POLITECNICO
DI TORINO

Master's Degree in Electronic Engineering

Master Thesis

Design of the frontend for LEN5, a RISC-V Out-of-Order processor

Supervisor
Prof. Maurizio Martina

Candidate
Marco Andorno

Academic year 2018-2019

Abstract

RISC-V is a free and open source Instruction Set Architecture, which has sparked interest all over the community of computer architects, as it paves the way for a previously unseen era of extensible software and hardware design freedom. One of its main strength points is the vast modularity implemented in terms of different ISA extensions, which aim to cover a very broad range of applications. This allows designers to tailor the architecture according to their specific needs, without constraining them to support unnecessary instructions.

Being RISC-V a relatively new ISA, a limited number of cores is available at the moment, and in particular very few of them are open sourced. So the main motivation for this work is the contribution to this open source hardware community, by means of the design of an Out-of-Order RISC-V core as general purpose as possible.

The core is a 64-bit processor, supporting the G extension, which is a shorthand for the base integer (I), multiply and divide (M), floating point (F) and atomic (A) extensions. One goal of this project, which will be carried out alongside two colleagues, is to eventually include support also for the operating system, by implementing the yet unstandardized Privileged ISA, for the experimental vector extension (V) and possibly for a matrix extension to be defined from scratch. These last design choices are motivated by the lack of open source cores supporting them, and the great advantage that such vectorized computation can provide in a world where the popularity and the performance needs of artificial intelligence and machine learning are ever-growing.

Moreover, the choice of designing an Out-of-Order core arises mainly as all modern processors are of such kind, as it has been the best compromise to efficiently exploit instruction level parallelism for decades. The goal is to implement both instruction issue and execution to be performed Out-of-Order, because this allows the highest performance gain. This design choice, of course, comes with a series of implications that will need accurate analysis and bench- marking, possibly by keeping everything as parametric and modular as possible: branch prediction, instruction queue management, memory hierarchy and cache organization are just some examples.

The final outcome of this work will be an in-depth exploration of the design space offered by such complex architectures, to actually experience firsthand the main issues and tradeoffs designers must face and to be prepared to offer a significant contribution to the state of the art of processor design. Moreover, the common hope is for this project to serve as the basis for future in-house development of a complete RISC-V-based platform here at Politecnico di Torino. As mentioned above, the entire work will be open source and available on a GitHub repository.

Acknowledgements

Thanks everybody!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since their first development in the 1960s, *Out-of-Order* (also known as *dynamic scheduling*) microprocessors have become the main architectural paradigm used in high-performance CPUs, given their ability to hide pipeline latencies and allow for a faster program execution. Along with that, another key role in achieving high effective performance is played by the concept of *speculation* and in particular by branch prediction techniques, which improve the pipeline throughput by maintaining a constant instruction flow inside the processor.

Nowadays, almost every device of common use, from desktop computers, to laptops, to smartphones and tablets, contains some kind of Out-of-Order core which exploits such techniques to offer the computing power and pleasant user experience that the modern world demands. Of course, these architectural design choices come with the drawback of significant added hardware complexity, so there are still some very low power or very low cost microprocessors which do not employ them.

In order to deeply understand such complex architectures and explore the design choices that must be faced in order to achieve that final result, a very convenient way is to make use of an open-source Instruction Set Architecture (ISA), namely RISC-V, which in turn allows the design of open source hardware.

This is exactly the aim of this thesis work: to design a RISC-V core, featuring Out-of-Order execution and speculation to face the issues that such a project involves firsthand, and gain valuable experience in this field of computer architectures. Given its complexity, this work has been carried out by the candidate along with two other colleagues, each one developing a defined part of the core, to come up with the complete design. It is common hope for this project to also serve as the starting point for the future development of a RISC-V based platform at Politecnico di Torino, which could be used for a many different research purposes. For this reason, the entire design and its documentation will be available on a GitHub repository.

