CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Graphical RISC-V Architecture Simulator

Memory Model and Project Management

Jakub Dupák
dev@jakubdupak.com

May 2021

https://github.com/cvut/qtrvsim

Supervisor: Ing. Pavel Píša PhD.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Dupák Jakub** Personal ID number: **483785**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Graphical RISC-V Architecture Simulator - Memory Model and Project Management

Bachelor’s thesis title in Czech:

Grafický simulátor architektury RISC-V - pamětový model a vedení projektu

Guidelines:

RISC MIPS architecture simulator has been developed to teach the Computer Architecture (B35APO) subject. It allows the visualization of both simple and pipelined processor variants and activity including cache and peripherals visualization. RISC-V architecture is becoming the future choice for computer architectures education. The architecture is open from the very moment of design and its authors are authors of the textbook that is considered the standard of quality in computer architecture education.
1. Familiarize with RISC-V processor architecture and respective standards and actual textbook.
2. Redesign simulator memory model to allow its use for littleendian architecture, mapped filesystem files and 64-bit targets.
3. Update visualization of the processor core.
4. Update documentation and packaging of the project.

Bibliography / sources:

[1] Patterson, D. A., and J. L.: Computer Organization and Design RISC-V Edition, The Hardware Software Interface 1rd ed. Morgan Kaufman, 2017, ISBN: 9780128122754
[2] Patterson, D. A., and J. L.: Computer Organization and Design RISC-V Edition, The Hardware Software Interface 2nd ed. Morgan Kaufman, 2021, ISBN: 9780128203316
[3] Waterman, A., Asanovic, K.: The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, CS Division, EECS Department, University of California, Berkeley, 2020 https://riscv.org/technical/specifications/
[4] Kočí, K.: Graphical CPU Simulator with Cache Visualization, Master's Thesis,Czech Technical University in Prague
[5] QtMIPS - MIPS CPU Simulator for Education Purposes, https://github.com/cvut/QtMips/

Name and workplace of bachelor’s thesis supervisor:

Ing. Pavel Píša, Ph.D., Department of Control Engineering, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **01.02.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Pavel Píša, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgement / Declaration

I would like to thank my supervisor for continuous help and support, my colleague Max Hollmann, who worked in parallel on different parts of the simulator, for theoretical consultations, František Vacek for an introduction into the *svgscene* and *libshv* libraries, and Matěj Kafka for our discussions about code style and code organization and his comments on this text.

In addition, I would like to acknowledge our entire study group, Max Hollmann, Matěj Kafka, Vojtěch Štěpančík and Jáchym Herynek, for endless technical discussions, mental support, and motivation always to try harder.

Finally, I would like to mention my high school informatics teacher, Vladimír Drápalík, who initially sparked my interest in computers.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague 21.5.2021

.....

Abstrakt / Abstract

Předměty spojené s architekturou počítačů vyučované na Fakultě elektrotechnické ČVUT využívají MIPS jako modelovou architekturu pro demonstraci principů činnosti počítačů. Důvodem je její jednoduchost a také fakt, že autorem je jeden z autorů celosvětově uznávané učebnice. Tito autoři se svými studenty navrhli novou architekturu – RISC-V, která lépe splňuje požadavky pro široce použitelnou a výkonnou architekturu a která je zároveň pod publikována otevřenou licencí. Cílem této práce je navrhnout a implementovat změny vhodné k převedení aktuálně používaného simulátoru *QtMips* na instrukční sadu RISC-V. Práce se zabývá především paměťovým modelem, vizualizací jádra procesoru a správou projektu.

Klíčová slova: RISC-V, architektura počítačů, CPU simulátor, memory-model, SVG, Qt, QtRVSim, QtMips

Computer architecture lectures at the Faculty of Electrical Engineering CTU are using MIPS ISA to demonstrate the internal principles of computers. This is due to MIPS simplicity and the fact that it was designed by one of the authors of a worldwide recognized textbook. Those authors, together with their students, have designed a new architecture, RISC-V, which not only satisfies requirements for usable and performant ISA but is also published under an open license. This thesis aims to design and implement simulator changes desirable to switch the currently used simulator *QtMips* to the RISC-V ISA. It focuses on the memory model, core visualization, and project management.

Keywords: RISC-V, computer architecture, CPU simulator, memory-model, SVG, Qt, QtRVSim, QtMips

Contents /

Tables / Figures

Chapter 1
Introduction

The Computer Architectures course at the Faculty of Electrical Engineering CTU has been using MIPS architecture to demonstrate processor operations for more than a decade. The MipsIt[1] simulator provided with the selected textbook[3] was used at the beginning. However, it became outdated, and no alternative capable of pipeline and cache visualization, which would be suitable for the course practices, was found. Therefore, Ing. Karel Kočí started a thesis project[1] to develop an up-to-date replacement and finally, he released a MIPS simulator *QtMips*, which is currently in use. He started the thesis with these words: *"Computers are dominating a lot of industry sectors, including engineering as they are essential production tools. At least basic knowledge of the inner working of a processor and ability to predict its influence on performance, security and safety consequences is important for each programmer expert, computer, processor and embedded systems designer and advanced user."* Since then, the importance of computer architecture knowledge has kept increasing. Many fields that require a more profound understanding of hardware principles are growing fast:

- The Internet of Things (and embedded development in general) now affects almost every sector of the economy[17]. Miniature computers are being incorporated in almost every new consumer product—from cars through washing machines and electric kettles to special water pipes[18]. Work with embedded computers cannot rely on vast layers of abstraction, and significant knowledge of hardware is necessary.
- After many years of x86_64 supremacy, Apple presented an ARM-based laptop CPU with surprising specifications[19], raising new challenges in the processor design, operating systems, drivers, and compilers.
- The design of domain-specific chips is also a growing market, where RISC-V found its place. Even here, in the Czech Republic, many companies including Honeywell[2], S3[3], Codasip[4] and Expressif[5] are offering jobs related to processor chip and ASIC design. Codasip and Expressif are known to work with RISC-V chip designs.

Understanding computer design is not only important for developers working close to the hardware. David Patterson and John L. Hennessy, laureates of the Turing Award 2017 for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry, say that anybody who aims to write performant software should learn the internal principles of computers: "*While programmers could ignore the advice and rely on computer architects, compiler writers, and silicon engineers to make their programs run faster or be more energy-efficient without change, that era is over. For programs to run faster, they must become parallel. While the goal of many researchers is to make it possible for programmers to be unaware of the underlying parallel nature of the hardware*

[1] https://www.eit.lth.se/fileadmin/eit/courses/eit090/MipsIt/MipsITEnvRef.html
[2] https://www.honeywell.com/cz/en
[3] https://www.s3connectedhealth.com/s3-group
[4] https://codasip.com/
[5] https://www.espressif.com/

they are programming, it will take many years to realize this vision. Our view is that for at least the next decade, most programmers are going to have to understand the hardware/software interface if they want programs to run efficiently on parallel computers"[6, pg. xi] Performance is important in many kinds of modern software, from realistic games, real-time high-resolution multimedia processing, virtual reality and server applications to complex scientific simulations and machine learning.

