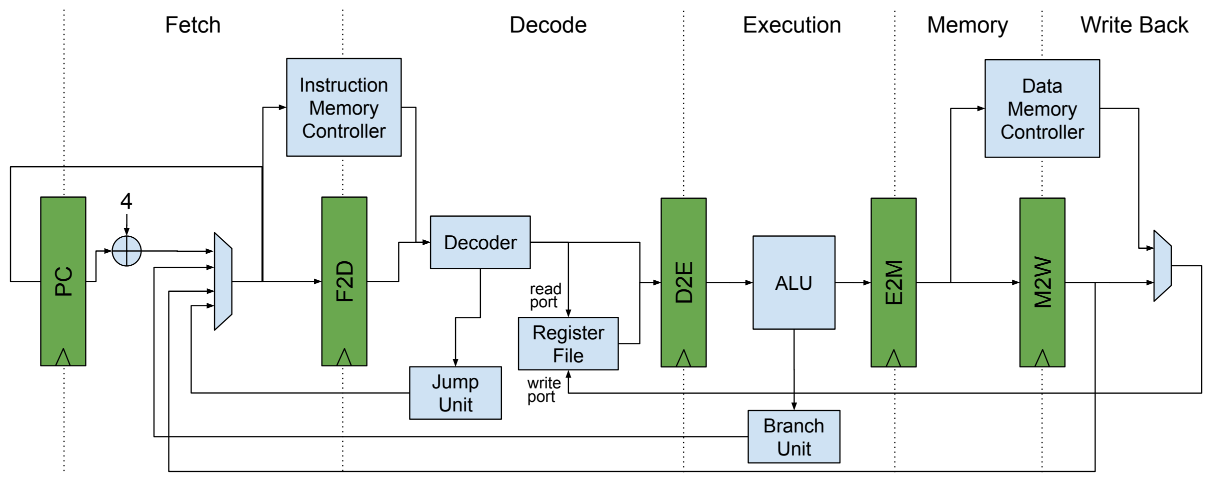


Figure 2.14: Pipeline with early PC redirection

in an overall reduction of performance.

Next, Figure 2.14 shows the implementation of new redirection paths in the 5-stage pipeline design. Apart from the new redirection from the decoder stage, there are some modifications in the fetch stage. All the redirections now are done before the instruction memory access, reducing the penalty of all PC redirections one cycle. Even though we are incrementing the IPC, we are also increasing some path delays. There is a new path that goes through all the execution stage and also the fetch stage and the instruction memory controller. These new paths must be carefully examined to determine if the trade-off of IPC and maximum clock frequency is worthwhile.

Branch prediction is a hardware mechanism that predicts the address of the instruction following the branch. A combination of different specified circuits is used to make a prediction. In this section, we will focus on the Branch Target Buffer (BTB), Branch History Table (BHT), and Return Address Stack (RAS), since these predictors are the most commonly used in in-order processors.

- **Branch Target Buffer** remembers recent target PCs for a set of control instructions. It has a cache like structure where every entry has the instruction PC and the target PC of the last execution. Thus, the next time the core executes the instruction located at the same PC, it can know which target PC was used the last time, and use this information as a prediction. The BTB has to be updated at every PC misprediction with the new target PC or eliminate the entry if it is not a control instruction.

- **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. The two-bit version of this predictor is the more commonly used. This version introduces a little bit of hysteresis that helps in eliminating the influence of unusual patterns. BHT is implemented with an array of saturated counters indexed with the least critical bits of the instruction. These saturated counters are updated every time that a conditional branch instruction

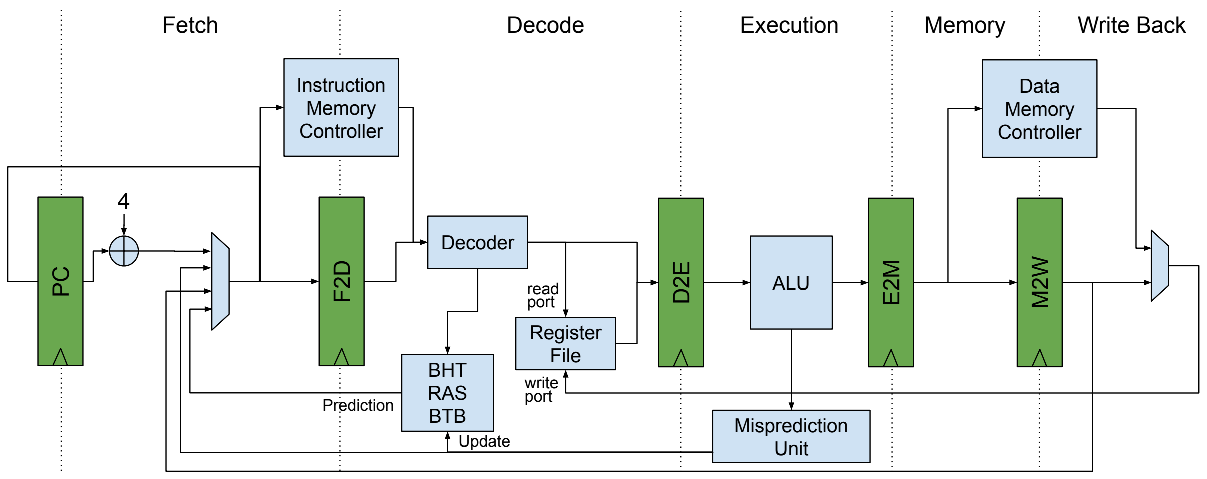


Figure 2.15: Pipeline with early PC redirection and branch predictors

is resolved. If the result is taken, the counter related to the instruction is incremented, otherwise it is decremented. The most significant bit of the counter is taken to make a prediction. If this bit is one, the result of this branch instruction condition is probably taken. If it is zero, then it is probable that it is not taken.

- **Return Address Stack** is a predictor that helps on return instructions destination predictions. BTB does not predict return instructions well, because the return address is not always the same (different parts of the program can call the same function). Return instructions almost always return to the last procedure call instruction, and return address stacks are highly accurate. Every time a CALL instruction is executed, its return address is pushed onto the stack. Every time a RETURN instruction enters the pipeline, the next address is popped off the stack, and the processor continues fetching from it.

In Figure 2.15, there is an example of a pipeline implementation using BTB, BHT, and RAS modules. This baseline 5-stage pipeline also integrates the early PC redirection modifications. In this example, the predictors are placed in the decoder stage, because there is no PC redirection penalty in this stage. The updated data comes from the execution stage. It can not be done earlier because to compute the condition of a conditional branch or to know the destination for jumps, we need a value from a register. This data will be available at the earliest on the execution stage.

2.3.3 First Level Caches

Accessing to main memory is needed to fetch each instruction and for every memory-access instruction, e.g., Load and Store. However, the latency of the main memory is considerable and has increased relative to the register delay in the last few decades [24]. Cache memories exploit temporal and spatial locality in memory accesses to reduce the average latency and energy in accessing data from the main memory. Cache memory is a faster, smaller memory that is located closer to a processor core. It stores copies of the data from frequently used main memory locations. The cache can store a small amount of memory, although the amount of memory cells needed is still very big. To be implementable and not consume a lot of area and power, caches are implemented using SRAM. Since SRAMs have a noticeable delay, caches can have multicycle access latencies to achieve higher frequencies.

Caches designs revolve around several different properties. The first one is associativity, which is the number of entries that can be mapped to a single address of the main memory. It can go from a direct-mapped cache, where each block of data of the main memory can only go to a determinate entry, to a fully associated cache, where each main memory block can go on any cache set. In the middle, there is N-way associativity where the main memory block can get to N entries.

Secondly, it is the replacement policy. Since more than one memory block is mapped to the same cache entry, in some cases, it has to evict some data located inside the cache to store the new one. There are different policies to do that replacement of data inside the cache (e.g. random, least recently used, FIFO, etc.).

Finally, nearly every processor uses the concept of virtual memory. Virtual memory is a memory management technique, which creates an illusion of an immense main memory for the users. It is essential for modern software development to abstract over different memory sizes in machines.

Implementing this technique requires a translation mechanism from virtual to physical memory addresses that are usually done in the Memory Management unit (MMU). Some parts of the MMU are located in the caches. Thus, caches have to deal with physical or virtual depending on the implementation chosen. There are different types of caches regarding if the index and tags correspond to physical or virtual addresses:

- Physically indexed, physically tagged (PIPT) cache design: it uses the physical address for both the index and the tag. It is simple and avoids problems with aliasing. However, it is slow because the translation from virtual address to physical address has to be done before the cache access.
- Virtually indexed, virtually tagged (VIVT) cache design: it uses the virtual address for both the index and the tag. This scheme can result in much faster lookups as address translation is not needed. However, it can suffer from some aliasing as different virtual addresses may refer to the same physical address.
- Virtually indexed, physically tagged (VIPT) cache design: it uses the virtual address to index the data and tag array and uses the physical address for the tag comparison. The advantage of this design is that it has lower latency than PIPT caches as the cache line can be accessed in parallel with the Translation Lookaside Buffer (TLB) translation. However, the tag cannot be compared until the physical address is available. Finally, the cache size has to be equal or lower than the page size multiplied by the cache associativity to avoid aliasing.
- Physically indexed, virtually tagged (PIVT) cache design: it is often claimed in literature to be useless and non-existing.

In this thesis, we will consider VIPT designs for first level caches since it has lower latency and prevents aliasing as we have explained in the previous paragraphs.

2.4 Hardware Description Language

In the computer engineering field, a hardware description language (HDL) is used to describe the structure and behavior of digital logic circuits. HDL gives a precise, formal description of a digital circuit that enables an automated analysis and simulation. Also, a circuit described with HDL can be synthesized into a netlist (a specification of physical, electronic components, and the connecting rooting of the components), which can then be placed & routed to produce the set of masks used to manufacture the integrated circuit.

2.4.1 SystemVerilog

SystemVerilog [31], standardized as IEEE 1800, aims to provide a well-defined and official IEEE unified hardware design, specification, and verification standard language. It is commonly used in the semiconductor and electronic design industry as an evolution of Verilog. The language is designed to coexist and enhance the hardware description and verification languages (HD-VLs) currently used by designers while providing the capabilities lacking in those languages. SystemVerilog enables the use of a unified language for abstract and detailed specification of the design, specification of assertions, coverage, and testbench verification based on manual or automatic methodologies.

2.4.2 Chisel

Chisel [8] is an HDL that allows advanced circuit design and generation. The same design can be reused for both ASIC and FPGA implementations.

Chisel adds hardware construction primitives to the Scala programming language. It gives the power of a modern programming language to the computer architects that enables designing complex, parameterizable circuit generators that produce synthesizable Verilog. This generator methodology allows the conception of re-usable components and libraries, increasing abstraction in design while maintaining fine-grained control.

Chisel is powered by FIRRTL (Flexible Intermediate Representation for RTL), a hardware compiler framework that performs optimizations of Chisel-generated circuits and supports custom user-defined circuit transformations.

2.4.3 Bluespec

As Bluespec is the HDL used in this thesis, we want to take a more in-depth look. Bluespec is an Event Driven Language (EDL) toolset for ASIC and FPGA design, which offers significantly higher productivity via a radically different approach to high-level synthesis. Bluespec has two different syntaxes that are interchangeable: Bluespec SystemVerilog (BSV) [23] and Bluespec

Haskell (BH, or "Bluespec Classic"). In this project, we decided to use BSV since it is more similar to modern HDL. BSV is based on guarded atomic rules and most of its syntax is inherited from SystemVerilog. Its toolchain has been developed by Bluespec Inc. and used in industry for almost 20 years. It has also been proven in production designs like Flute [16] and Piccolo [17].

The Bluespec compiler (BSC) emits standard Verilog for maximum compatibility with any synthesis toolchain and comes with an included simulator ("bluesim"), standard library, and TCL scripting support ("bluetcl"). The BSC has an open source [15] version that includes all the necessary tools for simulation and synthesis.

BSV has some similarities to object-oriented programming languages such as C++ and Java. BSV modules are like objects. These modules can be manipulated only by its interface methods preventing the manipulation of the internals. This allows us to change a module for another with different functionality. It can be done without further modifications in the code if both are using the same interface. The other attractive property of BSV is guarded atomic actions, also known as rules. A method can be invoked by a rule or another method. Rules and methods can only be executed if all its implicit guards (execution condition) and the guards of all the actions inside them are accomplished. This directly implies that every action executed has to be able to execute and commit all the actions that are called inside it. It is as if either all of the methods inside an action were applied successfully or none were applied. Parallelism is achieved through concurrent execution of non-conflicting rules. To sum up these concepts, we will use a 32-bit divider implementation. The code of *Div* interface is shown in Listing 2.1. It has two methods, *start* and *getResult* that can be used from another method or with a rule.

```
interface Div;
    method Action start(Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
```

Listing 2.1: Divider interface in Bluespec

A possible implementation of the divider is shown in Listing 2.2. We implement the famous binary long division where N is divided by D, placing the quotient in Q and the remainder in R. Inside the module, there is the implementation of the two methods of the interface and a rule, which is responsible for implementing the divider functionality. The method *start* saves the variables N and D into registers and sets the variables to the initial state. This method has a guard that depends on the variable busy. If busy is false, the divider is ready to be used, and it is true while the divider is performing a division. Method *getResult* gets the result and returns the resulting value and clears the variable busy if the division is done. The division is done when i is equal to zero and is still busy (the result is not given yet). Finally, the actual implementation of the divider is done in the rule *doDiv*, which is computing the result iteratively. It is only active when i is not zero, meaning that the division is started and not finished.

```
module mkDiv (GCD);
    Reg#(Bit#(32)) Q <- mkReg(0);
    Reg#(Bit#(32)) R <- mkReg(0);
    Reg#(Bit#(32)) D <- mkReg(0);
    Reg#(Bit#(32)) N <- mkReg(0);
    Reg#(Bit#(6)) i <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);
    rule doDiv (i != 0);
        let r = R << 1;
        r[0] = N[i-1];
        if(r >= D) begin
            R <= r - D;
            Q[i-1] <= 1;
        end else begin
            R = r;
        end
        i <= i - 1;
    endrule
    method Action start(Bit#(32) n, Bit#(32) d) if(!busy);
        i <= 32;
        N <= n;
        D <= d;
        Q <= 0;
        R <= 0;
        busy <= True;
    endmethod
    method ActionValue#(Bit#(32)) getResult if(busy && i==0);
        busy <= False;
        return Q;
    endmethod
endmodule
```

Listing 2.2: Divider algorithm implementation in Bluespec

Next, let's assume that the divider *mkDiv* is placed into a pipeline. In this pipeline, there is one process using the *start* method to launch a division and another process that is collecting the results. These processes will be implemented as rules. The implementation of the divider *mkDiv* does not allow concurrency divisions where more than one division starts before producing the end result. In order to improve the throughput of the pipeline, it is necessary to improve the divider to allow concurrent executions. To do that, we can duplicate the logic to allow two concurrent divisions. In Listing 2.3 the code of this divider is shown. In this case, there are two instances of *mkDiv* with the necessary logic to run in parallel. This new *mkTwoDiv* divider is using the same interface. Thus, it can replace the old one in the pipeline without any further change.

```
module mkTwoDiv(Div);
    Div div1 <- mkDiv;
    Div div2 <- mkDiv;
    Reg#(Bool) inTurn <- mkReg(True);
    Reg#(Bool) outTurn <- mkReg(True);

    method Action start(Bit#(32) a, Bit#(32) b);
        if(inTurn) begin
            div1.start(a,b); inTurn <= !inTurn;
        end else begin
            div2.start(a,b); inTurn <= !inTurn;
        end
    endmethod

    method ActionValue#(Bit#(32)) getResult;
```

```
        let res = ?;
        if(outTurn) begin
            res <- div1.getResult; outTurn <= !outTurn;
        end else begin
            res <- div2.getResult; outTurn <= !outTurn;
        end
        return res;
    endmethod
endmodule
```

Listing 2.3: Divider with two concurrent divider cores

An alternative to increase the throughput consists in implementing a divider that takes fewer cycles to compute a division. It is worth mentioning that with this faster divider, the interface will be the same as the action that is using the methods does not know how many cycles a division takes. Thus, it will be blocked until all the guards of the methods used are ready.

