BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DESIGN OF SUPERSCALAR RISC-V PROCESSOR

NÁVRH SUPERSKALÁRNÍHO RISC-V PROCESORU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DOMINIK SALVET**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

155087

Institut:	Department of Computer Systems (DCSY)
Student:	**Salvet Dominik, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Embedded Systems
Title:	**Design of Superscalar RISC-V Processor**
Category:	Computer Architecture
Academic year:	2023/24

Assignment:

1. Familiarize yourself with the open instruction set RISC-V developed by the RISC-V International consortium.
2. Review various designs and implementations of superscalar architectures.
3. Design a new architecture of a 32b superscalar RISC-V processor.
4. Implement the designed microarchitecture as a model using the Verilog/SystemVerilog programming language.
5. Verify and demonstrate the functionality of the implemented processor model.
6. Evaluate implemented solution and discuss possible future extensions.

Literature:

- According to supervisor's advice.

Requirements for the semestral defence:

- Items 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Jaroš Jiří, doc. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	30.10.2023

Abstract

This thesis deals with designing and implementing a superscalar RISC-V processor microarchitecture focused on environments with constrained resources. For that, the microarchitecture exposes a dual-issue seven-stage pipeline with in-order instruction execution. It is described in SystemVerilog and can be easily simulated on a computer. Using prepared tools, the created processor model runs RISC-V assembly programs compiled by GCC. Based on conducted testing without special compiler assistance, the processor executes 0.88 instructions per cycle on average, providing 22.6 % higher performance than its scalar counterpart. Considering that the microarchitecture also avoids unnecessary specialization, it provides a good base that can be further extended and optimized based on the profiling of expected programs, leading to optimal performance and use of resources.

Abstrakt

Tato práce se zabývá návrhem a implementací superskalární mikroarchitektury RISC-V procesoru zaměřené na prostředí s omezenými zdroji. Za tímto účelem mikroarchitektura definuje sedmistupňovou zřetězenou linku s dvojitým vydáváním instrukcí, které vykonává v pořadí. Je popsána v jazyce SystemVerilog a lze ji snadno simulovat na počítači. Pomocí připravených nástrojů pouští vytvořený model procesoru programy napsané v RISC-V jazyce symbolických adres zkompilované GCC. Na základě provedeného testování bez speciální asistence kompilátoru procesor provede v průměru 0,88 instrukcí za cyklus, čímž poskytuje o 22,6 % vyšší výkon než jeho skalární protějšek. Vzhledem k tomu, že se navržená mikroarchitektura také vyhýbá nadměrné specializaci, poskytuje dobrý základ, který lze dále rozšiřovat a optimalizovat na základě profilování očekávaných programů, což vede k optimálnímu výkonu a využití zdrojů.

Keywords

Superscalar processor, RISC-V instruction set, instruction pipelining, in-order execution, dual-issue architecture, open-source hardware, SystemVerilog, simulation testbench

Klíčová slova

Superskalární procesor, instrukční sada RISC-V, zřetězené zpracování, vykonávání instrukcí v pořadí, dvojité vydávání instrukcí, otevřený hardware, SystemVerilog, simulační testbench

Reference

SALVET, Dominik. *Design of Superscalar RISC-V Processor*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Jiří Jaroš, Ph.D.

Rozšířený abstrakt

Procesor může být chápán jako mozek dnešního technologického světa, zejména pak počítačového průmyslu. Ať už se jedná o drobná zařízení k osobnímu použití, průmyslové stroje, nebo superpočítače, všechny závisí na kvalitě a schopnostech procesorů, které je pohánějí. V některých situacích je kladen důraz na nízkou spotřebu, někdy je důležitá deterministická doba reakce na různé události, jindy je zapotřebí vysokého výkonu. Toto zaměření konkrétního procesoru velmi významně zasahuje už do jeho raných vývojových fází.

Každý procesor používá nějakou instrukční sadu, která přesně definuje jaké operace, označované také jako instrukce, procesor umí vykonávat a software může používat. Konkrétní implementace dané instrukční sady se nazývá mikroarchitektura. Tu lze podle počtu vykonávaných instrukcí za jeden hodinový cyklus procesoru rozdělit na skalární a superskalární. Skalární mikroarchitektura označuje procesory, které vykonávají nejvýše jednu instrukci za hodinový cyklus, superskalární pak zvládnou více.

Tato práce se zabývá návrhem a implementací superskalární mikroarchitektury procesoru zaměřeného na prostředí s omezenými zdroji, což zahrnuje zejména nízkou spotřebu elektrické energie. Procesor používá otevřenou instrukční sadu RISC-V, která se v poslední době stala velmi populární v mnoha doménách. K jeho popisu je využit jazyk pro popis hardwaru SystemVerilog. Navržená mikroarchitektura nese označení *Super RISC-V*, což má zdůrazňovat její superskalární schopnosti.

Vytvořený model procesoru definuje sedmistupňovou zřetězenou linku s dvojitým vydáváním instrukcí, které vykonává v pořadí. I když se jedná o superskalární architekturu, zaměření na prostředí s omezenými zdroji má nezanedbatelný dopad na její podobu. Hlavní dotčenou vlastností procesoru je deterministické plánování instrukcí do jeho dvou vykonávajících „slotů“ pouze na základě adres instrukcí. Za určitých podmínek to může vést ke sníženému výkonu, protože procesor nemůže dynamicky změnit plánování instrukce do jiného slotu. Proti tomu lze však ve většině případů bojovat kompilátorem, který tuto vlastnost procesoru vezme do úvahy. Díky takovému omezení má ovšem procesor nižší nároky na potenciální hardwarové zdroje včetně spotřeby. Další způsob, kterým se procesor snaží snížit své teoretické nároky na spotřebu, jsou podmíněné zápisy do téměř všech jeho registrů. To znamená, že se zápisy do registrů provádějí pouze tehdy, když se jejich hodnota může skutečně využít, čímž se zamezí zbytečnému přepínání hodnot signálů. Tento přístup bývá ještě typicky využit nástroji pro syntézu hardwaru k dalšímu prospěchu.

Architektura procesoru je rozdělena do čtyř hlavních jednotek, kde se každá stará o vymezenou činnost a navzájem spolu komunikují. Tři z nich jsou navíc rozloženy do více stupňů zřetězené linky. První jednotka se stará o vyzvednutí dvojic instrukcí z paměti, jejich případné uložení do fronty instrukcí a následné doručení do jednotky pro dekódování. Ta dekóduje dvojici instrukcí, vyčte požadované operandy z registrů a provede kontrolu různých typů hazardů ve zřetězeném zpracování. Dále se instrukce přesunou do vykonávající jednotky, která provádí samotný výpočet a která umí předčasně zpřístupnit výsledky starších instrukcí, aby novější nemusely čekat až na skutečný zápis do registrů s operandy. Poslední významná jednotka spravuje datový přístup do paměti. Přijímá požadavky od vykonávající jednotky a vrací jí výsledky. Ke komunikaci s pamětí procesor používá dvě samostatná rozhraní (instrukce a data) omezené formy AMBA 3 AHB-Lite protokolu.

Výsledný model procesoru je otestován simulací pomocí nástroje Verilator. Používá se 10 vlastních testů v podobě programů napsaných v RISC-V jazyce symbolických adres, které v případě bezchybného běhu vrací nulový kód (v případě chyby nenulový). Tyto testy jsou při simulaci doplněny i několika tvrzeními, která musí při běhu procesoru vždy platit, aby se předcházelo vzniku nedovolených stavů procesoru.

Na základě 10 vlastních testů byla také naměřena data o výkonu procesoru. Ukazuje se, že procesor poskytuje o 22,6 % vyšší výkon než jeho skalární protějšek. Tohoto výkonu procesor dosahuje díky své schopnosti vykonávat v průměru 0,88 instrukcí za cyklus. Vzhledem k tomu, že se navržená mikroarchitektura také vyhýbá nadměrné specializaci, model procesoru poskytuje dobrý základ, který lze dále rozšiřovat a optimalizovat na základě profilování očekávaných programů. Přinejmenším se očekává, že přidání predikce skoků bude mít obecně velmi výrazný pozitivní dopad na výkon. Vytvořený model procesoru je k dostání jako otevřený hardware pod permisivní licencí, což spolu se srozumitelným zdrojovým kódem umožňuje urychlit jeho další vývoj.

Design of Superscalar RISC-V Processor

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of doc. Ing. Jiří Jaroš, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Dominik Salvet
May 16, 2024

Acknowledgements

I want to express my gratitude to my thesis supervisor, doc. Ing. Jiří Jaroš, Ph.D., for his willingness and professional guidance throughout the work on the thesis. Also, I want to thank Ing. Lukáš Valach for his valuable advice during the initial work on the processor microarchitecture design and Ing. Jakub Horký for his suggestions for processor testing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the computing industry, not many factors affect its end products as much as the instruction set architecture (ISA) does. It is a bridge between software and hardware, and hence, it affects the form of both. Due to that, there is a strong demand to have an ISA designed well for its purpose, which often means finding a good balance between software requirements and hardware complexity. Even though some may be completely unaware of ISA presence, there is at least one deployed in practically any existing computer system.

An ISA describes operations, also called instructions, which are available for software and must be implemented by hardware. A particular hardware implementation of an ISA is called microarchitecture. The microarchitecture significantly participates in the final form of the processor. It strongly affects the processor's performance, power consumption, and area size. Moreover, some ISA concepts may get further reinterpreted in individual microarchitectures as long as the microarchitecture behaves as the ISA states.

The goal of this master's thesis is to design, implement, and test a microarchitecture of a superscalar RISC-V processor. A superscalar processor, unlike a scalar processor, can execute multiple instructions at a time. I chose a superscalar architecture because it is one of the most used techniques for increasing processor performance and, as such, is commonly used as a building block for many advanced techniques. RISC-V is a free ISA, which is so flexible that it allows a diverse range of possible microarchitecture implementations. RISC-V was my choice because I like its well-thought-out principles. Besides, it is probably today's most popular free ISA, and this trend does not seem to change anytime soon. The motivation behind the whole thesis is to explore the superscalar architecture and the RISC-V instruction set themselves. These two traits indicate that the submitted solution is built on a solid foundation.

Despite the recent open-source software expansion, the processor industry is still a relatively closed and commercial environment. It is understandable since, compared to software, generally, there is more at stake here. Once a faulty processor is manufactured, there is no easy way back. One of the added values of this work is that all associated materials are provided in a public Git repository[1] under a permissive license to promote open-source hardware and, hopefully, also enrich the related community. Furthermore, it makes reading this thesis easier as readers also have access to all source code files. Contributing to the overall usability, only open-source tools are required to build and simulate the created processor model.

[1]The thesis materials are available at `https://github.com/dominiksalvet/super-riscv`. Since further development is expected, please use the `master-thesis` tag to browse the version described here.

The thesis is divided into several chapters, each dealing with a defined area of interest. In chapter 2, selected parts of the RISC-V instruction set are introduced. Chapter 3 examines one of the most common processor architectures — scalar processors — and, based on that, covers superscalar processors and their representatives. Chapter 4 describes the designed superscalar RISC-V microarchitecture and its respective implementation in a hardware description language. Finally, in chapter 5, testing and demonstration of the processor work are covered. That also includes evaluating the processor design and performance based on performed measurements.

Chapter 2

Description of the RISC-V Instruction Set

RISC-V (pronounced "risk-five") is an open ISA developed by the RISC-V International association[1]. This association cares about RISC-V promotion and further development. As RISC-V ISA, with many relevant tools, is provided for free under a permissive license, anyone can adopt it: ranging from individual open-source developers who only want to share their work to companies with commercial uses, which may even include chip manufacturing. Nevertheless, some may decide to purchase a RISC-V International membership to gain associated strategic advantages such as having a stronger voice when deciding on the development direction [15].

Even though RISC-V is a relatively young ISA — originally started in 2010 at the University of California, Berkeley [27] — it has earned a lot of attention from both the industry and academic spheres. It builds on the Berkeley RISC (reduced instruction set computer) processor series, of which RISC-V represents the fifth. The most important reasons why this ISA was created were connected with the shortcomings of commercial ISAs of that time [25]. That included implications of their commercial principles, as compared in table 2.1, and their technical aspects as well. Due to that, RISC-V authors carefully examined the impacts of ISA design choices on historical processor architectures and the current ones, too, not to incorporate known mistakes in the emerging ISA. Each such decision is mentioned in RISC-V specifications with the associated rationale.

Classic commercial IP license	**RISC-V commercial IP license**	**RISC-V open source IP license**
Fees for ISA	No fees for ISA	No fees for ISA
Fees for microarchitecture	Fees for microarchitecture	No fees for microarchitecture
Warranty & indemnification (limited)	Warranty & indemnification (limited)	No warranty & indemnification

Table 2.1: **Comparison of processor IP (intellectual property) licenses.** It is worth pointing out that it is practically impossible to have an open-source microarchitecture without having the deployed ISA open first. Reproduced from [23].

[1]The RISC-V International website – https://riscv.org

The key RISC-V trait is the extensive use of so-called ISA extensions. Although the ISA base, which always must be present, aims to be domain agnostic (i.e., be as universal in terms of potential applications as possible), the extensions enable the domain-specific focus of a RISC-V processor. That considerably increases the overall flexibility of the ISA since simple implementations may get by with the base while more complex ones utilize the extensions. Besides this, the concept of extensions simplifies the development of RISC-V software tools due to reusability and fragmentation prevention. Also, companies that work with the RISC-V architecture reap the benefits connected with this ISA property and provide RISC-V processors for multiple target domains, each with a possibly different set of extensions. In addition, they may offer customization services, where a given processor core is extended with a required set of features.

Company representatives of today, whose business is based on RISC-V, may include SiFive, Codasip, and Andes Technology. There is also significant interest from big technology companies like Intel, NVIDIA, and Western Digital. More subjects that expressed support for RISC-V are listed on the RISC-V International website. Most of the end products of their effort — manufactured RISC-V processors — are deployed in various embedded systems (for instance, disk controllers). RISC-V cores are part of many microcontrollers and SoCs (systems on a chip), and lately, they have started to appear in the application domain, which demands running operating systems such as Linux [4].

The most important and stable parts of RISC-V ISA are structured in two volumes. The first volume [25] is about the unprivileged part of the ISA and is a primary information source for the following text of this chapter. It defines principles, instructions, and standard extensions that are generally usable regardless of privilege configuration. The second volume [26] covers the privileged part. It defines privileged levels and instructions, handling of interrupts and exceptions, memory management, and such. Privileged concepts get explained only as soon as needed. Both volumes are available to download for free on the RISC-V International website.

2.1 RISC-V Instruction Set Classification

Despite RISC-V's optional support of variable-length instructions, it is a typical representative of a RISC architecture. Base integer ISA has fixed length instructions by design. Instructions with lower lengths are defined in an extension to provide support for compressed forms of some RISC-V instructions connected with respective benefits. On the other hand, higher length instruction space is currently considered not frozen and rare to see being used in practice. However, it is a feature that may be useful in the future should new needs arise.

RISC-V adopts load-store memory access, which means it defines only a few instructions to communicate with the main memory. The most important such instructions are for storing and loading data of a given width. Optionally, if an extension is implemented, it may also define instructions for atomic memory operations. Due to the load-store type, there are not many addressing modes available either. In general, RISC-V offers the following four addressing modes (others exist but may be considered derivatives of those four):

- **Register addressing** uses a value directly stored in the concerned register.
- **Immediate addressing** extracts an immediate value stored in the current instruction word and, optionally, transforms it slightly before use.