RISC-V architecture is becoming the future choice for computer architecture education. The architecture is open from the very moment of design. It was designed by PhD students of David Patterson and John L. Hennessy; the authors of the textbook that is considered the standard of quality in computer architecture education. Both Faculty of Electrical Engineering and Faculty of Information Technology intend to start teaching their computer architecture lectures using RISC-V shortly. I have joined this effort with my bachelor's thesis. My task was to help switch the currently used MIPS CPU simulator to RISC-V. As stated in the formal assignment, my primary focus was on memory subsystem simulation and project management.

The thesis is structured according to the subtasks of assignment guidelines. Given that the subtasks are mostly unrelated subproblems, analysis and solution descriptions are provided for each subtask individually.

The text begins with a brief overview of the RISC-V ISA, followed by the core part of the thesis — the redesign of the memory model. The memory subsystem has been reworked to support more CPU configurations and memory-mapped files. The next chapter focuses on a new approach towards core visualization and differences in the typical RISC-V core organization CPUs that had to be addressed. The graphics of the CPU core is newly based on SVG files, which are interpreted by the simulator. The final two chapters present changes to the project structure, build system and packaging. The project was restructured and upgraded from deprecated QMake to CMake. The testing suite was extended and automated using GitHub continuous integration. Packages and building service configuration have been updated for the CMake build and RISC-V edition of the simulator, and two new packages have been introduced.

Chapter 2
RISC-V ISA

RISC-V has been designed for education since day one. The first line of the first specification version states: "*RISC-V is a new instruction set architecture (ISA) designed to support computer architectures research and education*"[9, 8]. No previously available ISA was well-fitting for teaching. x86, ARM, and MIPS are too complex for teaching examples, and they are protected by intellectual property. Designing a new processor core using these architectures poses a great legal (and financial) challenge. That is not true for RISC-V. It starts from a simple set of instructions and extends it orthogonally. It is open-source and royalty-free. Anyone can design a RISC-V core, even a high school student[7, 8:12], and publish it freely[1][2][3] or profit from a proprietary design[10]. The article [10] clearly shows that major world companies see RISC-V as something worth investing in. Alibaba, Amazon, AMD, Google, Hewlett Packard Enterprise, IBM, Microsoft, NVIDIA, Qualcomm, Samsung, and Western Digital are all members or sponsors of the RISC-V foundation.[6, pg. xiii]

2.1 Brief RISC-V ISA Overview

RISC-V is a load-store reduced instruction set computer architecture. It provides a set of basic integer instruction sets with different register widths (32bit, 64bit, 128bit, and reduced 32bit for embedded computers). Those are the minimal mandatory sets of instructions for any implementation. It provides basic ten ALU operations with register and immediate operands, load and store instructions, jumps and conditional branches relative to the program counter, memory fence, ECALL[4] and EBREAK. The instructions operate on 32 (reduced: 16) general-purpose registers, where the zero register is hard-wired to zero value. For simplicity, integer operations cause no arithmetic exceptions, and all operations are defined. A program can check for all problematic situations simply itself. All other instructions are optional extensions. Some opcode ranges are guaranteed to be kept free for custom domain-specific extensions. This organization is advantageous in multiple ways. Students can work with a minimal and simple, yet complete, set of instructions. For industry, this allows designers to put on the chip only needed circuits, saving silicon, power and potentially improving safety by reducing the attack surface. Standard extensions (some of them not stable yet) include IEEE 754-2008 floating-point operations, bit operations, a vector extension, a compressed (16bit) instructions extension, and even a JIT and a cryptography extension. Such combinatorial variety would significantly complicate compiler support. Therefore *RISC-V International* provides "architectural profiles" that designate the most common configurations.[7, 12:27]

[1] `https://github.com/lowRISC/rocket`
[2] `https://github .com/riscv-boom/riscv-boom`
[3] `https://github.com/chipsalliance/Cores-SweRV-EH2`
[4] Call to the execution environment, e.g., system call.

The instruction formats are optimized for simple instruction decoding. Here is an example from the specification: "*The RISC-V ISA keeps the source (rs1 and rs2), and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediate used in CSR instructions (Chapter 9), immediate is always sign-extended and is generally packed towards the leftmost available bits in the instruction and allocated to reduce hardware complexity. In particular, the sign bit for all immediate is always in bit 31 of the instruction speed sign-extension circuitry.*"[2, pg. 15]

Instruction encoding is ready for variable length instructions (arbitrary multiples of 16 bits). The standard[2, pg. 8] provides encoding rules for up to 176bit. Longer instructions have reserved instruction space, but the encoding has not been defined yet.

Chapter 3
Memory Model Redesign

Task 2 of the thesis assignment required me to redesign the memory system to support simulation of little- and big-endian, 32- and 64-bit targets, and the usage of memory-mapped files[1]. To satisfy the requirements, the memory hierarchy and the read/write API had to be reworked. New types and classes have been introduced to produce cleaner and safer API. Finally, a considerable amount of refactoring and documenting was needed.

3.1 Original Model

The MIPS simulator emulates only 32-bit big-endian CPUs. It uses word-sized[2] functions for all memory-related operations. This way, it can transfer data through return values. The main memory is naturally represented by an array of words. In the case of a read, the memory subsystem always loads the whole word. Smaller reads are extracted from it. When data write size is not a complete 32-bit word, the whole word containing the current value has to be read first. Then the new value is packed into it, and it is written back. Given that offset accesses and memory map are not supported, the main memory can internally have the same endianness as the host machine (the computer running the simulator). When the simulated and the host system endianness mismatch and the access size is smaller than a word, the top-level functions compensate for the difference.

The memory subsystem consists of components, e.g., cache, memory (RAM), address space mapping component and peripherals. All components implement a `MemoryAccess` interface which defines the read/write API. Some components can form chains by accepting another instance of `MemoryAccess` as their backing memory (multi-level cache, cache and RAM). A component that receives a request either resolves it or invokes its backing component. Memory is addressed by a 32bit integer. The CPU core typically has data memory and program memory entry points.

3.2 Modified Memory Hierarchy Overview

The new model recognizes two types of memory interfaces: `FrontendMemory` and `BackendMemory`. Every CPU core configuration has three mandatory frontend memory components. Two of them are the entry points for memory operations for the program and the data, respectively. The entry point starts a chain of memory components of arbitrary length, each backed by the next one. For the core, the chain is invisible, and it only interfaces with the top-level component. The chain is terminated by a memory data bus. The data bus for the data memory is the third component mentioned. The frontend memory has three characteristics. Except for the memory bus, each frontend

[1] `https://www.man7.org/linux/man-pages/man2/mmap.2.html`
[2] 32-bit integer

component must have a frontend component backing it. The memory bus is backed by zero or more backend devices (explained later). All addresses used in the frontend are full memory addresses. By the word *full*, I mean the numerical value observable by a C program (up to address translation[1]), in contrast to a local relative offset. The `Address` datatype is used to store the address and pass it as an argument. Finally, frontend memory instances are likely those found on the CPU chip itself (cache, TLB).

Example. In the simplest CPU with no cache and joint memory address space, all three components coincide. They are all references to the same object, the data bus. It is both the entry point and the exit point of the chain.

Example. A more common configuration is a split single-level cache with a single memory bus (joint memory address space). The caches are the entry points here, and both of them are backed by the bus. With multi-level caches, we only have to add more cache objects to the chain. If we were to implement a TLB, it would become the top-level component of the chain.

Example. When simulating a CPU with the program and the data memory split, we create them with separate memory buses and pass the *data* one to the core. The core needs an access to the *data* memory data bus to emulate functionalities like file access using memory map at runtime. The separate program memory is read-only at the simulation runtime.