Ephemeral History Register (EHR) Registers are updated at the end of the clock cycle. Thus, it can not be used to share data between rules in the same clock cycle. Since two rules can fire in different cycles because the guard conditions can be different, a wire can not be used for sharing data either. To properly schedule and share data between rules, Bluespec uses the Ephemeral History Register(EHR) [27]. Using EHR, it is possible to select the scheduling of different rules running concurrently. A two-port EHR is shown in Figure 2.16. EHR has several writing and reading ports. A EHR of N ports has N reading ports and N writing ports. Each port is ordered using its number, where the smaller port is the port number 0 and the larger port is the port number N-1. The reading and writing priorities are specified in the Equation 2.3 where $r[i]$ is a reading port, $w[i]$ is a writing port and $<$, meaning that the action performed on the port on the left is done earlier than the action performed on the right port. Then, in this particular example:

- The read is performed before the write on the same port (as a normal register)
- A reading port will return the value stored into the last cycle if any lower port is written during the actual cycle.
- A reading port will return the value of the biggest written port on this cycle. The write port has to be smaller than the reading port.
- The next cycle value will be the value written by the biggest port.

$$\begin{cases} r[i] < w[i] \\ w[i] < r[j] \qquad \forall\, i < j \\ w[i] < w[j] \end{cases} \tag{2.3}$$

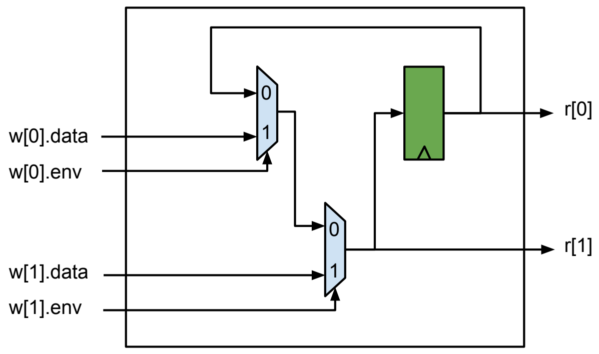


Figure 2.16: Internals of a two port EHR

2.5 RTL Simulation, FPGA Emulation and Gate Level Simulation

In processor design, it is essential to test the design constantly to find errors and to evaluate the performance implications of the applied modifications. Thus, it is crucial to execute pieces of code and programs during the implementation of a design. There are different ways to run a program on top of a processor in the implementation state. Each strategy has its properties in terms of simulation speed, the information provided, and the effort needed.

In the early stages, the most commonly used strategy is RTL simulation. RTL simulation is the most straightforward way to simulate a design, since the only necessary thing is the RTL design. A computer performs this type of simulation. It makes a model of the completed design modeling the registers and wires as variables. Also, it can give the cycle state of each register for debugging purposes. It is a cycle-accurate simulation, but it is impossible to compute the sub-cycle delays, since the RTL simulation tool does not have a technology model. RTL simulations are in general quite slow, since the computer has to calculate the value of all the registers and wires one by one for each cycle. Thus, it is not suitable for simulating the execution of large programs. If the state of the internal wires is not essential, it is possible to have a higher-level model of the design that still models the clock cycles accurately with faster simulation speed, meanwhile losing the ability to see the internal wires.

Sometimes, it is necessary to emulate the processor design on an FPGA to get a large program simulation. FPGAs can give a really good emulation performance by literally mapping the circuit of the design on it. FPGAs can achieve tens of Megahertz, which can be hundreds or thousands of times faster than RTL simulation. However, more effort is needed to obtain an FPGA emulation running. It is due to the necessity of using hard IPs for external communication and to obtain resulting data. Also, it is not possible to know the value of all the wires or registers cycle by cycle because the FPGA does not have enough outputs. Therefore, possible errors are hard to find and solve. Apart from that, the FPGA synthesis of the design is quite slow, and for small runs, the gain on simulation speed with respect to the RTL simulation is not worth it.

Finally, 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 the most accurate one, including timing information. It uses the module of the technology standard cells and hard IPs. It can detect paths that do not meet the required timing constraints, find glitches, and find possible propagation of X values (e.g. uninitialized registers). In Table 2.1, there is a summary of all the different simulations and emulations approaches, as well as 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

3 Related Work

Nowadays, with the help of RISC-V and its open-source nature, universities and research centers are starting to develop open-source processors and platforms. The researchers are implementing from simple power-efficient in-order cores to very complex superscalar out-of-order solutions. Also, researchers are implementing SoCs for multi-core architectures, even NoC for many-core solutions. This open-source environment is pushing computer architecture research into a promising future. In this section, we will describe the different solutions proposed by the different research centers and universities. We first introduce the RISC-V ISA and then survey some of its implementations.

3.1 RISC-V ISA

RISC-V is an open ISA that is freely available to academia and industry. RISC-V is a new ISA that was initially designed to support computer architecture research and education, but it is sophisticated enough to also become a standard in industry as well. It is an ISA for direct native hardware implementations. It is not targeting a particular microarchitecture style or an implementation technology. Although it is not specifically optimized for a particular microarchitecture, it can be efficient in any of these.

The RISC-V ISA is defined as avoiding implementation details as much as possible (although the commentary is included in implementation-driven decisions). It should be read as the software-visible interface to a wide variety of implementations, rather than the design of a particular hardware artifact. The RISC-V manual is structured in two volumes. There is the unprivileged volume [33] that covers the design of the base unprivileged instructions, including optional unprivileged ISA extensions. Unprivileged instructions are those that are generally used in all privilege modes in all privileged architectures, though behavior might vary depending on privilege mode and privilege architecture. The second volume provides [34] the design of the privileged architecture.

The unprivileged ISA it can be separated in 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.

The implementation of these extensions introduces some changes in the structure of the processors. All the extensions affect the decoder (it needs to decode the new instructions) and the back-end, adding new processing units. For example, the F and D extensions need specific

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

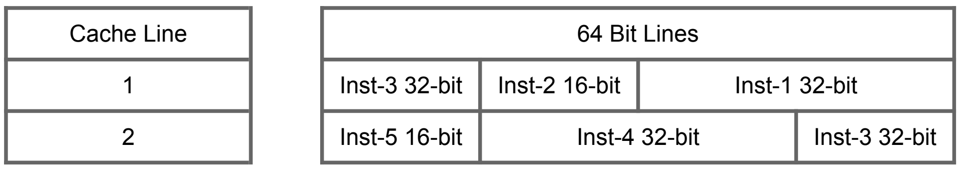


Figure 3.1: Misaligment example of a 32-bit instruction using RISC-V C extension

circuitry to perform the floating-point operations.

However, the only extension that introduces significant changes in the front-end is the C extension. This extension adds the 16-bit instructions, which misaligns the 32-bit instructions. Thus, the branch predictors and the fetch mechanism have to take into account this factor. Also, the inclusion of 16-bit instructions can misalign 32-bit instructions and split it into two different words or cache lines. In Figure 3.1, there is an example of two consecutive instruction cache lines where the instruction number 3, a 32-bits instruction, is split into both lines. It needs to be taken into account for the design of the fetch mechanism.

3.2 In-Order Designs

In-order designs are commonly chosen for academia since the complexity is not very high. It can be quickly implemented and verified thanks to its simple nature. We want to present some in-order designs to see what academia is achieving in this type of design.

3.2.1 Ariane Core

Ariane [36] (now known as CORE-V CVA6) is a 6-stage, single issue, in-order Central Processing Units (CPU), which implements the 64-bit RISC-V instruction set. Also, it fully implements the ISA extensions M and C specified in User-Level ISA version 2.1, as well as the draft privilege extension 1.10. It fully implements the three privilege levels, machine, supervisor, and user, to fully support a Unix-like operating system.

Ariane core was designed for the propose of running a full operative system at a reasonable speed and IPC. To achieve this necessary performance, Ariane has a 6-stage pipelined design. It has two fetch stages, two decoding stages, one stage of execution (although there are paths that can take more than one cycle, such as the divider), and a commit stage. The first-level instruction cache is a 4-way associative and data cache is 8-way associative. Instruction-cache and data cache have 1-cycle and 3-cycle hit latencies, respectively.

Ariane has been fabricated using the 22nm Fully Depleted Silicon On Insulator (FD-SOI) node of GlobalFoundries. The design was signed-off with a 902MHz worst case at 0.72 V, 125 °C and SSG. Nevertheless, it can achieve a clock frequency of 1.7 GHz with an operational voltage of 1.15V and using the body-biasing technique. The final netlist contains 75.34% LVT (low voltage threshold) and 24.66% SLVT (super low voltage threshold) cells.

The pipeline of Ariane is shown in Figure 3.2. In the first stage, the PC is computed. It can be selected form the branch predictors, branch misprediction module, or the exception PC from Control and Status Register (CSR). Also, the access to the instruction cache is done in the first cycle. Since the instruction cache is VIPT, the instruction TLB is also accessed in parallel. Taking into account that the instruction cache takes one cycle, it needs to be registered at the output to avoid long critical paths. This modification has the side-effect that even on a correct control flow prediction, it loses a cycle, as it can not calculate the next PC in the same cycle that it receives the data from the instruction cache. With the additional compressed instruction set extension, this is usually not a problem as (with a fetch width of 32 bit) it is fetching 1.5 instructions on average, and approximately 70% of all instructions are compressed. At the second stage, a pre-decoding is applied to the instruction to get the information needed for the branch predictors. Finally, the instruction is enqueued to an instruction queue. Also, in this stage, there is the logic for compressed instructions and the request of the next line if a 32-bit instruction is unaligned between two different blocks.

In the decoding stage, there is the logic to re-align the instruction with the decoders for 32 and 16-bit instructions. The issuing state performs the lookup and destination push of the scoreboard along with the register file reading. Ariane has an out of order write-back, since it has several functional units that can run in parallel. It is done like that since some of the functional units take several cycles. Thus, there is a small Re-Order Buffer (ROB) to handle the out-of-order commit. On the execution stage, it has an ALU, a multiplier unit, a Branch Predictor unit, a Floating Point Unit (FPU) and a Load Store Unit.

In the commit stage, write-back conflicts are resolved through the ROB. The commit stage reads from the ROB and commits all instructions in the program order. Stores and atomic memory operations are held in a store buffer until the commit stage confirms their architectural commitment. Finally, the register file is updated by the retiring instruction. To avoid artificial starvation because of a full ROB, the commit stage can commit two instructions per cycle. With this architecture, they are claiming a performance of 1.65 DMIPS/MHz depending on the branch-prediction configuration and the load latency (number of registers after the data cache).

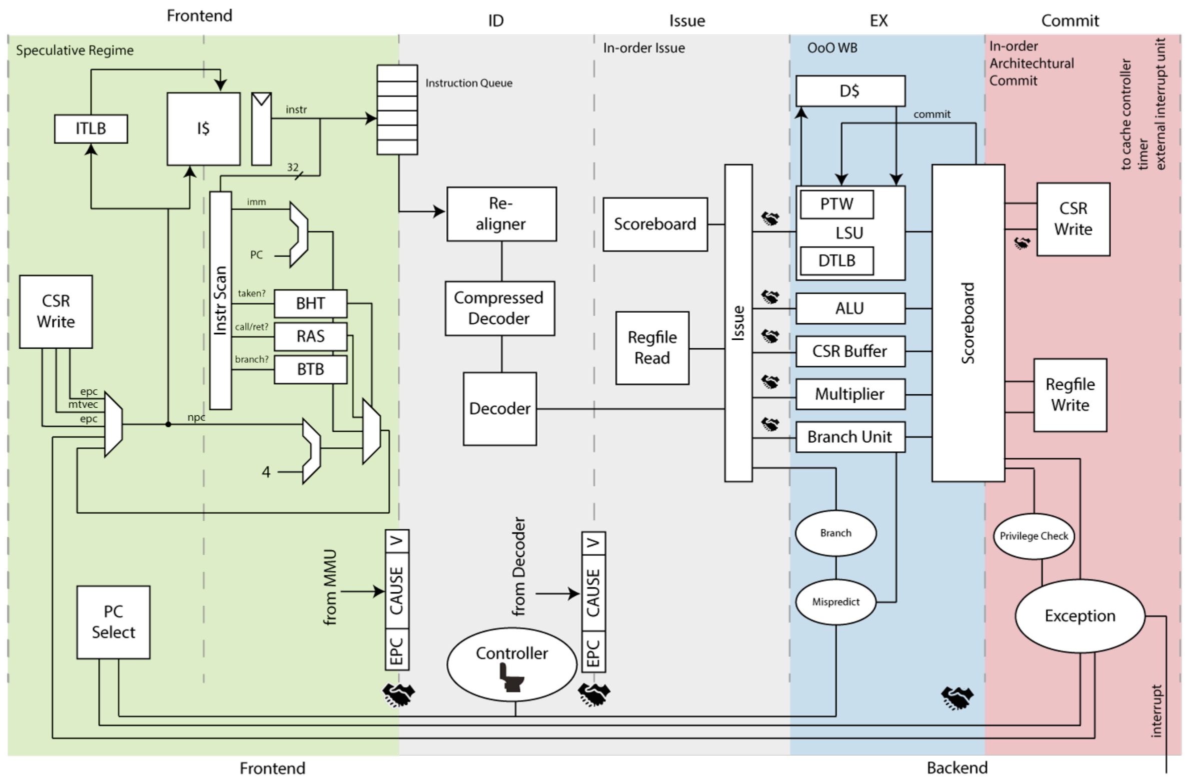


Figure 3.2: Blockdiagram of Ariane. Source: [36]

3.2.2 Rocket64 Core

In this section, we introduce the RISC-V Rocket core [2], a microarchitecture developed by the Berkeley Architecture group. Rocket is a 5-stage in-order scalar core generator. Rocket implements the RV32G and RV64G ISAs, depending on the configuration chosen. It has a front-end with several branch predictors, MMU that supports page-based virtual memory along with a non-blocking data cache. Rocket supports different privilege levels. Machine mode, supervisor mode, and user mode are supported. The Rocket chip generator is itself written in Chisel, although it finally produces synthesizable Verilog RTL.

Rocket generator has an enormous configurability, a large number of parameters are exposed. The optional support of some ISA extensions such as M, A, F, and D can be individually selected. Different branch predictors such as BTB, BHT, and RAS are provided. Also, the caches and TLB sizes can be configured, as well as the number of floating-point pipeline stages. The implementation of the floating-point pipeline makes use of Berkeley's Chisel implementations.

Moreover, Rocket can be considered as a library of processor components. Several modules that were initially designed for the Rocket generator are also used in other designs. This includes the TLBs, page table walker, instruction cache, non-blocking data cache, some modules for the implementation of the privileged level (i.e., the CSR file).

The Rocket core is sometimes described as a 6-stage pipeline [18] with the addition of a *pcgen* stage. While it is useful to lay out the figure in this way, the stage is perhaps best considered as part of the other stages and is not a distinct pipeline stage in the traditional sense. The front

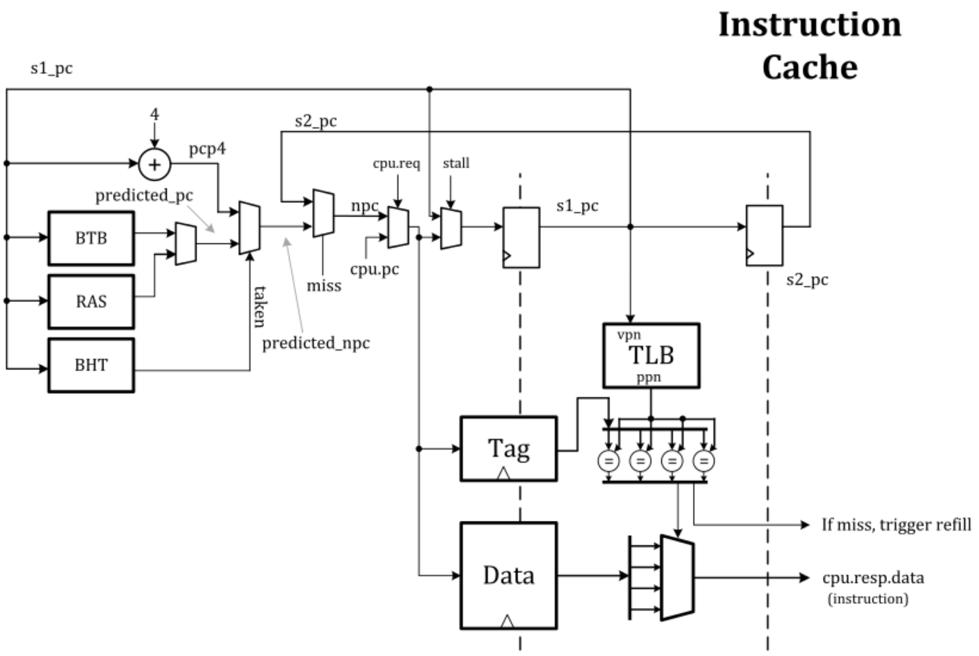


Figure 3.3: Blockdiagram of Rocked core frontkend. Source: [18]

end is shown in Figure 3.3. It has the PC generation with a BTB, BHT, and RAS predictors. Also, it has the Instruction cache and instruction TLB. Since the instruction cache is VIPT, the access of the TLB and the data arrays on the cache is performed in parallel to decrease the latency. The output of the cache is registered to eliminate a possible critical path through the cache SRAMs.

