FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	RISC-V CPU superscalar microarchitecture design
Student:	Tomáš Věžník
Supervisor:	Ing. Michal Štepanovský, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer engineering
Department:	Department of Digital Design
Validity:	until the end of summer semester 2022/2023

Instructions

RISC-V CPU superscalar microarchitecture design
1. Describe the basic principles of superscalar processors.
2. Design a microarchitecture of a superscalar processor supporting the RV32I instruction set (or its subset) with emphasis on its simplicity.
3. Describe the proposed microarchitecture in Verilog/SystemVerilog.
4. Demonstrate the functionality of your proposal using simulation.
Consult your work with the supervisor.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 30 May 2022 in Prague.

Bachelor’s thesis

RISC-V CPU SUPERSCALAR MICROARCHITECTURE DESIGN

Tomáš Věžník

Faculty of Information Technology
Computer engineering
Supervisor: Ing. Michal Štepanovský, PhD
May 11, 2023

Citation of this thesis: Věžník Tomáš. *RISC-V CPU superscalar microarchitecture design.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I want to express my sincere gratitude to my thesis supervisor, Ing. Michal Štepanovský, PhD., for their encouragement and support. Their feedback and constructive criticism have helped immensely during the creation of this work.
I would also like to thank my roommate, Bc. Adam Plodek, for his ability to listen and bravely accept the role of a rubber duck.
Finally, I thank my family and friends for their love, support, and encouragement throughout my academic journey.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guidelines for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded a license agreement with the Czech Technical University in Prague on the utilisation of this thesis as school work under the provisions of Article 60(1) of the Act. This fact shall not affect the provisions of Article 47b of Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on May 11, 2023 . . . .

Abstract

This thesis aims to explain the superscalar processor's working principles and to design a microarchitecture based on the RISC-V RV32I instruction set architecture. The designed microarchitecture is called VTM (Veznik Tomas Microarchitecture) and is described using SystemVerilog hardware description language. The VTM is a dual-issue out-of-order superscalar microarchitecture that executes up to four instructions in the execution stage simultaneously. The primary output of this thesis is the VTM source code. The simulation, accompanied by the theoretical part of this thesis, serves as a learning tool for anyone who wants to understand the inner workings of superscalar processors, with students as the primary audience.

Keywords microarchitecture, superscalar processor, RISC-V, RV32I, out-of-order execution, SystemVerilog

Abstrakt

Zaměřením této práce je přiblížení činnosti superskalárních procesorů a navržení superskalární mikroarchitektury procesoru využívající instrukční sady RISC-V RV32I. Navržená mikroarchitektura se nazývá VTM (Veznik Tomas Microarchitecture) a je popsána jazykem SystemVerilog. VTM je superskalární mikroarchitektura, která zpracovává dvě instrukce najednou a ve vykonávacím stupni vykonává do čtyř instrukcí najednou. Hlavním výstupem této práce jsou zdrojové kódy popisující VTM. Zdrojové kódy spolu s teoretickou částí této práce mohou posloužit jako učební pomůcka, která ukazuje fungovaní superskalárních procesorů, cílená na studenty.

Klíčová slova mikroarchitektura, superskalární procesor, RISC-V, RV32I, vykonávání instrukcí mimo pořadí, SystemVerilog

Abbreviations

ARN	Architectural Registers
CDB	Common Data Bus
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DD	Decode & Dispatch
EEI	Execution Environment Interface
GS	Global Signals
IIB	Instruction Issue Bus
ISA	Instruction Set Architecture
MMU	Memory Management Unit
PC	Program Counter
PCI	PC Interface
RISC	Reduces Instruction Set Computer
ROB	Reorder buffer
RQ	Register Query
RRN	Renamed Registers
RRQ	Renamed Register Queue
VTM	Veznik Tomas Microarchitecture

Introduction

The processor, a flattened rock with trapped lightning inside, tricked into thinking.[1]

As modern processor architectures get increasingly complicated year over year, keeping track of all the technologies used is important. Therefore, creating tools and materials explaining the trends and technologies used in modern devices is more than welcome.

The main goal of this thesis is to design and implement a simple superscalar processor microarchitecture based on the RISC-V RV32I instruction set architecture. Additional goals are to simulate the implemented microarchitecture and to explain the working principles of a simple superscalar processor microarchitecture.

Instruction Set Architecture (ISA) describes the "visible parts", an interface between the hardware and software of a computer system. It defines internal processor storage organisation, memory alignment, memory addressing modes, etc. ISA is an abstraction that only describes the behaviour of a system but does not specify precise implementation. Not being reliable on one implementation allows the reusability of a binary code on multiple machines that support the same ISA. Currently, there are two main trends in ISA design, complex instruction set computer (CISC) and reduced instruction set computer (RISC). [2, 3, 4]

CISC is an instruction set architecture commonly used in desktop and server computers. The main representative of CISC is the x86_64 instruction set architecture, now mostly used by Intel and AMD. CISC instruction set architectures commonly have variable instruction lengths, multi-cycle instruction execution, and extensive memory access modes. Common instructions are shorter and simpler to implement, and specialised instructions are longer and more complex. The size of one instruction varies from 1B to 16B. This allows for high code density but increases complexity regarding instruction decoding. [3]

RISC focuses on the simplicity and highly-optimised instruction set. RISC instruction set architectures commonly have fixed-length instructions, simple instruction encoding, and single clock cycle instruction execution. RISC architectures heavily rely on the ability of a processor to execute multiple instructions at once and the optimisations that code compilers provide. [5, 3]

RISC-V (pronounced "risk-five") is a new open and free standard ISA originating at UC Berkley. Originally designed to aid in education and computer research but turned into a fast-growing trend. As the name suggests, RISC-V is reduced instruction set computer (RISC) based ISA that aims to be simple, extensible, and scalable. [4, 6]

The RISC-V utilises a more modular approach than other instruction set architectures (ISAs). At the centre of RISC-V ISA is the base integer ISA and built onto it are optional extensions ISAs. The base integer ISA defines only the necessary set of instructions needed by the majority of operating systems. There are two primary base integer variants, RV32I and RV64I, where the main difference is the width of integer registers and the size of addressable memory space. The RV32I and RV64I use 32-bit or 64-bit wide integer registers and address space, respectively. RV32E base integer ISA, a subset of RV32I, is targeted at microcontrollers with half as many integer registers and future RV128I base integer ISA with support for 128-bit address space. All base integer ISAs use a two's-complement signed integer representation and either little-endian or big-endian memory systems. [4, 6] *"To support general software development, a set of standard extensions are defined to provide integer multiply/divide, atomic operations, and single and double-precision floating-point arithmetic."* [4]

The RISC-V ISA focuses more on the software-visible interface and avoids implementations detail where possible. [4] This gives the designer of a particular hardware implementation more freedom and the ability to tailor the ISA to their needs with the RISC-V ISA extension approach.

Microarchitecture is a high-level description of a computer's design, such as the memory system, memory interconnect, and central processing unit (CPU) design. Microarchitecture is a specific implementation of ISA; for example, two processors can share ISA and execute the same binary code but have different microarchitectures, resulting in completely different performances. [2]

Chapter 1

RISC-V RV32I

This chapter describes a subset of RISC-V RV32I instructions; this subset is implemented in the designed microarchitecture.

The base integer ISA RV32I utilises 32 integer registers with a width of 32 bits each. Table 1.1 shows each register's number and application binary interface (ABI) name. ISA also defines an additional program counter (`pc`) register.

Table 1.1 RV32I register state with application binary interface and best practice use. [7]

Register	ABI name	Use
x0	zero	Hardwired zero
x1	ra	Return address for a call
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5 x6 x7	t0 t1 t2	Temporary registers
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10 x11	a0 a1	Function arguments / return values
x12–x17	a2–a7	Function arguments
x18–x27	s2–s11	Saved registers
x28–x31	t3–t6	Temporary registers
pc		Holds current instruction address

The RV32I uses six 32-bit instruction formats R-type, I-type, S-type, U-type, B-type, and J-type. The formats are designed to simplify instruction decoding by keeping source (*rs1*, *rs2*) and destination registers (*rd*) at the same positions across all the formats. In formats where immediate value is used, it is always sign-extended to 32 bits, and the most significant is used as a sign bit. [4]

The RV32I is a load-store architecture where every instruction performs implicit memory access so it can be fetched, but only LOAD and STORE instructions have direct access to the memory. Other instructions have access only to registers. [4]

The designed microarchitecture doesn't implement all RV32I instructions. The instruction FENCE is omitted because the microarchitecture implements only one hardware thread, thus making the FENCE instruction redundant. The microarchitecture also omits the ECALL and EBREAK instructions as they transfer control to the debug environment, which is not implemented. The unimplemented instructions are decoded as NOPs to ensure compatibility with the whole instruction set.

All implemented instructions are shown in Table 1.2 and are described in detail in the following subsections and in [4].

Table 1.2 Overview of all implemented instructions.

Category	*Subcategory*	*Encoding*	*Instructions*
Arithmetic and Logic Instructions	Register-Register	R-type	ADD, SLT, SLTU, AND, OR, XOR, SLL, SRL, SUB, SRA
	Register-Immediate	I-type	ADDI, SLTI, SLTIU, ANDI, ORI, XORI, SLLI, SRLI, SRAI, NOP
	Immediate only	U-type	LUI
	Address-Relative	U-type	AUIPC
Control Flow Instructions	Unconditional Branches	J-type	JAL
		I-type	JALR
	Conditional Branches	B-type	BEQ, BNE, BLT, BLTU, BGE, BGEU
Load and Store Instructions	Load instructions	I-type	LW, LH, LHU, LB, LBU
	Store instructions	S-type	SW, SH, SB

1.1 Arithmetic and Logic Instructions

This section describes instructions that perform basic arithmetic (adding, subtraction, etc.) and logical (binary and, or, xor, etc..) operations. Instructions in this section are divided into four groups based on how their operands are sourced:

Register-Register instructions source their operands only from registers and are described in section 1.1.1.

Register-Immediate instructions source one of their operands from a register, and a 32-bit sign extended value obtained from the instruction itself and are described in section 1.1.2.

Immediate Only instructions use a 32-bit sign extended value obtained from the instruction itself as their operand and are described in section 1.1.3.

Address-Relative Immediate instructions use the instruction's memory address and a 32-bit sign extended value obtained from the instruction itself as their operands. They are described in section 1.1.4.

Any arithmetic or logic instruction exceptions, such as overflow or division by zero [8], do not throw any exceptions because software can manage such exceptions, thus simplifying the hardware design. [4]

1.1.1 Register-Register Instructions

R-type format instructions that perform an arithmetic or logical operation on 32-bit values *src1* and *src2* from registers *rs1* and *rs2* respectively and store the final result in *rd* register. All

R-type instructions use the same *opcode*; therefore, the type of operation is defined by *funct3* and *funct7* fields. The Table 1.3 shows the order and sizes in bits of fields used by the R-type instruction format. [4]

Table 1.3 Encoding of R-type instructions. [4]

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD adds *src1* and *src2*. SUB subtracts *src2* from *src1*. SLT (set less than) results in 1 if *src1* < *src2*, 0 otherwise. SLTU (set less than unsigned) performs the same operations as SLT but treats *src1* and *src2* as unsigned integers. AND, OR, and XOR perform their typical logical bitwise operations. SLL (shift left logical) shifts the *src1* to the left by the amount in the lower 5 bits of *src2* and pads the lower bits with zeros. SRL (shift right logical) shifts the *src1* to the right by the amount in the lower 5 bits of *src2* and pads the upper bits with zeros. SRA (shift right arithmetic) shifts the *src1* to the right by the amount in the lower 5 bits of *src2* and copies the sign bit to the upper bits. [4]

1.1.2 Register-Immediate Instructions

The I-type format instructions perform an arithmetic or logical operation on 32-bit value *src1* from register *rs1* and 32-bit sign-extended I-immediate *imm* and store the final result in *rd* register. [4] Table 1.4 depicts the encoding of non-shift Register-Immediate instructions using the I-type instruction format. Table 1.5 illustrate the encoding of the shift Register-Immediate instructions that use the I-type instruction format but encode the I-immediate field differently.