The other part of the memory model is the backend memory. Real-world examples are the main memory, memory-mapped I/O peripherals, and in our simulation also memory-mapped files from the host system. I often call backend memory instances *devices* as, in the real world, they are not permanent parts of the machine. Backend memory devices generally do not form chains. They can internally consist of multiple layers, but that behaviour is not a part of the backend memory interface. They connect to a memory data bus, listening for operation on a certain address range. In contrast to a typical simple hardware implementation, the address range filtering is performed by the memory bus rather than the components. In modern computers the situation is more complex (e.g. PCIe). Each component is addressed by a local relative offset starting at zero for each component. To refer to this offset, a type alias `Offset` is used for 64bit unsigned integer. The address to range conversion is a responsibility of the bus. Devices can be inserted or removed at any time, for example, by an operating system emulation.

3.3 Load / Store API

The requirement to simulate both 32- and 64-bit machines makes the simple solution of fixed-width accesses unusable. Using a wider access and ignoring part or multiple narrower accesses would produce misleading statistics, mainly in caches. It would also pose a problem for the compressed instruction extension and the vector extension. Based on the suggestion of my supervisor, I have designed the API inspired by the semantics of POSIX functions `read`, `write` and `memcpy`. It takes a destination, a source, a size, and, in addition to `memcpy`, also an options struct and returns a result struct. The API ensures that *size* bytes will be transferred from the *source* to the *destination*. In particular, a read transfers data from a simulated location specified by the source to a buffer pointed to by the destination and write transfers data from a source buffer to a simulated destination location. This way, arbitrary-sized types can be transferred,

[1] `https://en.wikipedia.org/wiki/Memory_management_unit`

ranging from a single byte to whole cache lines. Transfers larger than 64 bits are, however, more complex due to endian issues. For convenient use, the `FrontendMemory` interface provides simplified value-passing-based entry points for basic integral types. They provide a temporary buffer and handle the endianness internally. The complexity of the endianness simulation is discussed later.

```
uint16_t value = 42;
Write result res = some_frontend_memory.write(
    0xff_addr,
    &value,
    sizeof(value),
    WriteOptions()
);
// or for common integer types
some_frontend_memory.write_u16(0xff_addr, 42);
```

Example. *Demonstration of the new API usage.*

The special types `ReadOptions`, `ReadResult`, `WriteOptions` and `WriteResult` keep the API simple and extensible. Separating parameters and return values (in contrast to returning multiple values using side-effects) produces a much simpler and more predictable data flow. Both result types contain information on the successful access size. An access can happen on a boundary of some form of a region (a page, a physical RAM module) and succeed only partially. For more advanced uses, like a vector extension, partial success is a necessary feature. Write result struct also contains the information on whether the write changes the affected memory. The options struct currently only contains a flag determining the intended effects of the read. Internal reads should not cause any side effects, and their repetition should always return the same values. Internal reads are used, for example, by the GUI to display current memory data.

3.3.1 Endian Simulation

There are eight possible combinations of endianness for each memory data path. The host system, the simulated machine, and the final backend device can have any endianness. The tricky part is maintaining the correct results for unaligned and offset accesses.

The aim is to make the host and the simulated main memory equivalent, when they are interpreted as arrays of bytes. When the stimulated and the native endiannesses are the same, the situation is trivial, and no swaps are needed. Otherwise, we need to store the value to memory the same way the simulated machine would. When simulating little-endian on big-endian, we need to swap the value as it would be swapped when written by the simulated little-endian machine. Conversely, when simulating big-endian on little-endian, pre-swapping results in the value being stored in big-endian order as the host machine swaps the value again. (See the example below.)

All nonperipheral components (cache and main memory) store data in the same endianness. Therefore, no swap is required between them, and data can be copied fast. That is enforceable as the data is only accessible via the memory API. Peripherals have to behave in this manner externally, but they might be forced to use a different endianness internally. The reason may be that their internal registers are directly observable by the UI or entirely out of simulator control. The internal storage can be a

```
BIG on LITTLE
    REGISTER:           12 34 56 78
    PRE-SWAP:           78 56 34 12 (still in register)
    NATIVE ENDIAN MEM:  12 34 56 78 00 00 (native is LITTLE)
    READ IN MEM:              56 78 00 00
    REGISTER:                 00 00 78 56
    POST-SWAP:                56 78 00 00 (correct)

LITTLE on BIG
    REGISTER:          12 34 56 78
    PRE-SWAP:          78 56 34 12  (still in register)
    NATIVE ENDIAN MEM: 78 56 34 12 00 00 (native is BIG)
    READ IN MEM:              34 12 00 00
    REGISTER:                 34 12 00 00
    POST-SWAP:                00 00 12 34 (correct)
```

Example. *Correct handling of 4 bytes write followed 4 bytes read offset by 2 bytes. I assume memory to be zeroed initially. The visual offset represents the address offset. (This snippet is also a part of the inline code documentation.)*

memory-mapped file from the host system, or it can even lead to a real memory-mapped hardware, which dictates the endianness. As a proof of concept, I have created a backend device that uses an anonymous memory map to simulate the main memory instead of a malloc[1] allocated tree [1, pg. 26]. Another option is to map an executable in this way. Each backend memory device has a public `simulated_machine_endian` property for easier orientation. It can be constant (LCD), set according to host endianness (main memory), or set dynamically at initialization (mapped executable).

3.3.2 32-bit Accessible Peripherals

When switching to the new model, modifying the peripherals to be internally byte-addressable would be too complicated. Section 2.6 of the standard[2] allows declaring that some regions only support word access and raise an exception on a misaligned access. However, it was decided to keep the behaviour simple for a user and emulate the misaligned access within the peripherals.

3.4 Refactoring

To implement the new model, I had to perform a large amount of refactoring. I mainly intended to simplify the data flow, break the code into smaller components, introduce new types and type aliases, rename too short names to more descriptive ones, and provide more comments. In the following sections, I present the most conceptually essential changes.

3.4.1 Address Data Type

Prior to these changes, a 32-bit integer was used to store and pass a frontend memory address. This is problematic for three reasons. It is fixed to 32-bit and hard to change.

[1] `https://man7.org/linux/man-pages/man3/malloc.3.html`

```
QMap<std::uint32_t, hwBreak *> hw_breaks;
// vs
QMap<Address, hwBreak *> hw_breaks;
```

Example. *This collection controls program locations, where CPU core should stop execution and notify the execution environment. This feature is used mainly by debuggers.*

It provides no documentation (see example below). Also, the compiler cannot check its usage. Now, if a function requires a simulated address, either you already have an `Address`, or you have to construct it explicitly and take responsibility for its correctness.