1.1 The RISC-V ISA

RISC-V started as a summer research project in 2010 at UC Berkeley by PhD candidates Andrew Waterman and Yunsup Lee and professors Krste Asanović and David Patterson, but soon developed into a fully featured ISA, presented several years later in Waterman's dissertation [1].

Why a new ISA from scratch?

Today the goal of RISC-V is to become a universal ISA [2], able to suit all kinds of processors, from small embedded ones to high-performance cores, from single issue in-order to superscalar out-of-order microarchitectures. Moreover, it is also designed to be implementation independent, in order to work on FPGAs, ASICs and even future technologies, and to be compatible with a large number of popular softwares and programming languages.

How RISC-V intends to achieve that is by leveraging its two main strengths: first of all it is a completely *open source* ISA, meaning that no single company has control over its development and future, and secondly it is *modular*, in the sense that the base instructions are frozen and will stay the same, while new extensions are available and will be developed to expand the capabilities of the ISA (see 1.1.1).

RISC-V belongs to a non-profit foundation, composed by many different corporate members as well as other non-profits and academic institutions, which together aim at maintaining the stability of the ISA, evolving it when necessary and trying to make it ever more popular. For more information, refer to https://riscv.org/.

Add other advantages of the ISA (see Reader chapter 1)

1.1.1 Extensions

Most ISAs are *incremental*, meaning that, in order to ensure compatibility, every new processor must implement new ISA extensions as well as all the extensions introduced in the past, which leads to an accumulation of very rarely used instructions and a subsequent waste of hardware complexity and area. A clear example of this inflation is the growth of the number of instructions in the x86 ISA (figure 1.1).

One the other hand, as stated above, RISC-V is a *modular* ISA: a small number of base instructions (called RV32I, RV64I or RV128I for 32, 64 and 128-bit processors respectively) must be implemented by all instances of RISC-V processors and are guaranteed to never change in the future, while on top of that, designers can freely choose to include support or not for each of the other optional extensions, some of which have already been frozen, while others are still in development. Table 1.1 contains a list of available extensions at the time of writing.

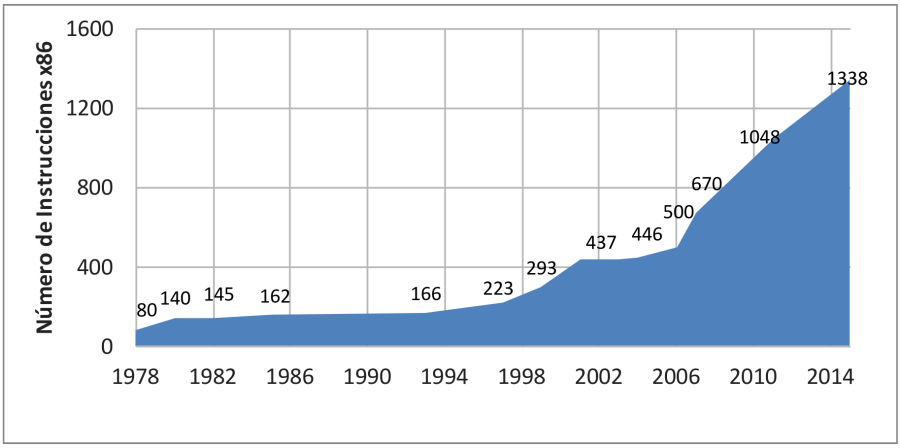

Figure 1.1. x86 instruction count over time [2, p. 3]

Name	Description
I	Base integer instruction set, including arithmetic and logic instructions, jump, branch and control transfer instructions and some miscellaneous general management ones.
M	Integer multiplication and division extension.
A	Atomic extension for atomic memory operations, for process synchronization.
F	Single-precision floating point extension.
D	Double-precision floating point extension.
G	Shorthand for all the previous ones. The processor designed for this thesis supports the RV64G ISA.
Q	Quad-precision floating point extension.
L	Decimal floating point extension.
C	Compressed instructions extension.
B	Bit manipulation extension.
J	Dynamically translated languages extension.
T	Transactional memory extension.
P	Packed-SIMD extension.
V	Vector extension.
N	User-level interrupts extension.
H	Hypervisor extension.