Table 1.4 Encoding of I-type instructions with 12-bit I-immediate. [4]

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src1	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src1	ANDI/ORI/XORI	dest	OP-IMM

Table 1.5 Encoding of I-type instructions with immediate encoding shamt and direction. [4]

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

ADDI, SLTI, SLTIU, ANDI, ORI, XORI, SLLI, SRLI, and SRAI all perform the same operations as their non-I counterparts but use *imm* instead of *src2* for calculations. SLLI, SRLI, and SRAI encode the shift value *shamt* in 5 lower bits of the *imm* and in 7 upper bits of the *imm* is encoded the direction of the shift. [4]

1.1.3 Immediate Only Instructions

The RV32I provides the instruction LUI to calculate any address in available 2^{32} address space. LUI is a U-type format instruction that uses 32-bit sign-extended U-immediate *imm* and stores its results into *rd* register. [4] Table 1.6 depicts encoding of the LUI instruction.

Table 1.6 Encoding of the LUI instruction. [4]

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI

LUI (load upper immediate) takes 20 lower bits of the *imm* and pads them with zeros to form a 32-bit value. [4]

1.1.4 Address-Relative Immediate Instructions

The RV32I provides the instruction AUIPC to calculate a value relative to the AUIPC instruction program address in the available 2^{32} address space. AUIPC is a U-type format instruction that uses 32-bit sign-extended U-immediate *imm* and stores its results into *rd* register. [4] Table 1.6 depicts encoding of the AUIPC instruction.

Table 1.7 Encoding of the AUIPC instruction. [4]

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	AUIPC

AUIPC (add upper immediate to pc) computes the same value, from its immediate, as LUI and adds to it the program address of the AUIPC instruction. [4]

1.1.5 NOP

NOP (no operation) is I-type instruction designed only to advance `pc` or any other relative counters. [4] Table 1.8 shows that the NOP instruction is effectively ADDI instruction with all source operands set to zero and destination register set to `x0`, which ISA dictates as constant zero, making the NOP result redundant.

Table 1.8 Encoding of NOP instruction. [4]

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

1.2 Control Flow Instructions

This section describes instructions used to change control flow, meaning they dictate the order in which instructions are executed.

1.2.1 Unconditional Branches

Unconditional branches always change the program's flow control based on the instruction's operands.

JAL (jump and link) is J-type format instruction. JAL encodes the 32-bit sign-extended J-immediate *imm* as a 32-bit signed *offset* aligned to multiples of two. Omitting the 0 bit of *imm* expands the possible address range and allows the *offset* to form a jump address range of ±1 MiB around the JAL instruction. The address of the following instruction (`pc`+4) is stored into *rd* register. [4] Table 1.9 depicts the encoding of the JAL instruction and shows how the J-type instruction format is derived from the U-type instruction format as the only difference is the way the immediate value is encoded.

Table 1.9 Encoding of JAL instruction. [4]

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

JALR (jump and link register) is I-type format instruction. JALR encodes the 32-bit sign-extended I-immediate *imm* as a 32-bit signed *offset*, which is added with the value *base* of the *rs1* register then setting the least-significant bit to 0 to form an effective jump target address. Calculating the jump target address based on register value than on the JALR instruction address allows JALR to jump anywhere in the available 2^{32} address space. The address of the following instruction (`pc`+4) is stored into *rd* register. [4] Table 1.10 depicts the encoding of the JALR instruction.

Table 1.10 Encoding of JALR instruction. [4]

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

1.2.2 Conditional Branches

Conditional branches allow for additional logic when calculating jump target addresses. All conditional branch instructions use a B-type instruction format. B-type format encodes the 32-bit sign-extended B-immediate *imm* as 32-bit signed *offset* aligned to multiples of two. The branch instructions decide based on values *src1* and *src2* from *rs1* and *rs2* registers, respectively. The effective jump target address is calculated by adding the *offset* and the address of branch instruction giving it a target range of ±4 KiB. [4] Table 1.11 depicts the encoding of all conditional branch instructions and shows how the B-type instruction format is derived from the S-type instruction format (shown in Table 1.13) as the only difference is the way the immediate value is encoded.

BEQ (branch equal) takes a branch if the *src1* is equal to the *src2*. BNE (branch not equal) takes a branch if the *src1* is not equal to the *src2*. BLT (branch less than) takes a branch if the *src1* is less than the *src2*. BLTU (branch less than unsigned) performs the same operations as BLT but treats *src1* and *src2* as unsigned integers. BGE (branch greater equal) take a branch if the *src1* is greater or equal to the *src2*. BGEU (branch greater equal unsigned) performs the same operations as BGE but treats *src1* and *src2* as unsigned integers. [4]

Table 1.11 Encoding of B-type instructions. [4]

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

1.3 Load and Store Instructions

Load and store instructions provide direct access to the memory. [4]

All loads are I-type format instructions encoding the 32-bit sign-extended I-immediate *imm* as a 32-bit signed *offset* added with *base*, value from *rs1* register, form effective memory address. The instruction field *width* defines if 32, 16, or 8 bits are loaded from memory. The value loaded from memory is copied into the *rd* register. [4] Table 1.12 depicts the encoding of LOAD instructions.

Table 1.12 Encoding of LOAD instructions. [4]

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

LW (load word) loads 32 bits (word) from memory. LH (load halfword) loads 16 bits (halfword) from memory and sign-extends the value to 32 bits. LHU (load halfword unsigned) performs the same operations as LH but extends the value with zeros instead of the halfword sign bit. LB (load byte) loads 8 bits from memory and sign-extends the value to 32 bits. LBU (load byte unsigned) performs the same operations as LB but extends the value with zeros instead of the halfword sign bit. [4]

Store instructions use an S-type instruction format. S-type format encodes 32-bit sign-extended S-immediate *imm* as a 32-bit signed *offset* which added to *base*, value from *rs1* register, form effective memory address. The value to be stored *src* is loaded from *rs2* register. The instruction field *width* defines if the lower 32, 16, or 8 bits of *src* are being stored in memory. [4] Table 1.13 depicts the encoding of STORE instructions.

Table 1.13 Encoding of STORE instructions. [4]

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

SW, SH, and SB store 32-bit (word), 16-bit (halfword), and 8-bit (byte) values, respectively, in memory. [4]

1.4 Pseudoinstructions

Many other instructions can be implemented as pseudoinstructions using already existing instructions. For example, move instruction MV *rd, rs1* is equivalent to instruction ADDI *rd, rs1, 0*, instruction NOT *rd, rs* can be implemented as XORI *rd, rs1, -1* or J instruction as JAL where *rd*=x0. Therefore reducing hardware design complexity and offloading it to a programmer. [4]

Chapter 2

CPU Microarchitecture and Instruction Parallelism

This chapter introduces the reader to the basic principles of superscalar processors, namely the various types of CPU microarchitectures, instruction processing, and the problems and solutions to instruction-level parallelism.

2.1 CPU Microarchitecture

"The processor is the active part of the computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, etc." [9] A Central Processing Unit (CPU) is a core of a computer system that executes instructions of a computer program. [10]

For a processor to properly execute an instruction, several steps are needed; these steps form an instruction cycle:

1. Instruction fetch (IF). The instruction is fetched from memory or instruction cache at the address held by the PC register.

2. Instruction decode (ID). The instruction is decoded. In this step, information about source and destination registers, immediate values, and type of operation is extracted from the instruction.

3. Execution (EX). This step performs arithmetic and logical operations. If the instruction addresses memory, the effective address is calculated, and in the case of a branch instruction, PC is set to the correct value.

4. Memory access (MEM). Having effective memory address from EX step data are loaded or stored in memory or data cache.

5. Write back (WB). If the instruction writes into the register, the result of the operation performed in EX gets stored. [10]

Depending on the microarchitecture used, these steps may or may not correspond to the physical stages of the instruction processing.

2.2 Classes of Processor Microarchitectures

Scalar microarchitecture executes one instruction at a time. The scalar microarchitectures fetch at most one instruction per clock cycle. [9] The scalar microarchitectures can complete the instruction in one, illustrated in Figure 2.2, or multiple clock cycles as depicted in Figure 2.3. Figure 2.1 depicts an abstraction of a scalar microarchitecture. The scalar microarchitectures are designed to be simple and are used in elementary applications.

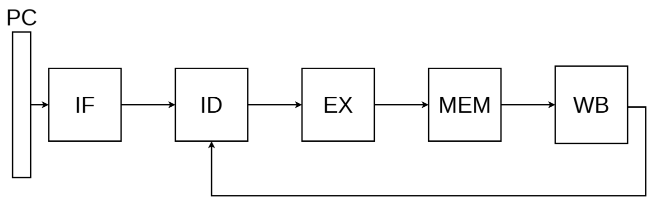


Figure 2.1 Abstraction of a scalar microarchitecture.

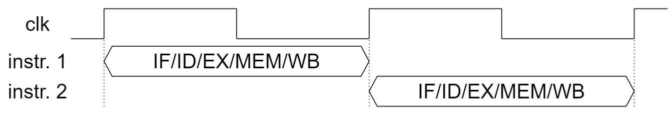


Figure 2.2 Instruction cycle of a single-cycle scalar microarchitecture. An entire instruction is processed in one cycle.

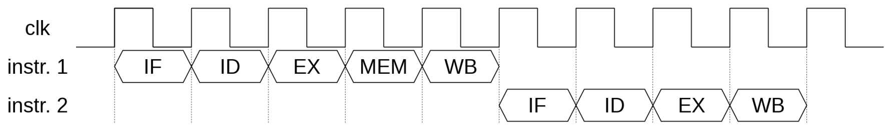


Figure 2.3 Instruction cycle of a multi-cycle scalar microarchitecture. Instructions are processed in a series of shorter cycles. In this illustration, instr. 1 traverses all steps of the instruction cycle and takes five clock cycles to finish, whereas instr. 2 requires only four clock cycles to finish as it does not access memory.