- **Base+offset addressing** uses the value of a given register as the base and adds an offset directly stored in the instruction word to it.
- **PC-relative addressing**[2] calculates the final value by adding an offset defined in the instruction word to the value of the PC register (useful for, e.g., branch instructions and position-independent code).

RISC-V uses common memory space for data and instructions. Input and output devices are mapped to this memory space as well. Implies that it follows the von Neumann architecture, which means that RISC-V needs to provide a mechanism for the memory writes and instruction fetches synchronization (e.g., for just-in-time compilation). This mechanism is provided in a separate extension.

RISC-V ISA avoids favoring a particular microarchitecture style to be as universal as possible. However, common microarchitecture styles were taken into mind during the ISA development, and they influenced the ISA (as often explained in commentary sections of the RISC-V documents). Therefore, occasionally a useful hint for the microarchitecture may be observed in the RISC-V documents, such as the recommended pair of multiplication instructions to obtain a full-size product.

2.2 Base Integer Instruction Set

RISC-V currently defines four base integer instruction sets, which are not compatible with each other. Nevertheless, they all have 32-bit integer-based instructions, most of which are present in all bases with the same or very similar encoding. Furthermore, all are comparably small and work with a flat address space that is byte-addressed. The RISC-V integer bases are the following:

- **RV32I** represents the default ISA base with a 32-bit address space and 32 registers at hand, each 32 bits wide. Since this base is used as the **selected base for this thesis**, the following text examines only the RV32I base.
- **RV32E** is a reduced version of RV32I, having only 16 registers. In other mentioned aspects, it stays the same.
- **RV64I** uses a 64-bit address space and, compared to RV32I, extends the width of registers to 64 bits. Also, RV64I defines new instructions to support work on both 32-bit and 64-bit data.
- **RV128I** supports a 128-bit address space with registers extended to 128 bits. This base is not frozen. It was created to prevent issues with insufficient address bits for memory addressing in the future.

When deciding which base would be used for this thesis, I quickly realized that a 32-bit one would be a good choice because a superscalar processor does not have to be 64-bit with the associated additional functionality. Then, when comparing RV32I and RV32E, I preferred the former due to its higher popularity. Moreover, RV32E advantages were not that important for this thesis, so overall, I considered RV32I a good starting point.

[2] PC – program counter (a register that holds the address of the current instruction)

RV32I might contain as low as 38 instructions[3], and as such, it may implement almost any extension. This trait is commonly used for unimplemented instructions' emulation. It is also beneficial for simple implementations where even operations like integer multiplication are not required. At the same time, RV32I was designed to be a sufficient compiler target.

2.2.1 Architectural Registers

The RV32I base defines a 32-bit PC register for holding the address of the current instruction. Among others, it is used for relative unconditional jumps and conditional branches. RV32I defines 32 general-purpose registers with a width of 32 bits. They are labeled `x0`–`x31` and are used as source and target operands of most instructions. The `x0` register holds zero value, ignoring any writes. That is very important to implement some operations using existing instructions; hence, it indirectly increases the number of available instructions.

If respective extensions are present, more registers are involved. Namely, it includes control and status registers (CSRs) used predominantly in the privileged part of RISC-V ISA, but they have some uses in the unprivileged part too. Next, there are 32 floating-point registers used for floating-point arithmetics. Another contributor to the register count may be custom extensions.

2.2.2 Instruction Formats

Like many RISC architectures, RV32I divides instructions into categories to speed up instruction decoding. Individual instruction categories have selected bits equal; other bits are used for further decoding. Generally, instructions are broken into these categories based on work with carried operands. RV32I recognizes six instruction formats in total; R-type, I-type, S-type, B-type, U-type, and J-type; as demonstrated in table 2.2. However, there are only subtle differences between S-type and B-type, and U-type and J-type consisting of extracting immediate values (offset of jumps has the least significant bit zero).

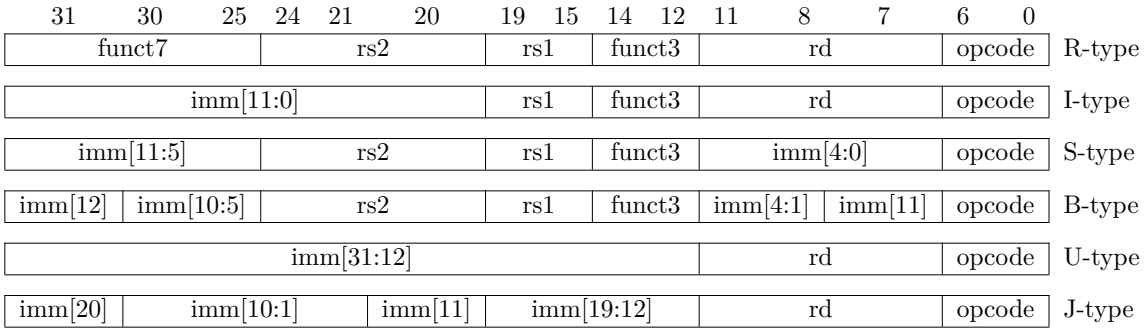

Table 2.2: **Instruction formats of the RV32I base ISA.** The *opcode*, *funct3*, and *funct7* bit fields are used for instruction decoding. The *rs1*, *rs2*, and *rd* fields represent indexes of two source registers and one destination register, respectively. An *imm[x]* bit represents the x-th bit of the immediate value stored in the instruction. Reproduced from [25].

Even though the arrangement of some bit fields in table 2.2 may seem inconsistent at first glance, simple rules are followed to allow efficient instruction decoding. If present, opcode and register fields always have dedicated placements. All immediate values are to be extended to 32 bits. Lower bits not present in the instruction word are filled with zeros,

[3]RV32I contains 40 unique instructions; however, some may be merged in simple implementations.

and sign extension is utilized to generate higher bits. Despite the variety of immediate values, their sign bits are always held at the same place in the instruction word (bit 31). The rest immediate value bits were placed to maximize the reuse factor.

The listing of all RV32I instructions would be out of the scope of this thesis. However, it is appropriate to know the differences between individual instruction formats. The formats, with instruction examples, are described in the following list:

- **R-type** instructions are known as a register-register type. They read two register values, perform an operation, and write the result to a register. E.g., subtraction instruction with an assembly form `sub x2, x3, x5`[4] (it means $x2 \leftarrow x3 - x5$).
- **I-type** gathers immediate-register types of instructions. It reads one source register and an immediate value delivered in the instruction. The value may represent a signed 12-bit number or, to some extent, be a part of the instruction encoding. The result is stored in a register. Load word instruction `lw x7, 11(x13)` presents an I-type example (meaning $x7 \leftarrow [x13 + 11]$).
- **S-type** reads two registers and a 12-bit offset value present in the instruction word. It is used for storing data in memory; no register write is performed. For example, store word instruction `sw x13, 17(x19)` that performs $[x19 + 17] \leftarrow x13$.
- **B-type** uses similar operands as the S-type. They are, however, used for conditional branches. For instance, the branch-if-less-than instruction `blt x23, x29, label` that causes jump to `label` if $x23 < x29$.
- **U-type** format uses a 20-bit upper immediate value embedded in the instruction. The result is written to a register. An example might be the load-upper-immediate instruction `lui x31, 0x815c5` that assigns `0x815c5000` value to `x31`.
- **J-type** uses a 20-bit offset value carried in the instruction for unconditional jumps. One register is used as a "link". E.g., the jump-and-link instruction `jal x5, label` stores the address of the following instruction to `x5` and jumps to `label`.

Apart from native instructions, RISC-V ISA mentions assembler pseudo-instructions, too. The pseudo-instructions get translated by an assembler into one or more native instructions. For example, the pseudo-instruction `not x9, x11` translates to the native instruction `xori x9, x11, -1`, which performs bitwise XOR operation (`-1` extends to 32-bit value filled with ones).

2.3 RISC-V Extensions

The instruction set extensions concept makes RISC-V a modular ISA. This is important for the RISC-V flexibility since it allows simpler and more complex implementations with the same ISA. To incorporate extensions into RISC-V, the instruction encoding space is broken into three disjoint categories:

- **Standard encodings** are used by RISC-V International for standard extensions. They do not collide with each other for a selected base ISA.

[4] Register names used in practice may differ based on the used application binary interface. Nonetheless, they are always mapped to the canonical `x0`–`x31` registers.

- **Reserved encodings** are not currently in use but can be utilized for the encoding space of the future standard extensions.
- **Custom encodings** are ready for custom extensions. Developers may utilize this encoding space for their specific functionality.

During the development, a standard extension goes through several states. It starts as a *draft*, where incoming changes are to be applied. Later, it shifts to the *frozen* state, where significant changes are not expected. Then, it ultimately goes to the *ratified* state. For instance, the extension for CSR instructions was created by detaching from the base ISA during its ratification process.

Frequently, standard extensions have a single letter assigned based on their focus. The most eminent extensions include the **"M"** extension, used for integer multiplication and division; the **"A"** extension, which is used for atomic memory operations; the **"F"** and **"D"** extensions, used in single and double precision floating-point arithmetics compliant with the IEEE 754-2008 standard [10]; the **"Zicsr"** extension that enables work with CSR registers; the **"Zifencei"** extension, which adds support for instruction-fetch fences; and the **"C"** extension that defines compressed 16-bit substitutes for most used 32-bit instructions. Derived RISC-V modularity is indicated in figure 2.1.

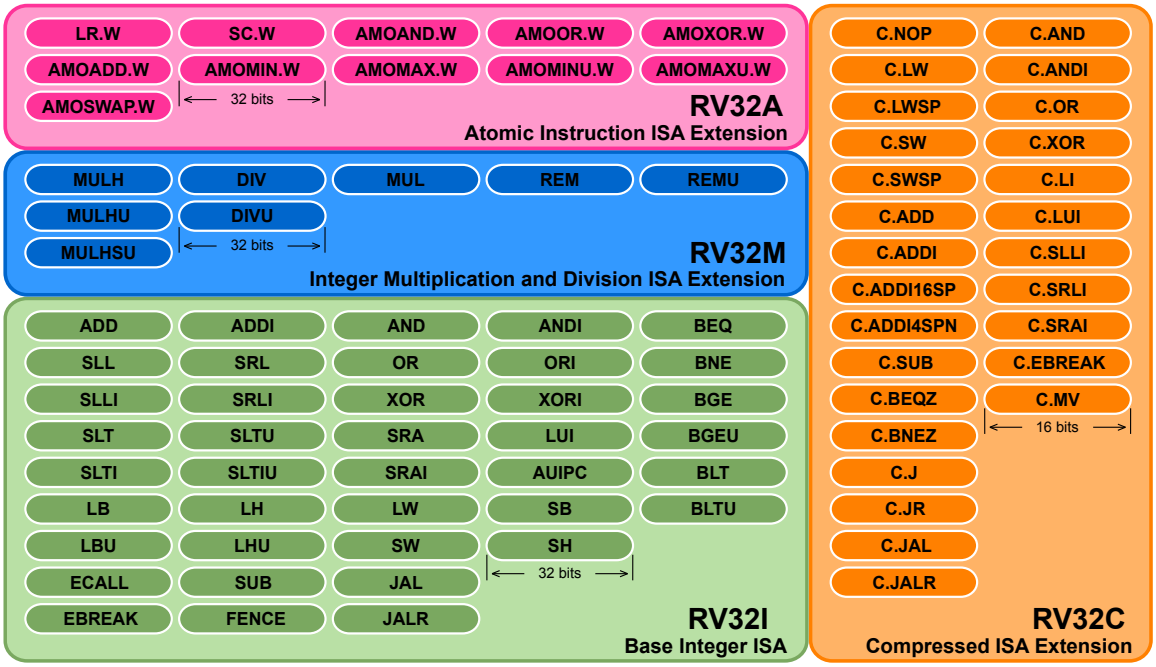

Figure 2.1: Demonstration of the RISC-V modularity on the RV32IMAC configuration with individual instruction contributions of each extension. Adapted from [6].

This thesis does not directly require any ISA extensions to fulfill the goals stated in chapter 1. And since it would make the design only more complex, no extension to the RV32I base is employed. However, it should be noted that the base itself might not be sufficient when more practical topics should be solved. Moreover, the concept of extensions is critical for this contemporary ISA and hence deserves to be mentioned.

Chapter 3

Architecture of Superscalar Processors

Superscalar processors can execute multiple instructions simultaneously (i.e., per clock cycle). This acceleration exploits instruction-level parallelism observed in the instruction stream and was preceded by some processor evolvement. In times of scalar processors, rough processor performance was measured in cycles per instruction (CPI) rather than in instructions per cycle (IPC) since, typically, more than one clock cycle was required to execute an average instruction. With the advent of superscalar processors, the preferred metric for rough processor performance has inherently changed to IPC. However, it is essential to note that IPC is just a fraction of the attributes required to obtain the actual processor performance. In the past, it was even shown that a processor clock rate is not a good performance indicator either — a phenomenon known as the megahertz myth [19]. More complex methods, such as running a sophisticated benchmark, are necessary.

Superscalar execution may be implemented in several ways. The most commonly used deployment of superscalar execution may be found in RISC and CISC (complex instruction set computer) processors. They divide further into processors with in-order execution, explored in section 3.2, and out-of-order execution, mentioned within subsection 3.3.1. Relevant processor representatives are provided in table 3.1. Regardless of a particular implementation, such processors dynamically check the data dependencies of instructions and ensure the safe execution of multiple instructions. The assistance of a compiler is beneficial but not necessary.

	RISC	**CISC**
In-order execution	MIPS R8000	Motorola 68060
Out-of-order execution	PowerPC 603	Cyrix 6x86

Table 3.1: Examples of comparably old superscalar processors based on other significant processor traits. Cyrix 6x86 was released in 1995, and others in 1994. In those years, out-of-order designs started to appear on a large scale.

VLIW (very long instruction word) processors are related to superscalar ones to some degree since VLIW instruction words explicitly contain more operations to be performed in parallel. Contrary to RISC and CISC, however, VLIW architecture demands intensive compiler assistance in resolving data dependencies and operations scheduling. In this manner, VLIW hardware is simpler but relies on static compiler decisions, which sometimes

are not very precise during actual program execution. Furthermore, VLIW architectures do not scale very well because they usually use fixed-length instruction words with a fixed number of possible parallel operations.

3.1 Introduction to Scalar Pipelined Processors

Before explaining superscalar approaches, it is favorable to start with scalar architectures, from which they evolved. Scalar processors can execute one instruction per cycle at most. This is an inherent limitation of the scalar design and a theoretical maximum of their instruction processing power. It is given by their ability to process only one data item at a time. After all, that is why they are labeled scalar.

Similarities were observed across all instructions. Each instruction might be broken down into steps needed to achieve the desired effect. To summarize, every instruction is fetched from memory and decoded, then performs some calculations during which it may access memory, and finally, stores the result somewhere. Based upon that, engineers swiftly recognized that multiple instructions might be processed concurrently; however, each is in a different stage of processing. This approach is illustrated in figure 3.1 and is recognized as instruction pipelining. Since the recognition of pipelining possibilities, practically all processors have utilized pipelining: ranging from older ones like Hitachi SH-2 (1993) to more contemporary ones like ARM Cortex-A77 (2019). Pipelining is so essential for performance that even the vast majority of microcontrollers currently use it.