On the back-end there are 4 stages shown in Figure 3.4. There is a simple decoder on the decoding stage, since it is not implementing the C extension. Also, there is the Regfile reading along with the lookup and setting of the scoreboard. On the execution stage, there are multiple processing units as an ALU, a multiplication and division unit, and a branch unit in charge of the misprediction detection. Bypasses are implemented in this stage too. In this design, there is a memory stage, since it does not have out-of-order write-back. This stage allows us to mask the access time penalty of the data cache. This data cache has a load and store queue, allowing non-blocking cache execution. Finally, in the commit stage, the data is written-back on the register. Apart from this, there is the Rocket Custom Co-processor Interface (ROCC). ROCC interface facilitates communication with coprocessors/accelerators. Such accelerators include crypto units (e.g., SHA3) and vector processing units. The ROCC interface accepts coprocessor commands that are emitted by committed instructions to run on the Control Processor. Any ROCC command will be executed by the coprocessor (barring exceptions thrown by the coprocessor); nothing speculative can be issued over ROCC.

This core has been taped out [20] along the 64-bit Hwacha vector accelerator using a TSMC40GPLUS process. It claims a performance of 1.72 DMIPS/MHz and a maximum frequency of 1.3 GHz when it is running with an operational voltage of 1.2V, at which point the SRAM array becomes the speed-limiting factor.

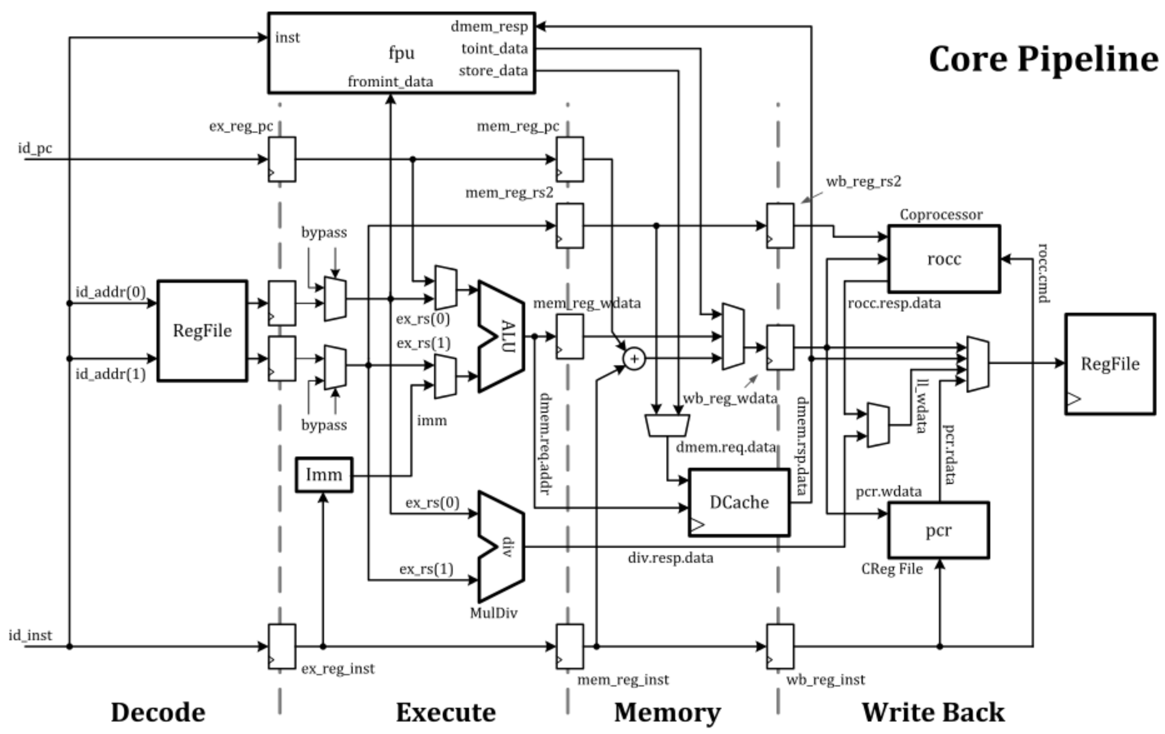


Figure 3.4: Blockdiagram of Rocked core backend. Source: [18]

3.2.3 DRAC Core

DRAC is RISC-V general purpose processor capable of booting Linux jointly developed by the Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC), Centro de Investigación en Computación del Instituto Politécnico Nacional de México (CIC-IPN), Centre Nacional de Microelectrònica (CNM), and Universitat Politècnica de Catalunya (UPC).

The design incorporates a 5-stage single-issue in-order Lagarto pipeline that implements the 64-bit RV64IMA scalar RISC-V ISA, as well as the associated instruction and data caches, a unified L2 cache and the peripherals required to connect the processor with external devices such as main memory, JTAG, UART, and an SD card. Lagarto I [26] is a 64-bit in-order single-issue scalar core based on RISC-V ISA. The design was taped-out on TSMC 65nm technology node with a base frequency of 200 MHz.

Five pipeline stages compose the design: fetch, decode, read register, execution/memory-access, and write-back. The pipeline is shown in Figure 3.5. In the fetch stage, there is the PC generation with the BTB, BHT, and RAS predictors. In the decoder, the stage is performed the instruction decode and the JAL redirection. The third stage is used for the reading of the register file. Next, in the Execution stage there are the ALU, the data cache requests, and the branch unit are responsible for correcting the branch mispredictions. The last stage is the write-back.

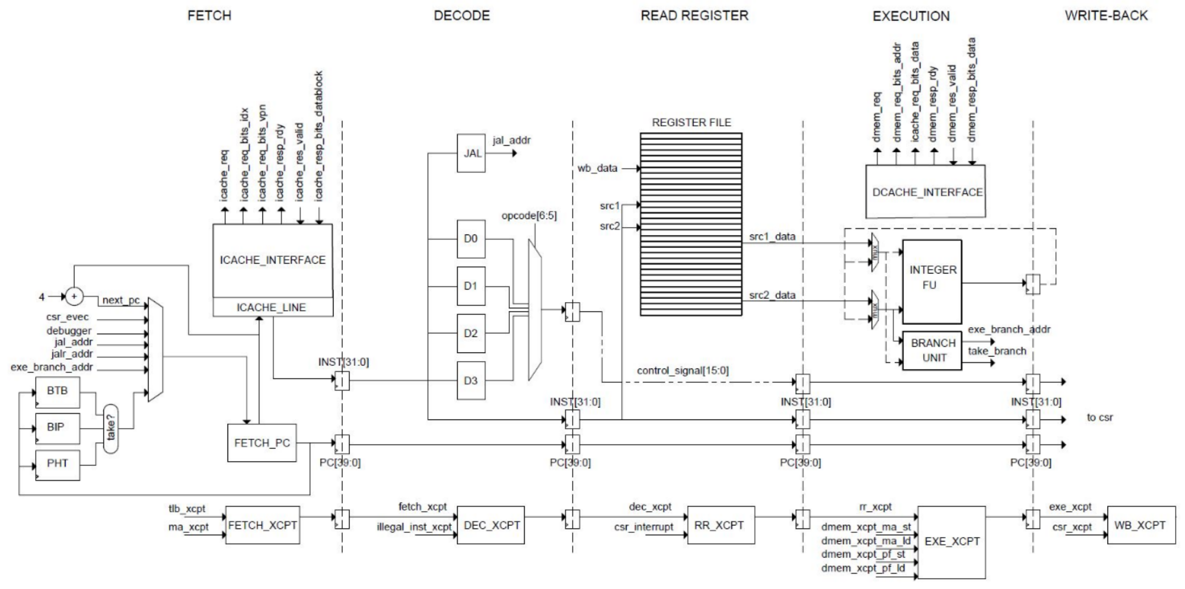


Figure 3.5: Lagarto pipeline

3.3 Out-of-Order Designs

Since in-order cores have limited IPC performance, some universities and laboratories have implemented more complex out-of-order designs. These designs are much more complex to implement and even harder to verify. However, there are outstanding designs which we introduce next.

3.3.1 RiscyOO

RiscyOO [37] is a RISC-V out-of-order core designed in the MIT Computer Science & Artificial Intelligence Lab. It uses a framework called Composable Modular Design (CMD). CMD has the following main properties:

- The interface methods of modules provide instantaneous access and perform atomic updates to the state elements inside the module.
- Every interface method is guarded, i.e., it cannot be applied unless it is ready.
- Modules are composed together by atomic rules, which call interface methods of different modules. A rule either successfully updates the state of all the called modules, or it does nothing.

CMD designs are described using Bluespec SystemVerilog and are compiled into RTL, which can be run on Field-Programmable Gate Array (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.

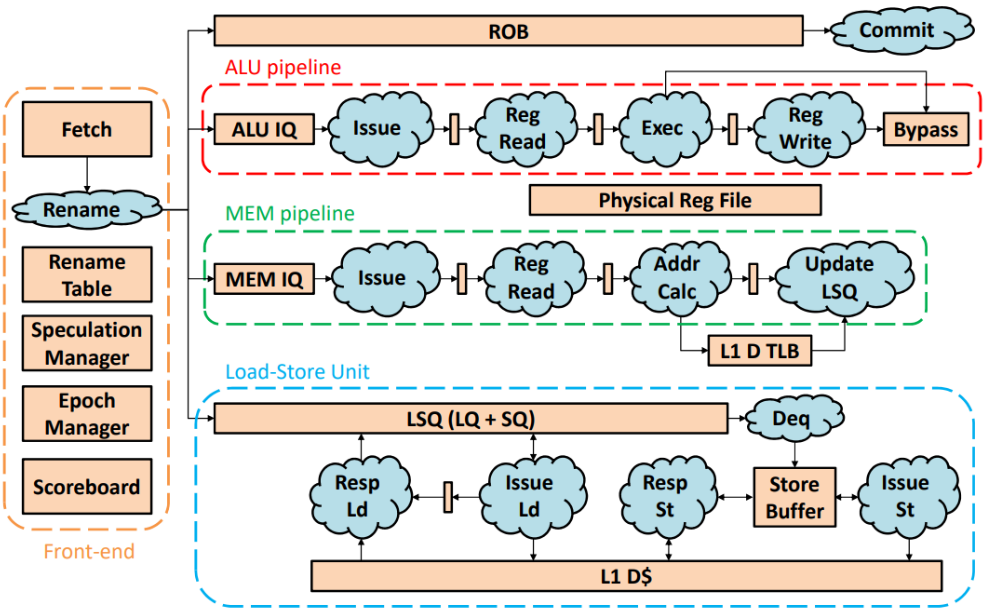


Figure 3.6: RiscyOO structure diagram

RiscyOO boots Linux and runs on FPGAs at a frequency of 25 MHz to 40 MHz. The overview of the design is shown in Figure 3.6. It can also be synthesized into an ASIC target with several variants of it in a 32 nm technology to run at 1 GHz to 1.1 GHz. RiscyOO has been evaluated in the FPGA environment. The performance evaluation shows that it beats in-order processors in terms of IPC, but will require more architectural work to compete with wider superscalar commercial ARM processors. However, thanks to 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.

3.3.2 Berkeley Out-of-Order Machine (BOOM)

The Berkeley Out-of-Order Machine (BOOM) is an open-source RV64GC RISC-V core written in the Chisel. BOOM is synthesizable and parameterizable. It is designed at the University of California, Berkeley in the Berkeley Architecture Research group, its focus is to create a high performance, synthesizable, and parameterizable core for architecture research. Like most contemporary high-performance cores, BOOM is superscalar and out-of-order. It uses the Rocket generator as a library of components, allowing to use all the cache hierarchy and uncore (controllers, I/O) to quickly bring up an entire multi-core processor system able to boot Linux.

There are three versions of the BOOM core. There is a diagram of the three versions in Figure 3.7. The first one, and also the oldest, is BOOMv1 [6]. 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). In this technology, the core is achieving a maximum frequency of 1.5 GHz, the same as the in-order core Rocket, as both have the same critical path on the SRAM memories.

The second generation was named BOOMv2 [7]. This generation is based on the information

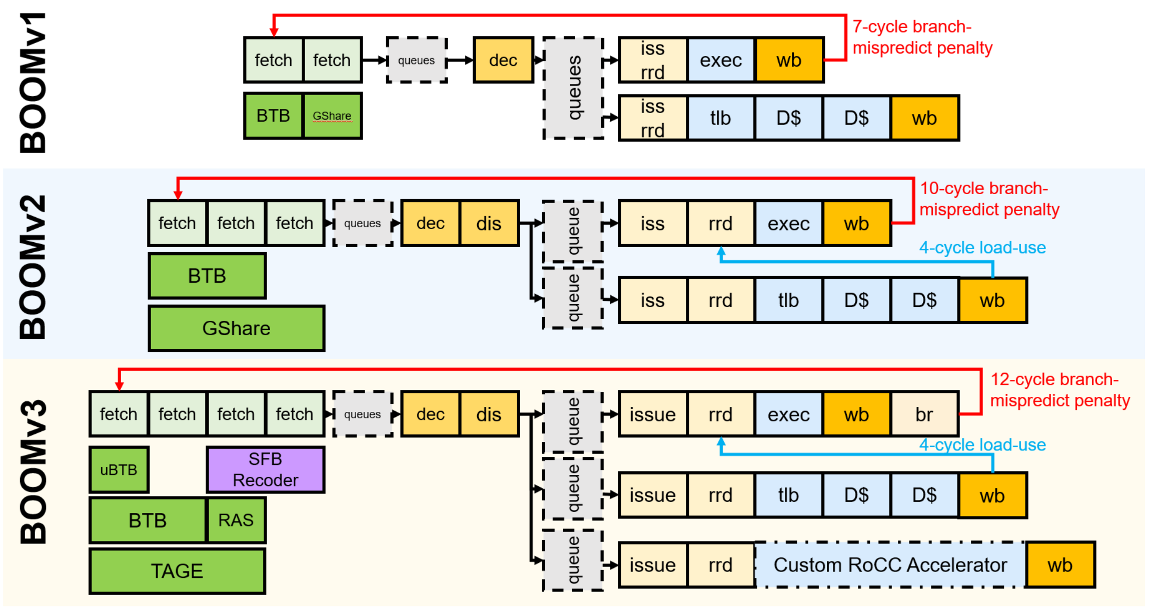


Figure 3.7: Evolution of the BOOM pipeline

collected through synthesis, place, and route using a commercial TSMC 28 nm process of the first generation. BOOMv2 has a redesigned larger front-end with a modified version of branch predictors. Also, it has a new custom register file to reduce the size and access time. Finally, it includes a distributed issue queue that reduces the critical path while increasing the number of instructions that can be scheduled to respect the original unified issue queue. BOOMv2 reduces the total fanout-of-four by 24%, but it has a reduction in the IPC of 20%. This IPC reduction can be recovered by tuning the processor parameters, such as the number of BTB entries or the register renaming's pipeline depth. These modifications can increase the IPC in exchange for an increased clock period.

The current version of the BOOM microarchitecture (SonicBOOM, or BOOMv3) [38] is performance competitive with commercial high-performance out-of-order cores, achieving 6.2 CoreMark/MHz, 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 adds support to RISC-V C extension, modifying all the front-end and the branch predictors' structure. The most significant contribution to overall core performance is the inclusion of a high-performance TAGE [29] branch predictor, with a speculatively updated and repaired global-history vector driving predictions. It is synthesized at 1GHz with the same TSMC 28 nm process as BOOMv2.

3.4 SoCs and NoCs

Nowadays, academia is also involved in the design of SoCs, NOCs, and accelerators part of cores and processors. These SoCs and NoCs are usually used in other open-source projects increasing the open hardware community.