Pipelined scalar microarchitecture utilises stages (Figure 2.4), which overlap parts of the instruction pipeline across instructions. The instruction pipeline of a simple scalar microarchitecture using instruction pipelining is demonstrated in Figure 2.5. [10, 2] The pipelined microarchitectures with a high number of stages are called superpipelined. An example of a processor with superpipelined microarchitecture is Intel® Pentium® 4 with Hyper-Pipelined Technology (20-stage pipeline). [11]

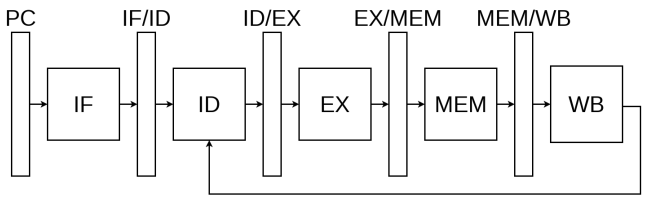


Figure 2.4 Abstraction of a scalar pipelined microarchitecture. [10]

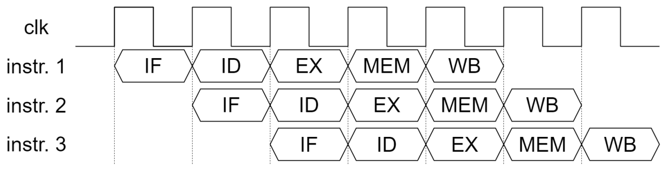


Figure 2.5 Instruction cycle of a pipelined microarchitecture.

Superscalar microarchitecture simultaneously processes multiple instructions, meaning each step of the instruction pipeline processes more than one instruction. The superscalar microarchitectures are often pipelined, meaning each stage processes multiple instructions as depicted in Figure 2.6, with an instruction cycle similar to Figure 2.7, to achieve even better performance. This allows for exploiting instruction-level parallelism.

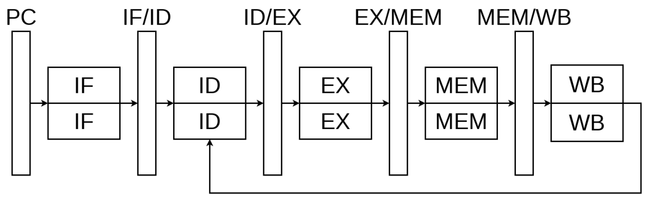


Figure 2.6 Abstraction of a superscalar pipelined microarchitecture. In this case, all instructions require five clock cycles to finish. However, several instructions are processed simultaneously, thus improving the throughput significantly.

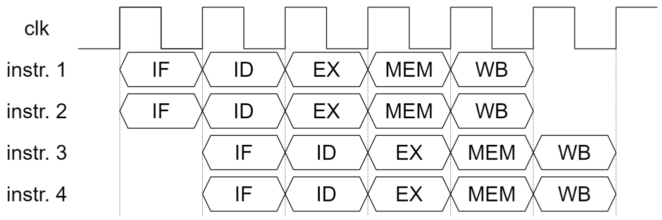


Figure 2.7 Instruction cycle of a superscalar pipelined microarchitecture.

The instruction level parallelism aims to break the program into several parallel running instruction streams, increasing the system's overall throughput. However, the exploitation of instruction-level parallelism carries its problems, which are addressed in the following sections.

2.3 Out-of-order Execution

With the scalar pipelined processors, if an instruction is stalled, all the instructions after that instruction are also stalled. This is often caused because the instruction does not have valid operands. Even if the following instruction has valid operands, it has to wait its turn. This problem is caused by in-order execution, where instructions must follow the program order. [2, 10]

Instead of stalling instructions with valid operands, we want to execute them even if the preceding instruction is stalled. This is called out-of-order execution. The out-of-order is commonly used in superscalar pipelined processors where multiple instructions are fetched, decoded, and issued together, but diverge at the execution step breaking their order. Instructions must be reordered after completion to preserve the program order. [2, 10]

2.3.1 Data Dependencies and Data Hazards

If multiple instructions are executed simultaneously, some may be dependent on each other. These situations create hazards. If every instruction depended on the preceding instruction, it would create an instruction chain equivalent to in-order execution. This shows that the number of dependencies determines how much parallelism can be done. If two instructions have no dependency on each other, they are independent and can be executed simultaneously without issues. [2]

Thus, inter-instruction dependencies must be resolved. Otherwise, they create hazards which can break the illusion of in-order execution.

2.3.1.1 Data Dependencies

Data dependencies occur when two or more instructions are truly data-dependent. Then, they must be executed in order. The data dependencies limit how much parallelism can be employed. [2] The Code listing 2.1 shows data-dependency between the ADDI and SUB instructions. The SUB instruction cannot be executed simultaneously with the ADDI instruction because the value of *s5* as seen by the SUB instruction is invalid until precedent ADDI instruction writes it.

Code listing 2.1 Example of data-dependent instructions.

```
ADDI s5, a0, 12
SUB  s6, s5, s4
```

2.3.1.2 Name Dependencies

Name dependencies occur when two or more instructions try to use the same register or memory location, but there is no data flow between these instructions. There are two types of such name dependencies: an antidependence and an output dependence.[2]

The antidependence is a name dependence that occurs when a later instruction reads the destination of the preceding instruction. [2] For example, the Code listing 2.2 shows that the ADDI instruction reads the *s5* register, and the SUB instruction writes to the *s5*. If both instructions were executed simultaneously, the SUB instruction could write to the *s5* register before the ADDI could read it. This would result in an incorrect value.

Code listing 2.2 Example of an antidependency.

```
ADDI s4, s5, 12
SUB  s5, a1, a2
```

The output dependence is a name dependence that occurs when two or more instructions write to the same register or memory destination. [2] For example, the Code listing 2.3 shows ADDI and SUB instructions both write to *s5* register. If the SUB instruction were completed before, the *s5* would be later rewritten by the ADDI, resulting in the wrong value compared to the program order.

Code listing 2.3 Example of an output dependency.

```
ADDI s5, a0, 12
SUB  s5, a1, a2
```

The name dependencies, compared to data dependencies, do not transfer any values between each other. The name-dependent instructions can be executed simultaneously if the naming dependencies are resolved. [2]

2.3.1.3 Data Hazards

Data hazards result from not handling the name or data dependencies resulting in a different program order or incorrect results. [2] Even when executing instructions out-of-order, the program order must be preserved as it would be impolite to change programmers' work without their permission.

There are three types of data hazards:

RAW (read after write) The Code listing 2.4 shows an example of the RAW hazard. When executed simultaneously, the SUB instruction could read the *s5* register before the ADDI instruction wrote a new value, thus getting the old value.

Code listing 2.4 Example of RAW hazard.

```
ADDI s5, a0, 1
SUB  s6, s5, a1
```

WAW (write after write) The Code listing 2.5 shows an example of the WAW hazard where both instructions write to the same *s5* register. When executed simultaneously, the SUB instruction could write first and the ADDI second, breaking the program order and leaving the *s5* register with the wrong value.

Code listing 2.5 Example of WAW hazard.

```
ADDI s5, a0, 1
SUB  s5, a1, a2
```

WAR (write after read) The Code listing 2.6 shows an example of the WAR hazard. When executed simultaneously, the SUB instruction could write its result to the *s5* register before the ADDI instruction had read it, leaving the result of the ADDI instruction incorrect.

Code listing 2.6 Example of WAR hazard.

```
ADDI s4, s5, 1
SUB  s5, a1, a2
```

RAR (read after read) is not a hazard, as reading a value after another instruction's read does not change the value. (The RAR was added for completion.)

2.3.2 Control Dependencies

The control dependencies arise from branching. Instructions must respect the program order.

The Code listing 2.7 shows an example of control dependency. The SUB instruction is control dependent on BNE instruction. It thus cannot be executed in parallel with ADDI instruction (before BNE), even when there is no data dependency between SUB and ADDI. Moreover, ADDI instruction is not control dependent on BNE, and thus, cannot be moved after BNE, so its execution would be controlled by BNE.

Code listing 2.7 Example of a control dependency.

```
start:
    ADDI s5, a0, 1
    BNE  a1, a2, end
    SUB  s4, a3, a4
end:
    ANDI s6, a5, 2
```

The control dependencies need to be preserved to ensure the program order. The instructions can be executed together but must be completed in the program order.

2.3.3 Tomasulo’s Algorithm

Tomasulo’s algorithm is a scheme for out-of-order execution. The algorithm resolves RAW hazards by keeping track of operand availability and utilises hardware register renaming to eliminate WAW and WAR hazards. Robert Tomasulo created it for the IBM 360/91 floating-point unit, shown in the Figure 2.8, but it’s easy to adapt for general purposes. [2]

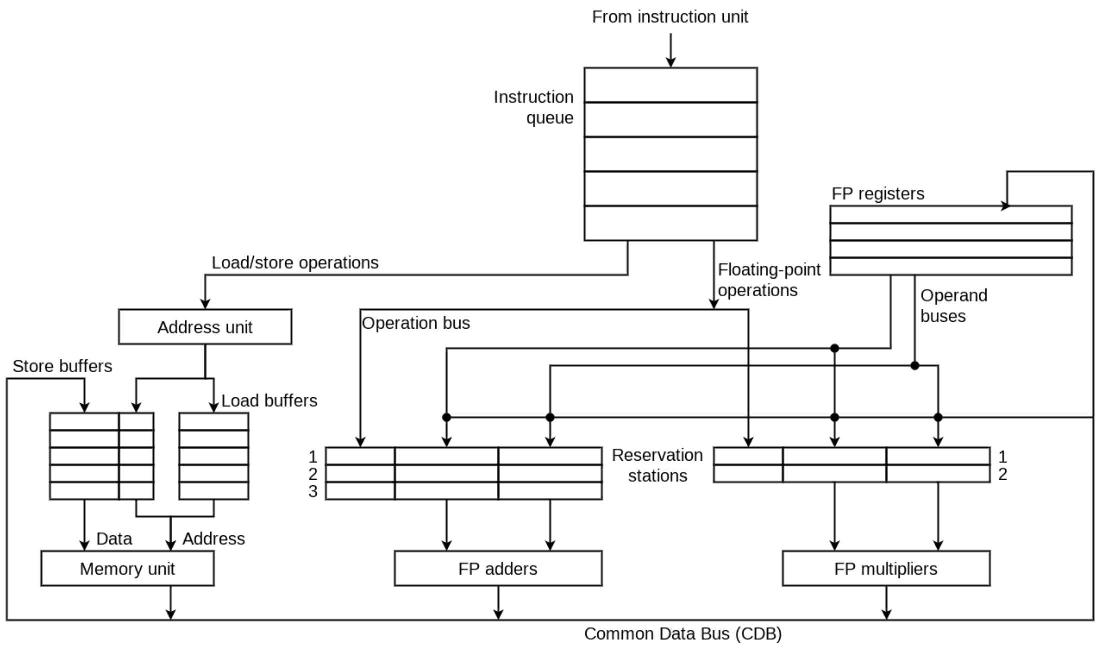


Figure 2.8 The basic structure of a RISC-V floating-point unit using Tomasulo’s algorithm. [2]

Issue (Dispatch) New instruction fetched from the head of the FIFO queue of decoded instructions is issued to a matching reservation station (if there is enough space). If the operand values are in registers, they are also issued. If the operands are unavailable, the reservation stations snoop the Common Data Bus (CDB) for the required valid operands. The Issue step renames registers eliminating WAR and WAW hazards. [2]

Execute When all operands are available, the reservation station relays the instructions with valid operands to the respective execution unit. Waiting until all instruction operands are available resolves the RAW hazard. [2]

Write result Available results are written to the CDB and propagated to the rename registers, reservation stations, and store buffers waiting for that result. Store buffers buffer store operations until the store address and the store result are available. Then the result is written to memory as soon as the memory unit is free. [2]