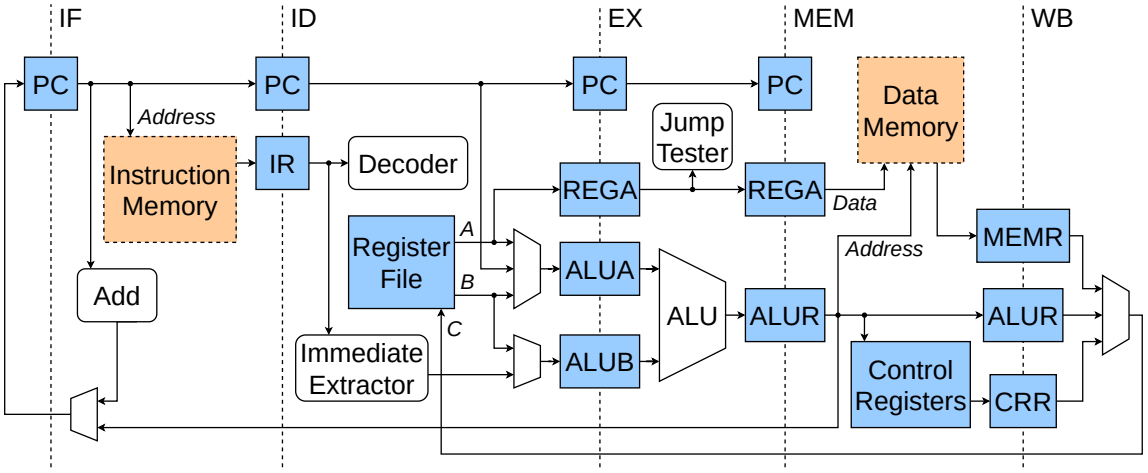

Figure 3.1: **Pipelined architecture of a scalar RISC processor.** It uses classic five-stage design — IF, instruction fetch; ID, instruction decode; EX, execute; MEM, memory access; and WB, writeback. The stages work concurrently. Instruction memory and data memory present respective memory interfaces. Adapted from [17].

The number of pipeline stages in general-purpose CPUs (central processing units) may vary significantly. Early microprocessors typically had only two independent units: one for fetching instructions and the other for their execution [3]. Later, the number of pipeline stages was continually growing until Intel Pentium 4 "Prescott" with its 31 pipeline stages appeared in 2004. A pipeline of such a length encountered diminishing returns due to relatively frequent pipeline hazards solving (pipeline hazards are explained later in subsection 3.1.2). Since then, pipeline lengths have stabilized at 10–20 stages.

3.1.1 Pipeline Stages

One of the most widespread pipeline architectures is a classic five-stage RISC pipeline (almost identical to one in figure 3.1). It is suitable for understanding how pipelining works in general. In such an architecture, each instruction goes through the following stages:

1. In the **instruction fetch (IF)** stage, the instruction is fetched from memory to the processor. Also, the address of the next instruction is computed.
2. The fetched instruction is decoded in the **instruction decode (ID)** stage, extracting an immediate value from the instruction word and preparing source register operands.
3. Then, the instruction is executed in the **execute (EX)** stage. That includes operations like subtraction, bit shifting, and memory address computation. Also, jump instruction conditions are evaluated here.
4. Instructions that access memory can do so in the **memory access (MEM)** stage. If the jump instruction condition was evaluated as positive in the EX stage, the target address is propagated to the IF stage. Also, access to auxiliary control and status registers may be performed here.
5. In the **writeback (WB)** stage, the instruction result is written to the destination register if not previously written elsewhere (e.g., to memory).

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
i_1	IF	ID	EX	MEM	WB				
i_2		IF	ID	EX	MEM	WB			
i_3			IF	ID	EX	MEM	WB		
i_4				IF	ID	EX	MEM	WB	
i_5					IF	ID	EX	MEM	WB

Table 3.2: **Pipelined execution of five independent instructions.** Each instruction has its color. The t_n denotes a processor cycle number, and the i_n denotes the n-th instruction of the stated instruction batch.

The processing of multiple instructions is demonstrated in table 3.2. As far as there is no conflict among the instructions, all are processed exactly as depicted. This way, the maximum number of instructions in progress is five (as observed at time mark t_5). Should the conflict occur, it must be dealt with, which could also change the execution schema.

3.1.2 Pipeline Hazards

In ideal cases, scalar pipelined processors may execute one instruction per cycle. That is, however, not possible due to hazards, which occur regularly during the program execution in pipelined processors. Three types of hazards are recognized:

- **Control hazard** is closely related to jumps. It happens when a jump is taken, but incorrect instructions are prepared in the preceding pipeline stages.

- **Structural hazards** occur when multiple instructions require the same processor resource. This may happen, for example, when there are two consequent multi-cycle instructions like integer division.
- **Data hazard** happens when there is a data dependency between instructions in a way that pipelining would break data consistency across those instructions. Based on data access in program order, there are three categories of data hazards[1]. They are listed below, each with an example of a triggering instruction sequence:
  - Read after write (true dependency): $\boldsymbol{r1} \leftarrow r2 + r3$, $r4 \leftarrow \boldsymbol{r1} + r6$;
  - Write after read (anti-dependency): $r1 \leftarrow \boldsymbol{r2} + r3$, $\boldsymbol{r2} \leftarrow r5 + r6$;
  - Write after write (output dependency): $\boldsymbol{r1} \leftarrow r2 + r3$, $\boldsymbol{r1} \leftarrow r5 + r6$.

These pipeline hazards, however, are not written in stone. There are several methods used in scalar designs to eliminate them. The most universal, while also the least effective, is called pipeline bubbling. The processor must be able to detect hazards, and once one is detected, it stalls the affected stage (and stages before) until the hazard gets resolved. Branch predictors, described in subsection 3.3.3, are very helpful for reducing the occurrence of control hazards. Data hazards may often be successfully minimized using a technique called operand forwarding. It makes the stages' results immediately available for preceding stages without having to wait for register writes. Even so, the processor still has to wait for some results (for instance, when reading data from memory).

3.2 Superscalar Approaches for In-Order Execution

Even though superscalar execution can take many forms, for further explanation is beneficial to build on the scalar pipelined design, as described in section 3.1, and double all relevant construction blocks. This maximizes the potential of superscalar execution, which also means there is nearly no loss of generality. After that, it is necessary to handle emerged pipeline hazards and decide on allowed two-instruction bundles that can be run in parallel. This is known as a dual-issue pipeline. In general, should an architecture enable more instructions to be run in parallel, we are talking about multiple-issue pipeline design.

Such a design usually does not change the order of instructions when executing them (i.e., it follows the program's instruction order). This approach is known as in-order instruction execution, which is also referred to as static instruction scheduling. It is called static scheduling because a compiler is ultimately responsible for creating instruction sequences, whether or not it takes the target microarchitecture into account. Actual examples of some superscalar in-order processors are provided in section 3.4.

Also, the number of cycles required to perform implemented instructions plays an important role in multiple-issue designs. More such variability usually means that there must be more independent functional units in the design to which instructions are dispatched. It also complicates the pipeline control logic. That is one of the reasons why this thesis works only with basic instructions. Among them, there are only a few instructions — like loading data from memory — that could take more than one cycle.

[1] It is worth mentioning that the pipeline in figure 3.1 might not suffer from some types of data hazards at all. They, however, may appear once some microarchitectural changes are made.

3.2.1 Basic Dual-Issue Pipeline Design

Since the mentioned dual-issue in-order design is also the most relevant for the thesis, this text is primarily dedicated to that configuration. As indicated sooner, it has many similarities with the scalar pipelined architecture introduced in section 3.1, so it makes sense to use the scalar architecture as a reference when explaining the dual-issue design additions. Nonetheless, the text intentionally avoids sliding into explaining specialized concepts and details. These are explored later when needed.

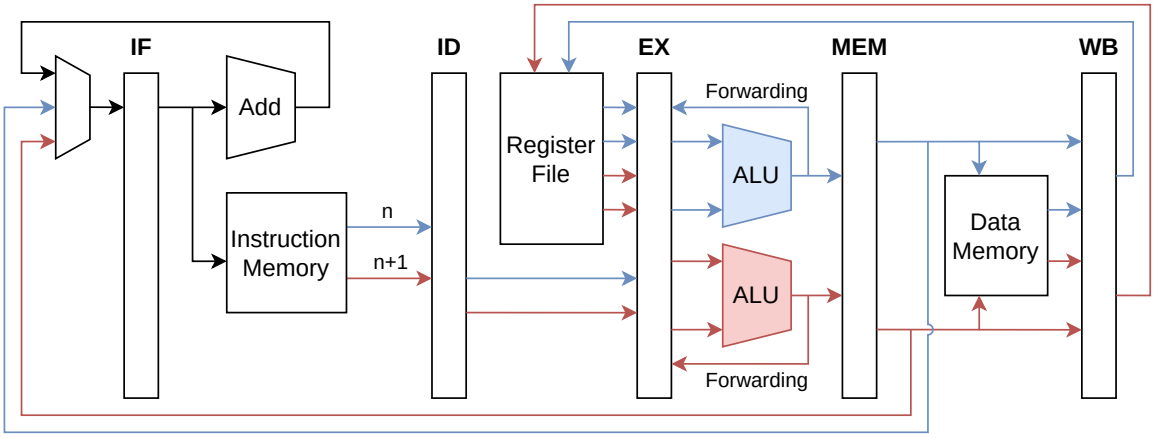

Figure 3.2: **Generic dual-issue pipeline design with five stages.** Blue and red colors represent two individual pipes that can operate simultaneously. The IF, ID, EX, MEM, and WB acronyms highlight the individual pipeline stages.

Both the scalar and the chosen superscalar architecture use pipelining. The layout of such a dual-issue superscalar pipeline design is depicted in figure 3.2. This pipeline can execute two program-ordered instructions simultaneously under certain circumstances, like the absence of data dependencies. It is worth pointing out that the blue and red pipes in the figure are of general use (i.e., both can execute any implemented instruction). In many real designs, however, some pipes cannot accept and execute all instructions — as in the case of the original Intel Pentium (1993) with its U-pipe and V-pipe [7]. As a rule of newer designs, branch and memory access instructions tend to be restricted to a particular pipe. Regardless, all superscalar designs must have solved an identical set of questions not present in scalar designs.

3.2.2 Changes Compared to Scalar Pipelined Processors

When it comes to the scalar to superscalar pipelined architecture transformation, the crucial challenges are already known, as are their potential solutions. Hence mostly, it depends solely on the designers' decisions (and their motivation) on how to solve them. These challenges are demonstrated using the generic superscalar dual-issue design introduced sooner in this section, but with some effort, they may be generalized:

- The first issue correlates with **instruction fetching**. Instead of one, it is required to fetch at least two per cycle. In simpler implementations, it can be easily solved by increasing the width of data transferred from the instruction memory. However, if an instruction cache is in use, we must also handle or prevent the situation of reading instructions when they are located in different cache lines. Besides, in case a branch

predictor is installed, it is highly advantageous to use a trace cache for non-sequential instruction fetches [16].

- Another issue faced would be **decoding two instructions** simultaneously. Whereas decoding itself presents no big problem, solving data dependencies might be considerably more difficult. Apart from usual checks inside each pipe, cross-pipe checks, which are even applied within the instruction pair itself, must take place as well. Furthermore, the two execution pipes may not be identical and able to run all instructions, so additional scheduling control logic may be attached. In any case, twice as many register-read ports are needed.

- The **execution of instructions** brings one prominent issue — operand forwarding. With a multiple-issue design and a higher number of pipeline stages, the complexity of operand forwarding across pipes and their stages dramatically increases. For simpler dual-issue designs, it may be feasible to employ complete forwarding. For more complex ones, pipes might be grouped into forwarding clusters, in which complete forwarding takes place. This would bring a new possibility of instruction scheduling since dispatching instructions with data dependencies to the same forwarding cluster would be desirable. Otherwise, stalling may occur.

- Whenever an **instruction should be stalled**, the relevant instruction pair may be stalled as a whole. That stalling would automatically carry over to previous pipeline stages as well. There is, however, space for improvement (at the cost of some control logic). If the younger instruction of the two is stalling, the older one may leave its instruction bundle and continue with the execution, as indicated in figure 3.3. A similar mechanism is mandatory for the decoding phase to prevent deadlock caused by data dependencies within the instruction pair itself.

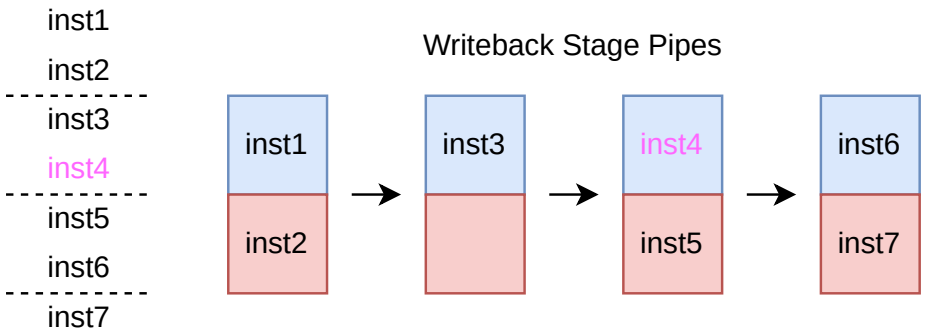

Figure 3.3: **Relaxed dual-issue in-order execution.** The instructions executed are represented by *inst1–inst7*. Pink *inst4* was stalled for one cycle, which caused transferring it to a later instruction bundle. Blue and red colors mark the writeback stages of the two pipes — it demonstrates how individual instructions are finished.

- Having **jump instructions** supported in both pipes brings increased complexity of control logic. It even must correctly manage two jumps in a single instruction bundle. Furthermore, if dedicated hardware for jump address computation is involved, it might be duplicated. When using a branch predictor, supporting jumps in both pipes would lead to multi-ported memories for such prediction, which could notably increase required hardware resources.

- When **accessing memory** from both pipes, it eventually requires adding a second port to data memory. If no data cache is involved, it may be harder to achieve such memory access. Hence, especially in simpler implementations, memory instructions are either serialized in the memory access stage or dispatched only to the selected pipe with exclusive memory access.
- When the **final results** are available, both must be written to the register file at the same time. This is accomplished by adding a second register write port.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
i_1	IF	ID	EX	MEM	WB		
i_2	IF	ID	EX	MEM	WB		
i_3		IF	ID	EX	MEM	WB	
i_4		IF	ID	EX	MEM	WB	
i_5			IF	ID	EX	MEM	WB
i_6			IF	ID	EX	MEM	WB

Table 3.3: **Execution of six independent instructions with dual-issue pipelining.** Two instructions are always in the same pipeline stage. Each instruction has its color. The t_n denotes a processor cycle number, and the i_n denotes the n-th instruction in the program order of the stated instruction batch.

The dual-issue processing of multiple instructions is demonstrated in table 3.3. The depicted instructions hold no data dependencies, which enables finishing two instructions per cycle. The total number of possible instructions in progress is 10. Compared to the scalar design introduced in section 3.1, the maximal theoretical throughput of executed instructions in the dual-issue superscalar design is double. As positive as it may sound, due to the mentioned challenges above, it is very hard to reach such acceleration.