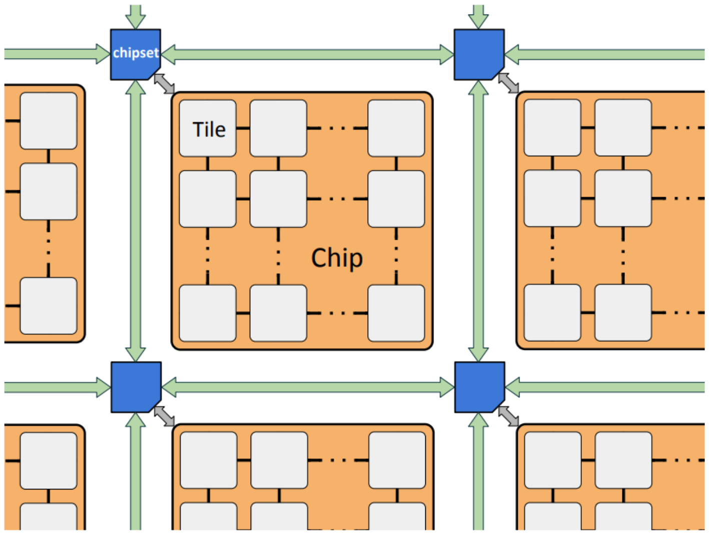


Figure 3.8: OpenPiton Architecture. Multiple manycore chips are connected together with chipset logic and networks to build large scalable manycore systems. OpenPiton's cache coherence protocol extends off chip. Source: [3]

3.4.1 lowRISC SoC

lowRISC [22] SoC is a 64-bit Rocket-based SoC design. It offers an FPGA-ready SoC distribution, with open source peripherals such as SD and Ethernet, and documentation and tutorials. It can be used as a testbed for new ideas, including tagged memory and minion cores. The project aims to deliver complete design to an equivalent quality standard as similar proprietary, closed IP. The most recent version is 0.6v. Its implementation uses the RISC-V Rocket Core. It implements the RISC-V Privilege specifications version 1.10. It is an untethered design (i.e.,it can be run without a host) with the support of an SD card, UART console in the standard and VGA variants, a PS/2 keyboard controller, and it can execute Debian binaries among other features.

3.4.2 OpenPiton NoC

The OpenPiton [3] platform is a modern, tiled, many-core design. It is designed by Wentzlaff Parallel Research Group at Princeton University. Piton Project is composed of multiple open-source tools (hardware, firmware, and software), that allows users to build the OpenPiton processor. Open Piton uses the industry hardened OpenSPARC T1 [30] core, but can be replaced for other designs. OpenPiton has a distributed, directory-based cache coherence protocol (shared distributed L2) implemented across three physical, 2D mesh Networks-on-Chip (NoCs).

Along with the hardware description, the project has a new and modern simulation and synthesis framework, a modern set of FPGA scripts, a complete set of ASIC back-end scripts enabling chip tape-out, and full-stack multiuser Debian Linux support. This design was taped-out on the IBM 32nm SOI process with a target clock frequency of 1GHz. The tape-out includes a 25-core implementation. The area of a single tile is 1.17mm2.

The OpenPiton processor is designed as a highly scalable tiled architecture. OpenPiton can scale both the numbers of tiles in a chip and the numbers of chips. Figure 3.8, a 2D mesh interconnects the tiles within the chip. This 2D mesh is capable of scaling up to 256 nodes per dimension. Thus, the maximum configuration of a chip is a matrix of 256x256 tiles, making a total of 64Kibit $(64 \cdot 2^{10})$ tiles per chip.

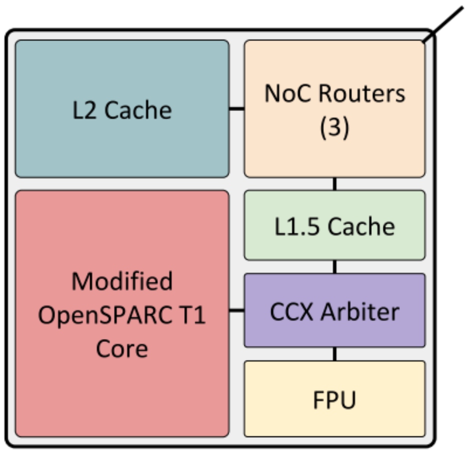


Figure 3.9: Arquitecture of a OpenPiton tile. Source: [3]

As mentioned, the default core used by the OpenPiton processor is an OpenSPARC T1 with some modifications. The original OpenSpark T1 is an octa-core design that uses write-through first level (L1) caches. To implement a coherent distributed system, OpenPiton integrates L1.5 caches that accept the write-through from the core. The tail, shown in Figure 3.9 have the modified core, a pipelined FPU, the L1.5 cache that implements the MESI coherency protocol, a module of the distributed L2 cache, and a three-port router that has communication with the L1.5 cache, L2 cache, and the tile 2D mesh network.

Multiple projects take advantage of OpenPiton using its open-source NoC. JuxtaPiton [21] enables Heterogeneous-ISA research implementing simultaneously two different types of tiles with PicoRV32 (RISC-V) and OpenSpark T1 (SPARC) on FPGA. Also, OpenPiton will release support for the Ariane RV64IMAC Core later on.

4 Experimental Environment

During this thesis, we make use of different tools and libraries. In this chapter we describe the environment used in the RTL design, the benchmarks for the IPC performance analysis, and the ASIC tool-flow.

4.1 RTL Environment

We have used Bluespec SystemVerilog to implement all our RTL designs. Bluespec SystemVerilog enables fast prototyping. It also gives the ability to change module implementation without modifying any other part of the design provided we maintain the interface.

In this project, we use Connectal [19] to avoid dealing with all the implementation of the different simulation environments. Connectal can generate a Bluesim simulation (behavioral simulation), a Verilog RTL code with the cycle-accurate simulation structure for Verilator [32] (an open-source C++ based RTL simulator), or the Verilog code with Hard IP blocks for FPGA synthesis using Xilinx proprietary tools.

4.2 Benchmarks

It is not easy to compare two processor designs by merely looking at their specifications and architecture. Also, sometimes it is difficult to know the impact of a microarchitecture modification on the processor performance. For this reason, it is essential to have a mechanism to measure the processor performance after each microarchitectural decision. For example, increasing the pipeline of a processor by an additional stage can reduce the critical path and, as a consequence, increase the maximum clock frequency achievable by the processor. However, we need to know the impact on IPC that this modification has. Using specific benchmarks, we can get this information and decide if the extra stage is worth it.

Benchmarks are designed to mimic a particular type of workload on a component or system. Benchmarks extract the critical algorithms of an application, containing the performance-sensitive aspects of that application. There are two main types of benchmarks: i) synthetic benchmarks, specially created programs to stress various components of the architecture, and ii) application benchmarks, which run real-world programs on the system. The application benchmarks have a better representation of real-world problems. Thus, it can provide an accurate view on the performance of the processor for a class of programs. Synthetic benchmarks are useful for testing individual processor parts since they can be focused only on some specific operations.

In this section, we will describe the benchmarks used in our project to evaluate different tradeoffs of our processor design. We have used EEMBC CoreMark, a compact benchmark, to quickly evaluate the performance impact of the different modifications made to the processor during this thesis. We have also used the EEMBC AutoBench suite to perform a more robust performance evaluation of the final design.

4.2.1 CoreMark

EEMBC's CoreMark [11] is a benchmark that measures the performance of Mcicrocontrollers (MCU) and CPUs used in embedded systems. It is intended to become an industry standard, replacing the Dhrystone benchmark [35]. CoreMark ties a performance indicator to the execution of a simple code, but rather than being entirely arbitrary and synthetic, the code for the benchmark uses basic data structures and algorithms that are common in practically any application. It uses the following commonly used 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 is designed to run on a wide range of devices from 8-bit microcontrollers to 64-bit microprocessors. CoreMark also sets specific rules about how to run the code and report results, thereby eliminating inconsistencies.

The CRC algorithm serves a dual function; it provides a workload commonly seen in embedded applications and ensures the CoreMark benchmark's correct operation, essentially providing a self-checking mechanism. Accurately, to verify the correct operation, a 16-bit CRC is performed on the data contained in elements of the linked-list.

To ensure that compilers cannot pre-compute the results at compile-time, every operation in the benchmark derives a value that is not available at compile time. Furthermore, all the code used within the timed portion of the benchmark is a part of the benchmark itself (no library calls).

This benchmark 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.

4.2.2 EEMBC AutoBench Performance Benchmark Suite

AutoBench 1.1 [10] is a suite of benchmarks created by EEMBC (Embedded Microprocessors Benchmarks) focused on the performance evaluation for microprocessors in automotive, industrial, and general-purpose applications. It has 16 benchmark kernels grouped into three different groups: generic workload tests, basic automotive algorithms, and signal processing algorithms. We will use this benchmark suite for the evaluation of the core performance for the different modifications. The test suite is composed of non-floating-point tests: Bit Manipulation (bitmnp), Cache "Buster" (cacheb), CAN Remote Data Request (canrdr), Finite Impulse Response (FIR) (aifirf), Pointer Chasing (pntrch), Pulse Width Modulation (PWM) (puwmod), Road Speed Calculation (rspeed), Table Lookup and Interpolation (tblook), Tooth to Spark (ttspark). Also, it is composed of floating-point benchmarks: Angle to Time Conversion (a2time), Basic Integer and Floating Point (basefp), Fast Fourier Transform (FFT) (aifftr), Inverse Discrete Cosine Transform (iDCT) (idctrn), Inverse Fast Fourier Transform (iFFT) (aiifft), Infinite Impulse Response (IIR) Filter (iirflt), and Matrix Arithmetic (matrix).

4.3 ASIC Tool-Flow Environment

This section describes the setup used for the ASIC synthesis and Place & Route phases. To do the ASIC synthesis, we use the Genus [4] tool from Cadence version 19.11-s087_1. For the Place & Route phases, we use the Innovus [5] version 19.11.000 also from Cadence.

4.3.1 Synthesis Input Files

The information needed in the synthesis phase is the following:

- RTL files (Verilog, SystemVerilog, or VHDL.) of the design
- Liberty files for standard cells and hard IP blocks. They contain timing and power information.
- LEF files for standard cells and hard IP blocks. They contain physical information as area and pin location. Optional in the synthesis phase, but they improve the quality of the netlist.
- Captable file. It contains information about interconnections parasitics (capacitance and resistance). Also optional for synthesis, but needed for a better quality of the generated netlist.
- SDC file. It describes the timing constraints: clock domains, the frequency of each clock signal, primary inputs, and outputs delay.
- Synthesis scripts in Tcl language with the specific tool commands.

4.3.2 Standard Cell Libraries

In this project, we make use of 22FDX technology, a 22nm Fully-Depleted Silicon-On-Insulator (FD-SOI) technology node from GlobalFoundries. There are several standard cell libraries provided by GlobalFoundries for the technology 22FDX as shown in Table 4.1.

- number of tracks: 12 tracks for high speed; 8 tracks for high density.
- CPP: 104
- Threshold voltage: low-Vt; ultralow-Vt for high speed.
- Corner: voltage, process, and temperature conditions.

Table 4.1: List of Standard Cell libraries in 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
12T	Low	Typical	0.8 V and 25°C
12T	Low	Fast	0.88 V and -40°C
12T	Low	Slow	0.72 V and 125°C
12T	SuperLow	Typical	0.8 V and 25°C
12T	SuperLow	Fast	0.88 V and -40°C
12T	SuperLow	Slow	0.72 V and 125°C

Libraries are provided for logic cells (*core cells*) and cells for low power techniques (*coarse grain cells*, e.g. level shifters, always-on buffers, etc.). For this project, only core cell libraries are used with these three corners: typical, best-case, and worst-case with NLDM models.

4.3.3 Hard IP Blocks

In this thesis, we used hard IP blocks for the SRAM arrays in the caches. The SRAM hard IP blocks are generated from memory compilers provided by Invecas or Synopsys. These compilers generate the IP blocks with all the needed information for synthesis: Liberty files for different corners, LEF files, and Verilog files (for RTL simulation). Table 4.2 lists the memory blocks used in the evaluated 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.

SRAM block name	Words	Width	Description
IN22FDX_R2PV_NFKG_W00064B064M02C128	64	64	1R1W Data Array
IN22FDX_R2PV_NFKG_W00064B052M02C128	64	52	1R1W Tag Array
IN22FDX_R2PV_NFKG_W00064B004M02C128	64	4	1R1W Metadata Array
sp_l_64x64	64	64	1RW Data Array; Low Vt
sp_l_64x52	64	52	1RW Tag Array; Low Vt
sp_ul_64x64	64	64	1RW Data Array; Ultralow Vt
sp_ul_64x52	64	52	1RW Tag Array; Ultralow Vt

5 Microarchitectural Design-Space Exploration of the Riscy Processor

The aim of this project is to analyze and improve the Riscy in-order core taking into account the ASIC synthesis results in 22FDX technology. First of all, an analysis of the architecture has to be made. This analysis will identify possible changes in the design to improve the IPC. Also, we can make a preliminary hypothesis of the potential critical paths on the design. However, it is necessary to do an ASIC target synthesis of the design to validate the critical paths hypothesis obtained in the previous analysis.

During this chapter, we mention different versions of the Riscy processor, depending on the modifications applied to it. The original version of the Riscy, clean of changes, is denoted *Riscy v1* throughout this thesis. Next, we start with an in-depth analysis of Riscy v1.

5.1 Original Riscy

Riscy in-order is a single issue 5-stage core. It implements the RV64I integer ISA along with the complete implementation of the G extension (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 BSV, taking advantage of the modularity and atomicity given by the properties of this language.

5.1.1 Pipeline Description

The 5-stage pipeline is shown in Figure 5.1. The first stage in the pipeline is the Fetch stage, which allocates the PC generation along with the instruction cache access. Riscy implements BHT and BTB branch predictors that improve the PC generation. The access to the branch predictors is performed in the fetch stage. However, the BHT result is used in the decode stage when it is known if the instruction is a branch or not.

The Decode stage is the second stage of the pipeline. It is a complex stage because it performs a large number of actions. The actions are performed in this sequence:

1. In this stage, the instructions cache's responses arrive as the cache access only has a one-cycle latency. In particular, the instruction cache response has a width of 64-bit. Since Riscy implements compressed instructions, the decoder stage has dedicated logic,

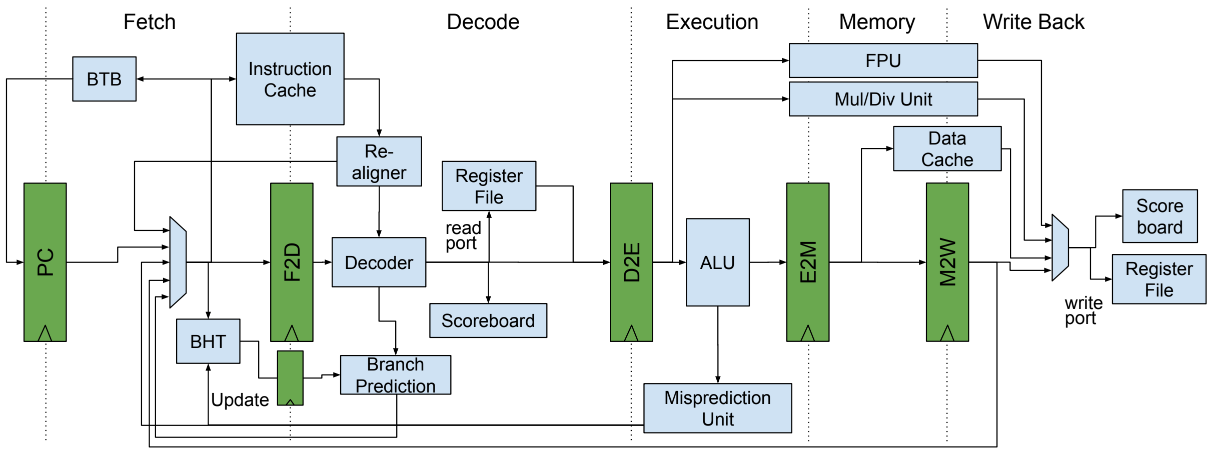


Figure 5.1: Diagram of the original Riscy v1 pipeline

named re-align module in the schematic, to select the instruction inside the cache response. Furthermore, this logic has to make a new instruction cache request if the instruction to be fetched is split into two instruction cache requests (shown in Fig 3.1).

2. When the instruction is selected from the instruction cache response, the decoding logic will unpack the information inside the instruction.

3. This information is used along with the BHT output generated in fetch state to generate a prediction for the next PC. It also redirects the PC if a JAL instruction is detected.

4. The decoded information is used to look up the data dependencies on the scoreboard. Also, the Register File is accessed in parallel. In this design, the Register File has a bypass logic to reduce the data hazard penalty in one cycle. If there is not any data dependency, the destination register is pushed in the scoreboard, and the instruction is moved to the Execution stage.