2.3.3.1 Register Renaming

As described above, register renaming is a technique that allows for the minimisation of hazards. The registers can be divided into logical or architectural, defined by ISA, and physical, the total register storage implemented. [10, 2, 12]

There are two commonly used methods for register renaming. The first method uses a physical register file larger than the logical register file, and a mapping table is used to associate the physical register with a logical register. Physical registers with no logical register mapped to them are recorded in a free list. When an instruction writes in a logical register, a physical register is removed from a free list. And a link between the physical and logical registers is added to the mapping table. The mapping table looks up the already-linked register to source proper instruction operands. After the physical register is not in use, it is returned to the back of the free list. [10, 2, 12]

The second method uses a physical register file the size of the logical register file. Additionally, a buffer, typically called reorder buffer, that keeps entries of issued instructions and is commonly implemented as a circular FIFO queue that ensures a program order. This approach also uses a mapping table to indicate how logical registers are linked to physical registers. When an instruction finishes execution, the result is written into the reorder buffer and an accordingly mapped physical register. If the reorder buffer has an incomplete instruction at the head, it blocks the completion of other instructions. Suppose the instruction at the head of the reorder buffer is completed. In that case, the instruction result is written into the appropriate logical register, and the mapping between the logical and physical registers is erased. When the reorder buffer is full, the instruction issue is stalled. [10, 2, 12]

2.4 Speculative Execution

Speculative execution is when we don't want to stall instructions until a branch target and branch condition are calculated. For example, assume a superscalar processor that fetches and issues two instructions simultaneously. With such a processor, the Code listing 2.8 shows a simple example. If the processor does not use speculative execution, it is forced only to issue the BNE as it does not know the branch result. This approach is limiting; therefore, we want to avoid it. With speculative execution, BNE and SUB can be issued simultaneously, but the SUB instruction must be somehow marked in case the BNE takes the branch skipping the SUB instruction. If the branch was taken, all marked instructions must be invalidated, as they are considered speculative. [13, 2]

Code listing 2.8 Example of a speculative execution.

```
    BNE  a1, a2, skip
    SUB  s4, a3, a4
skip:
    ANDI s6, a5, 2
```

2.4.1 Tagging

Tagging is a common approach to indicating what instructions are speculatively executed. The more branches we try to speculatively execute more tags we need. [2]

The Figure 2.9 shows how each subsequent branch adds another unique tag. After the branch result is known, the speculative branches are resolved based on the tag information. Assuming the first branch was speculated correctly, Tag1 is cleared, but if the second branch takes the non-speculated path, then the processor resolves the branch based on Tag2.

2.4.2 Branch Prediction

With speculative execution, we can minimise the number of wrongly taken branches using branch prediction. There are two approaches to branch prediction:

Static branch predictors that predict the branch outcome using the static information from the program itself, such as binary flags inserted by the compiler or the sign of immediate operand (indicating the branch direction) of branching instructions.

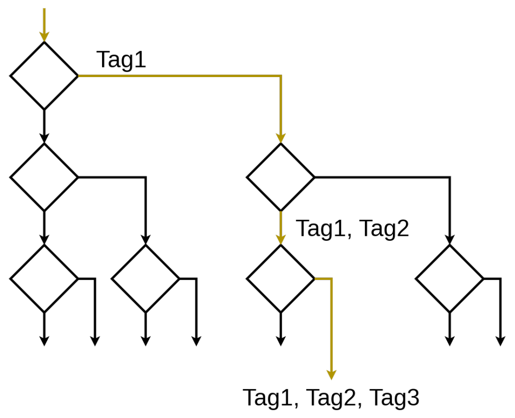


Figure 2.9 Tag accumulation example. [14]

Dynamic branch predictors use and update information based on the running program's actual behaviour. These predictors usually predict branch behaviour based on the branch result history. The dynamic branch predictors are commonly used as cache-like structures that return taken or not taken based on the branch instruction address. [12]

The branch predictors can vary from simple 2-bit branch predictors, implemented using a simple finite state machine, to tagged hybrid predictors, which combine multiple branch predictors into one with predictions based on a branch history length.

Chapter 3

Implementation

This chapter describes the components of the designed microarchitecture and the instruction execution process.

The designed microarchitecture is called the VTM (Veznik Tomas Microarchitecture). The VTM is a dual-issue superscalar processor microarchitecture based on the RISC-V RV32I ISA. The VTM is designed as a power-on reset architecture running the program on startup, where the program has access to the whole address space.

The VTM uses a variation of a modern adaptation of Tomasulo's scheme. It uses register renaming, reservation stations, and a reorder buffer to eliminate data and control hazards.

The VTM uses a simple branch predictor that assumes every branch is not taken to prevent stalling on every branch instruction. The simple branch prediction allows for speculative execution where a one-bit tagging is used. The one-bit tag allows only one branch instruction to be executed, but other branch instructions can be staged in the Issue step. If the tag is active, no other branch instruction can be issued because the execution of multiple branch instructions would require a vector of tags.

The following steps create the VTM instruction cycle:

Load Two instructions are loaded from the Instruction Cache to the Decoder if present otherwise, the cache loads the instructions from the Memory and then forwards them to the Decode.

Decode The two instructions are simultaneously decoded.

Fetch The types of instructions are recognised and values are loaded from registers.

Prepare All the necessary information about the instructions is packed. In this step, the dependencies between the two instructions are resolved. If necessary, the destination registers are renamed.

Issue If all the requirements are met, the instructions are sent to their respective reservation stations and are queued into the Reorder buffer.

Execute If the instruction has all operands valid and the respective execution unit is free, it is executed; otherwise, it waits for valid operands. This step breaks the order of instructions and creates an "execution pool" of instructions that are either being executed or are waiting for execution.

Complete After the instruction is executed, the Reorder buffer changes the status of the instruction to complete.

Write back If the instruction at the head of the Reorder buffer is completed, the instruction result is written into the architectural portion of the Register File.

Figures 3.1 to 3.3 depict possible instruction cycles used in the VTM. The `A` and `B` blocks symbolise one instruction each because of the VTM's dual-issue microarchitectural design. The unnamed purple blocks symbolise a pool of processed instructions in the system, as the number of instructions in these blocks depends on the program being run.

The Figure 3.1 shows the default instruction cycle where no problems or hazards are detected. Both instructions are processed simultaneously and only diverge at the Execution step and are reordered back into the program order in the Write back step.

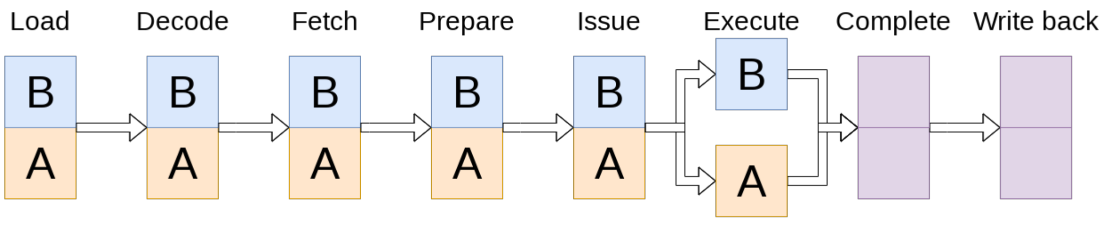


Figure 3.1 Instruction cycle processing two instructions through the whole microarchitecture.

The Figure 3.2 shows a variation of the instruction cycle where a potential problem arose. This can happen when a reservation station doesn't have enough space for another instruction (instruction `B` in this case) and only issues one instruction into the execution pool.

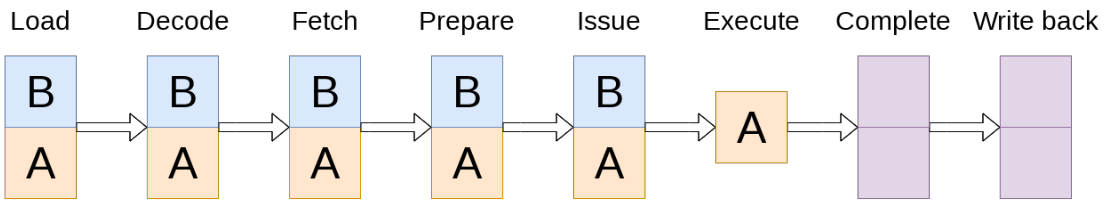


Figure 3.2 Instruction cycle processing two instructions until a potential hazard is detected.

The Figure 3.3 shows a variation of the instruction cycle. Similarly to the Figure 3.2, if all reservation stations are out of space or the reorder buffer is full, the microarchitecture falls into the Stall loop (the green arrow in the Figure 3.3), which essentially waits for free space. Additionally, if an error is detected (primarily an invalid instruction), the microarchitecture falls into the Break loop (the red arrow in the Figure 3.3). The Break loop disables the Load of new instructions, letting the already-issued instructions finish but halting the rest of the program's execution.

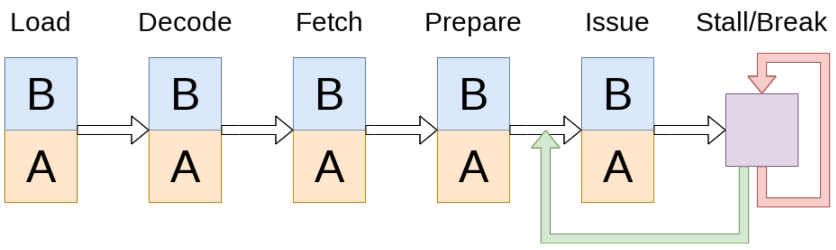


Figure 3.3 Instruction cycle processing two instructions until a potential hazard is detected or an error occurs.

3.1 Components

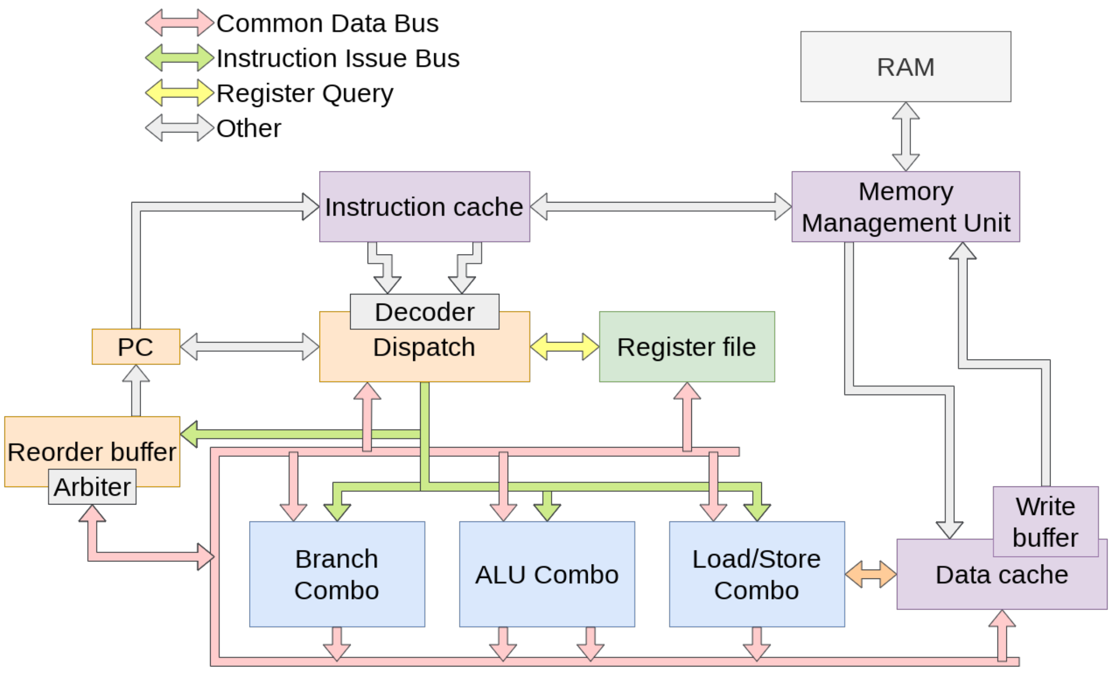


Figure 3.4 The VTM microarchitecture overview.