Table 1.1. RISC-V ISA extensions [2]

1.1.2 Comparison with other ISAs

Arguably the two most popular ISAs at the present time are Intel x86 and ARM, which are dominant in the desktop/laptop computers and smartphones/tablets markets respectively. The first significant difference between them and RISC-V is that they are *proprietary* ISAs, which means that whoever wants to design a processor based on such instruction sets is obliged to the payment of the required royalties. On the other hand, RISC-V is free for everyone.

For what concerns the microarchitectural standpoint, another major difference resides in the organization of the internal registers. First of all, RISC-V has 32 of them, twice as much as ARM has, and four times as much as x86. A higher number of registers greatly simplifies assembly language programming and compiler writing. Moreover, the first of those registers, register `x0`, is hardwired to zero, which allows for a significant reduction in instruction count, as many instructions present in other ISAs, which do not have a zero register, can be synthesized using RV instructions with `x0` as an operand. As an example, RISC-V does not need a separate instruction in order to branch if the value of a register is zero: this operation can be obtained with the `beq` (branch if equal) instruction using `x0` as the second operand. The program counter (PC) in the RISC-V ISA is a separate register, and that prevents any instruction from being able to modify it and thus become a branch instruction, as is the case of the ARM ISA, reducing the complexity of the branch prediction hardware and avoiding the loss of one general purpose register.

By keeping simplicity in mind, RISC-V does not provide direct support for byte or half-word integer computation, which can be carried out using separate shift instructions, as they are not critical in terms of efficiency and energy consumption, as are for instance reduced-size memory accesses [2, p. 20]. In addition, multiplication and division are not present in the base ISA (they are comprised in the M extension), and that means that a full software stack can run even without them, which helps reduce the size of embedded chips where such operations are not needed.

Other instructions that the designers of RISC-V chose not to include are, among others, stack instructions, as the stack pointer is one of the general purpose registers and so is accessed as any other register, delayed load, as it is deemed as useless in modern deeply pipelined processors, and finally delayed branch and condition code instructions, which complicate the dependencies checking in Out-of-Order processors [2, p. 21].

It is quite clear that who conceived the RISC-V ISA adopted a philosophy of keeping it simple and that *less is more*, by targeted choices made by learning from the work achieved in the previous decades.

Chapter 2

Improving processor performance

The performance of a processor is defined by the the time it takes to execute a program. This time span, called *CPU time*, can be expressed as:

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Clock cycles}}{\text{Program}} \cdot T_{ck}$$

where T_{ck} is the clock period.

The first term can be decomposed further by computing the total number of instructions inside a program, called *instruction count* (IC), which is known given the assembly code of the program. From this figure and the total number of clock cycles, the average number of *clock cycles per instruction* (CPI)[1] can be derived. By factoring in these quantities, the final expression of CPU time is as follows [3, p. 53]:

$$\text{CPU time} = \text{IC} \cdot \text{CPI} \cdot T_{ck} \tag{2.1}$$

Equation (2.1) shows that the processor performance is directly and equally dependent on three factors:

- Clock period, which depends mainly on the implementation technology and the microarchitectural choices (e.g. pipeline depth).
- Instruction count, which is determined for the most part by the ISA (see section 1.1.2) and compiler technology.
- CPI, which is dependant on both the ISA and the architecture.

[1] Sometimes, also the inverse figure can be used, that is *instructions per clock* (IPC).

The goal is then to minimize each of these terms, but it is evident that none of these parameters can be modified without affecting the others, as many design choices influence many of them.

2.1 Instruction-level parallelism

Earliest processors executed instructions one at a time, fetching a new one only after the previous has finished, leading to a number of clock cycles per instruction greater than one, and in particular equal to the number of stages an instruction must get through. These processors, where CPI > 1, are called *subscalar*. To illustrate the situation, in the example of the classic 5-stage RISC pipeline (fetch, decode, execute, memory access, write back), a subscalar processor would execute three consecutive instructions as shown in figure 2.1, taking a total of 15 clock cycles.