In the Execution stage, the Arithmetic Logic Unit (ALU) and the branch misprediction module are located along with the request of the FPU and multiplication and division unit. The ALU and the branch misprediction module takes only one execution cycle while the other units have a multiple cycle execution. However, the instruction jumps to the next stage, even with it uses the multicycle units.

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 is also located here. The exception information is propagated from the stage where it is generated to the Write-Back stage. Here, if an exception is raised, the commit is not performed (the Register write is not done), and the PC is redirected using the exception handler PC from the CSRs.

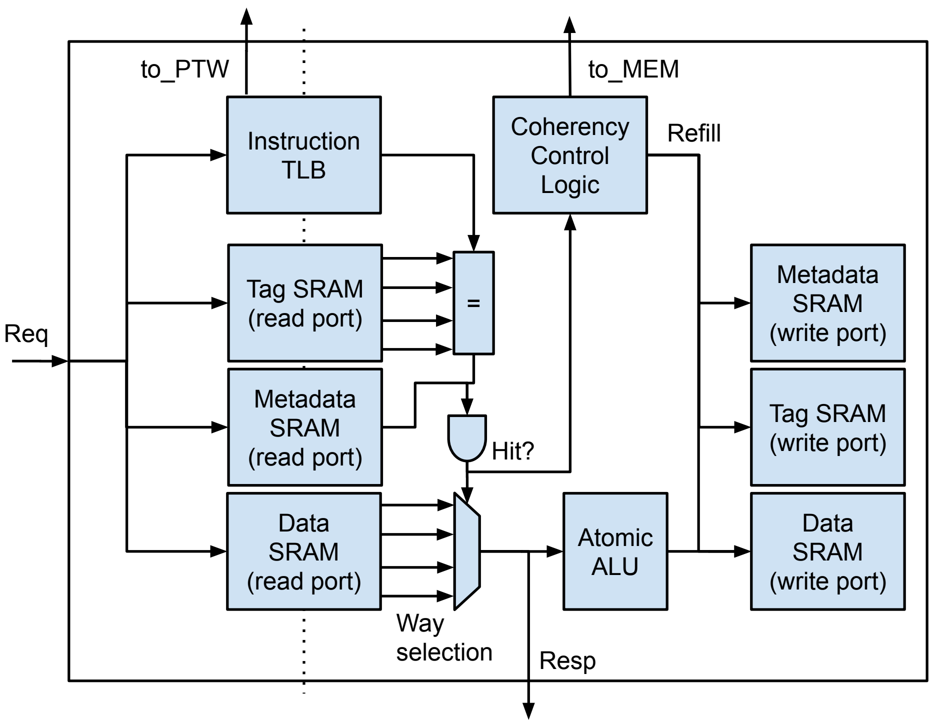


Figure 5.2: Block diagram of the Riscy's caches

Riscy v1 has a 32 cycle radix-4 booth multiplier and a 64 cycle divider. Since the CoreMark benchmark uses the multiplier intensively, we decide to replace the Booth multiplier with an emulated three-cycle pipelined multiplier. This permits us to identify other performance bottlenecks in the design and also provides a better comparison with other efficient implementations. The emulation of the three-cycles pipelined multiplier is done using the one-cycle Verilog multiplier and two empty pipelined stages. This one-cycle multiplier can be synthesized at higher frequencies, but the efficiency is quite poor. However, it can be used in this project as we do not focus on power efficiency and the area of the design.

In this design, the instruction cache and the data cache are identical except for minor differences in the memory translation mechanism. The data cache includes the PTW, which is also used by the instruction TLB. The caches are VIPT with one cycle of latency. In the basic configuration, the caches are 4-way set associative with 64 set for each way. The cache lines have a width of 512-bit, making a total of 16KBytes per cache. All the memory arrays inside the caches are implemented using SRAMs.

Figure 5.2 shows a block diagram of the one-cycle latency cache structure. Since we are not showing the PTW mechanism in this diagram, it can represent either the instruction cache or the data cache because they are identical. The data, tag, and metadata arrays are implemented using SRAMs. The request on the SRAMs and the access on the TLB is performed in the first cycle. On the second cycle, the SRAM gives the values asked the cycle before. The tags obtained are compared with the physical address given by the TLB.

5.1.2 Pipeline Implementation in BSV

The pipeline's implementation is mainly done using one rule per stage with all the functionality and a poison rule in charge of flushing the stage. Unlike the majority of processors, Riscy design does not have a centralized control logic for the pipeline. The control logic is distributed over all the rules. In this case, the execution of a stage starts if the actual state has data to process, and the following stage can accept the result, making a totally elastic pipeline. The registers between stages are composed of: a regular data register storing the data processed, an EHR with a valid control bit, and an EHR with a poison bit. Each stage rule has the same main guards:

- The valid bit of the actual state must be valid (the stage has data to process).
- The valid bit of the following stage must be invalid (the next stage has already processed all the data and is waiting).
- The Poison bit should be true (the instruction that is waiting to be processed has to be flush from the pipeline). This bit is set from older stages in the pipeline when the PC is redirected.

To achieve a correct pipeline scheduling of the rules, the valid bit EHR is written as it is shown in Listing 5.1. A stage uses the port number 1 to read and write its own valid bit EHR (performing as a register), the port 0 for an early state EHR to bypass the result (performing as a wire), and the port 2 for a future state EHR to avoid bypassing (performing as a register). This way, the stages are scheduled starting from the Write-Back and finishing with the Fetch achieving a correct pipeline behavior.

```
rule doDecode(f2d_valid[1]
            && !f2d_poisoned[1]
            && !d2e_valid[2]);
    // Set the state to wait.  Port 1 since is the actual stage
     fpcg2fbp_valid[1] <= False;

    ...

    // Redirection of the pc. Port 0 since is a previous stage
    fetch_pc[0] <= ....;
    fetch_active[0] <= True;
    // Pass to decode state. Port 2 because is the next stage
    fbp2fri_valid[2] <= True;
    fbp2fri_poisoned[2] <= False;
endrule
```

Listing 5.1: Reading and writing EHR inside a pipeline rule. Example of the Decode stage rule

5.1.3 IPC Analysis

Apart from a better multiplier and divider, the only important optimization that is missing in this design is bypassing. Bypasses eliminate many stalls and can significantly improve the IPC.

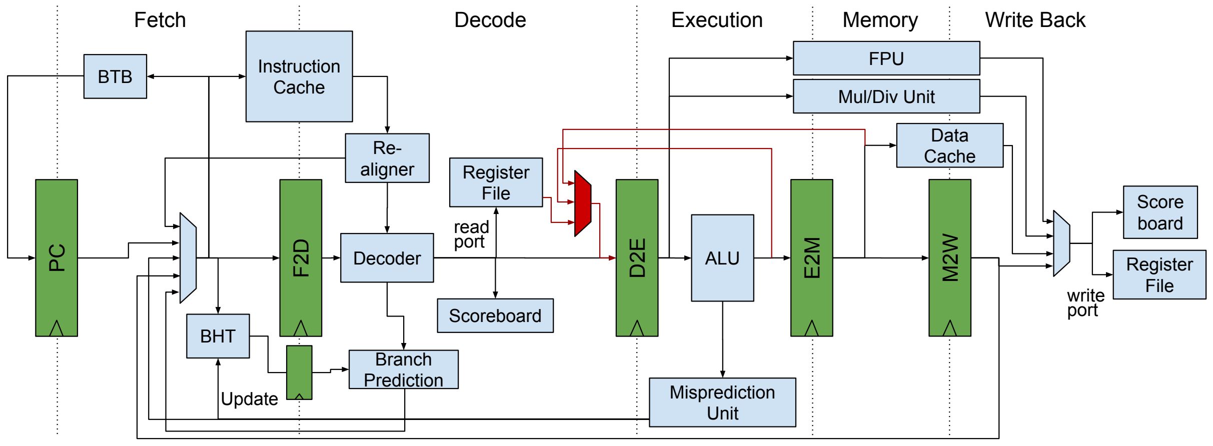


Figure 5.3: Diagram of the Riscy v1 pipeline with bypasses

As the pipeline uses a memory stage, in the worst case, the penalty for a data hazard is two cycles for an arithmetic operation performed by the ALU. These hazards can be easily removed by adding bypasses. In this case, it is necessary to include only two bypasses to complete all the forwarding. The first bypass goes from the end of the Execution stage to the end of the Decoder stage. The second bypass goes from the end of the Memory stage to the end of the Decoder stage. The pipeline with these bypasses is shown in Figure 5.3.

To analyze the improvements, we have run CoreMark in three different versions of the core: without bypasses, with only the bypass from Execution to Decode, and with the two bypasses implemented. To see the improvement on IPC, we use the CoreMark/MHz since it is independent of the frequency. Thus, it is proportional to the IPC. The Riscy v1 is achieving a Cormark/MHz of 1.765. Riscy v1, with only the Execution to Decoder bypass, has a speedup of 1.14X respect Riscy v1 achieving a Cormark/MHz of 2.006. Finally, with both bypasses, the CoreMark/MHz rise to 2.228 points achieving a speedup of 1.26X.

However, as we will show later, we are adding new large paths. We are also adding new logic to the operand selection in the Decoder stage. These two factors can decrease the maximum achievable frequency.

In this state, the core has a good shape in terms of IPC. Since it is a single in-order core, the only noticeable optimization will be adding a RAS branch predictor or an instruction queue, but since the pipeline is short, the IPC improvement will be very small.

5.1.4 Critical Path Analisis

In this section, we are doing a critical path analysis of the Riscy v1 with and without bypasses. The analysis is made by making an analytical hypothesis of the critical path and validating this hypothesis using the synthesis reports of the core. We are using this method to better understand those critical paths. It can not only be done by looking at the synthesis results, because the name of the wires or the wires itself can disappear after the synthesis optimizations. As a result, the only useful information on the synthesis reports is the start and endpoint of the critical paths and the modules where they pass through. Often this information is not enough to have a clear vision of the problem.

This design has two well-known spots that cause delay, if the FPU and the mult/div units are ignored. These two critical points are located in the two caches. In particular, in the tag comparison on the second cycle of the access. It is due to two factors; the considerable access time of the SRAM itself, which can be more than half of the cycle in a high-frequency design; and the comparison of the SRAM output tags with the physical tag from the TLB. A 52-bit comparison is an expensive operation that can introduce a significant delay. Thus, these two actions, which are performed in series, introduce a significant delay, and since the output of the cache is not registered, these paths through the cache can easily become critical paths of the core.

Before starting the detailed look on the pipeline, we need to describe two different types of paths that are encountered. The more obvious paths are those that are only transporting data. For example, there is the path form the D2E pipeline register, which goes through the ALU and finishes to register E2M. The second type of path is related to the control logic of the circuit.

Control Logic Paths The Control logic paths are not always easy to detect since there are not usually explicit in the code. As we have explained, in this project, we are using BSV with its guarded atomic action property. Guarded atomic action property generates a small control logic for each rule and method. This control logic is controlling mainly the write-enable bits for all the registers modified inside the action. These write-enable bits depends on the guard conditions.

In the first look, this control logic should not cause any problem since, in the worst case, the paths should finish on the register controlled with the write-enable bits. However, it is only valid in regular registers. BSV has EHR (explained in Section 2.4.3) used to communicate and schedule different rules and methods during the same cycle. The write-enable bit of these individual registers can propagate the delay to another rule or method since it can perform as a bypass.

To exemplify this phenomenon, we use the Riscy v1 pipeline shown in Figure 5.1 and its implementation using EHR. In this example, there is a large path that is not explicit in the code. The path starts on the Write-Back stage rule (doWriteBack). This rule reads the result of a load instruction from the data cache. The method used to read from the data cache has a guard that is set to valid when there is a hit. If there is no hit (meaning that the read request method of the cache can not be performed), the doWriteBack rule will not perform any action due to the rules' atomic property. Thus, the control logic of the doWriteBack rule needs to take into account this hit condition to control the write enables of all the register used. As we know, the hit condition has a long delay (SRAM delay plus the tag comparators). Then, the write-enable bit in the doWriteBack rule has a minimum delay equal to the hit condition delay.

Moreover, the guard in the Memory stage rule (doMem) depends on the valid EHR of the Write-Back stage, since doMEM can not be executed if the Write-Back stage is stalled. The valid EHR of the Write-Back stage performs as a bypass from the Write-Back to Memory stage. It means that there exists a path from the data cache SRAMs to the write enable bit of the Memory stage passing through the valid EHR of the Write-Back stage. Furthermore, this bypass on the control logic is happening in every stage, meaning that a path from the data cache SRAMs to the write-enable bit on the Fetch stage exists.

5.1.5 Hypothesis of Possible Critical Paths

Taking into account the internal caches problem and the control paths, the expected critical paths will have the starting point on the SRAM arrays on both caches. The first element to consider is the implementation of the A (atomic) RISC-V extension. Both caches implement an internal atomic mechanism since they are identical. This mechanism performs the data read, the atomic operation, and the data write in the same cycle. This path is considerable, and it can be a possible critical path.

Apart from the atomic mechanism, other paths are starting from the caches. These paths can be divided depending on which cache starts the paths.

On the instruction cache, there is a group of large data paths that use the PC redirection mechanism. The group goes from the data cache, passing through the re-align module, Decoder, and the Branch predictor module, which can redirect the PC. The path arrives at the PC selection and can finish to the instruction cache, the F2D register, or on the of the PC register passing through the BTB (which is the more costly action). The other critical path that can be detected is going through the control logic. It is also coming from the instruction cache, going to the re-align module, the Decode, and finally to the scoreboard. Inside the scoreboard is known if the decoder stage has to be stalled due to a data hazard. This stall is performed by not clearing the valid bit of the F2D register. Since this bit is implemented using an EHR is propagated to the Fetch stage control logic.

On the data cache, there only is a large path to consider. It is a control path that starts on the control logic of the Write-Back stage. It is the path used as an example on the section 5.1.4. One of the firing conditions of this stage is the hit on the data cache. This huge delay is propagated by the control logic of each stage, arriving finally to the Fetch stage.

The inclusion of the bypasses should only increase the control logic in the Decoding stage. Thus, the control logic paths from the instruction cache and from the data cache can be increased.

5.1.6 Synthesis Results

To verify the hypothesis made, we have performed two different syntheses of the Riscy v1 to see where the critical path of the design is located. One synthesis is done without bypasses and another with them. To perform the 22nm ASIC synthesis, we are using the SRAMs generated form the memory compilers and the standard cell libraries of 8-tacks, low Vt, and a typical corner at 25°C with a power supply of 0.8V.

The results of the Riscy v1 synthesis without bypasses shows a critical path that starts on the instruction cache tag array as we expected. Next, it goes to the Decode stage, and then to the fetch stage. Inside the fetch stage, it passes through the BTB and finishes on the PC register. This was one of the expected critical paths. As we commented early, only the delay of the tag SRAM is almost a 40% of the path delay. Thus, it is necessary to register the instruction

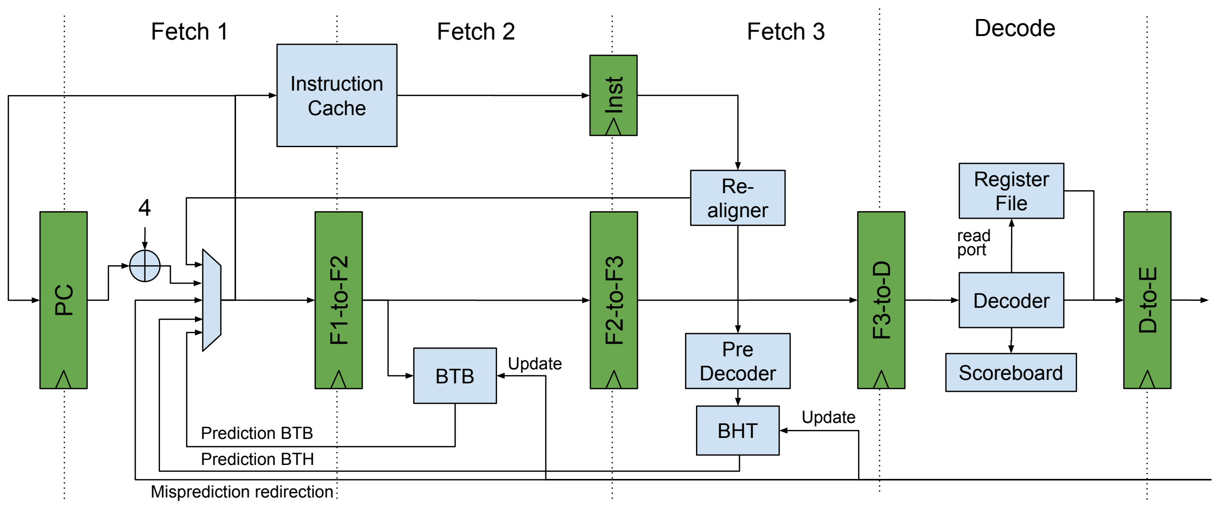


Figure 5.4: Block diagram of the new Riscy front-end

cache output to avoid the delay propagation to the Decoder stage.