3.1.1 PC

The Program counter (PC) holds the memory address of the next instruction to be processed. The Instruction Cache and the Dispatch read the PC. The PC has three ways to change its value as described in section 3.2.

3.1.2 Decoder

Decoder is a combination logic that takes the whole 32-bit instruction and decodes it into usable parts defined by RV32I. It also provides instruction flags to help with additional logic, allowing easier processing in Dispatch. Decoder processes instructions in two ways data and control. Data includes:

- Destination register (`rd`).
- Source registers (`rs1`, `rs2`).
- Immediate value which is arranged according to RV32I specification and sigh-extended to 32 bits (`imm`).

Control includes:

- Unique pseudo identifier (`pid`). Execution units use PID to discern which operation should be used.
- Destination station (`station`).

- Operation flags. `Writes`, `jumps`, `opimms` indicate if the given instruction writes to a register, is jump or branch instruction, or if the given instruction is arithmetical or logical and uses the immediate value, respectively.

3.1.3 Dispatch

The Dispatch works with two instructions simultaneously. The Dispatch takes the decoded information about the two instructions and marks them as one of four sequence types:

NJ1NJ2 where both instructions do not jump.

NJ1J2 where the first instruction does not jump, but the second does.

J1NJ2 where the first instruction jumps and the second does not.

J1J2 where both instructions do jump.

The VTM uses a very primitive branch predictor, assuming the branch is always un-taken. This predictor is implemented in the Dispatch using instruction tagging. The tag is set when a branch instruction is issued, and every instruction following the branch instruction is tagged as speculative. The tag is cleared when the Reorder buffer completes the branch instruction.

Depending on the status of the tag and the sequence type of processed instructions, the Dispatch behaves accordingly:

- For the NJ1NJ2, both instructions are issued as the tag is irrelevant because both instructions do not jump.
- For the NJ1J2, only the first instruction is issued if the tag is active. If the tag is inactive, both instructions are issued, and the tag is set to active.
- For the J1NJ2, if the tag is active, both instructions are stalled. If the tag is inactive, both instructions are issued, and the tag is set to active.
- For the J1J2, if the tag is active, both instructions are stalled, but only the first instruction is issued if the tag is not active.

To summarise the behaviour. If the tag is active and jump instruction is to be issued, then the Dispatch stalls because processing multiple jump or branch instructions would require multiple tags.

The Dispatch processes the two instructions in stages. Those stages are:

LOAD stage requests instructions from the Instruction Cache. If the Instruction Cache hits the Dispatch proceeds to the next stage; otherwise is stuck in LOAD.

FETCH stage processed the Register Query, described in section 3.2, to acquire values stored in source registers, and if the instruction writes, request a renamed register number to avoid hazards. FETCH also defines what sequence type loaded instructions create, which dictates if one, both, or no instruction is issued.

PREPARE stage assembles results of the Register Query with any other information that Reservation stations need (data, instruction address, source and destination registers, data validity, ...).

ISSUE stage checks the fullness of the destination Reservation station of currently processed instructions. If they have enough space, ISSUE will write instructions to Instruction Issue Buses, described in section 3.2; otherwise, it waits for them to empty.

STALL stage is to trap Dispatch in a loop that waits for signals that resolve the tag.

BREAK stage is a trap with no escape.

3.1.4 Reorder buffer

As The Dispatch issues instructions to Reservation stations, Reorder buffer (ROB) listens and stores instructions chronologically in a queue. Then ROB listens to Common Data Buses, described in section 3.2, to see if any of held instructions were completed, and if so, it is marked as finished. If the instruction at the head of the queue is finished, ROB will request CDB to complete the instruction. If the completed instruction writes, the result is stored in an architectural register in the Register File and a renamed register, used to ensure the hazard-free operation, is freed. Every completed instruction needs to be written to the CDB because other components (such as the Data Cache) listen for changes in the instruction status.

3.1.5 Combo Units

Combo units combine the Reservation station, execution unit, and Arbiter, which is described in section 3.2.1, into one component. The primary purpose of the combo unit is to manage wires between the Reservation station and the execution unit because not all connections from the Reservation station are necessary for the execution unit to function. The secondary purpose is to house an Arbiter that controls bus access. The insides of the Combo unit blocks from the Figure 3.4 are shown in the section 3.1.5, and the execution units are described in more detail in the following subsections.

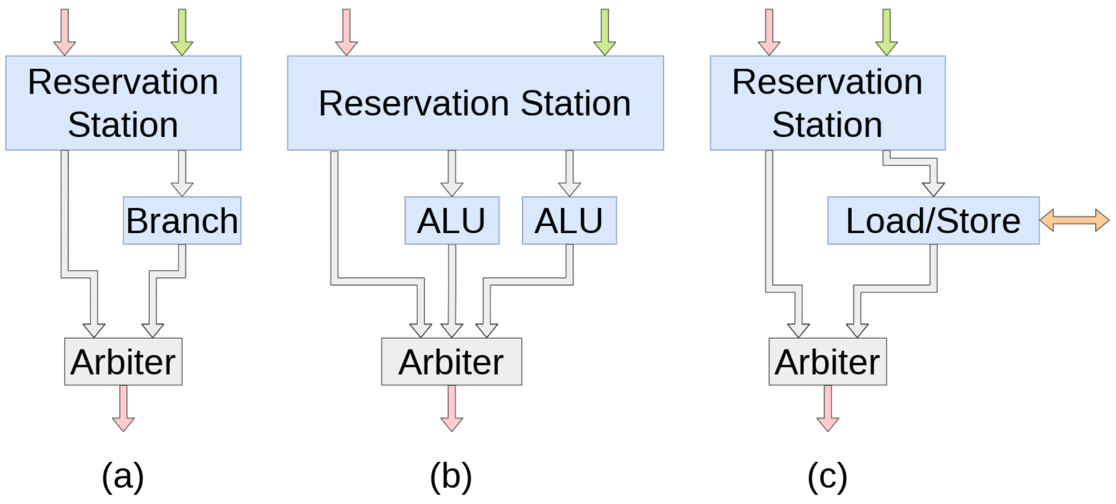


Figure 3.5 Execution Combos overview. The (a), (b), and (c) show the insides of the Branch Combo, ALU Combo, and Load/Store Combo, respectively, which are depicted in their block form in Figure 3.4.

3.1.6 Branch Unit

The Branch Unit is responsible for calculating unconditional and conditional branches. The Branch Unit executes eight instructions: JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, and BGEU.

From the microarchitecture perspective, unconditional and conditional branches are the same types that change the PC value, with the only difference being that JAL and JALR write to a register. This behaviour is not optimal as the unconditional branches always jump. The instructions following an unconditional branch instruction are, therefore, always speculatively executed. The Dispatch, or some preprocessor module, should instead precalculate the unconditional branch result, skip instructions after the unconditional branch, and load the instructions at the destination of the unconditional branch. Unfortunately, this revelation was discovered late in the implementation process and wasn't implemented.

3.1.7 Arithmetic and Logic Unit

An Arithmetic and Logic Unit (ALU) is an execution unit that executes arithmetic and logical operations. ALU performs twenty-one instructions: ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND, LUI, AUIPC. Some of these instructions perform essentially the same task (ADD, ADDI, AUIPC), so they could be grouped into one operation, but it would reduce the readability of the waveform diagram.

3.1.8 Load and Store unit

The Load and Store Unit (LS) manages all direct data access to the Memory. The LS executes eight instructions: LB, LBU, LH, LHU, LW, SB, SH, and SW. The LS communicates indirectly with the Memory because the Data Cache buffers memory reads and writes.

3.1.9 Reservation Stations

Reservation stations serve as instruction buffers and listen to CDBs for valid required operands. The reservation station uses a queue as a buffer. The Reservation station listens to IIB, and if the bus contains instructions for that type of station, it will be placed at the end of the queue. If the instruction at the head of the queue has all operands valid, it is propagated to the connected execution unit. The station listens to CDB for the result of the current head instruction to ensure it was executed. After observing the instruction result on CDB, the queue is advanced.

The Reservation stations report their fullness status to the Dispatch to prevent queue overflow.

There are two variations of the Reservation station. The simple single output station feeds operands only to one execution unit and a multi-station that feeds two execution units simultaneously.

3.1.10 Register File

The Register File contains 32 architectural registers (ARN) required by the RV32I and 32 renamed registers (RRN) used to resolve data hazards. The registers are numbered using a 6-bit vector where the most significant bit indicates if the register is architectural (0) or renamed (1). The Register File also contains renamed register queue (RRQ), which holds values of free RRN.

The Register File resolves the Register Queries. The query contains the numbers of the source and destination architectural registers. The Register File reads the number of the source registers. If the selected register is linked to a renamed register, the values of the renamed register are returned; otherwise, the values of the architectural register are returned. The returned values consist of the 32-bit register value, the validity bit of the register value, and the number of the register from which the values are read.

The Register Query can signal to the Register File that there is a need to rename a register. The Register File then pops a new renamed register number from the front of the RRQ and links the number with the architectural destination register. The popped renamed register value is also returned to the Dispatch through the Register Query.

Only the instructions that write into a register need renaming. The VTM is designed so that the number of processed instructions that write does not exceed the number of renamed registers.

Because the VTM can execute instructions speculatively, the Register File needs to keep track of what writes and renames are speculative. This is done with the help of the tag. If the instruction is speculatively executed and does write, then the linking to the renamed register must be done non-destructively in case the branch was taken. If the branch was taken, all tagged renaming is cleared, and the used renamed registers are pushed at the back of the RRQ.

The Register File listens to the CDB, where the instruction results are propagated. If the instruction result originates from the execution unit, it's written to a renamed register. The result is written to the architectural register if it originates from the ROB.

To ensure a proper renamed register linking the order of operations is as follows:

1. The Register File writes the results from the CDB to respective registers.
2. If the query source register matches the destination register, the Register File saves the current state of the destination register. No state is saved if the query source and destination registers are different.
3. The Register File renames the query destination register.
4. If the Register File stored query destination register state, then values based on the stored state are returned; otherwise, values based on the query source register are returned.

Code listing 3.1 depicts the ports of the Registers module where we can see the `DebugInterface`, which provides the state of the `x11` register used during testing.