Figure 2.1. Subscalar processor

Starting from the mid 80s, processor architects introduced *pipelining* to improve performance by overlapping the execution of different instructions. This overlap means that at any given point in time there can be multiple instructions running in different stages of the processor, that is *in parallel*, hence the term *instruction-level parallelism* (ILP), which is a fundamental concept in developing techniques to enhance processor performance. For the same example of figure 2.1, a pipelined processor could theoretically achieve a CPI of 1, executing one instruction for each clock cycle (see figure 2.2). Processors of this kind are called *scalar*.

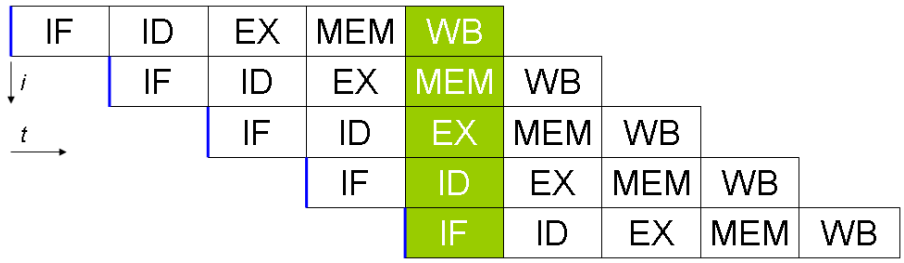

Figure 2.2. Scalar processor

In practice however, data and control dependencies between successive instructions could cause hazards and force the pipeline to stall, causing CPI to rise once again at values greater than one. There are mainly three types of hazards that can take place in a pipelined processor:

- **Structural hazards** arise when a hardware block is needed by two or more instructions at the same point in time. For instance, if a processor features only one memory block for both instructions and data, then two different instructions executing in the fetch and memory access stages could generate a structural hazard when trying to read from memory. Such hazards can either be easily solved (e.g. separate instruction and data memory in this example) or are known and accepted by the designers, given the limited hardware available.

- **Data hazards** in a simple pipelined processor occur when there is a *data dependence* between instructions, that is one instruction needs to read a value that provided by a previous instruction. For example, in

```
add    x1, x2, x3
sub    x4, x5, x1
```

  the `sub` instruction needs the value of register `x1` in the decode stage, but the previous `add` has not yet reached the write back stage and a data hazard is generated.

- **Control hazards** arise in the case of conditional flow changing instructions, such as branches, that prevent following instructions to be fetched until the new direction is resolved.

The real CPI a pipelined processor can achieve is then given by the sum of the ideal CPI and all the delays introduced by pipeline stalls caused by hazards io vole[3, p. 168]:

$$\begin{aligned} \text{CPI} &= \text{Ideal CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls} \\ &= 1 + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls} > 1 \end{aligned} \tag{2.2}$$

Those hazards become more frequent and more expensive to manage the more pipeline stages are introduced and that is a clear example of a tradeoff between two factors of the performance equation (2.1): a deeper pipeline shortens the critical path and thus reduces the clock period, but at the same time it increases the CPI. That is the reason why designers at some point had to find other architectural solutions to improve performance.

2.1.1 Multiple-issue processors

A processor featuring a single execution pipeline can only achieve a theoretical CPI of 1, but by duplicating the pipeline to include multiple execution units more than one instruction per clock cycle could be delivered. That is the idea that lies behind *multiple-issue* processors, that exploit ILP by executing independent instructions on separate execution pipelines.

Instructions that can be issued independently to the different pipelines are selected among a so called *basic block*, that is a sequence of instructions comprised between single entry and exit points (i.e. with no branches or jumps in between). Recalling the examples of the previous section, figure 2.3 shows the execution scheme for a multiple issue processor.