The synthesis done with bypasses shows the exact same critical path but with a larger delay because the bypasses add extra control logic to the Decoder stage. The maximum frequency achievable with the Riscy v1 design without bypasses is 667 MHz, while with bypasses, this frequency decreases a little bit, achieving a maximum of 633 MHz. In order to see larger paths that are not the actual critical paths, we need to first solve the known critical paths.

5.2 Frond-End Reimplementation

One of the most critical points in the pipeline is the front-end of the design. The SRAM inside the instruction cache has an output delay that consumes a huge part of the clock cycle. It is necessary to add a register at the output of the instruction cache to avoid the propagation of this delay further. Also, we decided to break the decoder stage in two to reduce the number of operations done in series in this stage (instruction re-alignment, instruction decoding, branch predictor redirection, look up to the scoreboard and reading the register file) and the control logic of this stage (data hazard detection). The new front-end is shown on Figure 5.4.

Now, there are three cycles of Fetch, along with a cycle of decoding. In the first cycle of Fetch, there is the PC generation and the instruction cache access. The branch predictors have been moved to the second stage to remove the extended access to the BTB in this first cycle.

On the second cycle of Fetch, the BTB and BHT are accessed. The BTB prediction is directly redirecting to the PC from this cycle using a bypass to reduce the prediction delay and avoiding a cycle penalty if a BTB prediction hits. The BHT result will be used in the next cycle since, at this point in the pipeline, it is unknown if the instruction is a branch. Also, the response of the instruction cache arrives in this second fetch cycle. As we want to avoid the propagation of the SRAMs delay, we decided to register the cache's output. It is worth mentioning that the instruction cache is still totally pipelined.

On the third fetch stage, the response of the instruction cache is ready to be used. In this stage, there is the re-align module. As a consequence of being one stage later in the pipeline, if the instruction is split into two requests, the penalty of requesting the next cache request increases by one cycle. Also, in this stage, there is the BHT PC redirection. In order to perform the prediction, a pre-decode of the instruction is performed. This pre-decoding of the instruction detects if the instruction is a branch or not. To mitigate a little bit the impact of compressed instructions as the branch predictor is not taking it into account, we decide to add a PC redirection to PC+2 if the pre-decoder detects a compressed instruction.

On the decoding stage, the instruction arrives already re-aligned. The decoder module can directly use it. The decoded instruction is used to access the register file and to detect data hazards on the scoreboard.

5.2.1 Instruction Cache Simplification

Another important module to be modified is the instruction cache. In the Riscy v1, the instruction cache and the data cache are identical. Both integrate all the write-back logic to evict the modified data and integrates the atomic mechanisms. The instruction cache does not need all this complexity since it can not have modified data inside. We have decided to simplify the instruction cache to remove the atomic and write-back capabilities to simplify and remove unnecessary internal paths.

5.2.2 Critical Path and Performance of Riscy with the New Front-End

With the last modifications, all the major critical paths of the Front-End have been removed. Although, now the pipeline has seven stages. The increment of stages has an impact on performance because the PC redirections have a more significant cycle penalty. The BTB redirection is still instantaneous. However, BHT and the JAL redirections are performed in the third stage of the pipeline, resulting in a one cycle penalty. Moreover, the mispredictions are corrected on the fifth stage in the pipeline, resulting in a three-cycle penalty. These extra penalties become worse with compressed instructions, which cause even more mispredictions.

We need to evaluate the impact on the IPC performance of the front-end modifications. To do so, we have run the CoreMark with the core implementing the new front-end. This version of the core does not have the bypasses implemented yet. The CoreMark result shows a performance of 1,692 CoreMark/MHz. The penalty on IPC performance of the new front-end is around 4% compared to the Riscy v1 without bypasses.

We have run a new synthesis of the Riscy with the new front-end without bypasses to validate the removal of the front end critical path. The synthesis result shows a critical path that starts on the data cache tag array and finishes on the same cache's data array. These two points are showing the critical path of the data cache atomic instructions mechanism. It is an expected critical path. Riscy with the new front-end and without bypasses can achieve a maximum of 818 MHz, a 23% improvement respect of the Riscy v1. If we take into account the IPC and frequency of this new version, the final performance (IPC·MHz) increases a 17.5% respect Riscy v1 without bypasses.

5.3 Back-End Modifications

As we have shown in Section 5.1.3 the bypasses are necessary for this design since the increment on IPC performance is noticeable. Thus, we need to discuss and solve the critical paths introduced by the bypasses.

5.3.1 Bypasses

We want to integrate the bypasses along with the new front-end implementing it in the same way as we have done in Riscy v1. We have implemented the execution to decode and the memory to decode bypasses. We expect a better performance in terms of IPC than the Riscy with the new front-end and. However, a smaller IPC respects the Riscy v1 with bypasses because of the penalties of the new front-end.

We want to compare the new IPC performance and maximum frequency concerning the other versions doing the same analysis. The CoreMark result shows a CoreMark/MHz of 2.108, a 25% improvement respect Riscy with only the new front-end although we have the expected reduction of a 5% respect the Riscy v1 with bypasses.

Also, we synthesized the core to see if we are introducing new long paths. The synthesis results show a critical path which limits the frequency of the core at 708 MHz. The critical path was expected. It goes from the tag array of the dcache to the Fetch state going through the control logic of each stage. With the bypass logic, the control logic of the data hazards detection on the decoding stage increases considerably and makes the control path from the Write-Back stage to the Fetch stage bigger. Now, it is bigger than the internal atomic path of the data cache, and it is shown in the synthesis reports.

5.3.2 Data Cache Modifications

To avoid all the data paths generated for the data cache, we have decided to register the output of the SRAMs. The data cache modified is shown in Figure 5.5 With this modification, the access time of the data cache is increased by one cycle. Also, the execution of the atomic instructions is divided into two cycles. The critical paths should be removed. However, all the memory and atomic instructions have a penalty of one cycle, which has a significant impact on performance since each memory or atomic instruction is stalling the pipeline one cycle.

We have run the CoreMark benchmark with this new data cache on the Riscy with the new front-end and with bypasses. It results in a CoreMark/MHz of 1.841. Thus, the extra latency cycle on the data cache is decreasing the IPC about 13%.

We have synthesized this version of the core to know the impact in terms of maximum clock frequency. The critical path is no longer through the data cache. The critical path is the control logic that fires the instructions cache's internal rules, which depends on the tag hit condition (tag SRAM access delay plus the tag comparison delay). This logic must be there and can not be simplified without significant changes in the cache's rule base mechanism. The maximum achievable frequency with the new data cache is 928 MHz. A 31 % improvement respect the

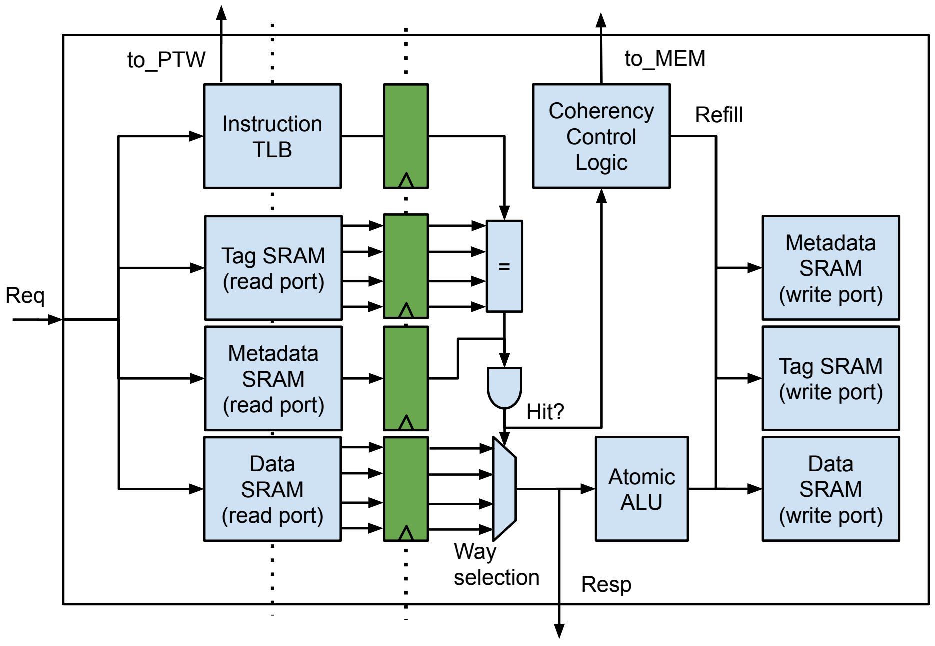


Figure 5.5: Block diagram of the new data cache

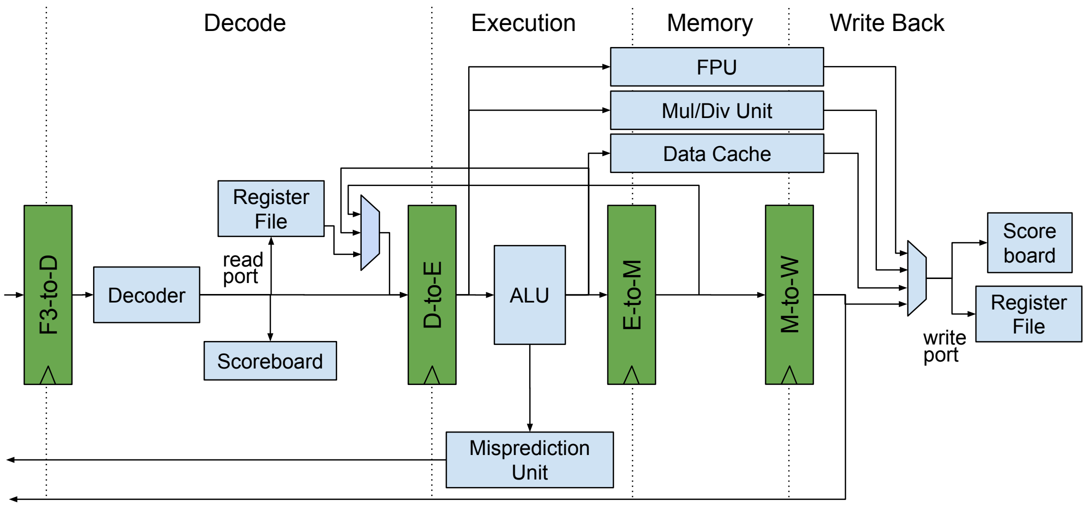


Figure 5.6: Block diagram of the new Riscy back-end

Riscy with the new front-end and bypasses.

5.3.3 Early Data Cache Access in Execution Stage

We noticed that the extra latency on the data cache had significant negative impact on performance. However, this extra latency can be hidden if the access to the data cache is started on the execution stage. We are adding extra delay to the access of the data cache by computing the access address in the same cycle. However, it is not the critical path, and synthesis shows the same maximum frequency is still 928 MHz. This modification allows some loads to not stall the pipeline. Stalls occur only when the load result is the source for the following instruction. It also happens with the one cycle latency cache. We are also stalling the consecutive memory instruction because the data cache is not pipelined to avoid the control logic delay propagation. This modification results in IPC's increment of 10% respect to Riscy with the new front-end, extra cycle latency on data cache, and bypasses. The resulting CoreMark/MHz is 2.03 points.

6 Evaluation

In this section, we discuss the final results obtained in this thesis. We also discuss each modification's intermediate results to show its impact on the performance per cycle and maximum clock frequency. We evaluated the following different Riscy design versions:

- **Riscy v1:** Original version of the core. It has a 5-stage pipeline, without bypasses and identical one-cycle latency cache for the instruction cache and the data cache.
- **Riscy v1-B:** Is the same core than Riscy v1 with the implementation of the bypasses from the Execution and the Memory stages to the Decode stage. *B* stands for Bypasses.
- **Riscy FE:** It has the new front-end with a simplified instruction cache. It does not have bypasses implemented. *FE* stands for Front-End.
- **Riscy FE-B:** It is the Riscy design with the new front-end version and with the bypasses implemented.
- **Riscy FE-B-DC:** It is the Riscy FE-B with the two-cycle latency data cache. *DC* stands for Data Cache.
- **Riscy v2:** It is the final version of Riscy proposed in this thesis. It has the new front-end resulting in a 7 stage pipeline. It has the simpler one-cycle latency instruction cache and the two-cycle latency data cache. Finally, the data cache access is done in the Execution stage.

All the different versions have a 3-cycle pipelined multiplier and a 64-cycle divider, which are not in the critical path. Also, all versions have the FPU provided by Bluespec to enable the F and D RISC-V extensions. However, our analyses will not take into account these extensions for two reasons. The maximum frequency achieved by the FPU is higher than the core frequency, and it is not the bottleneck in the design in terms of frequency. The second reason is the known errors on the FPU behavior in some non-numerical results such as infinite or NaN. Since the behavior is not the behavior specified in the RISC-V ISA, some benchmarks with floating-point operations do not finish correctly. Thus, we only run the non-floating point benchmarks for more consistent results. For the evaluation, we used the EEMBC CoreMark and the non-floating point benchmarks of the EEMBC AutoBench.

For the clock frequency and critical path analysis, we used an ASIC target synthesis done with the Genus 19.11-s087_1 from Cadence. We are using the 22FDX GlobalFoundries technology

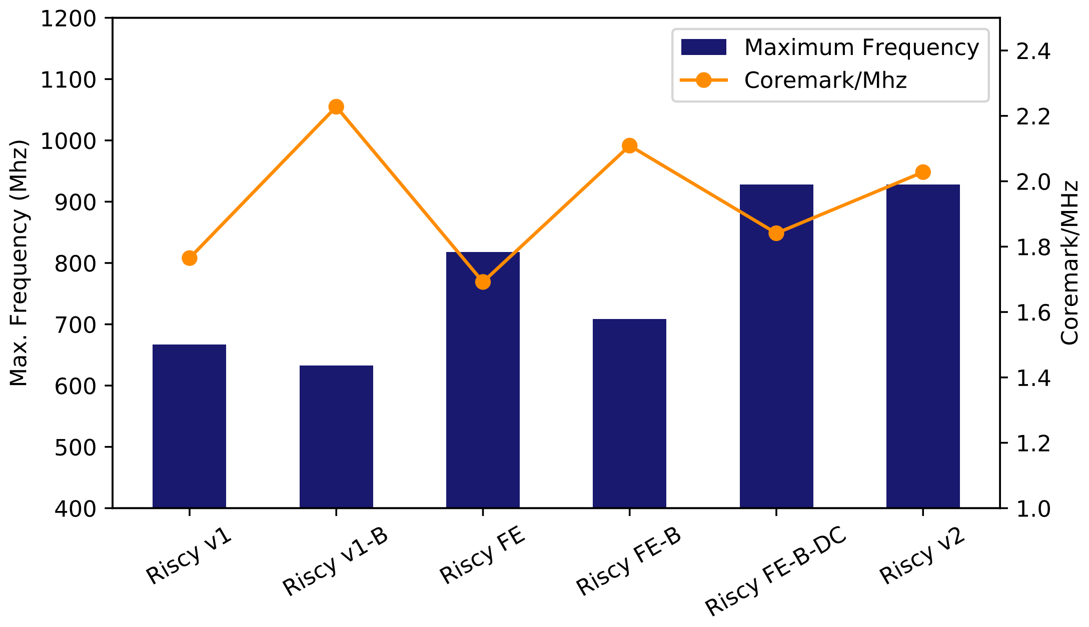


Figure 6.1: Evolution of the maximum frequency and IPC of the Riscy designs

node. The standard cell libraries used in all the analysis has 8-tracks, low Vt, a 104 CPP with the typical corner at 25°C and a power supply of 0.8 V. The Hard IPs used in the synthesis are the SRAMs for implementing the caches. In the case of the original cache and the two-cycle data cache, the SRAMs used has two ports (one read and one write). The simplified instruction cache uses a single port SRAM as it is not necessary to handle reads and writes in the same cycle.