Code listing 3.1 Register File module ports.

```
module Registers(
  GlobalSignals.rest global_signals,
  RegisterQuery.regs query1, query2,
  CommonDataBus.regs data_bus1, data_bus2,
  DebugInterface debug
);
```

3.1.11 Instruction Cache

The Instruction Cache buffers instructions into two mapped ways. The cache record consists of the instruction address and the instruction itself. The Instruction Cache does not require any validity bits because the Instruction Cache loads the instructions from the Memory and does not modify the data. Each load from memory switches the way used to ensure two instructions beside each other in memory but in different memory blocks can be loaded.

If a memory sector not containing instructions is loaded, then the Decoder can interpret random data as valid instruction.

3.1.12 Data Cache

The Data Cache combines load and store buffers from Tomasulo's design. The cache, however, works slightly differently than Tomasulo's because it manages tags. The Load/Store unit can speculatively change the cache's values, in which case the confirmation of the instruction's completion from ROB is needed. The Data Cache is connected to the CDB as shown in the Figure 3.4 and Code listing 3.2.

The cache record contains 32-bit data, data memory address, instruction address, tag, record state, and word select.

The word select determines what bits are valid for a given operation. The word select is either WORD (32-bit), HALFWORD (16-bit), BYTE (8-bit), or EMPTY (0-bit, error state).

The record state indicates the state of the data in a record. The record state is always one of the following:

VALID states that the data in the record are just loaded from the Memory.

WAITING is used when the data have been modified, but the instruction has yet to be completed by ROB.

INVALID is the default state of a cache record. This state is used for all records after the initial reset or if the record is tagged and the tag delete signal is issued.

MODIFIED signifies that the record has been modified and ROB has completed the instruction that changed the data.

The invalidation of the cache records follows a different order during load operations than store operations. Load operations invalidate records in this order:

1. INVALID
2. MODIFIED
3. VALID

Store operations invalidate records in this order:

1. MODIFIED, where the data address of the incoming store and record match,
2. INVALID
3. VALID
4. MODIFIED

Ejected MODIFIED records are sent into the Write Buffer. WAITING records are not invalidated as the cache is designed to be significantly bigger than the Reservation station feeding the LS.

■ **Code listing 3.2** Data Cache module ports.

```
module DataCache #(
  parameter cache_size = 16
)(
  GlobalSignals.rest global_signals,
  CommonDataBus.cache data_bus1, data_bus2,
  DataCacheBus.cache data_ls,
  DataMemoryBus.cache data_mem
);
```

3.1.13 Write Buffer

The Write Buffer is a queue of ejected MODIFIED Data Cache records. If the Data Memory Bus is unused, the Write Buffer writes the modified data to the Memory.

3.1.14 Memory Management Unit

The Memory Management Unit (MMU) manages access to the Memory. The MMU unifies the Instruction and Data Memory Buses into one to approach more traditional designs.

3.2 Connections

This section describes all buses and connections used in the VTM. The most significant buses are:

Common Data Bus (CDB) propagates results from execution units through the system. The VTM contains two Common Data Buses to allow for multiple writes simultaneously. Access to the CDB is managed through Arbiters. The first CDB is prioritised by Reorder buffer (ROB), and execution units prioritise the second CDB.

Code listing 3.3 Common Data Bus Wires.

```
interface CommonDataBus;
    wire [31:0] data, address, jump_address;
    wire [5:0] arn, rrn;
    wire [3:0] select;
    wire we;
endinterface
```

- `data` carries the result of the instruction if the instruction writes to a register,
- `address` is the instruction address and is used for identification purposes through the system,
- `jump_address` carries the result of a jump or branch instruction and is needed because jump instructions return two values, one that is to be stored in a register and another that replaces the current `pc` value,
- `arn` and `rrn` carry architectural and renamed destination register numbers respectively,
- `select` is used by Arbiters to communicate the availability of the bus,
- `we` enables writing into the Register File.

Instruction Issue Bus (IIB) is used to issue all the necessary information from Dispatch to the Reservation stations and the Reorder buffer (ROB). Similarly to the CDB, there are also two IIBs used to issue two instructions simultaneously to achieve superscalar behaviour.

Code listing 3.4 Instruction Issue Bus Wires.

```
interface InstrIssue;
    logic [31:0] data1, data2, address, imm;
    logic [5:0] src1, src2, arn, rrn;
    logic valid1, valid2, jump, tag;
    PID pid;
    Station stat_select;
endinterface
```

- `data1`, `data2` are used as operands in the instructions,
- `address` is the instruction address and is used for identification purposes and address-relative calculations,
- `imm`, short for immediate,
- `src1`, `src2` specify the source register numbers,
- `arn`, `rrn` specify the destination register numbers,
- `valid1`, `valid2` indicate the validity of the `data1` and `data2` respectively,
- `jump` specifies if the instruction jumps and is only used by the ROB,

- tag indicates that the instruction is being executed speculatively,
- pid is unique instruction operation identifier,
- stat_select defines for which reservation station the data is targeted.

Register Query (RQ) connects the Dispatch and the Register File.

Code listing 3.5 Register Query Wires.

```
interface RegisterQuery;
    logic [31:0] reg1_data, reg2_data;
    logic [5:0] ret_renamed, reg1_ren, reg2_ren;
    logic [4:0] reg1_num, reg2_num, reg3_num;
    logic reg1_valid, reg2_valid, get_renamed, tag;
endinterface
```

- reg1_data, reg2_data return register value with their respective validity reg1_valid, reg2_valid,
- ret_renamed returns number of newly linked renamed register if get_renamed is 1 otherwise returns nothing,
- reg1_ren, reg2_ren return architectural register number if the register is not renamed; otherwise return the number of the renamed register,
- reg1_num, reg2_num are architectural register numbers which DD uses to query the values,
- reg3_num is used for renaming purposes,
- tag indicates speculative renaming.

Global Signals (GS) connects all components.

Code listing 3.6 Global Signal Wires.

```
interface GlobalSignals(input logic clk, reset);
    logic delete_tagged, clear_tags;
endinterface
```

- clk is the global clock signal,
- reset is used to reset all components to their default state,
- delete_tagged signal is used when the branch is taken,
- clear_tag signal is used when the branch isn't taken.

PC Interface (PCI) connects the program counter with the Dispatch, Reorder buffer, and Instruction Cache.

Code listing 3.7 PC Interface Wires.

```
interface PCInterface;
    logic [31:0] jump_address, address;
    logic inc, inc2, wr;
endinterface
```

- jump_address is used to set the value of the PC,
- address outputs the current PC value,
- inc signals to the PC to add 4 to the current value,

- `inc2` signals to the PC to add 8 to the current value,
- `wr` signals to the PC to replace the current value with the `jump_address`

Less significant interconnects are:

Data Memory Bus connects the Data Cache with the Memory Management Unit (MMU).

Data Cache Bus connects the Load/Store unit (LS) with the Data Cache.

Instruction Memory Bus connects the Instruction Cache with the Memory Management Unit (MMU).

Instruction Cache Bus connects the Instruction Cache with the Decode.

Memory Bus connects the Processor with the Memory.

Execution Feed bundles wires that connect the Reservation stations with their respective execution unit.

3.2.1 Arbiter

The Arbiters manage access to CDB, to which multiple components often write simultaneously. The Arbiter is a modified version of the arbiter design [15]. Each component that writes to the CDB has its Arbiter with a unique arbiter address which dictates bus access priority. Priorities/addresses were selected by the expected utilisation of a given component and can be set with the `address` parameter viz. Code listing 3.8.

Code listing 3.8 Arbiter module parameter and ports.

```
module Arbiter #(
    parameter address = 4'b0000
)(
    input logic get_bus,
    inout wire [3:0] select,
    output logic bus_granted
);
```

Chapter 4

Simulation and Testing

This chapter describes the simulation processes and tests used.

The GitLab repository [16] contains all source and simulation code required and a user manual on how to run the simulation. The simulation requires the Vivado ML Edition [17] because it supports all used SystemVerilog features.

The simulation was used to validate the functionality of the VTM design. The validation was done using two SystemVerilog testbenches. The first testbench `TopTest` runs a single test and displays its waveform, an example of such simulation is shown by Figure 4.3. The second testbench `TopTestAll` runs RISC-V RV32I official tests for all implemented instructions, thus overseeing the overall VTM functionality. There is also the `Top` testbench, which runs any program, not just tests, and focuses on displaying the program's waveform.

Additionally to the testbenches, a "fake" RAM module was implemented, which loads a `.hex` file, which contains a program. The format of the used `.hex` file contains program instructions represented as 32-bit hexadecimal little-endian values with only one instruction per line. An example of a such file is given in the Code listing 4.1.

Code listing 4.1 .hex file format example.

```
ffff83b7
48771263
00600193
800000b7
00000113
```

If custom tests are used in the `TopTest` testbench, they must adhere to the pass/fail interface the testbench uses. The `TopTest` and `TopTestAll` testbenches control the value of the `x11` register. If the register contains value `0x0600d000` at the end of the simulation run, then the test passes. Any other values will fail.

4.1 RISC-V Official Tests

The RISC-V foundation provides its official tests (available at [18]) for each instruction set. These tests contain an assembly language file for each instruction in an official RISC-V instruction set. All tests must include the `riscv-test.h` header file containing the necessary macros used during testing. The RISC-V official header file had to be modified for my testing purposes because the official test relies on functionality absent in my design. An example of the modified header file is Code listing 4.2, where macros for successful and unsuccessful tests are defined. As mentioned before, the value `0x0600d000` is required for a test to be deemed successful because that is the value specified in the `RVTEST_PASS` macro.

Code listing 4.2 Example of a customized testing environment for the official RISC-V tests.

```
#define RVTEST_PASS                                                    \
        li a1, 0x60;                                                   \
        slli a1, a1, 8;                                                \
        addi a1, a1, 0xd;                                              \
        slli a1, a1, 12;                                               \
        ecall

#define TESTNUM gp

#define RVTEST_FAIL                                                    \
        addi a1,a1,0xBA;                                               \
        slli a1,a1,4;                                                  \
        addi a1, a1, 0xD;                                              \
        slli a1, a1, 16;                                               \
        ecall
```

In this case, the official tests for the RV32I instruction set were used, specifically the RV32UI tests, as the VTM does not support privileged mode. Because the official tests cover every instruction and a lot of edge-case scenarios that can result in data or control hazards, no other tests were needed. Code listing 4.3 shows one of 38 tests for the instruction ADD.

Code listing 4.3 Example of a section from a RISC-V RV32UI official ADD test.

```
80000440 <test_30>:
80000440:       01e00193         li     gp,30
80000444:       00000213         li     tp,0
80000448:       00b00113         li     sp,11
8000044c:       00e00093         li     ra,14
80000450:       00000013         nop
80000454:       00208733         add a4,ra,sp
80000458:       00120213         addi tp,tp,1 # 1 <_start-0x7fffffff>
8000045c:       00200293         li     t0,2
80000460:       fe5214e3         bne tp,t0,80000448 <test_30+0x8>
80000464:       01900393         li     t2,25
80000468:       10771a63         bne a4,t2,8000057c <fail>
```

4.1.1 Official Tests Compilation

The RISC-V official tests come with their `Makefile`, which builds tests for all the instruction sets. This is unnecessary as, for our purposes, tests for the RV32I suffice. Therefore the official `Makefile` was modified to compile only necessary tests. The compiled official tests need to be translated into the `.hex` files for the Vivado to load properly as memory. A custom Python script was created for this translation that takes binary `.elf` file and returns the respective `.hex` file.