3.3 Advanced Superscalar-Relevant Techniques

Many popular advanced techniques deployed in superscalar processors are not directly connected with the thesis. Nonetheless, some are mentioned in this section to extend the thesis's context. They are typically used to mitigate the negative effects of pipeline hazards or increase processor performance in another way. Usually, most such techniques are used together to reap the benefits of each, maximizing the total performance. Sometimes, superscalar processors tend to be even bound with the techniques explained herein.

If the reader wants to get even more familiar with the advanced techniques, I recommend having a look at [24] — a master's thesis on creating a simulator of a superscalar RISC-V processor. In the thesis, the advanced techniques are described in more detail, and basic ones are explained with a different point of view in mind, which might be beneficial for the reader too. Finally, the created simulator[2] *is very helpful in understanding all these.*

[2]The simulator may be downloaded from https://nextcloud.fit.vutbr.cz/s/eWSxejPno3pxnQW.

3.3.1 Out-of-Order Execution

Out-of-order execution (OoOE), also known as dynamic instruction scheduling, exploits instruction-level parallelism by allowing a processor to execute instructions out of program order. It effectively eliminates selected data hazards listed in subsection 3.1.2 and increases the utilization of the processor's functional units. As opposed to pipelining and superscalar execution, the OoOE popularity is slightly lower due to its higher demands on hardware resources. When OoOE is deployed, it typically comes also with a dedicated load-store unit and branch prediction, which are described later. Hence, its exemplary application may be found in high-performance processors. This has been true for a few decades; for instance, ranging from the first custom AMD (Advanced Micro Devices) processor with OoOE, AMD K5 (1996), to newer models, such as AMD Ryzen 5 3600 (2019). A similar procedure may be observed with Intel processors, too.

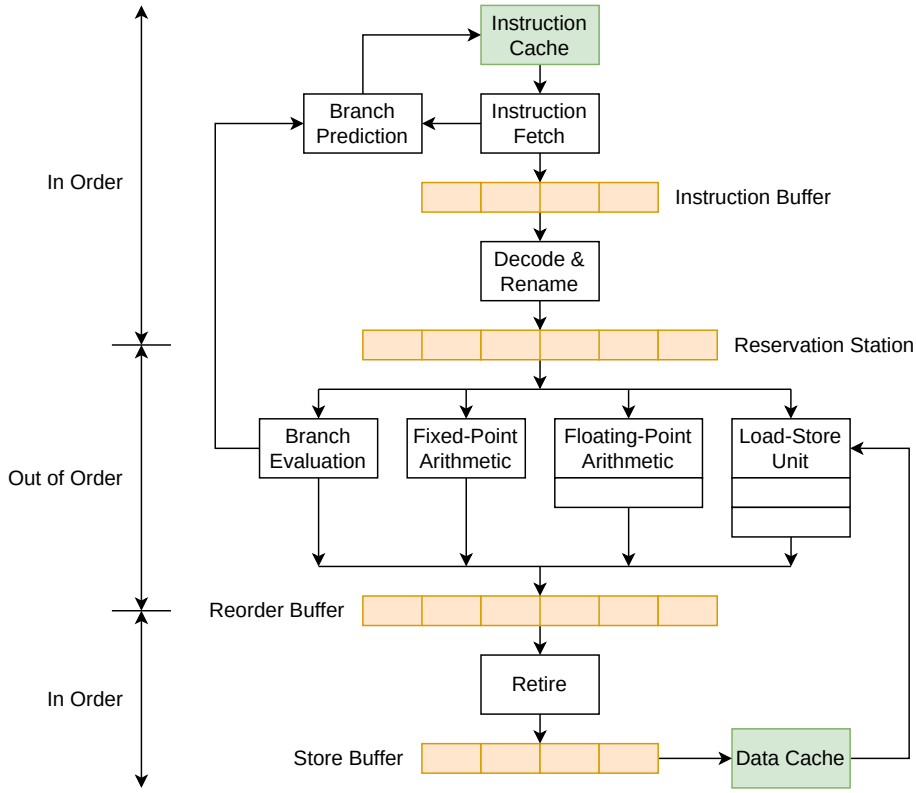

Figure 3.4: **Typical superscalar out-of-order processor structure.** This example has four distinct functional units, each with a potentially different number of pipeline stages.

Usually, OoOE methods fetch multiple instructions in order, execute them out of order, and store their results in order — all pipelined. Essentially, two OoOE methods are recognized. The simpler one is called *scoreboarding* [20], where instructions get executed as soon as they are not conflicting with others; otherwise, the affected instructions wait. A more advanced method is called *Tomasulo's algorithm* [21] and is considered the default in the OoOE area today. Compared to the former, it uses register renaming to dissolve false data dependencies, making full OoOE possible. True data dependencies are treated in a reservation station. A reorder buffer is used for storing instructions' results in the correct order. The architecture of such an out-of-order processor is shown in figure 3.4.

3.3.2 Load-Store Unit

Optimal access to memory is critical for processor performance. The most flexible means for achieving that is cache, which fundamentally decreases the average memory access latency. For data memory, deployment of a dedicated load-store unit (LSU) is also very beneficial. In fact, it is an essential component in out-of-order architectures. LSU deals with data dependencies on the memory level. It contains a *store buffer* to keep track of recent memory write operations and accelerate reading data of the same addresses. LSU also has a *load buffer*, which is used for speculative memory reads of addresses currently not present in the store buffer.

3.3.3 Branch Prediction

Pipelined processors without branch prediction fetch instructions in the order they are stored in memory. Hence, once a branch instruction is evaluated as taken, the following instructions in the pipeline are invalidated. The longer the pipeline is, the more cycles it takes to start executing the actual branch target. Branch prediction can combat such an effect by deciding which instructions to fetch immediately after the branch instruction. Later, if the genuine branch outcome matches the prediction, no execution delays occur. Until the prediction is confirmed, execution is speculative and must be treated as such.

In principle, branch prediction must provide guesses for two questions — whether a jump occurs and where it jumps. For this purpose, branch prediction maintains two respective tables. The first is recognized as the *branch history table* (BHT), and the second is called the *branch target buffer* (BTB). As prediction must be quick, these tables are read concurrently with instructions fetching. To index the tables, instruction addresses themselves are used. Since prediction for subroutines' return jumps would be inaccurate due to multiple entry points, the *return address stack* (RAS) is employed in relevant cases to accelerate subroutine calls.

3.4 Examples of Superscalar In-Order Processors

Since the thesis is focused on creating a superscalar processor model, it is reasonable to introduce some of the existing relevant examples first. The area of interest naturally falls to superscalar in-order RISC-V processors and their microarchitectures, as described later in this section. Nevertheless, it would be fair also to briefly mention some such in-order processors before the RISC-V era:

- **Alpha 21064** [14] was designed by DEC (Digital Equipment Corporation) in 1992. It is a 64-bit dual-issue RISC microprocessor with four functional units: integer, floating-point, address, and branch. It deploys pipes with up to 10 stages.
- **PA-7100** [28] was introduced by HP (Hewlett-Packard) in 1992. It implements 32-bit PA-RISC (precision architecture RISC) instruction set and two-way superscalar execution with less diversity than Alpha 21064 — only one floating-point and one integer instruction may be paired for execution. The pipeline has five stages.
- **UltraSPARC**[3] [8] was developed by Sun Microsystems in 1995 as a 64-bit four-issue processor with nine pipeline stages and the same number of functional units. Since

[3] SPARC – scalable processor architecture

it is more advanced than both above, it also possesses multimedia support and some capabilities found in out-of-order processors.

3.4.1 VeeR EH1 Processor

VeeR EH1, previously known as SweRV EH1, was created by Western Digital and, in 2019, published as an open-source processor[4]. Later, its maintenance was handed over to the CHIPS (Common Hardware for Interfaces, Processors and Systems) Alliance. VeeR EH1 represents an *RV32IMC_Zicsr_Zifencei* processor (RISC-V extension naming is explained in sections 2.2 and 2.3). It is described in SystemVerilog language [11].

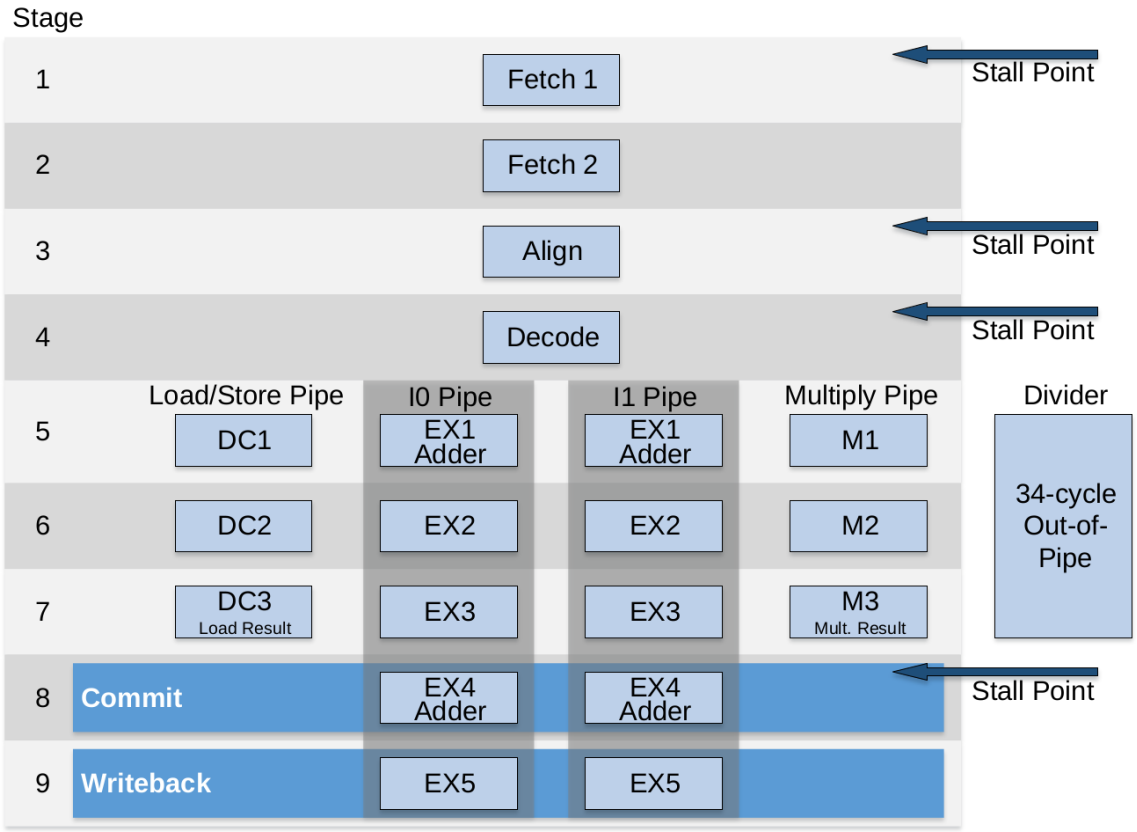

Figure 3.5: **VeeR EH1 nine-stage pipeline design.** For instruction execution, there are two shorter pipes (*load/store* and *multiply*), two extended ones (*I0* and *I1*) with result writing, and a sequential divider, which is not pipelined. Reproduced from [5].

VeeR EH1 is a dual-issue superscalar processor whose pipeline architecture is depicted in figure 3.5. It employs four ALU (arithmetic logic unit) engines. When operands are not ready for the first set of ALUs (EX1), the calculation is skipped until reaching the second one (EX4), where eventual stalling may be initiated. Along with installed operand forwarding from EX5 to EX4, this approach eliminates many read-after-write data hazards, leading to higher performance. That even applies to load-use hazards[5].

[4] VeeR EH1 repository – https://github.com/chipsalliance/Cores-VeeR-EH1

[5] The load-use hazard is a specific data hazard that happens when an instruction reads an operand whose value should be loaded from memory but is not yet.

The execution of VeeR EH1 is mostly in-order. Due to its dedicated LSU, it can perform non-blocking memory loads. It means the pipeline stalls only when requiring the loaded value; otherwise, the following instructions are executed, and the actual load is finished later. Among other advanced techniques deployed in the core, we can find branch prediction with a return address stack — both with configurable parameters.

Based on the user configuration, the VeeR EH1 core complex may contain an instruction cache with an optional ECC (error correction code). The data cache is absent. However, EH1 might include instruction and data tightly coupled memory (TCM) that serve as scratchpad memory [2]. External instruction and data memory interfaces may be picked from two popular AMBA (advanced microcontroller bus architecture) bus protocols. The core complex integrates, among others, a programmable interrupt controller for smoother interrupt handling.

From the commercial circles, SiFive E76 [18] design is similar to EH1, although it offers many additional features. For instance, it optionally might be equipped with an FPU (floating-point unit) and support for simple privileged applications.

3.4.2 biRISC-V Processor

biRISC-V is an open-source superscalar processor core published on GitHub[6] by an independent developer. It adopts dual-issue in-order execution of instructions. biRISC-V is an *RV32IM_Zicsr* processor (with some custom features) and supports user, supervisor, and machine privilege levels [26], allowing it to boot Linux. It is written in Verilog [9].

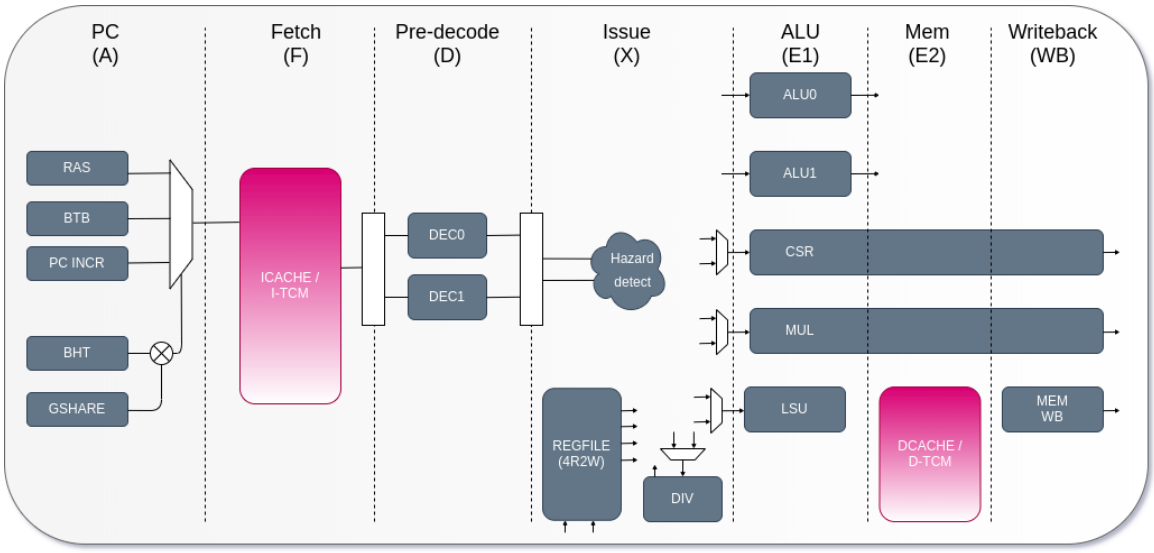

Figure 3.6: **biRISC-V seven-stage pipeline design.** It has five regular pipes with various latency and an out-of-pipeline divider. *Gshare* is a type of branch predictor. Reproduced from the biRISC-V repository.