The type checking is extra helpful when using slots[1]. The functions being connected are often from distant parts of the codebase (e.g., simulation and GUI), which might even be maintained by different developers.

```
void fetch_inst_addr_value(machine::Address);
void fetch_jump_reg_value(uint32_t);
void fetch_jump_value(uint32_t);
void fetch_branch_value(uint32_t);
void decode_inst_addr_value(machine::Address);
```

Example. *Some of the 58 lines declaring slots in the machine core.*

3.4.2 RegisterValue Data Type

`RegisterValue` type was introduced to abstract away the width of the simulated machine registers. All arithmetic operations are performed on a private 64-bit internal storage, regardless of the simulated machine's configuration. From the type safety point of view, `RegisterValue` is considered the most general integral value. All integers can implicitly *degrade* to it (i.e., implicit construction is allowed). In the other direction, we are assuming some interpretation of the value, and we have to do that explicitly. *Explicit* static casts to common integer types are implemented. All constructors, getters, and cast operators ensure proper sign extension as required by the RISC-V standard Section 5.2:

*These *W instructions ignore the upper 32 bits of their inputs and always produce 32-bit signed values, i.e., bits XLEN-1 through 31 are equal.*[2]. (In the 64bit instruction set, `*W` stands for instructions postfixed with `W`. Such instructions operate on 32bit values in 64bit registers.)

When simulating a 32-bit machine, we behave as if all instructions were of the `*W` type.

3.4.3 Cache Replacement Policy

Cache policy replacement handling was previously tightly integrated into the cache code. I have separated it into a decoupled component with no knowledge of the cache except its dimensions. Now every policy consists of tiny blocks of code, which can be tested separately. They are also much simpler to verify by reading. These advantages are desirable as the cache simulation is likely to be extended in the future.[2]

[1] Qt component communication primitive. Primarily used for asynchronous updates of the GUI.
[2] Even at this time, a Greek colleague is adding L2 cache simulation to the QtMips.

3.4.4 Extended Testing

Given the increased complexity of the memory subsystem, the current testing suite, which tested only a few configurations, seemed unsatisfactory. I have built a test generator, which tests the full cartesian product of the cache configurations. This turned out to be a good decision as it helped me discover and debug multiple serious problems. The execution time of all machine unit tests is still under one second, which is entirely negligible compared to the compilation time. To test the correct offset behaviour, I have devised a mechanism that decomposes a 64-bit integer to all subcomponents (32, 16, 8) as if they were read on each simulated endianness regardless of the native endianness. Automation of cache performance testing is problematic. Therefore, I have decided to only test for future changes. I have saved the current results to an array, and I compare them based on the order. Any change of the configuration matrix invalidates these values, and they have to be regenerated.

3.4.5 Qt5 Signal-Slot Syntax

This part is closely related to the introduction of the address data type. The original Qt4 string-based system has proven very error-prone, and it made it hard to track all code paths, where `Address` should replace `uint32_t`. With Qt4 syntax, type checks are performed when the `connect` function is called, and type errors only result in runtime warnings. Qt5 syntax[1] replaces the string names with C++ method referencing. Usage of the language native features instead of meta-object compiler immediately enables standard C++ compiler type checking. I have upgraded all connections to Qt5 syntax to benefit from the extra type checking. Fortunately, QtCreator[2] has a semi-automatic way of converting those syntaxes, and it worked most of the time.

```
connect(
    sender, SIGNAL( valueChanged( QString, QString ) ),
    receiver, SLOT( updateValue( QString ) )
);
```

Example. *Old syntax.*

```
connect(
    sender, &Sender::valueChanged,
    receiver, &Receiver::updateValue
);
```

Example. *New syntax.*

A minor disadvantage of the new syntax is that it does not support implicit overloading. It has to be done explicitly in quite a wordy way.

3.5 Backporting

All the changes described here are compatible with both, QtRVSim and QtMips. The git branch has been kept separate from all RISC-V-related changes. With minor mod-

[1] `https://wiki.qt.io/New_Signal_Slot_Syntax`
[2] An official Qt integrated development environment.

```
connect(ser_port, &machine::SerialPort::tx_byte,
        this, QOverload<unsigned int>::of(&TerminalDock::tx_byte));
```

Example. *Problematic overloading.*

ifications to the core that Ing. Píša offered to implement on his own, the new memory model can be backported to the CTU/QtMips repository. We intend to keep it in the main development branch for anyone who would contribute to the QtMips.

After a discussion among QtMips/QtRVSim developers, it was decided that it did not make sense to release the changes to students in the master branch. The gain for current teaching is minimal, and there is a risk of new bug introduction. We intend to concentrate our efforts on the QtRVSim.

Chapter 4
Core View Update (GUI)

The differences in the instruction sets of MIPS and RISC-V cause significant differences in a typical core organization. Task 3 of the assignment required me to reflect them in the core view. As part of those changes, I have implemented a long-planned switch to a system that builds the scene by interpreting an SVG file. I have recreated the visualization as a vector diagram with hooks to inject data from the simulator. To achieve a usable visualization and scene modification API, I have extended the SVG handling library. Now, a developer can edit the scene in a GUI editor and pay more attention to visual details.

4.1 RISC-V Differences

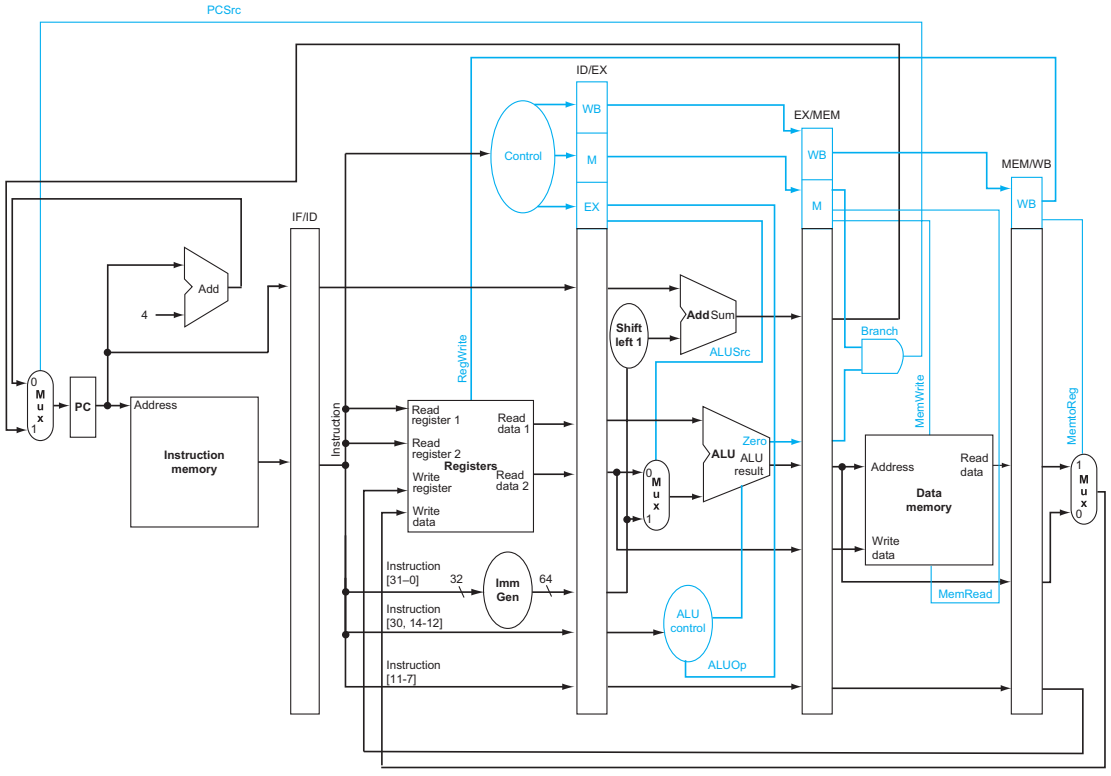

Figure 4.1. RISC-V pipeline data path from the coursebook. You can compare it with the original visualization of QtMips on the next page. (Source: [5, figure 4.49])

MIPS pipelined CPUs can branch directly from the decode stage using equality comparison on the register values. Meanwhile, RISC-V provides more branch instructions and uses ALU to determine whether they should be taken. Because ALU is in the execute stage, the whole branching mechanism occurs in later stages. Control signals

and branch destination address bus need to be delayed (saved to interstage registers). Forwarding to the decode stage, previously required by the `jump and link` instruction, is no longer needed in RISC-V. To replace the decode stage comparator, a zero signal from the ALU was added.