Requirements for compiling the RISC-V official tests and their simulation in the Vivado:

1. Vivado ML Edition (the edition used was 2022.2) [17]
2. Python 3.10 or higher [19]
3. `riscv64-unknown-elf-gcc`, other compilers may be used, but this is what the official test use by default; if another compilator is used, the `Makefile` needs to be modified

4. `riscv64-unknown-elf-binutils`
5. Git [20]
6. Cmake [21]

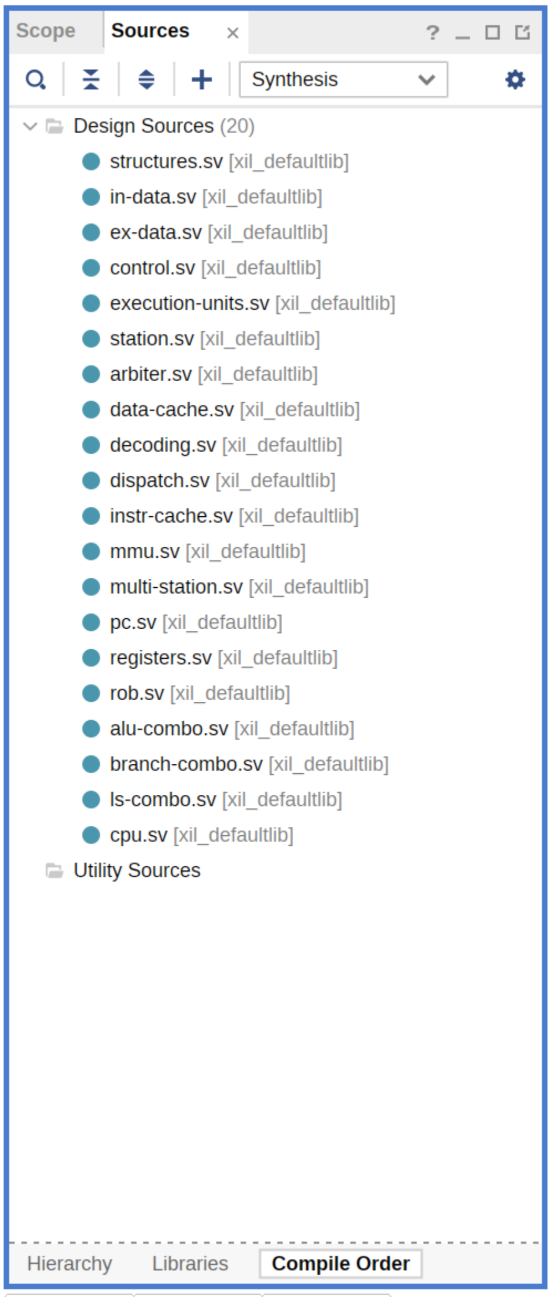
The following Code listing 4.4 lists command necessary to prepare the VTM-official repository [16] for simulation.

Code listing 4.4 Steps required to prepare everything needed for the Vivado simulation.

```
git clone https://gitlab.fit.cvut.cz/veznitom/VTM-official
cd VTM-official
git submodule update --init --recursive
./build-tests.sh
python3.10 elfhex.py
```

These are the steps to run a simulation in a Vivado (tutorial with all pictures available at [16]):

1. Create a new Viado project `File->Project->New` or use the Quick Start.
2. Name the project and select the project location.
3. Select `RTL Project` and check `Do not specify sources at this time`.
4. `Boards` and select any available board (preferably Artix-7 AC701).
5. `Finish` and now we have a new clear Vivado RTL project.
6. `Add Sources`.
7. Select `Add or create design sources`.
8. `Add Directories` and select the path to the `VTM-official/src` directory.
9. `Finish`
10. `Add Sources`
11. Select `Add or create simulation sources`.
12. `Add Directories` and select the path to the `VTM-official/sim` directory.
13. `Finish`
14. Reorder the compilation order as seen on the Figure 4.1.
15. Select desired simulation file in the Hierarchy window, right-click the simulation file and select `Set as Top` to make the file a simulation target.
16. Change the `absolute-path` in the path portion of the desired testbench to the absolute path to the test.
17. `Flow->Run Simulation->Run Behavioural Simulation` or use the Flow Navigator.
18. After the previous step, the simulation should start and open the waveform diagram. The Vivado should automatically load pre-configured waveform diagram configurations for the Top and TopTest testbenches. If not, repeat the same steps for loading the simulation files, but instead, load files and select `VTM-official/sim/Top.wcfg` and `VTM-official/sim/Top.wcfg`.



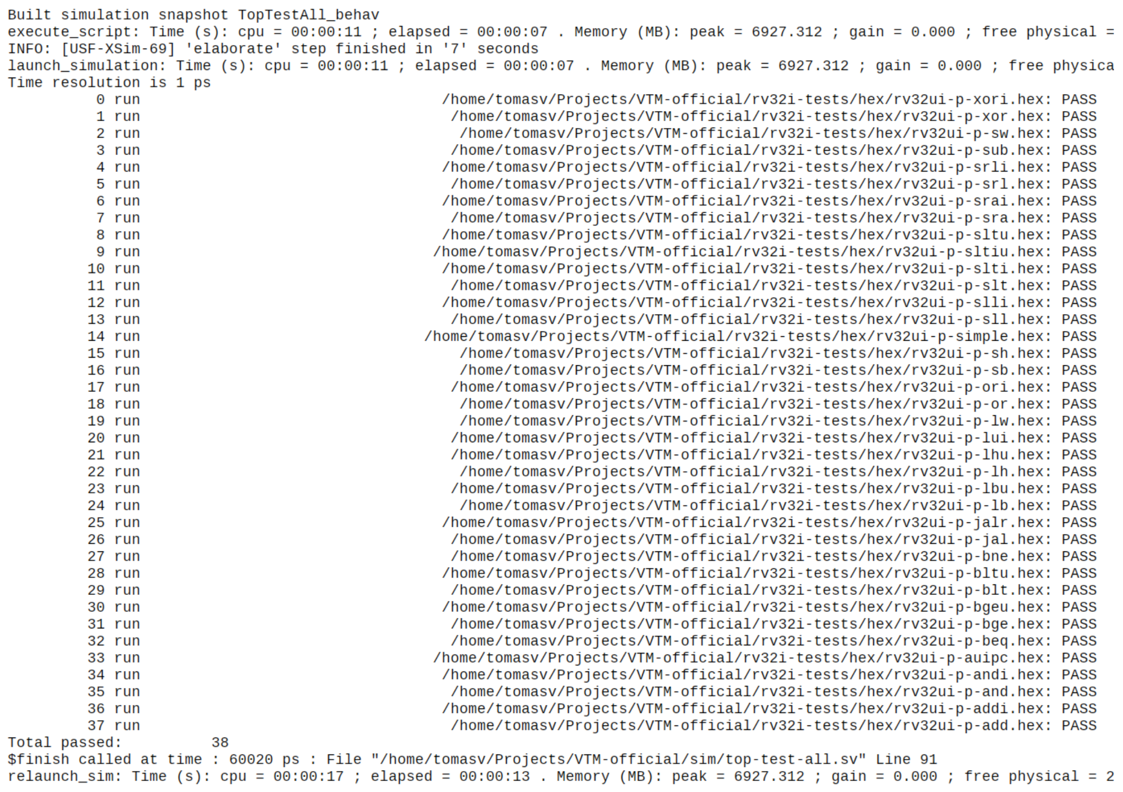
Design Sources (20)
- structures.sv [xil_defaultlib]
- in-data.sv [xil_defaultlib]
- ex-data.sv [xil_defaultlib]
- control.sv [xil_defaultlib]
- execution-units.sv [xil_defaultlib]
- station.sv [xil_defaultlib]
- arbiter.sv [xil_defaultlib]
- data-cache.sv [xil_defaultlib]
- decoding.sv [xil_defaultlib]
- dispatch.sv [xil_defaultlib]
- instr-cache.sv [xil_defaultlib]
- mmu.sv [xil_defaultlib]
- multi-station.sv [xil_defaultlib]
- pc.sv [xil_defaultlib]
- registers.sv [xil_defaultlib]
- rob.sv [xil_defaultlib]
- alu-combo.sv [xil_defaultlib]
- branch-combo.sv [xil_defaultlib]
- ls-combo.sv [xil_defaultlib]
- cpu.sv [xil_defaultlib]

Utility Sources

Figure 4.1 Vivado project compilation order.

4.1.2 Results

The RISC-V official test provided an excellent tool that helped discover various bugs and imperfections in the implementation. The Figure 4.2 shows a printed debug message containing the official test results for every implemented instruction. An additional test called `rv32ui-p-simple.hex` is present to control that tests function correctly.



```
Built simulation snapshot TopTestAll_behav
execute_script: Time (s): cpu = 00:00:11 ; elapsed = 00:00:07 . Memory (MB): peak = 6927.312 ; gain = 0.000 ; free physical =
INFO: [USF-XSim-69] 'elaborate' step finished in '7' seconds
launch_simulation: Time (s): cpu = 00:00:11 ; elapsed = 00:00:07 . Memory (MB): peak = 6927.312 ; gain = 0.000 ; free physica
Time resolution is 1 ps
          0 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-xori.hex: PASS
          1 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-xor.hex: PASS
          2 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sw.hex: PASS
          3 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sub.hex: PASS
          4 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-srli.hex: PASS
          5 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-srl.hex: PASS
          6 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-srai.hex: PASS
          7 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sra.hex: PASS
          8 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sltu.hex: PASS
          9 run                                 /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sltiu.hex: PASS
         10 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-slti.hex: PASS
         11 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-slt.hex: PASS
         12 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-slli.hex: PASS
         13 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sll.hex: PASS
         14 run                                /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-simple.hex: PASS
         15 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sh.hex: PASS
         16 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-sb.hex: PASS
         17 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-ori.hex: PASS
         18 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-or.hex: PASS
         19 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-lw.hex: PASS
         20 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-lui.hex: PASS
         21 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-lhu.hex: PASS
         22 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-lh.hex: PASS
         23 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-lbu.hex: PASS
         24 run                                    /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-lb.hex: PASS
         25 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-jalr.hex: PASS
         26 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-jal.hex: PASS
         27 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-bne.hex: PASS
         28 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-bltu.hex: PASS
         29 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-blt.hex: PASS
         30 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-bgeu.hex: PASS
         31 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-bge.hex: PASS
         32 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-beq.hex: PASS
         33 run                                 /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-auipc.hex: PASS
         34 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-andi.hex: PASS
         35 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-and.hex: PASS
         36 run                                  /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-addi.hex: PASS
         37 run                                   /home/tomasv/Projects/VTM-official/rv32i-tests/hex/rv32ui-p-add.hex: PASS
Total passed:          38
$finish called at time : 60020 ps : File "/home/tomasv/Projects/VTM-official/sim/top-test-all.sv" Line 91
relaunch_sim: Time (s): cpu = 00:00:17 ; elapsed = 00:00:13 . Memory (MB): peak = 6927.312 ; gain = 0.000 ; free physical = 2
```

Figure 4.2 Results of the TopTestAll testbench.