The pipeline architecture of biRISC-V is illustrated in figure 3.6. It has two integer ALUs, and one of each: CSR unit, multiplier, and an LSU. The division is executed in a dedicated unit placed out of the pipeline, meaning it has a higher execution latency. biRISC-V fetches instructions in 64-bit blocks, allowing it to receive two neighboring in-

[6] biRISC-V repository – https://github.com/ultraembedded/biriscv

structions at once. It can also finish the same number of instructions per clock cycle to provide sustainable dual-issue execution.

biRISC-V is also highly configurable — including but not limited to the number of pipeline stages and even various forwarding options. Also, it brings configurable branch prediction, instruction and data caches and TCMs, a popular AMBA bus protocol for core's interfaces, basic MMU (memory management unit) support, and others. biRISC-V was created with FPGA (field-programmable gate array) in mind.

Chapter 4

Microarchitecture Design and Implementation

The created microarchitecture adopts the RISC-V instruction set introduced in chapter 2. Based on the concepts explained in chapter 3, it exposes a dual-issue seven-stage pipeline with in-order instruction execution. The created processor is called *Super RISC-V* to highlight its superscalar capabilities. The processor's SystemVerilog [11] model description might be found on the included media storage as mentioned in appendix A (preferred) or in a Git repository as mentioned in chapter 1.

Although the processor embraces RISC-V, it only supports base integer instructions collectively labeled RV32I. There is also no support for any privileged concept from [26]. Hence, the processor does not support interrupts, exceptions, environment calls, memory protection, etc. As a result, `ECALL` and `EBREAK` instructions are implemented as NOP (no operation). Nevertheless, the superscalar principles do not rely on those, so it does not decrease the value of this work.

The processor architecture is divided into four main units, each taking care of a specific functionality and communicating with others. In addition, three of them are spread over several pipeline stages. This is depicted in figure 4.1. The essential four units of the processor are listed here:

- **Instruction fetch unit (IFU)** takes care of fetching instruction pairs from memory, possibly storing them in the instruction fetch buffer (described in section 4.3) and then delivering them to the decoding unit.
- **Decoding unit (DEC)** decodes an instruction pair, reads the required operands from the registers, and performs checks of various pipeline hazard types before letting it pass to the execution unit. DEC also initializes the first phase of operand forwarding, which is explained in section 4.4.
- **Execution unit (EXU)** performs the calculation itself and can forward results of older instructions to newer ones so that they do not need to wait for the actual operand registers writes. This unit also evaluates and performs branches. Its two pipes, also known as issue slots, are labeled EXU 1 and EXU 2 and they are identical. Nevertheless, they both form EXU as such.

- **Load-store unit (LSU)** manages data access to memory. It receives requests from the executing unit and returns results to it. It should be noted that this LSU is simpler than one explained in subsection 3.3.2.

The processor uses two isolated interfaces to communicate with memory. The first is for instructions and is controlled by IFU, the second is for data and is controlled by LSU. Both deploy a limited form of AMBA 3 AHB-Lite protocol [1]. More details are provided in section 4.5.

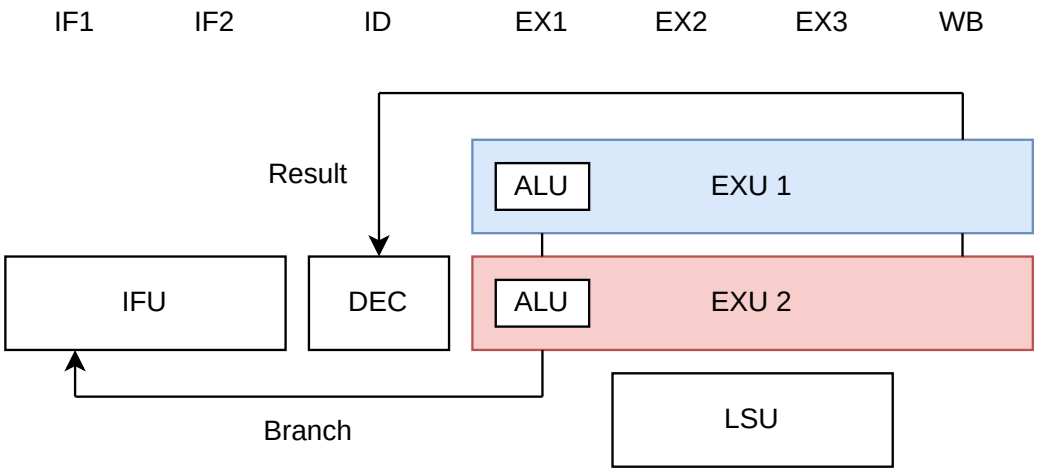

Figure 4.1: Pipeline diagram of the created processor with individual units broken into pipeline stages. IF1, IF2, ID, EX1, EX2, EX3, and WB represent pipeline stages. IFU, DEC, EXU 1/2, and LSU are the main processor units. Blue and red colors represent two issue slots for executing instructions.

4.1 Main Ideas of Created Microarchitecture

The main defined requirement for the presented superscalar processor was to be as simple as possible yet still practically usable. Using only essential superscalar approaches to reap the maximum associated benefits in an ideal case while ignoring complex structures capable of handling arbitrary cases. This strict requirement forced me to start looking at problems, especially performance-related ones, from a different point of view. Generally, the simpler the processor is, the fewer resources it requires. Hence, the idea of focusing the processor on environments with constrained resources such as power consumption or silicon area size.

Due to the complexity of this work, it must be stated that some techniques used in the microarchitecture are influenced primarily by good practices commonly used in the semiconductor industry [22]. I did not run processor synthesis or power analysis to provide exact measurements. This would require a large amount of additional work while not being part of the declared thesis goals.

4.1.1 Deterministic Issue Slot Scheduling

In minimal, the created processor must fetch two instructions per clock cycle. The used instructions are 32 bits wide, so need fetching 64-bit blocks (instruction pairs). In terms of

hardware resources requirements, the simplest way of doing so is by performing only fetches aligned to 64 bits and in case one instruction of the pair is unwanted, discard it.

The key decision connected with aligned fetches was to create a microarchitecture with deterministic instruction scheduling to the processor's two issue slots based exclusively on the instruction address. That immediately implies that the compiler has information on how generated instructions will be executed on the processor and, hence, it can perform many static optimizations very efficiently. Under certain conditions, on the other hand, it can lead to decreased performance because the processor cannot reschedule instructions dynamically to a different issue slot — it cannot merge and execute instructions from two different instruction pairs. Also, this prevents future support for compressed RISC-V instructions as they are 16-bit, which would break the static issue slot scheduling. At this cost, however, the theoretical required resources of the processor are lower and a specialized compiler can compensate for the mentioned issues.

4.1.2 Other Ideas for Low Power Consumption

Another method to reduce the processor's theoretical demands on power consumption is conditional writes to almost all its registers. This means that registers are only written when their value can really be used, thus avoiding unnecessary switching activity related to their change. This approach is also typically utilized by hardware synthesis tools for further benefit.

Even though some jump instructions might be evaluated and performed sooner in the pipeline, the microarchitecture defines a single point of jumps where all jump instructions have their operands ready. This decreases performance in some cases but helps keep the required hardware resources lower. Also, if branch prediction is added, this negative impact would be almost completely erased, not to mention the general benefits of branch prediction.

4.1.3 Ideas for High Processor Frequency

Despite the restricted hardware resources focus, the designed microarchitecture also has high frequencies in mind. That is why it contains seven pipeline stages and not less, enabling it to have optimal paths to memory. The memory paths are typically on the critical path[1] of designs and limit the maximal possible processor frequency the most.

Another potential limitation of high frequencies for dual-issue with this length of pipeline is operand forwarding. The processor needs to forward data from each concerned pipeline stage from both issue slots to both issue slots' operands, whose count might be up to four. There are 24 forwarding paths in total in the design and chances would be very high that one of them appears on the critical path of the design, should a standard approach be used. The proposed solution for this is introduced later in section 4.4.

Ultimately, both mentioned ideas for making high processor frequencies possible were also reflected during the early development stage of the processor when considering the required processor resources of each supported instruction. Instruction calculation results might be divided into two types — the first result type is written to a register (a subject of forwarding), and the second is used for addressing (using memory paths). The idea was to break these two types of calculations even on a hardware level since it would make them practically independent, bringing another potential improvement in frequency. Hence, the

[1] A critical path is the longest path in the design.

first type is calculated by ALU (arithmetic logic unit), and the second is calculated by AGU (address generation unit). Concrete used resources of instructions are stated in table 4.1.

Instructions	ALU		AGU		Extra Operands	CPU State Changes
	s1	s2	s1	s2		
LUI	0	imm				rd
AUIPC	PC	imm				rd
JAL	PC	4	PC	imm		PC, rd
JALR	PC	4	rs1	imm		PC, rd
Conditional branches	rs1	rs2	PC	imm		PC
Load instructions			rs1	imm		rd
Store instructions			rs1	imm	rs2	
Immediate arithmetic	rs1	imm				rd
Register arithmetic	rs1	rs2				rd

Table 4.1: **Used processor resources by each instruction.** The *s1* and *s2* labels are operand sources of computation units; *rs1*, *rs2*, and *rd* are register operands; and *imm* are immediate value operands.

4.2 Individual Pipeline Stages and Their Cooperation

As mentioned earlier, the created microarchitecture defines seven pipeline stages for enhancing its performance as instruction pipelining increases the maximal possible target frequency of the processor. Due to that, special attention was paid to balancing all pipeline stages' logic as the final timing is affected by the one with the longest logic latency.

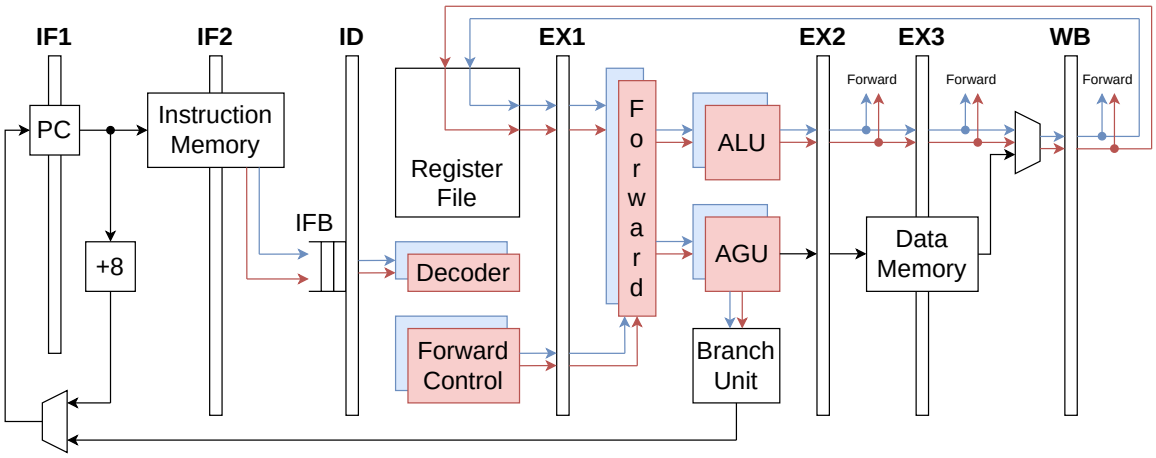

Figure 4.2: **Detailed pipeline diagram of the created processor.** IF1, IF2, ID, EX1, EX2, EX3, and WB represent pipeline stages. IFB is an instruction fetch buffer. Blue and red denote exclusive parts for the issue slots, others are shared. Please note that only one arrow is used for a pair of read operands.

Figure 4.2 shows the created microarchitecture with individual pipeline stages and their functional units. In the following subsections, each pipeline stage is described. Also, using an example program, their cooperative execution of instructions is explained after. Besides

others, these should answer questions raised in subsection 3.2.2 on challenges connected with dual-issue execution compared to the scalar one.

4.2.1 First Instruction Fetch Stage (IF1)

The IF1 stage contains the architectural PC register, which holds the address of the next fetched instruction. As instructions are 32 bits wide, its value must be divisible by four. If the PC value is also divisible by eight, we say its value is *aligned* and a 64-bit fetch is started. If it is divisible solely by four, the PC value is considered *unaligned*[2] and only the higher instruction of the pair is to be fetched. This is an implication of the principle explained in subsection 4.1.1.

In case a branch is indicated from the EX1 stage, the PC is updated with the jump address on the next clock cycle. If stalling is signaled, the PC value does not change. In other cases, the PC is increased by four or eight so that the new PC value is aligned. This ensures that the following fetching throughput is optimal.

4.2.2 Second Instruction Fetch Stage (IF2)

The IF2 stage finishes memory transfers started in IF1 and provides the fetched instruction pairs to ID. In case a stalling is signaled, it automatically stores the current pair in IFB (instruction fetch buffer). Once stalling ends, the first instruction pairs provided to ID are from IFB for correct program execution. More information about IFB and its mechanisms is provided in section 4.3.

4.2.3 Instruction Decode Stage (ID)

The ID stage decodes both instructions in the pair if their valid bits are set. This closely associates with the approach of conditional writes mentioned in subsection 4.1.2. If a valid bit is not set in IF2 (e.g., due to flushing or unaligned PC), the appropriate ID register values are not refreshed. They also further propagate to later pipeline stages.

Immediate operands are generated here. Also, register operands are read. The register file has four read ports for that. Additionally, it defines two write ports for writing results from the WB stage. These results are also forwarded to the read ports in the same cycle to ease later comprehensive forwarding control and related data paths. This approach might be recognized as cross-forwarding in other works [12].

Also, most pipeline hazards are solved in ID. If the newer (higher) instruction of the pair must be stalled for some reason, the older (lower) one is dispatched to EX1 for execution. If the older one is stalled, the whole instruction pair is stalled to keep the correct, in-order execution of the program. In case both instructions of the pair use LSU, they are serialized as LSU can accept only one request per clock cycle — the older instruction is dispatched to EX1 first, and the newer one is dispatched one clock later. The same solution is used when there is a true data dependency inside the instruction pair itself.

The ID stage also constantly checks for possible data hazards with existing results in the pipeline for used register operands. If not possible to forward them and dispatch the current pair immediately, it initializes stalling until the data is ready (e.g., when waiting for memory load). Note that the ID stage does not forward data itself, ID just prepares control values for the EX1 stage to actually forward it.

[2]Please note that if the PC value is not even divisible by four, it is caught by an assertion statement during simulation, which leads to its fatal exit.

4.2.4 First Execution Stage (EX1)

The EX1 stage is the first stage for instruction execution. Nevertheless, most of the instruction results are prepared in this stage. It contains two ALUs, two AGUs, and a shared branch unit. Before starting a computation, it finishes operand forwarding initialized in ID. From this stage further, there is a flag per each issue slot indicating whether its respective result is ready. Except for memory loads, this flag is set here in EX1. These flags are key for solving data hazards. More can be found in section 4.4, where forwarding is described.

Even though the branch unit is shared between issue slots, EX1 can evaluate and perform branches in the correct order from both issues in one clock cycle. In such a case, conditions are evaluated in ALUs, addresses are computed in AGUs, and the branch unit just evaluates all outcomes. If a jump should be performed from the older instruction of the pair, the newer one is discarded and its effects are canceled. Once a branch is confirmed, EX1, ID, and IF2 stages are flushed on the next clock cycle, giving us a *branch penalty* of three cycles. As indicated earlier, this is something that a branch predictor can fight with [13].