RISC-V ISA encodes the destination register number always to a constant location. This fact makes one multiplexer and its control signal redundant. RISC-V does not have a delay slot, and it has complex immediate decoding [2, section 2.3].

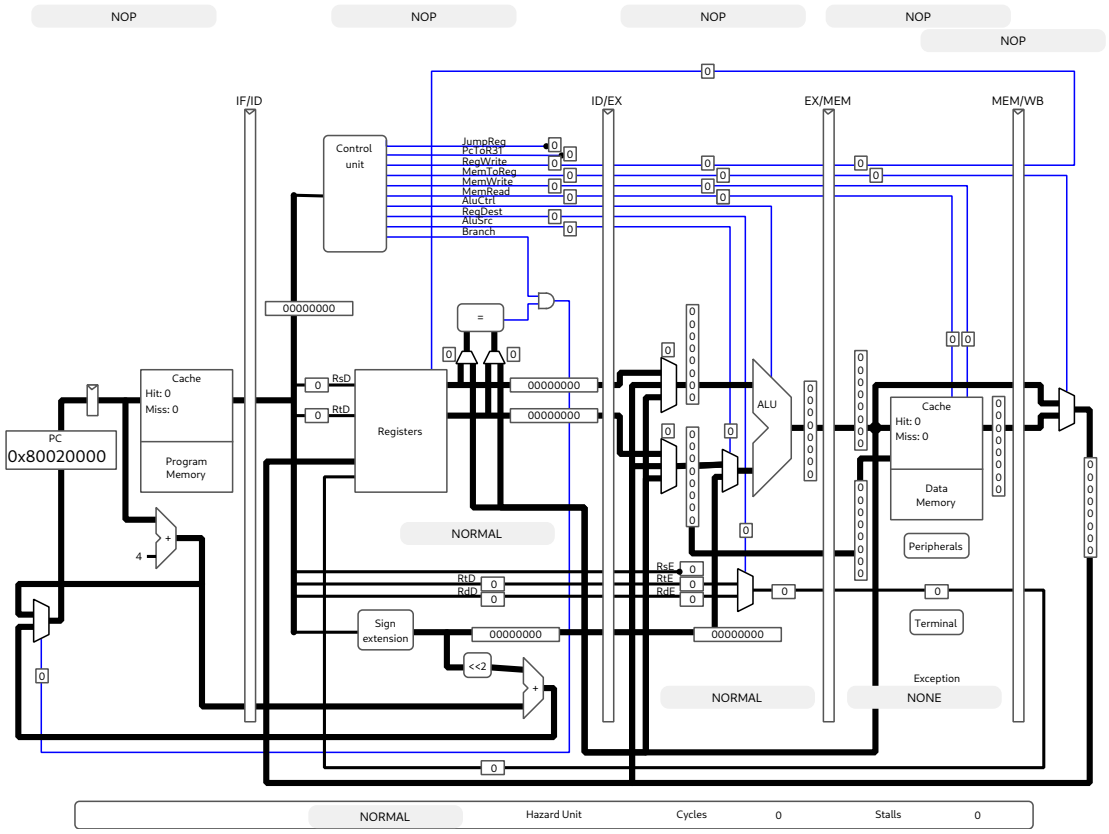

Figure 4.2. Original QtMips core view at maximum complexity setting.

4.2 QtMips C++-based Visualization Framework

The core view is described entirely in the C++ code. Ing. Karel Kočí created a framework of objects representing individual parts of the CPU core and macros used to assemble a core visualization from them. An example of such object is a multiplexer: It has a method to paint itself, it exposes points to connect wires, and it has a slot to update its state (in this case, which input is the output connected to). Then, macros are used to add the component to the screen. The macro allocates the object, connects it to signals, and saves a pointer to the component. Later, wires could be connected. Aside from the lack of documentation and occasional unclear naming, the framework seems well built. However, modification of the layout is not a simple task. The first problem is to match objects with components on the screen. The second problem is that every movement requires recompiling to ensure that the changes were correct. Adding simple details to the image means creating a class, allocating an object, and positioning it manually.

4.3 Svgscene Library and Its Usage

As a replacement for the direct creation of Qt objects, QtRVSim builds the Qt scene by parsing an SVG file. Dynamic values are updated using a custom API similar to XML Document Object Model (DOM)[15]. The library created for this purpose is based on `svgscene`[1], `libshv`[2] and QtSVG[3].

First, SVG is parsed, and each SVG element is translated to a Qt object with corresponding styles while preserving XML attributes. Then, the elements of interest are retrieved using the document traversing API. Then, one proceeds as with any other Qt graphics items and objects.

In the next section, I describe how the SVG files themselves are created. Before that, I present the document traversing API.

4.3.1 Document Traversing API

The key component of the API is `SvgDomTree`, which wraps the `QGraphicsItems` in SVG aware wrapper. `SvgDomTree` has methods to read XML and CSS attributes and their values and methods for searching the scene/document tree. Methods `find` and `findAll` search child elements using a naive depth-first-search[4] and return first, and all matching elements, respectively, wrapped as `SvgDomTrees`. The find methods accept the type of searched element as a template parameter, and the XML attribute name and value to search as function parameters. All parameters are optional. The default value for the template parameter is `QGraphicsItem`. The wrapped Qt object can be obtained by a call to the `getElement` method.

```
document
    .getRoot()
    .find<QGraphicsItem>("data-component", "data-cache")
    .find<SimpleTextItem>()
    .getElement();
```

Example. *Finds the element corresponding to the cache components and finds a text element inside.*

```
for (auto hyperlink : document.getRoot().findAll<HyperlinkItem>())
{
  this->install_hyperlink(hyperlink.getElement());
}
```

Example. *Real snippet from QtRVSim: This code searches all elements corresponding to XML hyperlinks[16, section 14] and calls a method to install them (bind them to Qt slots).*

[1] https://github .com/fvacek/svgscene
[2] https://github.com/silicon-heaven/libshv
[3] https://github.com/qt/qtsvg
[4] https://en.wikipedia.org/wiki/Depth-first_search

4.3.2 Error Handling

It can often happen that no element is found. Returning a null pointer would require an enormous amount of checks which would significantly complicate the API usage. Therefore, `SvgDomTree` can never contain a null. Any attempt to create `SvgDomTree` from null results in an exception being raised.

For compatibility with WASM, this safe behavior is only used at the top level. Internally, the `find` method, searching a single item, returns a null pointer when nothing is found. This is because WASM does not support exception handling.[1]

4.4 Core Diagram and SVG Image

Given that most parts of the image never change, it felt natural to replace them with a static vector image. SVG fits very well this use; however, direct use of an SVG was not ideal from a maintenance point of view. The advantage of a specialized diagramming tool over an SVG editor is that it provides better support for working with connection lines. This includes forced orthogonal lines, automatic line crossing, connection points for custom shapes, and moving the lines together with objects.