6.1 Frequency and IPC Evolution of Riscy

First of all, we comment and analyze the evolution of the performance in the different versions of the Riscy core. We compare the performance using the Coremark/MHz given by the EEMBC Coremark benchmark compiled without compressed instructions. The CoreMark/MHz is independent of the clock frequency and is proportional to the IPC. We avoid compressed instructions since the branch predictors of Riscy are not adequately designed to handle misaligned instructions. The noise produced by branch mispredictions makes the analysis more complex. However, we analyze this factor in detail in Section 6.2. To properly compare the clock frequency, we use the ASIC synthesis results. Figure 6.1 shows the results of Coremark/MHz and the maximum frequency of each version of the Riscy core. Figure 6.2 shows the speedup of each design version with respect to Riscy v1 in terms of the final performance. Performance is measured as the multiplication of the IPC and the maximum frequency.

The performance differences between Riscy v1 and v1-B, or between Riscy FE and FE-B show the huge impact of bypasses. Bypasses increment the IPC by 25%-26%. This improvement on IPC is due to eliminating the data hazards produced by arithmetic instruction source registers, even though the integration of the bypasses results in a penalty on maximum frequency. This frequency penalty is between 5% and 15%, depending on the case. These penalties are different

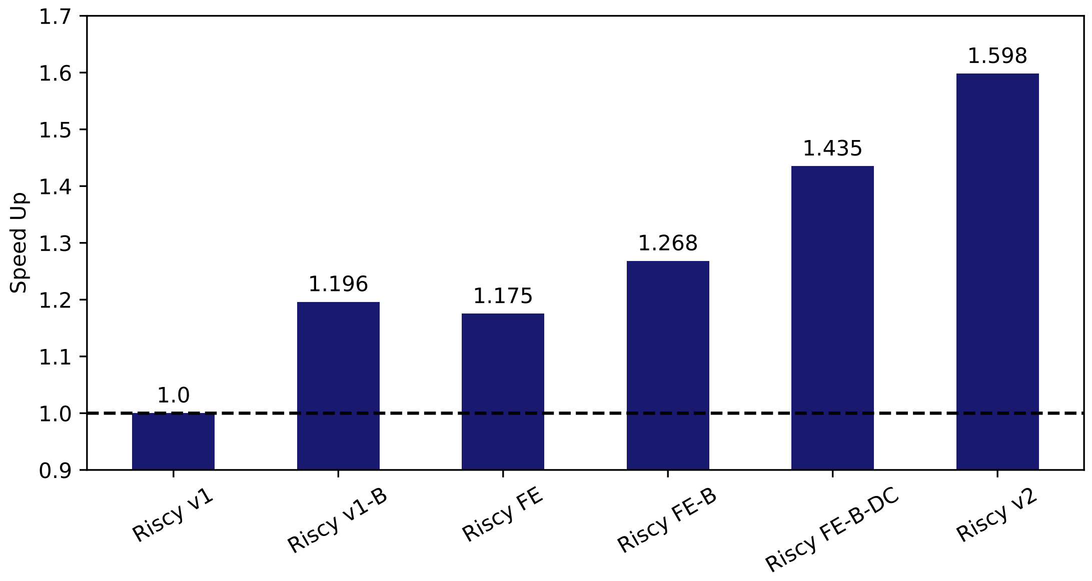


Figure 6.2: Evolution of the performance of the Riscy core designs

because the different Riscy versions do not have the same critical paths and can not be compared directly. Thus, we can see that the integration of bypasses has a bigger IPC improvement than the downgrade on frequency. As a result, the integration of bypasses improves the overall performance between a 7% in the case of the new front-end and a 20% in the old front-end.

To see the impact of the new front-end, we can either make the comparison between Riscy v1 and Riscy FE or the comparison between Riscy v1-B and Riscy FE-B. The CoreMark results show that the new front-end decreases IPC performance by about 4%-5%. This decrement is due to the increase in the number of stages in the new front-end and the later PC redirections compared to the original implementation. Unfortunately, the maximum clock frequency increases up to 23% without bypasses, and by 12% in case of bypasses. Thus, it results in an increment of the overall performance of 17% in the case of not implementing the bypasses. In the case of implementing bypasses, the increment is reduced to 4%. The improvements are different as the critical paths in the new front-end versions are not related to the front-end and can not be directly compared.

Finally, we can analyze the impact of adding a cycle of access latency in the data cache and where it starts the access. If we add an extra cycle of latency on the pipeline and the access is still starting in the Memory stage, stalling the pipeline one cycle for each memory instruction as a result, the IPC reduction is about 13%. This reduction is shown in the comparison between Riscy FE-B and Riscy FE-B-DC. In contrast, in Riscy v2, if we start the data cache access in the Execution stage, we are allowing the pipeline of the memory instructions followed by an arithmetic instruction. In this case, the IPC loss is only about a 4%. As Riscy FE-B-DC and v2 have the same critical path located in the front-end, the maximum frequency is the same. Riscy v2 has an improvement in terms of the maximum frequency of 31% with respect to the Riscy FE-B, which has a long critical path.

To summarize, Riscy v2 improves the maximum clock frequency by 39%, and the IPC by 15% with respect to the original Riscy v1, making the Riscy v2 60% faster than Riscy v1.

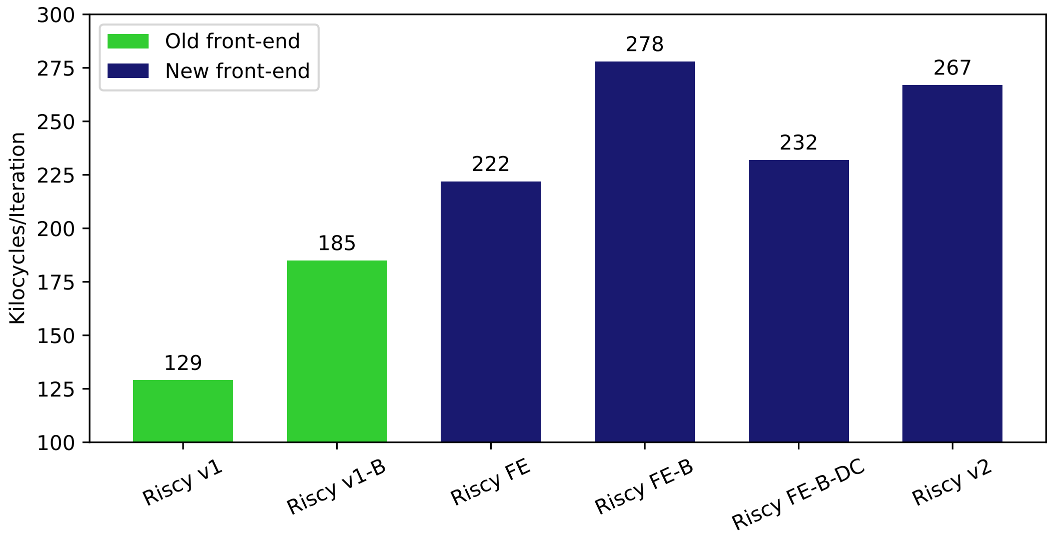


Figure 6.3: Extra stalled cycles per CoreMark iteration produced by the compressed instructions for the different versions of Riscy

However, directly comparing Riscy v1 with Riscy v2 is not a completely fair comparison as they have different IPC optimizations. To make an apple to apple comparison, we need to compare the Riscy v1-B with Riscy v2 as both implement the bypass optimization. Riscy v2 has a clock frequency that is 47% faster, but with a 9% IPC reduction, resulting in a 34% overall performance improvement.

6.2 Evaluation of the Compressed Instructions on Riscy

As mentioned earlier, the Riscy designs do not have a branch predictor adequately designed to handle misaligned instructions. To show this fact, we have executed Coremark with and without compressed instructions on the different versions of Riscy to understand the impact of the compressed instructions. Figure 6.3 shows the extra cycles per CoreMark per iteration produced by compressed instructions in each version. We are not showing the increment on IPC because the same increment of all cycles can produce a different percentage of increment depending on the baseline IPC.

The stalled cycles per CoreMark iteration produced by the compressed instructions in the new front-end is around 60% higher than in the old front-end versions. This is happening because the penalty for a misprediction is greater in the new front-end. This penalty would be more significant in the new front-end if we did not implement the PC+2 redirection in the third stage in case of a compressed instruction.

6.3 Performance Evaluation Using the EEMBC AutoBench

To get a more robust performance analysis, we execute the EEMBC AutoBench suite, since CoreMark is only one benchmark that possibly is not showing the real performance improvement

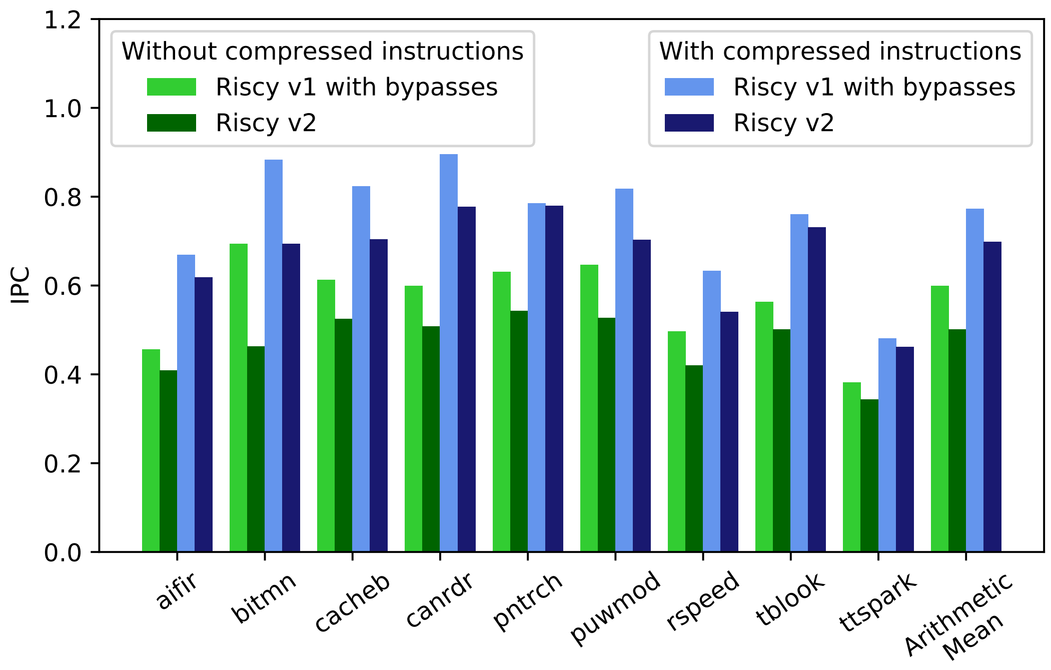


Figure 6.4: IPC of Riscy on the non-floating point benchmarks of the suite EEMBC AutoBench with and without using compressed instructions. At the right there are the arithmetic mean of the IPC obtained with the different benchmarks

in every environment. We have used only the non-floating point benchmarks to avoid issues with the FPU implementation. We decided to compare the Riscy v1-B and the Riscy v2, since both implement bypasses, unlike the Riscy v1. Figure 6.4 shows the IPC comparison of the two designs with and without compressed instructions. The final speedup of Riscy v2 over Riscy v1-B is shown in Figure 6.5.

As we already commented, the performance is better in both versions without compressed instructions. Moreover, the penalty of the compressed instructions on the Riscy v2 is bigger. This phenomenon is easy to detect if we take a look at the mean of the overall speedup. The Riscy v2 has a 1.23× speedup with respect to the v1-B design with compressed instructions. Without compressed instructions, the speedup increases to 1.33×. These results validate the CoreMark results discussed before.

Performance improvements are similar for all the different benchmark except for Bit Manipulation (bitmnp). The Bit Manipulation benchmark is not performing as well as the others with Riscy v2 because it has a lot of conditional branches that are dependent on a random value. The extra latency on PC redirection and misprediction detection on Riscy v2 explain this lower performance.

6.4 Synthesis Experiments with Riscy v2

To further increase the maximum frequency achieved by the core, it is possible to explore different libraries and options provided by the foundry. We synthesize Riscy v2 using different

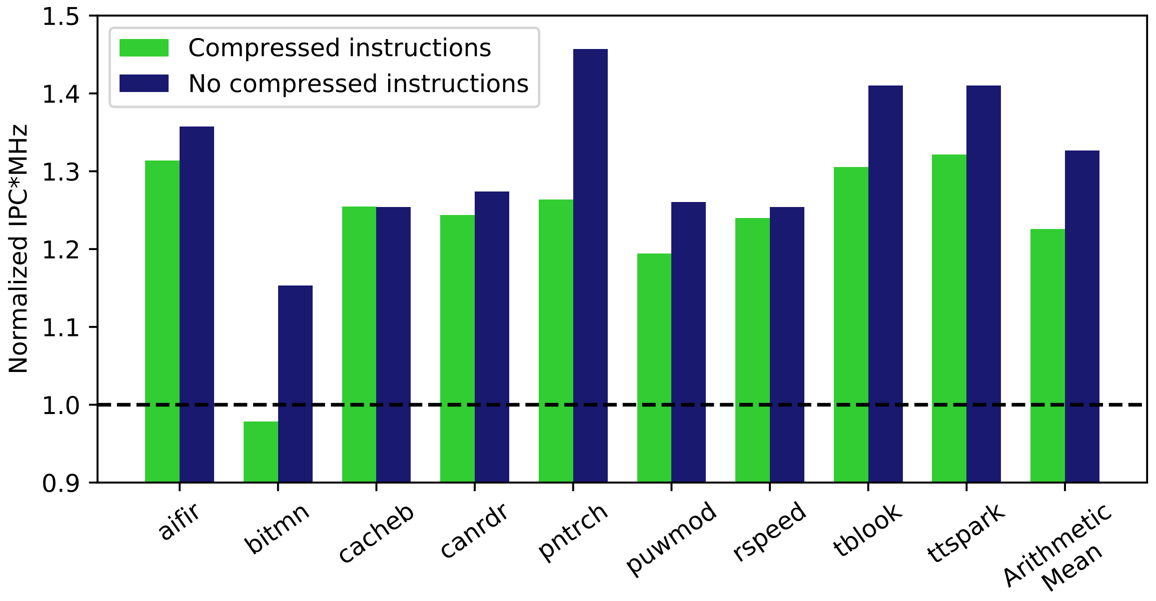


Figure 6.5: Speedup of Riscy v2 respect Riscy v1-B on EEMBC AutoBench with and without using compresed instructions

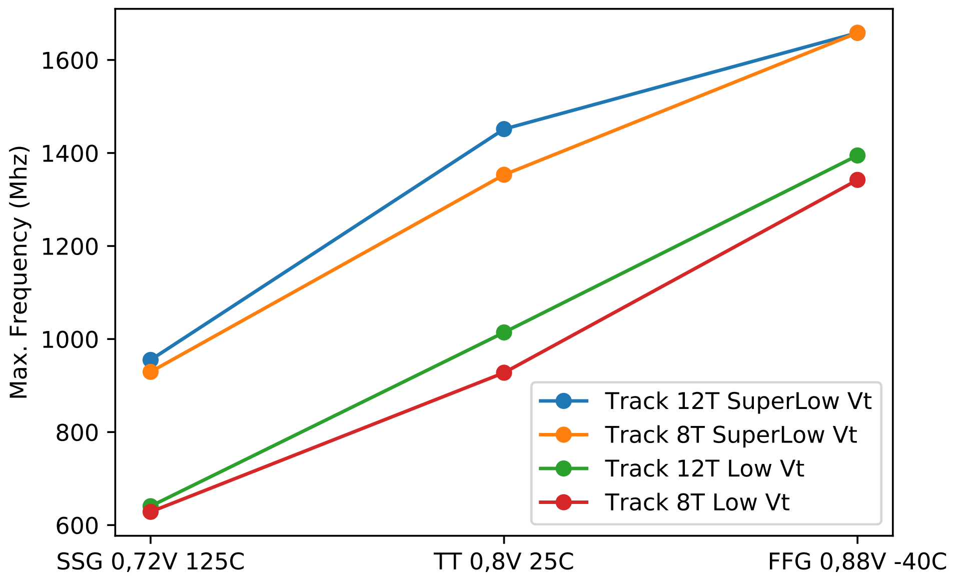


Figure 6.6: Maximum clock frequency depending on the library and PTV corner

combinations of 8-track and 12-track cells with low and superlow Vt. We have to mention that the single port SRAMs used on the simplified instruction cache has low and superlow Vt versions, while the dual-port SRAMs only have the low Vt version. In our experiments, we have used three different PVT corners: Typical with 0.8V and 25°C, fast with 0.88V and -40°C, and slow with 0.72V and 125°C. Figure 6.6 shows the maximum frequency achieved with the Riscy v2 with the different PTV corners and libraries.