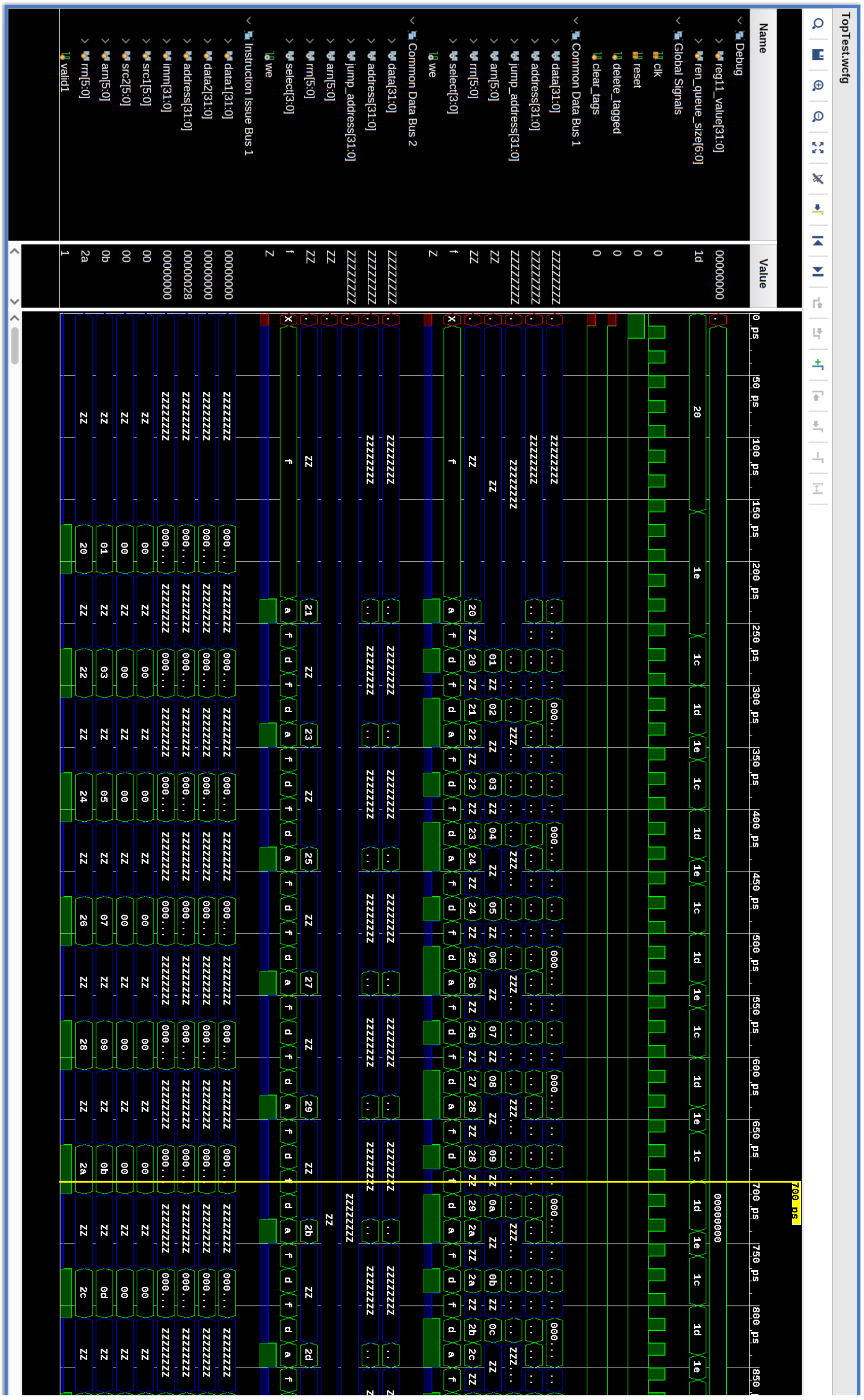


Figure 4.3 Showcase of a Vivado simulation waveform diagram. The test run in the figure is the official test for the ADD instruction.

Chapter 5
Conclusions

The main objective of this thesis was to design a simple superscalar processor microarchitecture based on the RISC-V instruction set architecture. Additional goals were to explain the working principles of superscalar processor microarchitectures, describe the designed microarchitecture in the SystemVerilog, and to simulate the design to demonstrate its functionality.

The main goal was fulfilled as the designed microarchitecture is described in the chapter 3 and in the [16] source files and was validated by official tests provided by [18]. The chapter 2 briefly explains every principle, technique, or scheme used in the designed microarchitecture fulfilling another set objective. The chapter 4 provides information on how the designed microarchitecture's functionality was verified and how to run the simulation. Overall, the thesis completes all the established objectives.

The resulting microarchitecture has room for improvement as the experience gained during the design process was initially missing. In the future, the main improvement to the implementation could be reducing the more advanced SystemVerilog features (interface modports, imports, ...) as the open-source compilators struggle or outright do not support these features. Since this thesis is aimed at anyone, the ability to run the simulation should not be locked to a specific advanced application suite like Vivado.

All source and simulation codes are available at the [16].

Bibliography

1. @DAISYOWL. *CPU tweet*. twitter.com, 2017. Available also from: https://twitter.com/daisyowl/status/841802094361235456. [cit. 04.04.2023].
2. HENNESSY, John L.; PATTERSON, David A.; ASANOVIĆ, Krste; BAKOS, Jason D.; COLWELL, Robert P.; BHATTACHARJEE, Abhishek; CONTE, Tom; DUATO, José; FRANKLIN, Diana; GOLDBERG, David; JOUPPI, Norman P.; LI Sheng, 1984; MURALIMANOHAR, Naveen; PETERSON, Gregory D.; PINKSTON, Timothy M.; RANGANATHAN, Parthasarathy; WOOD, David A.; YOUNG, Cliff; ZAKY, Amr. *Computer architecture: a quantitative approach*. Sixth. Cambridge, MA: Morgan Kaufmann Publishers, Elsevier, 2019. ISBN 9780128119051.
3. BLEM, Emily; MENON, Jaikrishnan; SANKARALINGAM, Karthikeyan. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 1–12. Available from DOI: 10.1109/HPCA.2013.6522302.
4. WATERMAN, Andrew; ASANOVIĆ, Krste (eds.). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. 2019. Version 20191213. Available also from: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf. [cit. 28.03.2023].
5. *GLOSSARY RISC*. Arm Limited (or its affiliates), 2023. Available also from: https://www.arm.com/glossary/risc. [cit. 28.03.2023].
6. *RISC-V About*. RISC-V International, 2021. Available also from: https://riscv.org/about/. [cit. 28.03.2023].
7. *RISC-V Assembly Programmer's Manual*. RISC-V International, 2023. Available also from: https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md. [cit. 28.03.2023].
8. *ArithmeticException Class*. Microsoft, 2023. Available also from: https://learn.microsoft.com/en-us/dotnet/api/system.arithmeticexception?view=net-8.0. [cit. 29.03.2023].
9. PATTERSON, David A.; HENNESSY, John L. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface [eBook]*. Elsevier Science & Technology, 2013. ISBN 9780124078864. Available also from: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=5376640. [cit. 30.03.2023].
10. BAER, Jean-Loup. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors [eBook]*. New York: Cambridge University Press, 2010. ISBN 9780511675461. Available also from: https://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=312518&lang=cs&site=ehost-live&ebv=EB&ppid=pp_Cover. [cit. 30.03.2023].

11. *Intel® Pentium® 4 Processor6x1 Sequence*. Intel Corporation, 2007. Available also from: `https://www.intel.com/Assets/PDF/datasheet/310308.pdf`. [cit. 01.04.2023].
12. SMITH, J.E.; SOHI, G.S. The microarchitecture of superscalar processors. *Proceedings of the IEEE*. 1995, vol. 83, no. 12, pp. 1609–1624. Available from DOI: `10.1109/5.476078`.
13. DUTTA ROY, Taposh. Instructional Level Parallelism. 2015. Available from DOI: `10.13140/RG.2.1.3338.9920`.
14. ING. MICHAL ŠTEPANOVSKÝ, Ph.D. *Superskalární procesory II: Zpracování paměťových instrukcí [online]*. Fakulta informačních technologií, České vysoké učení technické v Praze, 2021. Available also from: `https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska11-SuperscalarCPUs-II.pdf`. [cit. 09.05.2023].
15. ING. ALOIS PLUHÁČEK, CSc. doc. *Sběrnice [online]*. Katedra číslicového návrhu, Fakulta informačních technologií, České vysoké učení technické v Praze, 2020. Available also from: `https://courses.fit.cvut.cz/BI-JPO/@B201/media/lectures/JPO-1-11-K-V2.pdf`. [cit. 18.04.2023].
16. *VTM-official GitLab*. Tomáš Věžník, 2023. Available also from: `https://gitlab.fit.cvut.cz/veznitom/VTM-official`. [cit. 06.04.2023].
17. ADVANCED MICRO DEVICES, Inc. *Vivado ML Edition*. 2023. Version 2022.2. Available also from: `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools.html`. [Accessed 23.04.23].
18. *RISC-V official tests [online]*. RISC-V International®, 2023. Available also from: `https://github.com/riscv-software-src/riscv-tests`. [cit. 10.05.2023].
19. *Python Download [online]*. Python Software Foundation, 2023. Available also from: `https://www.python.org/downloads/`. [cit. 10.05.2023].
20. *Git Download [online]*. Software Freedom Conservancy, Inc., 2023. Available also from: `https://git-scm.com/downloads`. [cit. 10.05.2023].
21. *CMake Download [online]*. Kitware, Inc, 2023. Available also from: `https://cmake.org/download/`. [cit. 10.05.2023].

Attached Media Contents

```
code
├── build-tests.sh ........................ Bash script for compilation of the official tests.
├── elhex.py ................... Custom Python script to translate binary files to hex files
├── rv32i-tests ..................................... Folder for official RISC-V tests.
├── sim ....................................................... VTM simulation sources.
│   ├── fake-ram.sv
│   ├── top-test-all.sv
│   ├── top-test.sv
│   ├── top.sv
│   ├── Top.wcfg
│   └── TopTest.wcfg
└── src ............................................................. VTM source codes
    ├── components
    │   ├── arbiter.sv
    │   ├── cpu.sv
    │   ├── data-cache.sv
    │   ├── decoding.sv
    │   ├── dispatch.sv
    │   ├── execution-units.sv
    │   ├── instr-cache.sv
    │   ├── memory-management.sv
    │   ├── multi-station.sv
    │   ├── pc.sv
    │   ├── registers.sv
    │   ├── rob.sv
    │   ├── station.sv
    │   ├── alu-combo.sv
    │   ├── branch-combo.sv
    │   └── ls-combo.sv
    ├── interfaces
    │   ├── control.sv
    │   ├── ex-mem-data.sv
    │   └── in-data.sv
    └── structures.sv
thesis ............................................... Folder for LaTeXsource codes.
├── Images ................................................... Folder with images used.
└── Chapters ................................................ Folder with my LaTeXfiles.
```

```
│
│   │
│   ├── Text ................................................ Folder with template LATEXfiles.
│   ├── ctufit-thesis.cls
│   ├── ctufit-thesis.tex
│   └── veznitom-assignment.pdf
└── thesis.pdf
```