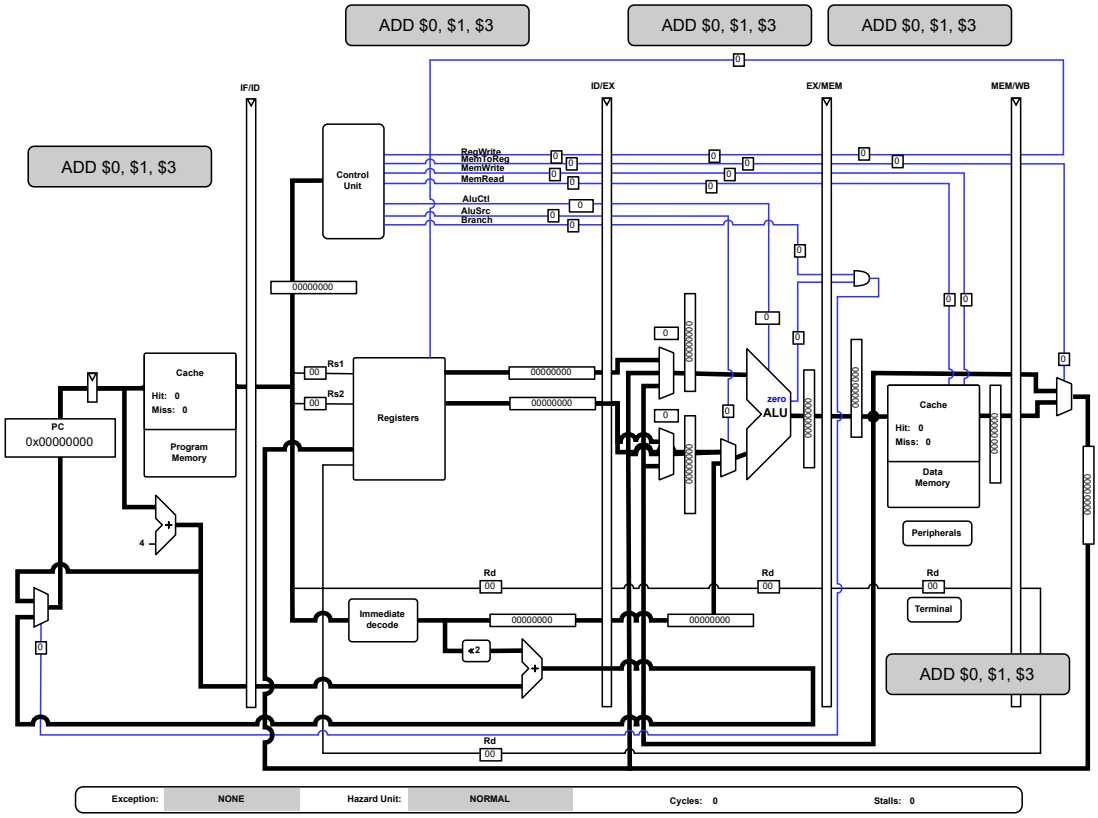

Figure 4.3. New SVG-base coreview.

I have decided to use diagrams.net (formerly known as draw.io). It is a free, open-source, and well-documented tool working in a browser or locally. Diagrams.net has many useful features and plugins. It allowed me to keep all configurations (single cycle CPU, pipelined CPU, and pipelined CPU with forwarding) in a single diagram, reusing

[1] `https://emscripten.org/docs/porting/exceptions.html`

the common parts. I have annotated every part to belong to a subset of the listed configurations. It can easily display each of them or all at once (for layout debugging purposes). Hyperlinks are used to open related sections of the GUI (for instance, memory view by clicking on memory). Data attributes are used to annotate components with updatable values and their data sources. A downside is that I cannot modify all objects of the same "class" at once. I have to replace each instance separately or copy-paste the style. Furthermore, the options to control the resulting SVG are limited. For example, the cache statistics had to be created as a custom element to ensure that all text elements are children of the cache component.

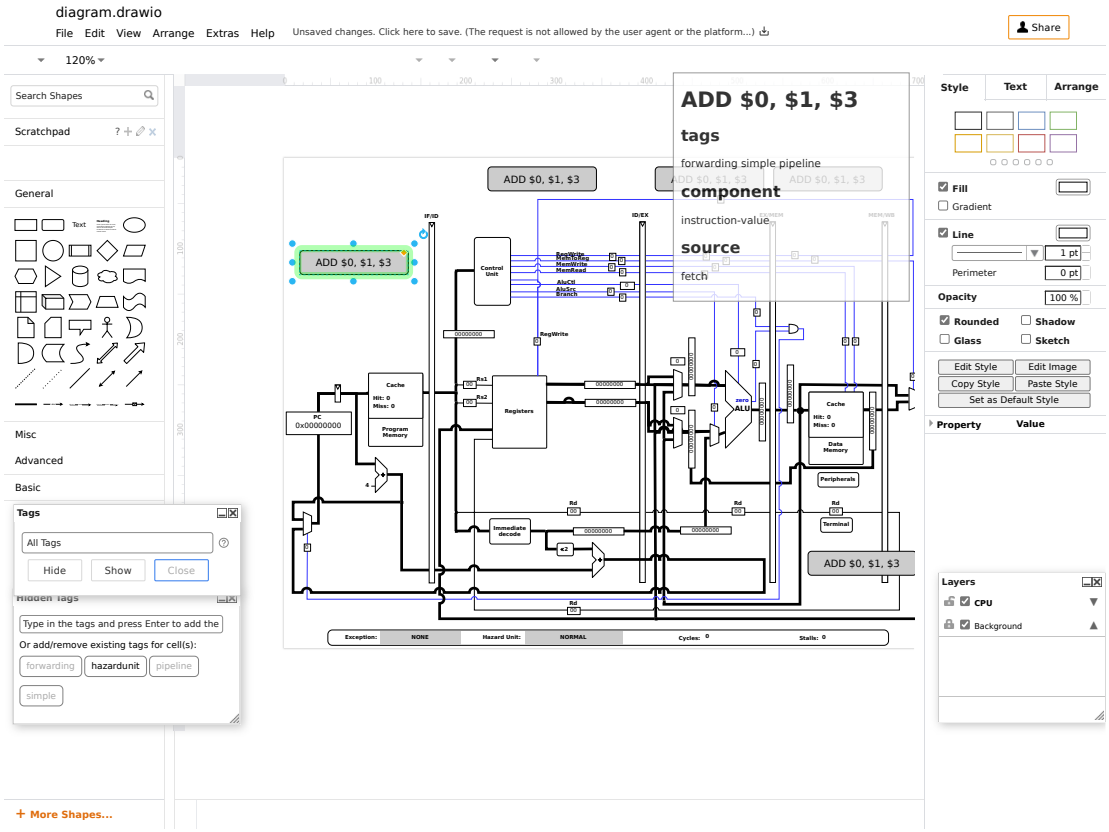

Figure 4.4. Diagrams.net editor.

The diagram source file is located in `extras/core-graphics`. The usage of diagrams.net and the exporting process is documented in `docs/developer/coreview-graphics/using-drawio-diagram.md`.

All shapes that I have used (fully custom or compositions of simpler shapes) are available on the GitHub[1].