As we can see, the 12-track superlow Vt library is the fastest implementation, having a typical frequency of 1451 MHz. However, since the frequency is not far from the 8-track superlow Vt

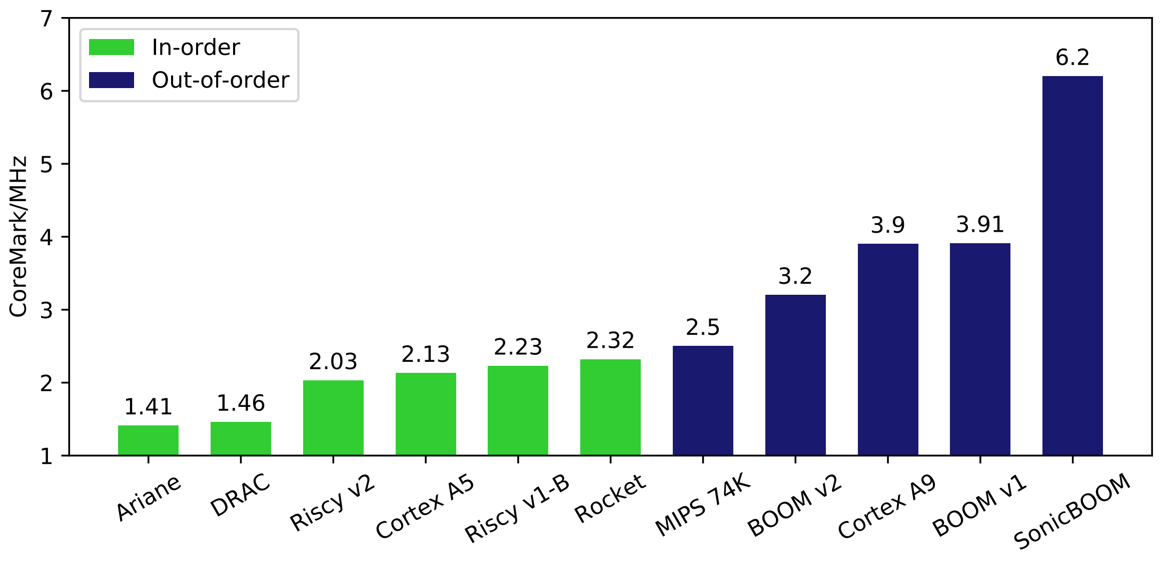


Figure 6.7: Riscy CoreMark/MHz of the version v2 and v1-B compared to other cores

library, it is recommended to use the 8-track library because the power consumption and the area used are significantly lower. In the 8-track superlow Vt configuration, Riscy is achieving 1353 MHz in the typical corner, 1658 with the faster corner, and in the worst case, it is achieving 930 MHz.

6.5 Riscy v2 Comparison with Other Cores

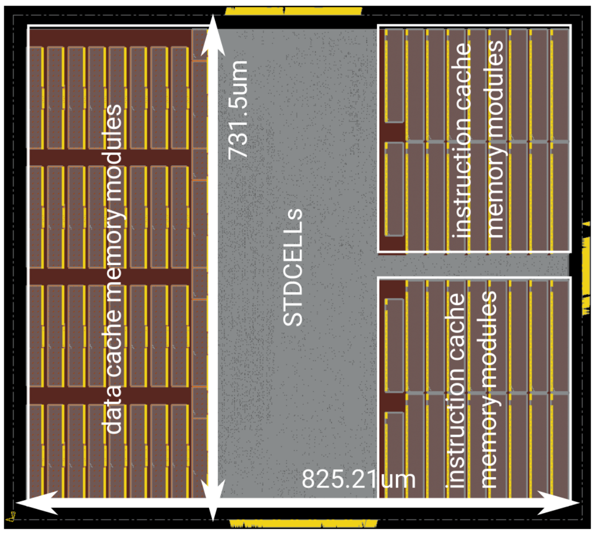
Comparing different cores' performance is always difficult because the performance results usually depend on the benchmark and processor designs can be optimized for specific workloads. Also, it is not always easy to find all the benchmark results for a particular core. Thus, we decide to use CoreMark to make the comparison easier because it is usually available for all the cores. Figure 6.7 shows the CoreMark result for each core.

Inside the in-order cores, Riscy is not far from the performance per cycle achieved by Rocket. However, it is difficult to compare the maximum frequency between them, because we are using a different fabrication technology. In any case, both Riscy v2 and Rocket achieve +1,5 GHz in a good PTV corner. Finally, we notice that it is necessary to implement an out-of-order design to achieve better CoreMark/MHz scores with these high frequencies.

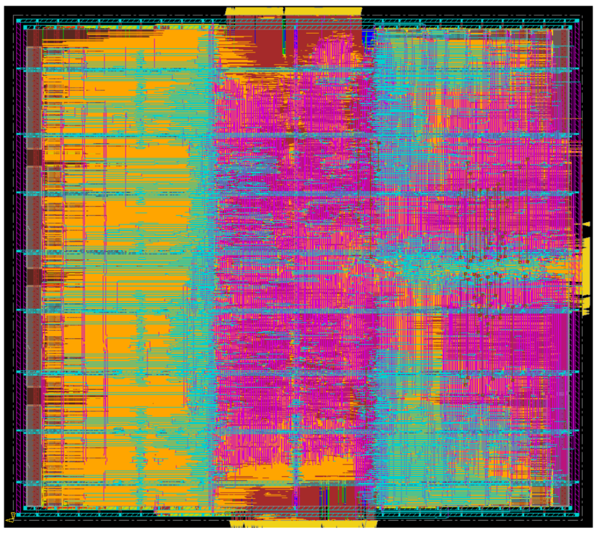
6.6 Place and Route of Riscy v2

The last experiment with Riscy is the actual Place & Route of the core. We are using the Riscy v2 to make the Place & Route. This Riscy v2 design does not have the FPU. We use Innovus version 19.11.000 in order to make the Place & Route of the core along with the netlist of Riscy v2 obtained with the synthesis using the 8-track libraries, super low Vt, and typical corner at 0.8V and 25 degrees Celsius.

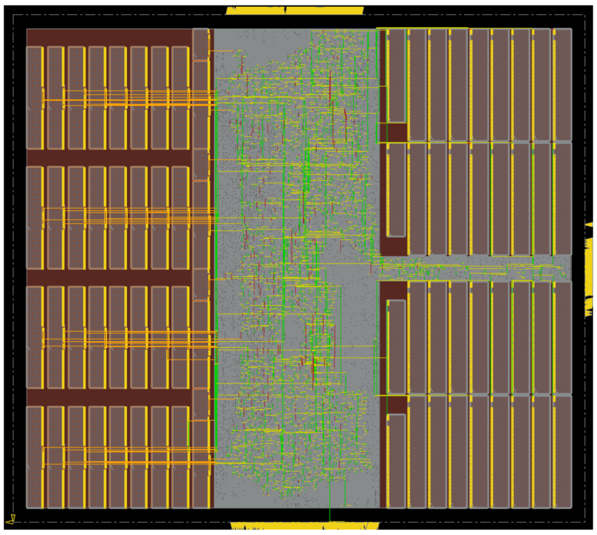
The design has been done with a clock of 750 ps, 1.33 GHz. After the Place & Route, the


data cache memory modules
731.5um
STDCELLs
instruction cache memory modules
instruction cache memory modules
825.21um

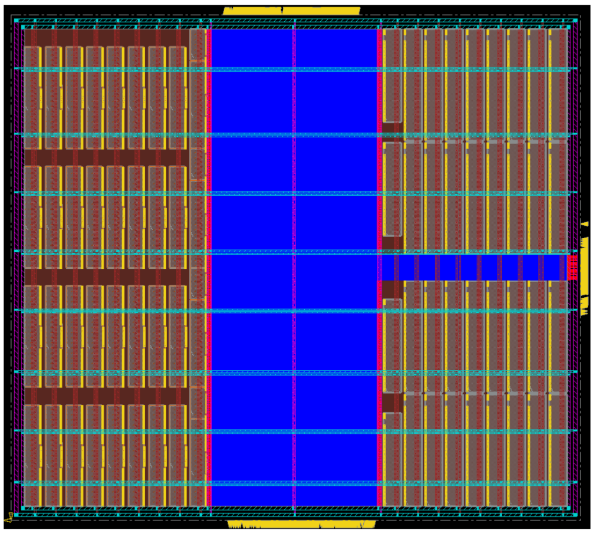
(a) PnR without routing



(b) PnR with all the interconnections



(c) PnR without routing



(d) PnR with all the interconnections

Figure 6.8: Place and route of the Riscy v2 without FPU with the 8 tracks, super low Vt libraries

worst negative slag is positive, meaning that the circuit does not violate any timing constraints. The placing of the SRAMs has been done manually to reduce the antenna effect errors (which can potentially cause yield and reliability problems during the manufacture of MOS integrated circuits [12]) and to accomplish the timing constrains. All the SRAM macros of the same cache memory are in the same region, reducing the amount of routing needed for the interconnection.

The final area utilization is about 65%. The final area is 0.603 mm^2. The SRAM macros are consuming nearly half of the area, 0.265 mm^2. The power ring for the core is using 0.060 mm^2, the physical cells (I/O connections, fillers, end of rows) occupies 0.088 mm^2, the blockages 0.102 mm^2, and the rest, 0.176 mm^2, is used for the standard cells that implement the logic of the circuit. The final power consumption is 430.5 mW running at 1.33 GHz at 0.8 V and 25 degrees Celsius.

Figure 6.8 shows the floorplan of the Riscy v2 core. Figure 6.8a shows the macros and the area used with standard cells. Also, it shows the dimensions of the floorplan. In Figure 6.8b, there is all the routing on each metal layer. Figure 6.8c shows the clock distribution. We can see how the clock is achieving all the flip-flops and the SRAM macros. Finally, Figure 6.8d shows the power distribution of the chip. There is a power ring at the border of the core, which is distributing the power along the chip. Each SRAM macro has two power lines to distribute the power better.

7 Conclusions

In this thesis, we have explored the design space of the Riscy core for a modern 22nm ASIC technology target. The final design has a performance improvement of around 60% with respect to the original Riscy design. When compared to the original Riscy design with the same IPC optimizations, around 34% performance improvements are achieved. This improvement is mainly achieved thanks to the improved maximum clock frequency of the design. Riscy v2 has a more mature 7-stage pipeline taking into account the SRAM timing limitations. It scores a 2.03 CoreMark/MHz in the EEMBC CoreMark benchmark. Moreover, it can achieve more than 1.3 GHz in a typical PVT corner using a technology node of 22nm from GlobalFoundries.

During the improvement process, we applied several modifications to Riscy. We have evaluated the impact of each modification in terms of performance per cycle and in terms of maximum frequency. We have learned that IPC optimizations usually increase the complexity and the size of the core logic, which often decreases the maximum frequency due to new critical paths. Moreover, we have learned that a design modification for increasing the frequency usually requires some paths to be split adding registers to these paths. These new registers add extra latency to that path, add extra hazards to the pipeline, and increase the number of stalled cycles of the pipeline. Also, this extra latency usually adds an extra penalty to pipeline flushes.

Moreover, we have shown the significant timing limitations introduced by the SRAM memories, and provided techniques to mitigate these limitations to further improve the clock frequency. We have also shown the critical paths generated by the control logic, inherent in BSV elastic pipeline designs implemented by a cascading of rules. However, the modularity and the atomic properties of BSV make the improvement process much more comfortable since it is possible to replace modules with different functions, even latency, without making any change to the rest of the design. Working with BSV reduced the implementation time noticeably.

Through the understanding of all the trade-offs between IPC and clock maximum frequency and timing limitations, we designed Riscy v2 with a balance between performance per cycle and maximum frequency, achieving the objectives of this thesis.

7.1 Future Work

Even though we achieved significant performance and frequency improvements throughout this thesis, some aspects of the implementation can be further improved. In Riscy v2, the limiting module in terms of frequency is the instruction cache and its BSV rule-based mechanism. The control path generated in the cache propagates the SRAMs delays, along with the delay generated for the tag comparison. It is necessary to find a rule-base mechanism of the instruction cache that does not propagate this delay to increase the maximum frequency even further.

It would also be interesting to implement a superscalar issue in Riscy and apply the same analysis performed in this project. Since the complexity in the core increases considerably, the new critical paths probably would not be related to the caches. Moreover, it will be interesting to apply the same analysis to the RiscyOO out-of-order design as it uses the same CMD concept than Riscy in-order.

Another potential improvement in the Riscy design is a branch predictor that considers compressed instructions, which are not taken into account in terms of prediction in the actual implementation. Other cores such as Ariane take advantage of compressed instructions to allow multiple instruction fetch per cycle, making a wider fetch mechanism.

The final goal will be to fabricate Riscy v2 and measure the real performance on a real chip. There are several steps needed to make a successful tape-out. Those steps are not done yet. For example, the verification done in Riscy v2 is not enough to ensure that the behavior of the processor is the behavior specified in the RISC-V ISA. Also, gate-level simulations are required to verify that the synthesis and PnR tool flow is working correctly.

Bibliography

[1] L. Amarú, P. Vuillod, J. Luo, and J. Olson. Logic optimization and synthesis: Trends and directions in industry. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1303–1305, 2017.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[3] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872414. URL `http://doi.acm.org/10.1145/2872362.2872414`.

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

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

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

[7] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. 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`.

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

[9] F. Dartu, N. Menezes, J. Qian, and L. T. Pillage. A gate-delay model for high-speed cmos circuits. In *31st Design Automation Conference*, pages 576–580, 1994.

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

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

[12] S. Fang and J. P. McVittie. Thin-oxide damage from gate charging during plasma processing. *IEEE Electron Device Letters*, 13(5):288–290, 1992.

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

[14] Gage Hills, Andrew Wright, Samuel Fuller, Mindy Bishop, Tathagata Srimani, Pritpal Kanhaiya, Rebecca Ho, Yosi Stein, Denis Murphy, Arvind Arvind, Anantha Chandrakasan, and Max Shulaker. Modern microprocessor built from complementary carbon nanotube transistors. *Nature*, 572:595–602, 08 2019. doi: 10.1038/s41586-019-1493-8.

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

[16] Bluespec Inc. Flute: A Open-source RISC-V CPU, . URL `https://github.com/bluespec/Flute`.

[17] Bluespec Inc. Piccolo: A Open-source RISC-V CPU, . URL `https://github.com/bluespec/Piccolo`.

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

[19] Myron King, Jamey Hicks, and John Ankcorn. Software-driven hardware development. 02 2015. doi: 10.1145/2684746.2689064.

[20] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. 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.

[21] Katie Lim, Jonathan Balkind, and David Wentzlaff. Juxtapiton: Enabling heterogeneous-isa research with RISC-V and SPARC FPGA soft-cores. *CoRR*, abs/1811.08091, 2018. URL `http://arxiv.org/abs/1811.08091`.

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

[23] R. Nikhil. 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.

[24] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. ISBN 1558800698.

[25] Jason Poovey, Thomas Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, 29:18 – 29, 11 2009. doi: 10.1109/MM.2009.74.

[26] Cristóbal Ramírez, César Hernández, Carlos Rojas Morales, Gustavo Mondragón García, Luis A. Villa, and Marco A. Ramírez. Lagarto i - una plataforma hardware/software de arquitectura de computadoras para la academia e investigación. *Res. Comput. Sci.*, 137: 19–28, 2017.

[27] D. L. Rosenband. The ephemeral history register: flexible scheduling for rule-based designs. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 189–198, 2004.

[28] David Schor. 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/`.

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

[30] Inc. Sun Microsystems. OpenSPARC^{TM} T1 Microarchitecture Specification. Part No. 819-6650-11, April 2008.

[31] 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.

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

[33] Andrew Waterman and Krste Asanović. 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/`.

[34] Andrew Waterman and Krste Asanović. 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/`.

[35] Alan R. Weiss. Dhrystone benchmark: History, analysis, scores and recommendations.

[36] F. Zaruba and L. Benini. 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.

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

[38] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. *Fourth Workshop on Computer Architecture Research with RISC-V.*

[39] Y. Zu, W. Huang, I. Paul, and V. J. Reddi. 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.