4.2.5 Second Execution Stage (EX2)

If applicable, the EX2 stage initializes data memory access in LSU. The AGU result from EX1 is stored in EX2 and used for memory addressing. Whether an instruction of the instruction pair uses LSU or not, its progress throughout the pipeline is always tracked in the EXU unit. No unexpected events during the transfer that could break this synchronization are supported now — more information is provided in section 4.5 on deploying the AHB-Lite bus protocol.

4.2.6 Third Execution Stage (EX3)

If data memory transfer was initialized in EX2, the memory response is available in the EX3 stage. In the next clock cycle, eventual data loaded from memory is also available for forwarding, so setting all instruction result ready flags here. That implies the *load-use latency* is two cycles.

4.2.7 Writeback Stage (WB)

The WB stage principally writes results prepared in the previous pipeline stages to the register file using its dedicated two write ports. These writes include cross-forwarding on the register file side and their values are also normally forwarded to EX1, meaning the WB logic and paths are around the same as other stages, not shorter.

If there is a "write after write" conflict within the instruction pair, only the result of the newer instruction is written to the register file. This is currently handled in the register file itself.

4.2.8 Pipelined Execution of Instructions

To better understand the overall instructions execution of the created processor, figure 4.3 shows an example program and the distribution of its instructions to the processor's pipeline stages at a selected time. The example program adds two vectors, scaling the first one by two before. The figure also demonstrates the static partitioning of the instructions into the issue slots using blue and red colors. The colored numbers might represent source code

lines rather than instructions' addresses. Each instruction pair is separated from the others by small horizontal lines. The grayed-out numbers in parentheses are lines of jump targets.

```
 1  beqz x10, done (12)
 2  lw x21, 0(x11)
 3  lw x22, 0(x12)
 4  addi x11, x11, 4
 5  addi x12, x12, 4
 6  slli x21, x21, 1
 7  addi x10, x10, -1
 8  add x23, x21, x22
 9  sw x23, 0(x13)
10  addi x13, x13, 4
11  j next_loop (1)
12  ...
```

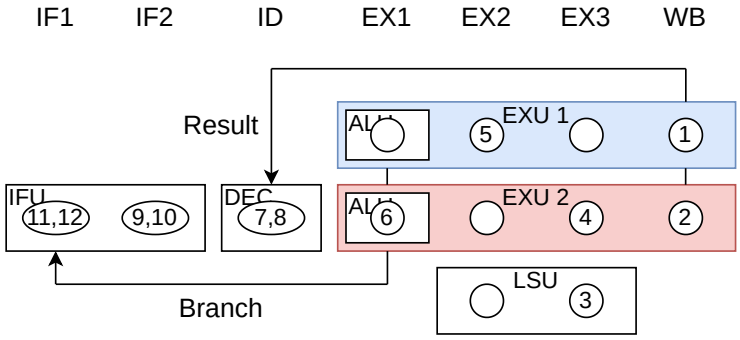

Figure 4.3: **Execution of instructions in pipeline stages.** On the left, an assembly form of a numbered instruction sequence is depicted. Their distribution to the processor pipeline at a selected time of execution is on the right.

As can be seen, the first four instructions are executed without any delays. The instruction packet (pair) `<addi (5), slli (6)>` is split because `slli (6)` instruction has to wait for the `lw (2)` result, while `addi (5)` has all operands ready. The `slli (6)` instruction is dispatched as soon as the load result may be forwarded. It is also dispatched alone and not together with `addi (7)` because of the processor's static issue slot scheduling. The rest instructions are considered prefetched and might not even be executed, like in the case of an unspecified instruction on the line `12`.

4.3 Instruction Fetch Buffer Mechanisms

The instruction fetch buffer (IFB) is a part of the IFU unit and it serves for temporarily storing fetched instructions when they cannot be dispatched for execution. This happens when later units are stalling, which might occur quite often (e.g., for the DEC unit). If there was no IFB, these instructions would need to be thrown away and fetched again once the stalling ends — this would be a big waste of power consumed during instruction fetching, not to mention the significant negative impact on performance.

A simplified architecture of the created IFB is illustrated in figure 4.4. IFB stores whole instruction packets into two dedicated items, conceptually forming a queue. The reason there are exactly two IFB items connects with how IFU stalling works. The IFU unit may have a new fetched instruction packet prepared on the memory output in IF2 when a stall signal is received. It also may have prepared signals in IF1 for initializing a new transfer. The instruction packets of both such transfers must eventually fit into IFB on stall and further instruction fetching must be prevented. A very important relevant design decision is to keep a dedicated register in the ID stage for storing an instruction pair instead of adding the third IFB item and reading its value. This would be a source of unnecessary switching activity in the ID stage.

IFB contains write/read pointer registers and an empty flag signal derived from a few registers but is considered a register for its simplicity. Together with stall and flush signals, these drive push and pop logic. The IFB registers are also practically the only sources of

a memory input signal that enables fetching the next instruction since IFB full and almost full indicators are derived from them. Together with the PC register used for addressing, these compose short paths from the processor to the instruction memory not to limit the processor's maximum frequency.

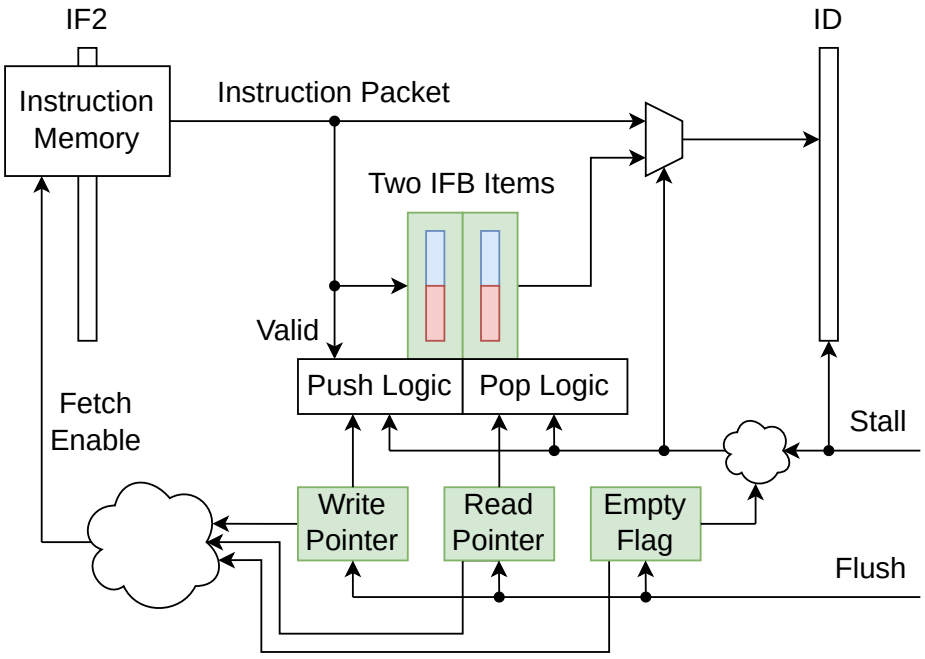

Figure 4.4: **Instruction fetch buffer and its connections to pipeline stages.** Green parts constitute register or derived register values of IFB. Clouds substitute simple combinational logic.

4.4 Two-Stage Operand Forwarding Approach

Operand forwarding can be quite demanding for superscalar architectures with longer pipeline stages. It is composed of control paths, data paths, and associated control logic. There is one control path and one data path per each result present in the pipeline. A control path includes a valid and ready flag of the result and its destination register index. A data path provides the result itself. As the created processor uses complete forwarding, for each source operand, control logic collects all control and data paths to provide correctly forwarded data.

Compared to common approaches that typically do the above in one stage, the created processor splits it into two. It removes dependencies rooted across control and data paths, shortening the longest forwarding-related path. The control paths are collected and evaluated in the ID stage, where possible stalling is initiated. It is worth mentioning that this evaluation is performed in parallel with decoding. The data paths are routed to the EX1 stage, where correct operand values are used. They are wired here to plain multiplexers, leaving enough latency reserve for computation units in the stage.

The fundamental requirement for the two-stage forwarding to work properly is that all involved stages with results are either working or stalling, there must be no deviation. In the created processor, it is even simpler since currently there are no events in the EXU unit that

would cause stalling. That must be ensured because the two-stage forwarding uses EXU ready flags[3] in its first phase (ID) and expects that the respective forwarded results will be shifted to the next pipeline stage for the second phase (EX1). Control paths for initiating such forwarding are routed from the EX1, EX2, and EX3 stages. Their respective data paths of forwarded results are routed from the EX2, EX3, and WB stages. That implies that the forwarding effectively works even for immediately adjacent instruction pairs. It is suggested to inspect figure 4.5 where this principle is illustrated.

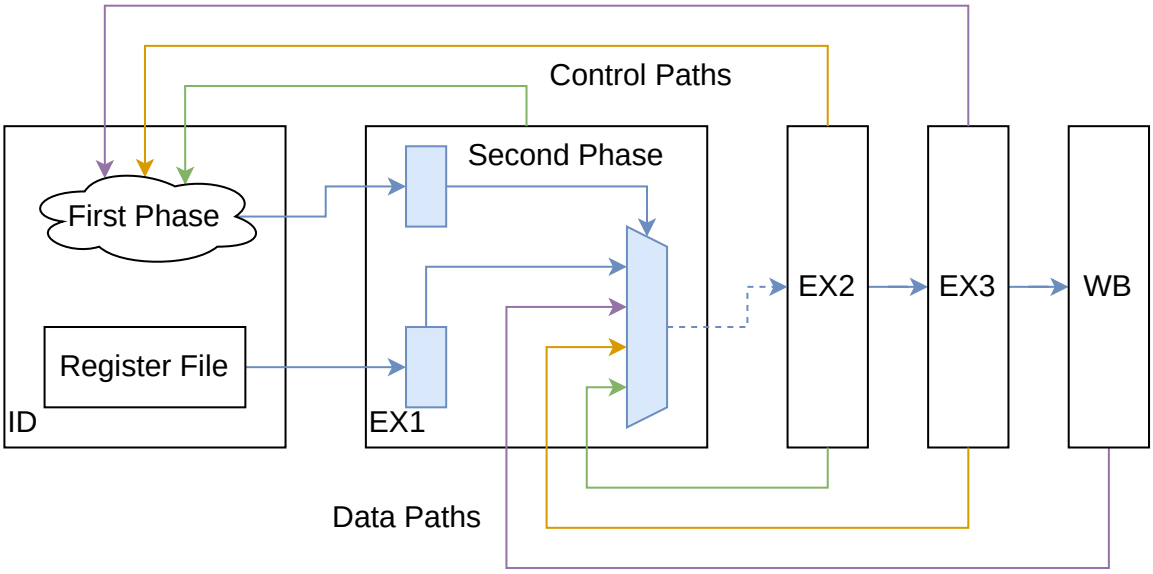

Figure 4.5: **Two-stage operand forwarding diagram.** Blue parts represent a trace of a single operand of one issue slot. Green, orange, and purple are control paths and their corresponding data paths of individual pipeline stage results.

4.5 Deploying the AHB-Lite Bus Protocol

Although RISC-V defines a common address space for instructions and data, the created microarchitecture defines two isolated interfaces to enable parallel instruction fetch and data access. To support better integration with already-existing peripherals, both interfaces deploy a limited form of AMBA 3 AHB-Lite bus protocol [1], an open standard for on-chip buses developed by ARM. Each AHB-Lite transfer consists of an address and data phase, separated at least (and typically) by one clock cycle. The protocol has a pipelined nature, enabling it to overlap the mentioned phases. This perfectly fits into pipelined processor designs. An example of AHB-Lite transfers is demonstrated in figure 4.6.

The IFU and LSU units are responsible for the protocol compliance. The protocol's address and data phases are directly mapped to their IF1/IF2 and EX2/EX3 stages, respectively. The associated logic from registers to the interface and vice versa must be simple to provide high processor frequencies. This was a subject of intense efforts during the initial work on the design. IFU approach is described in section 4.3 above. LSU only performs adjustments of loaded data before storing it in a register, other paths are clean.

[3]The result ready flag says that the result will be available for forwarding in the next pipeline stage on the next clock cycle.

If the loaded data had been adjusted in the following stage, it could have worsened the WB timing, further relating to forwarding timing. Consequently, adjustment of the data immediately is considered a good compromise.

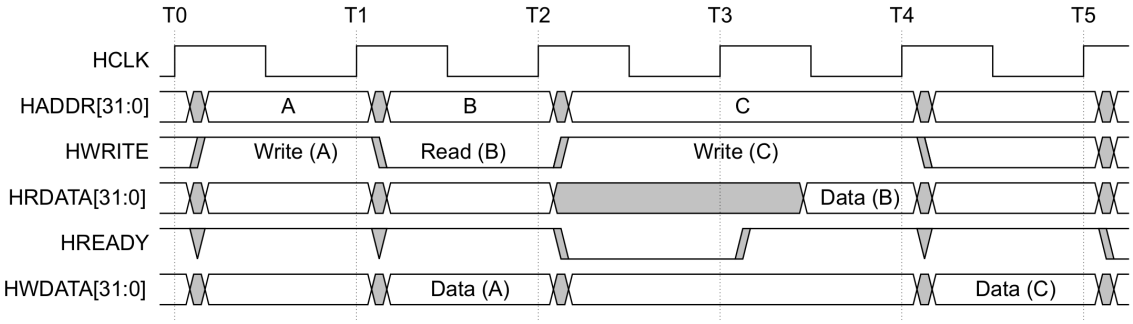

Figure 4.6: **Example AHB-Lite communication with multiple transfers.** It captures the overlapping capabilities of the protocol. For example, at time *T1*–*T2*, transfer *A* is being finished while transfer *B* starts. Reproduced from [1].

Since the AHB-Lite support is limited, some of its features are not currently supported in the core. The first significant one is transfer error signalization using the *HRESP* signal, which would require implementing some of the RISC-V privileged concepts first. The second constitutes wait states during the transfers defined by the *HREADY* signal. This would require adding more stall sources to the pipeline. Thus, the core expects that all transfers are always successful and last exactly two cycles. It is worth pointing out that some work has already been done to add support for the wait states. The pipeline control itself is prepared for that, the support in IFU and LSU units, however, is missing. I preferred to spend more time improving processor testing and performance measurements, explored in the following chapter 5.

To be complete, the processor interface also includes other signals than AHB-Lite related. It contains a global clock signal, a global reset signal, and a reset vector port, whose value is written to the PC register on active reset.

Chapter 5

Testing and Demonstration of Processor Work

A simulation environment has been developed to substantiate that the created processor works. The environment is built around Verilator, a fast Verilog/SystemVerilog simulator. For viewing the simulation progress, the GTKWave application is used. Both tools are free open-source software. Also, a custom build system has been established so the user does not have to run all simulation commands manually. Appendix B describes how to use the created build system to run the processor's simulation on a computer and what is required before doing so.

Verilator accepts an HDL (hardware description language) design and transforms it into a C++ model, which is then compiled using a C++ compiler into an executable simulator binary. This process is illustrated in figure 5.1. As I had not worked with Verilator before, I needed to build everything from scratch and test my attempts a lot. Nevertheless, the outcome is a simulation environment tailored to the created processor's needs. Namely, it includes many useful simulation options, the processor and its AHB-Lite memory integration, and the ability to compile RISC-V assembly programs and then run directly on the created processor.