[1] `https://github.com/jdupak/Diagrams.net-CPU-scheme-kit`

Chapter 5
Project Management

The title of the thesis makes me responsible for project management. This chapter discusses various subtasks not fitting under task four of the assignment. Task four (packages and documentation) is covered by the next chapter.

This part mainly relates to the directory structure and the process of building. The chapter starts with project structure changes, aiming to keep unrelated files in separate directories, especially in the project's root directory. Source code has been separated from support files, and a new subproject for shared code with a new internal compiler-compatibility library has been introduced. The project has been upgraded from deprecated QMake to CMake. Finally, a CI multi-platform testing procedure was set up.

5.1 Project Structure and Common Libraries

I have moved all source subprojects to the `src` directory, 3rd party code, and git submodules to `external` directory, and support content to `extras`. In `src`, I have introduced a new subproject, `common`, for small custom libraries shared between the subprojects. Previously, such code was placed in the machine (previously qtmips_machine) subproject, which was already conveniently included in other subprojects. However, I found this to be problematic. The concerned code had no direct bindings to the machine, and it should have been tested separately.

One of the libraries I have created in `common` is called `polyfills`. Its purpose is to abstract away compiler API differences for nonstandard features. Having compiler detecting `ifdef`s within the project code is hard to read, debug and test. Instead, the code is placed in a header file in `polyfills` directory, all compiler-dependent intrinsic are unified under a common interface and tested. In addition to development testing, these tests allow users compiling the code themselves to verify proper support on their current platform.

Example: mulh64. in `polyfills/mulh64.h` contains a set of functions calculating high bits of 64-bit multiplication. These functions are required to implement the RISC-V multiplication extension. However, the functionality is not directly available in the C or C++ standard. The implementation uses 128-bit integers on GCC compatible compilers (if available), intrinsic for some sign combinations on MVSC and manual fallbacks otherwise. This functionality is platform dependant, and as such, it must be covered by testing.

Example: Byteswap. `polyfills/byteswap.h` provides integer byte-swapping functions. For GCC (and compatible compilers, e.g., Clang) and MVSC, compiler builtins are used. Otherwise, optimized fallbacks are provided. I have considered using the fallbacks only; however, swapping is necessary on hot code paths, and experiments at `https://godbolt.org` showed that some major compilers could not optimize it properly.

Example: Endian detection. Including the file `polyfills/endian_detection.h` ensures that either macro `__LITTLE_ENDIAN__` or `__BIG_ENDIAN__` is defined. If detection fails, the compilation immediately stops with a clear error message.

Example: Endian utils. `common/endian.h` library wraps the low-level C functions (unified in `polyfills`) into a better abstracted C++ and provides additional functionalities, e.g. conditional byte swap `byteswap_if(T value, bool cond)`.

Polyfills library is considered low-level, and it does not introduce any new behavior. It only unifies the interfaces. More high-level libraries are placed elsewhere in `common`.

5.2 CMake Build System Generator

As part of project changes, I have suggested moving from QMake to CMake. QMake is now deprecated and is no longer developed. Qt itself moved to CMake. CMake is the most widely used build system generator for C++. It is well supported by development tools (IDEs, GitHub Actions, Nix), and it can generate various kinds of build files. Even though the CMake syntax contains many historical relicts, I believe that the current configuration is more straightforward to maintain than the QMake one.

In the following paragraphs, I provide some details on QMake deprecation, present some problems that CMake helped me solve more elegantly, and in the end, I provide sources I found helpful when learning CMake.

Note: *I admit that my knowledge of QMake is slim, and I have no base to claim that the presented problems could not be solved in QMake. However, learning proper use of QMake seemed to be a comparable task to learning CMake. Given the deprecation of QMake and better support of CMake, I have decided to favor CMake.*

5.2.1 Qt Switch to CMake

In 2019, the Qt company decided to switch the build of the Qt framework itself to CMake. An article[4] published in early 2021 presented the results. The main reason for the switch was the popularity of CMake among Qt project developers. One example is the KDE community, which is behind many popular Qt programs used on Linux. Another reason was that Qt did not want to develop and maintain a build tool and instead focus on the development of the framework. The switch had a noticeable influence on CMake itself. CMake now supports tools needed by Qt (e.g., the meta-object-compiler[1]) out of the box. Qt also influenced support of pre-compiled headers, Unity builds, and iOS support.

5.2.2 Build Targets Relationships

Previously all relationships between build targets were specified manually by relative paths (see example below) in each built target that used them. Instead, CMake brings the concept of targets. A target is a collection of information necessary to build (called PRIVATE properties) and link (called PUBLIC properties.) an executable or a library. All properties are specified at the point of target creations. In other parts of the build configuration, targets are only referred by their names.

[1] Qt C++ preprocessor that enables reflection features and is needed to make most of the Qt functionalities (like signal-slots) work.

```
LIBS += -L$$OUT_PWD/../os_emulation/$${LIBS_SUBDIR}  -los_emulation
LIBS += -L$$OUT_PWD/../machine/$${LIBS_SUBDIR} -lmachine -lelf
LIBS += -L$$OUT_PWD/../assembler/$${LIBS_SUBDIR} -lassembler -lelf

PRE_TARGETDEPS += $$OUT_PWD/../os_emulation/$${LIBS_SUBDIR}/\
                  libos_emulation.a
PRE_TARGETDEPS += $$OUT_PWD/../machine/$${LIBS_SUBDIR}/libmachine.a
PRE_TARGETDEPS += $$OUT_PWD/../assembler/$${LIBS_SUBDIR}/\
                  libassembler.a
```

Example. *QMake code used to link dependencies of the* `gui` *executable.*

```
target_link_libraries(gui
      PRIVATE machine os_emulation assembler)
```

Example. *CMake alternative to the previous example. Notice that* `elf` *is not mentioned in the new version. The LibElf library is not directly used by the* `gui` *target, and therefore it is not mentioned in its CMakeLists.txt. It is hidden in the* `machine` *target as a transitive public dependency.*

5.2.3 Configure-time choice of dependencies

Typically, QtRVSim uses the system LibElf library[1]. However, in special cases, we want to fall back to a self-build, statically linked version. Originally, this was the case of the WASM release, which cannot dynamically link libraries. In QtMips, this was not resolved as the WASM target was new. There was a git branch where LibElf was built and linked unconditionally. Now in CMake, I have a target interface, which is used the same way in the whole project. At configure time, the interface either points to the system library or a local CMake target. Another advantage is that if CMake fails to find LibElf in the system, it automatically falls back to a static version. It is also helpful for Windows and macOS. As my Greek colleague informed me, there no ARM version of LibElf in Homebrew. I also expose a CMake configuration option to force static version usage.

5.2.4 Config Defaults And Overrides

CMake provides a mechanism to specify the default properties of the whole project. I use this to set warning levels, C++ standard, and add runtime sanitizers to debug builds.

5.2.5 Build and Run Tests

QtMips uses Bash scripts to build and run tests. It has to create a test build directory, invoke QMake and make, run the tests and determine results. CMake integrates a tool for test management called CTest. Test run and build tasks are now part of the same build file (make/Ninja).

[1] A library for handling ELF executables.

5.2.6 Learning CMake

My experience with CMake before this project had only been with trivial projects. Therefore, I had to understand it enough to make the switch possible. Here are some resources that I found helpful:

Modern CMake online book[1] provides a complete basic introduction to CMake as a build system generator as well as a programming language. It highlights modern features that make development much more effortless.