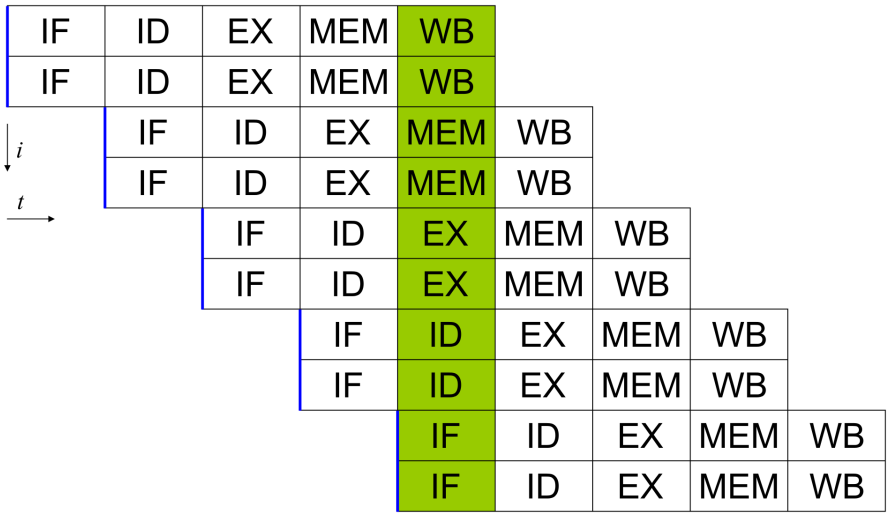

Figure 2.3. Multiple-issue processor with two pipelines

Two main different approaches exist to multiple issue processors:

- **VLIW (Very Long Instruction Word)** processors, also known as *static* multiple-issue, rely on software to discover ILP chances at compile time, thus avoiding increased hardware complexity. The compiler groups instructions that can be executed in parallel in a single long packet-like instruction (hence the name VLIW), that is then split and issued to the different execution units at run time. Despite many efforts, however, such static techniques reveal efficient only for specific applications presenting a high level of data parallelism [3, p. 168], mainly because the compiler software needs a perfect knowledge of the underlying architecture in order to efficiently exploit ILP.
- **Superscalar** processors, also known as *dynamic* multiple-issue, on the other hand, rely on dedicated hardware to exploit ILP at run time. Instructions

belonging to a basic block are inserted into a *window of execution* (WOE), from where instruction that can run in parallel thanks to no data dependence are selected and issued to the respective following pipeline stages. This dynamic approach has been shown to work better than a static one, at the cost of a significant hardware complexity overhead.

Multiple-issue processors can achieve a CPI lower than 1 (usually expressed at this point as IPC, greater than 1) thanks to duplicate hardware units that also lower the impact of structural hazards, but they are nonetheless subject to data and control hazards. Instructions belonging to the same basic block are very likely to depend upon one another, as they are part of the same piece of program, and as such the amount of ILP in contiguous instructions of a basic block is usually very small, neglecting almost totally the performance improvements of multiple issues.

2.2 Dynamic scheduling

2.3 Speculation

Chapter 3

Proposed design

3.1 General scheme

3.2 PC gen stage

3.3 Fetch stage

3.3.1 Branch Prediction Unit (BPU)

3.3.2 Instruction cache interface

3.3.3 Fetch unit

3.4 Execution stage

3.4.1 Branch unit

Chapter 4

Results

4.1 Simulation

4.1.1 Fetch unit

4.1.2 BPU

4.2 BPU benchmarking

4.3 Synthesis results

Chapter 5

Concluding remarks

5.1 Future work

Bibliography

[1] Waterman A., *Design of the RISC-V Instruction Set Architecture*, PhD diss., Electrical Engineering and Computer Sciences, University of California at Berkeley, 2016, UCB/EECS-2016-1.

[2] Patterson D., Waterman A., *The RISC-V Reader: An Open Architecture Atlas*, First edition, Strawberry Canyon, 2017.

[3] Hennessy J., Patterson D., *Computer Architecture: A Quantitative Approach*, Sixth edition, Morgan Kaufmann, 2017.

[4] Mittal S., "A Survey of Techniques for Dynamic Branch Prediction", *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, 2019.