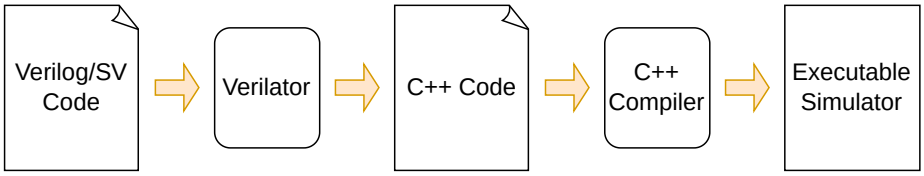

Figure 5.1: Processor simulation workflow using Verilator.

For actual processor testing, RISC-V programs were created and used. They are written specially so that their zero return codes are composed of the results of all executed instructions, if possible. More on RISC-V programs and their structure can be found in section 5.2. These programs are accompanied by design assertion statements that can be enabled in the simulation environment. They check, for example, that it is never pushed to full IFB. Testing on an FPGA was also considered but decided against since more work needs to be done on the processor before advancing its development to that stage.

5.1 Introduction to Simulation Testbench

The created simulation environment is built on a simulation testbench. A testbench is an HDL module containing the top-level module intended for testing. Such a module is often called DUT (design under test), and besides it, the testbench also contains auxiliary modules and logic needed for testing. Using a testbench is common practice since, compared to DUT, it might contain non-synthesizable constructs. In this particular case, DUT is the processor, auxiliary logic is composed of dual-ported AHB-Lite memory and structures needed for controlling simulation.

Since Verilator is used for simulation, an additional C++ wrapper had to be created. This wrapper instantiates the testbench module and creates a sense of time flow for it. It drives the global clock signal. The remaining signals are driven from non-synthesizable testbench parts, keeping the testbench module as universal as possible for potential future support of other simulators. The C++ wrapper also generates a signal trace file, if activated. It could have been done universally in the testbench module as well but Verilator provides a compressed file format with much better loading times. An example of such a file loaded in GTKWave is demonstrated in figure 5.2. Please note that a custom configuration file for GTKWave is used so the view looks well organized, helping with debugging.

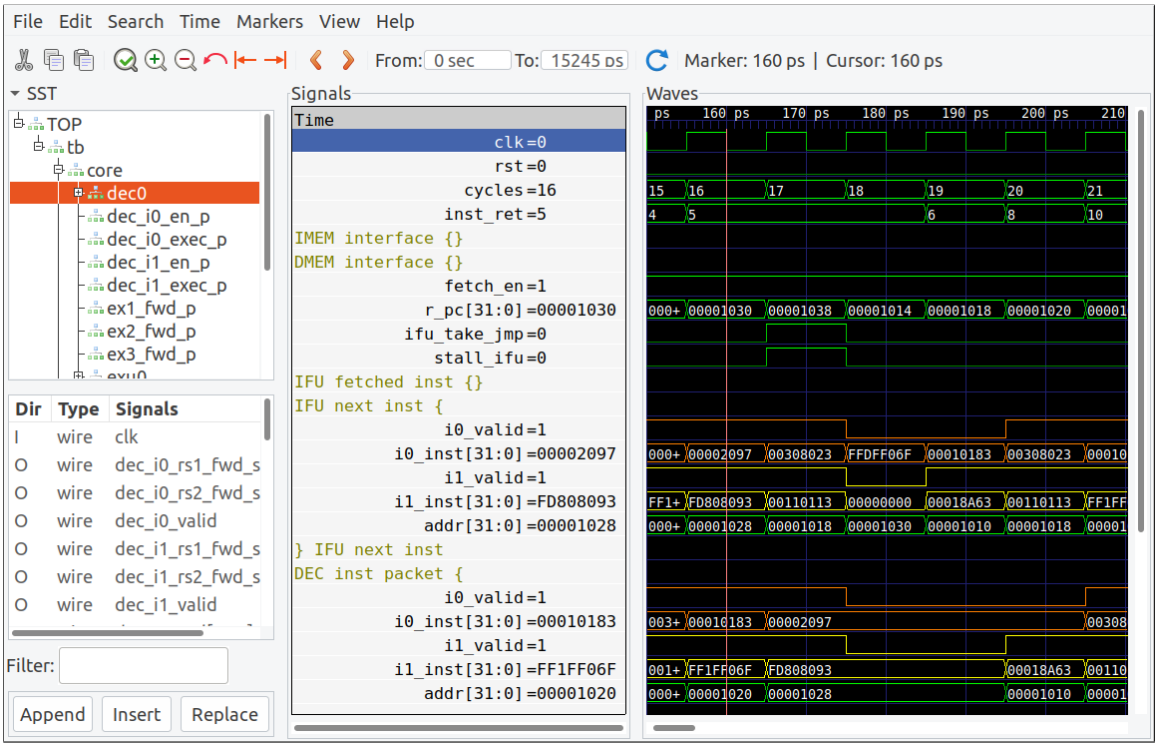

Figure 5.2: **Viewing the processor simulation progress using GTKWave.** The `inst_ret` signal counts the number of executed instructions. Its value changes on clock cycles demonstrate the processor's superscalar capabilities.

A program executed on the processor also needs to tell the simulation environment somehow to stop the simulation. Generally, this is achieved using so-called "semihosting" on more advanced processors. A simpler method, however, is deployed instead — a mailbox concept. The testbench constantly checks the processor-memory communication and once

a particular address, known as a mailbox address, is written to, it performs additional actions. Currently, there are two mailbox addresses — one for stopping the simulation, and one for printing an ASCII character to the computer's standard output. When stopping the simulation, the value written to the address is used as a return value of the simulation, expecting zero value for a successful program run.

To load a given RISC-V program binary into the processor's memory before starting the simulation, it must be in a special hexadecimal format defined by SystemVerilog (SV) standard [11]. It serves for loading data from a file to a specified SV memory array. An example of such a file is provided in listing 5.1. Fortunately, the RISC-V GCC tools mentioned in section 5.2 allow us to obtain such a format simply by transforming the compilation output binary file.

```
@00001000
17 1F 00 00 03 2F 0F 00 97 1F 00 00 93 8F CF FF
93 05 00 00 63 D0 E5 05 33 06 BF 40 93 06 00 00
13 87 16 00 63 46 C7 00 93 85 15 00 6F F0 9F FE
93 97 26 00 B3 87 F7 01 03 A8 07 00 83 A8 47 00
63 D6 08 01 23 A0 17 01 23 A2 07 01 93 86 16 00
6F F0 1F FD 13 05 00 00 63 08 0F 02 03 A6 0F 00
93 05 10 00 63 D2 E5 03 93 96 25 00 B3 86 F6 01
83 A6 06 00 63 C8 C6 00 13 86 06 00 93 85 15 00
6F F0 5F FE 13 05 10 00 97 25 00 00 93 85 85 F7
23 A0 A5 00 6F F0 DF FF
@00002000
1E 00 00 00 1C 00 00 00 04 00 00 00 08 00 00 00
FD FF FF FF 14 00 00 00 58 01 00 00 04 00 00 00
4E FE FF FF 2B 00 00 00 FF 6A 01 00 02 00 00 00
03 00 00 00 00 76 06 00 21 02 00 00 6A 0D 00 00
E9 FF FF FF 22 00 00 00 6C 03 00 00 69 03 00 00
14 00 00 00 00 00 00 00 FC FF FF FF E4 10 00 00
56 56 FF FF 04 00 00 00 01 00 00 00 02 00 00 00
FE FF FF FF FE FF FF FF 09 00 00 00
```

Listing 5.1: Example of a hexadecimal file that can initialize the processor's memory before running simulation. The file is a plain text file, not a binary one. It represents a bubble sort program, containing its instruction and data sections at selected addresses.

5.2 Testing Processor Using RISC-V Assembly Programs

The main part of the processor testing represents running RISC-V assembly programs. Almost every such program is designed so that all its instructions contribute to the final return value, possibly detecting faulting processor implementation. We call them "self-check tests" as they formally create sequences of instructions that always return zero code, which indicates success. They are compiled using the GCC compiler suite extended by RISC-V support. An example created RISC-V program is shown in listing 5.2. It is a "Hello World" program and due to its nature, this particular test does not have self-check abilities comparable to the rest. It is, however, a suitable example to present here.

```
.section .mailbox, "aw", @nobits
mb_halt: .word 0
mb_putc: .word 0

.section .text
.global _start
_start:
    la x1, mb_putc
    la x2, msg

.balign 8
    lb x3, 0(x2)
print_loop:
    beqz x3, print_finished
    sb x3, 0(x1)
    addi x2, x2, 1
    lb x3, 0(x2)
    j print_loop

print_finished:
    la x1, mb_halt
    li x2, 0
halt_loop:
    sw x2, 0(x1)
    j halt_loop

.section .rodata
msg: .ascii " ---------------------------------------------- \n"
     .ascii "|                                              |\n"
     .ascii "|   Hello World, I am Super RISC-V processor!  |\n"
     .ascii "|                                              |\n"
     .ascii " ---------------------------------------------- \n"
     .byte 0
```

Listing 5.2: **Example RISC-V assembly program.** The program starts its execution at the address corresponding to the `_start` label. Its `print_loop` is optimized to load data while recovering from branch penalty. Keep in mind that the program uses some pseudo-instructions. Besides others, the example also demonstrates how the mailbox concept is integrated into programs.

Since the programs need to be adjusted for this new existing processor with a specific environment, a custom linker script is used so the compiler knows how to do that. Currently, all programs use a common linker script illustrated in listing 5.3. Bear in mind that the linker script configuration must always match the configuration of the simulation environment kept in the testbench module. Otherwise, the processor, e.g., could start execution at a different address than the program is located.

```
OUTPUT_ARCH(riscv);
ENTRY(_start);

SECTIONS
{
    . = 0x1000;
    .text : { *(.text) }
    . = 0x2000;
    .rodata : { *(.rodata) }
    .data : { *(.data) }
    .bss : { *(.bss) }
    . = 0x3000;
    .mailbox : { *(.mailbox) }
}
```

Listing 5.3: **Linker script used for RISC-V assembly programs compilation.** Individual sections, then declared in assembly programs, have four-kilobyte gaps between them. These gaps may be increased for bigger instruction or data sections. In such a case, the testbench configuration must be changed too.

The testing itself was conducted using individual tests explored in the following subsection 5.2.1. It found some bugs in the design, which were then fixed. Mainly, it was load instructions that did not work at all. It turned out that it was just a typographical error in an EXU signal name. The second interesting bug found was connected with the `SRA` instruction that causes an arithmetic right shift, which preserves the sign of the value. However, an unsigned operand was used for that in ALU. After fixing these, all created tests are passing.

5.2.1 List of Used RISC-V Test Programs

Various test programs were created to verify the processor functionality. Some are highly specific to the processor, and some are more general, demonstrating a few tasks normally solved in computer science. There are also "fast" variants of these more general programs. These variants are carefully optimized for the created processor to provide better performance. There are 10 tests in total:

- **all_inst** tests all implemented RV32I instructions. It also uses almost all relevant pseudo-instructions to test their use as well.
- **array_sum** and **array_sum_fast** perform the sum of a given array of integer values (512 items in default) and compare it to the reference result. The array_sum_fast test also uses intensive loop unrolling, demonstrating the maximal performance potential of the processor in its current state for solving practical problems.
- **bubble_sort** and **bubble_sort_fast** implement the bubble sort algorithm over a given array of a selected size (30 in default) and then check it is sorted.
- **fib_seq** and **fib_seq_fast** compute numbers of the Fibonacci sequence and compare results with the prepared ones stored in the memory. The maximal unsigned 32-bit range is used, hence computing the first 48 numbers.

- **halt** just tests stopping the simulation with zero return code.
- **hazards** is an exhaustive test of all possible pipeline hazards of the created processor. Most important tests are focused on operand forwarding, inter-packet data hazards, load-use hazards, LSU structural hazards, and control hazards caused by jump instructions. This test is very aware of the processor microarchitecture.
- **hello_world** prints a "Hello World" message to the computer's standard output. The total number of message characters is 250. The program can be inspected directly in listing 5.2, and its output can be found in the appendix section B.2.

Some already-existing RISC-V tests were also considered to be reused but because the processor is completely a fresh one, it needs some microarchitecture-specific and simple tests first. However, they are intended to extend the current ones in the further development stages of the processor. Adopting tests not originating from a processor's author also improves the overall processor testing reliability.

5.3 Evaluation of Processor Design

Two basic types of performance measurements were also conducted based on the programs introduced in subsection 5.2.1 above. The first type observes the average IPC (instructions per cycle) during the programs' execution. The second type compares the performance of the created superscalar processor with an imitation of a scalar processor with as many identical aspects as possible, showing the exact benefits of superscalar possibilities on particular programs. The testbench module tracks the processor execution and makes these measurements possible.

Test Name	Spent Cycles	Executed Instructions	IPC
all_inst	251	257	1.02
array_sum	4119	2572	0.62
array_sum_fast	694	1132	1.63
bubble_sort	8584	4859	0.57
bubble_sort_fast	5975	4394	0.74
fib_seq	9016	7307	0.81
fib_seq_fast	7511	7308	0.97
halt	12	4	0.33
hazards	383	503	1.31
hello_world	1524	1260	0.83
Average			**0.88**

Table 5.1: Measured average instructions per cycle for each test program.

In table 5.1, the IPC of the created processor is deduced. On average, it can execute 0.88 instructions per cycle. Its best result of 1.63 IPC is achieved in the *array_sum_fast* test. Such performance is reached by applying hints provided in subsection 5.3.1.

Since the processor currently does not achieve an average IPC higher than one, as it has had other priorities in mind, comparing it with existing dual-issue processors does not make sense, even more so when they use branch predictors for example. In addition to that,

a fair comparison would need to be set up in the first place, implying running a standardized benchmark. The processor and its simulation environment are not prepared for that now.

Instead, I decided to demonstrate the tangible benefits of the processor's superscalar execution. In the source code of the processor, there is a part of the code that can be uncommented and the processor then behaves like it was a scalar one. It is present in the DEC unit and causes DEC to always dispatch at most one instruction to the EXU unit for execution. Then, these two cores face each other in table 5.2. The average improvement seen in the superscalar variant is 22.6 %. The lowest improvement 6.3 % belongs to the *bubble_sort* test due to its intensive jumping and related penalties, so the dual-issue execution does not have many opportunities to get into effect. The *bubble_sort_fast* test is deprived of that negative trait.

Test Name	**Spent Cycles**		**Improvement [%]**
	Scalar	**Dual-Issue**	
all_inst	333	251	24.62
array_sum	4632	4119	11.08
array_sum_fast	1237	694	43.9
bubble_sort	9162	8584	6.31
bubble_sort_fast	7482	5975	20.14
fib_seq	11372	9016	20.72
fib_seq_fast	11139	7511	32.57
halt	13	12	7.69
hazards	582	383	34.19
hello_world	2024	1524	24.7
Average			**22.59**

Table 5.2: Improvement of performance for each test program when using the created superscalar processor compared to its scalar variant.

5.3.1 How to Create Optimal Programs for the Processor

Since the created microarchitecture is focused on restricted resources, its performance must be caught up by clever programs' approaches. Some of such approaches are applied in the fast variants of the test programs. Please note that this was intended from the early stages of the processor development — not to burn power on something that can be done statically with satisfying results. To fully reap the relevant profits, a customized compiler must be created to generate optimized programs automatically.

The observed performance bottlenecks of the processor are mainly composed of data and control hazards, strongly influenced by the arrangement of instructions. Due to expected higher occurrences, the processor performance is most sensitive to the inter-pair data hazards. If the inter-pair hazard is inevitable, it is beneficial if there is a data hazard with the following instruction pair as well, since the second instruction of the current pair will be executed alone anyway. It is recommended to avoid placing two load/store instructions in one instruction pair since it causes serialization, too. Due to those, sometimes it is beneficial to manually align a particular block of instructions to the fetch block size (i.e., 64 bits) using the "`.balign 8`" assembler directive. Over and above, general tricks for increasing performance help as well. For example, loop unrolling, not branching in a common case, and loading data from memory before performing a jump.

Chapter 6

Conclusion

The goal of this thesis was to design, implement, and test a superscalar RISC-V processor. First, the RISC-V instruction set was introduced with its 32-bit RV32I integer base used in this thesis. Then, basic concepts of scalar processors, such as pipelining, were explained. Based on that, the superscalar processors and their approaches were explored together with relevant examples. Once the common ground was built, the created microarchitecture was described — its pipelined architecture, dealing with hazards, and the established processor interface. After, the prepared simulation environment was introduced to prove the processor, written in SystemVerilog, works as intended. RISC-V assembly programs were used for that. Ultimately, the mentioned backs up that the thesis goal was met.

The created open-source processor represents a good superscalar base for low-power applications. It exposes a dual-issue seven-stage pipeline with in-order instruction execution. Its pivot trait is static instruction scheduling to its two issue slots based exclusively on the instruction address. Also, an uncommon two-stage operand forwarding was deployed not to limit the processor timing by forwarding. Although its AHB-Lite support is limited, it is essential for integration with already-existing peripherals. As the microarchitecture avoids deeper specializations, it can be further extended and optimized concerning the actual needs. Therefore, a strong emphasis was also on creating readable source code.

The processor work was tested and demonstrated in a simulation environment crafted precisely for the processor's needs. It is based on the Verilator simulator. The environment contains a custom dual-ported AHB-Lite memory, a mechanism for processor printing to standard output, and basic processor performance measurement logic. RISC-V GCC was used to compile created RISC-V assembly programs for testing and collecting performance data. The conducted performance tests showed the processor can execute 0.88 instructions per cycle on average. This value, however, can be further improved even by better-optimized programs — a prototype test program used to demonstrate the processor performance possibilities showed the result of 1.63 instructions per cycle.

During this work, I learned that designing a practically usable processor from scratch is very time-consuming. I was building only the microarchitecture for several months, mostly due to tuning details that could have had bad power or frequency impacts. Due to all the efforts, however, I suppose the final design could hit at least one gigahertz if manufactured on a standard process node used for similar processors. Also, there should be no obstacles in using the processor in an FPGA project now.

As mentioned several times in the text, the main processor's shortcoming in terms of performance is the absence of a branch predictor. This is something I want to work on in the future. The processor also needs more RISC-V functionality, such as machine privilege

level, to be considered full-featured. In addition to that, several smaller tasks recorded in the source code `TODO` comments will also require my attention.

To push the processor closer to its real-life use, several further steps, possibly initiated by another master's theses, need to be taken. These include creating a compiler tailored to the processor, setting up and running synthesis and power analysis to prove used concepts, establishing a sophisticated verification environment, and creating an FPGA product prototype based on the processor to demonstrate its capabilities.

Bibliography

[1] ARM. *AMBA 3 AHB-Lite Protocol: Specification* [online]. Issue A. 2006 [cit. 2024-05-01]. Available at: https://developer.arm.com/documentation/ihi0033/a.

[2] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M. and Marwedel, P. Scratchpad Memory: Design Alternative for Cache on-Chip Memory in Embedded Systems. In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*. New York, NY, USA: Association for Computing Machinery, 2002, p. 73–78. CODES '02. DOI: 10.1145/774789.774805. ISBN 1581135424.

[3] Brandejs, M. *Mikroprocesory Intel 8086 – 80486 [Intel's Microprocessors, 8086 – 80486]*. 1st ed. Grada, 1991. Nestůjte za dveřmi. ISBN 80-85424-27-4.

[4] Canonical. *Canonical enables Ubuntu on SiFive's HiFive RISC-V boards* [online]. June 2021 [cit. 2023-01-09]. Available at: https://canonical.com/blog/canonical-enables-ubuntu-on-sifives-hifive-risc-v-boards.

[5] *RISC-V VeeR EH1: Programmer's Reference Manual* [online]. Revision 1.9. CHIPS Alliance, December 2022 [cit. 2023-03-30]. Available at: https://github.com/chipsalliance/Cores-VeeR-EH1/blob/main/docs/RISC-V_VeeR_EH1_PRM.pdf.

[6] Corpeño, E. *An Introduction to RISC-V—Understanding RISC's Open ISA* [online]. June 2022 [cit. 2023-01-15]. Available at: https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture.

[7] Fog, A. *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers* [online]. 3. Technical University of Denmark, November 2022 [cit. 2023-01-24]. Available at: https://www.agner.org/optimize/microarchitecture.pdf.

[8] Greenley, D., Bauman, J., Chang, D., Chen, D., Eltejaein, R. et al. UltraSPARC: the next generation superscalar 64-bit SPARC. In: *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*. 1995, p. 442–451. DOI: 10.1109/CMPCON.1995.512421. ISBN 0-8186-7029-0.

[9] IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*. 2001, p. 1–792. DOI: 10.1109/IEEESTD.2001.93352.

[10] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*. 2008, p. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[11] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*. 2018, p. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[12] Kra, Y., Shoshan, Y., Rudin, Y. and Teman, A. HAMSA-DI: A Low-Power Dual-Issue RISC-V Core Targeting Energy-Efficient Embedded Systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*. 2024, vol. 71, no. 1, p. 223–236. DOI: 10.1109/TCSI.2023.3323425.

[13] Lalja, D. J. Reducing the branch penalty in pipelined processors. *Computer*. 1988, vol. 21, no. 7, p. 47–55. DOI: 10.1109/2.68.

[14] McLellan, E. The Alpha AXP architecture and 21064 processor. *IEEE Micro*. 1993, vol. 13, no. 3, p. 36–47. DOI: 10.1109/40.216747.

[15] RISC-V International. *Membership* [online]. December 2022 [cit. 2023-01-04]. Available at: https://riscv.org/membership/.

[16] Rotenberg, E., Bennett, S. and Smith, J. Trace cache: a low latency approach to high bandwidth instruction fetching. In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 1996, p. 24–34. DOI: 10.1109/MICRO.1996.566447. ISBN 0-8186-7641-8.

[17] Salvet, D. *Návrh modelu procesoru [Design of the Processor Model]*. Brno, 2020. Bachelor's thesis. Masaryk University, Faculty of Informatics. Supervisor Přenosil, V. Available at: https://is.muni.cz/th/uqqoh/?lang=en.

[18] SiFive. *SiFive E76 Core Complex Manual* [online]. Version 21G3.02.00. 2022 [cit. 2023-03-30]. Available at: https://www.sifive.com/cores/e76.

[19] Smith, T. Megahertz myth: There is more to a PC's virility than clock speed. *The Guardian* [online]. February 28 2002, p. 9, [cit. 2023-01-24]. Available at: https://www.proquest.com/newspapers/online-megahertz-myth-there-is-more-pcs-virility/docview/245771186/se-2.

[20] Thornton, J. E. Parallel Operation in the Control Data 6600. In: *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*. New York, NY, USA: Association for Computing Machinery, 1964, p. 33–40. AFIPS '64 (Fall, part II). DOI: 10.1145/1464039.1464045. ISBN 9781450378888.

[21] Tomasulo, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*. 1967, vol. 11, no. 1, p. 25–33. DOI: 10.1147/rd.111.0025.

[22] Unnikrishnan, S. and Iype, S. M. *Reducing Power Hot Spots through RTL optimization techniques* [online]. February 2024 [cit. 2024-05-08]. Available at: https://www.design-reuse.com/articles/55643/reducing-power-hot-spots-through-rtl-optimization-techniques.html.

[23] Urquhart, R. *Open Source vs Commercial RISC-V Licensing Models* [online]. Codasip, November 2020 [cit. 2023-01-20]. Available at: https://codasip.com/2020/11/26/open-source-vs-commercial-risc-v-licensing-models/.

[24] Vávra, J. *Graphical Simulator of Superscalar Processors.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Jaroš, J. Available at: https://www.fit.vut.cz/study/thesis/21991/.en.

[25] Waterman, A. and Asanović, K., ed. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213.* RISC-V Foundation, December 2019.

[26] Waterman, A., Asanović, K. and Hauser, J., ed. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203.* RISC-V International, December 2021.

[27] Waterman, A., Lee, Y., Patterson, D. A. and Asanović, K. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA.* UCB/EECS-2011-62. EECS Department, University of California, Berkeley, May 2011. Available at: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

[28] Weissmann, P. *OpenPA: The book of PA-RISC* [online]. Release 2.8.1. January 2022 [cit. 2023-03-29]. ISSN 1866-2757. Available at: https://www.openpa.net/print.html.

Appendix A

Contents of Included Storage Media

The storage media contains the following directories:

- `/measured` — Measured processor performance data
- `/specs` — Relevant specification documents for the work
- `/thesis` — Source files for generating this thesis document
- `/src` — Created source code files including a `Makefile`
  - `/src/out` — Built processor model and tests (Ubuntu 22.04)
  - `/src/rtl` — The processor SystemVerilog design files
  - `/src/tb` — Base used for processor testing
  - `/src/tests` — Individual tests with their linker script
  - `/src/util` — Useful files for processor simulation

Appendix B

User Manual

To see how a processor works, the easiest way is to get it under simulation. Since even that process might be exhausting, a custom build system has been created during the Super RISC-V processor development. This significantly saves time for users as they typically need to enter only one command, the build system will do the rest.

B.1 Simulation Requirements

All that is needed to bring the Super RISC-V processor to life is stated here:

- **Make**[1] for custom build system
- **Verilator**[2] for translating processor design files to C++ model
- **GCC**[3] for C++ processor model compilation

Optionally:

- **RISC-V GCC** for RISC-V programs compilation
- **GTKWave**[4] for displaying signal traces

B.1.1 How to Install Required Software on Ubuntu

Ubuntu 22.04 was chosen as a reference system for this work. All mentioned tools are available for it. Nevertheless, other systems might be used if the tools are available. To install Make, Verilator, GCC, and GTKWave on Ubuntu 22.04, use the following command in a terminal emulator:

```
sudo apt install make verilator gcc gtkwave
```

[1] GNU Make website – https://www.gnu.org/software/make/
[2] Verilator website – https://www.veripool.org/verilator/
[3] GCC website – https://gcc.gnu.org
[4] GTKWave website – https://gtkwave.sourceforge.net

Installing RISC-V GCC requires more manual steps. It might be obtained precompiled directly for Ubuntu 22.04 from its GitHub releases[5]. During the development, the following archive was used:

```
riscv32-elf-ubuntu-22.04-gcc-nightly-2024.03.01-nightly.tar.gz
```

Once downloaded, it must be extracted and its `riscv/bin` directory path added to the `PATH` environment variable. This may be achieved by modifying `.bashrc` file in the current user home directory. The following line should be append:

```
export PATH="<extracted-directory-path>/riscv/bin:$PATH"
```

B.2 How to Run Simulation

Copy files mentioned in appendix A to the system used for experiments. Then, open a terminal emulator in the `src` directory, where a `Makefile` is located. Run `make hello_world`, which translates the SystemVerilog (SV) processor description into a C++ model, compiles the C++ model, and runs a processor simulation with a precompiled "Hello World" program loaded in the processor's memory. All generated files can be found in the `src/out` directory. This is a convenient way of testing that the build system works. Among others, the output should include something like this:

```
./out/build/Vtb +verilator+noassert +verilator+rand+reset+0 +test+path=
   utils/hello_world.hex
 ------------------------------------------------
|                                                |
|   Hello World, I am Super RISC-V processor!    |
|                                                |
 ------------------------------------------------
- tb/tb.sv:156: Verilog $finish
Simulated cycles: 1524
Executed instructions: 1260
```

B.2.1 Individual Makefile Targets

Since the build system uses Make, it was designed so that if some dependencies are missing, they are automatically built/rebuilt if needed (e.g., running a simulation requires building a C++ model first). However, the `Makefile` also contains targets for individual steps:

- `make`/`make verilate` — translate the SV processor description into a C++ model
- `make build` — compile the C++ model into an executable processor simulator
- `make sim` — compile a RISC-V program and run processor simulation with it
- `make debug` — run processor simulation with a RISC-V program and create files for processor debugging (disassembled RISC-V program, signal trace waves file)

[5]RISC-V GCC releases – https://github.com/riscv-collab/riscv-gnu-toolchain/releases

- `make waves` — run processor simulation with a RISC-V program and display signal trace waves in GTKWave
- `make clean` — clean all generated files

Note that `sim`, `debug`, and `waves` targets compile a RISC-V program before running the simulation. They use the RISC-V GCC for that and it must be installed on the system before running those targets.

B.2.2 Supported Makefile Macros

The `Makefile` also accepts the following macros to adjust its behavior:

- `ASSERTS=1` — enable SV assertions
- `MAX_CYCLES=<value>` — max cycles of simulation
- `TEST_NAME=<name>` — test to be run on the processor
- `WAVES_FILE=<path>` — path of output signal trace waves file
- `X_VAL=0|1|2` — unknown signal values in SV are replaced with: 0 — zeros, 1 — ones, 2 — random values
- `SEED=<value>` — seed used for any randomized event

An interesting macro that deserves more attention is `TEST_NAME`, which defines what RISC-V program is compiled and run by the processor during the simulation. The list of possible test names includes file names present in the `src/tests` directory without their `.s` file extensions. It is possible to create a new RISC-V program easily by adding it here. The default test name is `hello_world`.

B.2.3 Examples of Running Simulation

Run the `fib_seq.s` test with processor asserts enabled:

- `make sim ASSERTS=1 TEST_NAME=fib_seq`

Display waves of the `hello_world.s` test in GTKWave:

- `make waves`

Generate debug files (disassembly and waves files) when running `hazards.s`:

- `make debug TEST_NAME=hazards`

Run bubble sort test optimized for Super RISC-V and limit cycle count:

- `make sim TEST_NAME=bubble_sort_fast MAX_CYCLES=7500`

Let a test fail[6] due to constraining its cycle count:

- `make sim TEST_NAME=all_inst MAX_CYCLES=20`

[6] Note that it is normal for Verilator to use an abort signal when the simulation ends with an error code.

B.3 Automated Testing

Since the created `Makefile` is targetted mainly for direct use by users, an automated way of processor testing was also required. The `src/run_tests.sh` shell script does exactly that. It uses already-existing targets of the `Makefile` to support its reuse.

To use the test runner, first, change the directory to `src` (if not already done). Then execute `./run_tests.sh`. It automatically builds the C++ processor model, collects tests from the `tests` directory, and runs them appropriately. The main report is printed directly to the terminal, all output can be found in a generated log file.