CMake and Qt article by KDAB[2] provides more detail on CMake usage with Qt.

C++Now 2017: Daniel Pfeifer Effective CMake[3] conference talk presents and explains the concepts of modern, target-oriented CMake. I found the talk very useful to understand which parts of CMake I should use. CMake legacy API can sometimes be confusing.

Mastering CMake: Ken Martin, Bill Hoffman is a reliable resource on CMake. I have not read it directly, but many helpful StackOverflow posts and blogs cite from it.

5.3 GitHub CI Tests

Providing students with a malfunctional simulator can profoundly affect students' understanding of the subject, and their motivation to continue. Every developer makes mistakes, and therefore our only chance to reduce the risk is proper testing. Given that GitHub now provides unlimited CPU time to run continuous integration tasks (CI), it makes perfect sense to automate testing on every push. Moreover, CI can run the tests on multiple platforms to eliminate accidental platform-dependent code. From the table with download statistics (B.1 in the appendix), it is evident that we cannot dismiss proper Windows support.

Therefore, I have implemented a CI testing procedure using GitHub Actions[4]. It builds QtRVSim for Ubuntu, macOS, and Windows and runs all tests on every git push. After examining the download statistics (B.2), Ubuntu 18 was added as a test platform in addition to Ubuntu 20. In addition, the WASM target is built. It does not contain tests, but given many specific problems of the WASM compilation, testing the ability to build it seems appropriate. In case of problems, the author is notified by email.

The CI manifest can be later used for the continuous delivery of binaries for Windows and macOS. The CI procedure already exposes the resulting files for debugging purposes, e.g., examining macOS bundle structure or manually testing WASM release. This topic is also discussed in the next chapter.

5.4 Logging Library

The introduction of the library was not intended to be part of my thesis. However, during my work, I have met and sometimes created many temporary `printf` statements. Some of them were commented out, some in `#if false` or `if (false)`. Such a situation is not ideal for maintenance, so after consulting with my supervisor, I started searching for a logging library. First, I have collected requirements from my colleague, my supervisor, and myself:

[1] `https://cliutils.gitlab.io/modern-cmake`
[2] `https://www.kdab.com/wp-content/uploads/stories/KDAB-whitepaper-CMake.pdf`
[3] `https://www.youtube.com/watch?v=bsXLMQ6WgIk`
[4] `https://docs.github.com/en/actions`

- Lightweight — we only need simple logging with minimal overhead.
- Readable log statements that can serve as comments.
- Runtime filtering by category.
- Crossplatform — e.g., there is no standard output by default on Windows for GUI applications.

`https://cpp.libhunt.com/libs/logging` provided me with a reasonable listing of logging libraries. After analyzing the available libraries, I have found that the main differentiator is the printing mechanism. In C++ , there are generally 3 options: `printf`, `std::iostreams`, `std::fmt/fmtlib`[1]. `std::fmt` is only available in C++20. The `fmtlib` library is the most performant with very readable `printf`-like syntax, and support for C++ custom object printing; however, it would bring another 16 thousand lines of code to the project. `std::iostreams` support C++ object printing; however, the syntax is highly unreadable, and statefulness makes them hard to use. `printf` does not support C++ objects and may be problematic on Windows.

After long consideration, I have chosen the logging library already present in Qt. It supports `printf` syntax and provides a function to print Qt types. It is already part of the codebase. And it has a powerful category filtering mechanism that does not use program arguments. It is configured to use a native sink on all platforms. It even works in the WASM release, where it uses the JavaScript console.

I have added simple wrapper macros `LOG`, `INFO`, `WARN`, and `ERROR` which implicitly use a category defined in the file.

Other serious candidates were: Necrolog[2] (very lightweight with regex filtering, developed by a coworker of my supervisor, `std::iostreams`) and spdlog[3](`fmtlib`, no category filtering).

[1] `https://github.com/fmtlib/fmt`
[2] `https://github.com/fvacek/necrolog`
[3] `https://cpp.libhunt.com/spdlog-alternatives`

Chapter 6
Packaging and Documentation

This chapter is related to task four of the assignment. It starts with an analysis of the QtMips solution and the current package management situation on major operating systems. Given that we are releasing QtRVSim as a completely new project, package configuration files had to be modified and building services set up. I have simplified the release process and made it platform independent and with a single source of truth for shared information. Based on my analysis and my current knowledge, I have introduced two new packages. Finally, to build on older distribution, some parts of the code and CMake scripts needed modifications.

Regarding documentation, I tried to document everything in place of usage with code comments. For more complex operations, there are new documentation pages in the *docs* directory. I have also updated the `README.md`; however, my information on simulation itself is limited due to the delay of my colleague working on the core.

6.1 QtMips

QtMips is built, and its binary packages are distributed using openSUSE Build Service (OBS)[1] for the following Linux distributions: Debian, Fedora, Raspberian, SUSE Linux Enterprise, and openSUSE. For Ubuntu Launchpad[2] is used. OBS also builds distribution-independent AppImage. Binaries for Windows, macOS, and WASM are built manually and uploaded to GitHub releases.

For Debian, QtMips uses the so-called `native` format. It requires the *debian* directory to be in the project root and source archives. The distribution maintainers disfavor native package use for software not directly related to the particular distribution. It is preferred to store package files out of project root, and it is considered during the quality review. Solving this problem would make it easier to get the package to official repositories[3].

6.2 Linux Distributions Coverage Analysis

According to Distrowatch[12], the top ten Linux distributions are based on these independent[4] distributions: Debian, Arch, Fedora, and openSUSE. From these distributions, QtMips omits only Arch Linux.

[1] `https://build.opensuse.org/`
[2] `https://launchpad.net/`
[3] An information from my supervisor.
[4] Not derived from another distribution. Packages for the independent distributions will usually work on the derived ones as well.

Another group of package managers, which are currently gaining popularity, is that, independent of particular distributions. They mainly aim at isolation, reproducibility, and security. The most known of those are Flatpack[1], Snap[2], Nix[3] [4] and Guix [5] [6].

Package management on Windows and macOS is currently neglected. Homebrew[7] is the primary package manager on macOS. On Windows, official software distribution channels are Microsoft Store and WinGet. To support Microsoft Store, it would be necessary to compile QtRVSim as a Universal Windows Platform (UWP)[8] application. That is generally not a problem as Qt officially supports UWP. The only problem here is to fix minor incompatibilities with the MVSC compiler. From unofficial package managers, the most used is Chocolatey[9]. Chocolatey is recognized by Microsoft, and it is even the official way to install software on Windows runners of GitHub Actions[10] which do not yet support WinGet[11].

Given my personal experience with Arch Linux and Nix, I have decided to add those packages as part of this task. In future work, I intend to add support for more of the mentioned package managers. Table B.1 in appendix suggests that QtMips has many Windows users. Therefore, better Windows support might be appreciated.

In the appendix, there are tables of download statistics from GitHub and Launchpad (B.1, B.2). Unfortunately, it is not possible to obtain statistics from OBS, which would provide a broader image of the Linux situation.

6.2.1 NIX Package

Nix package manager allows users to install software on any Linux distribution, regardless of its package and dynamic library management. This is a great advantage for the project as we can provide a package of which we can be sure that it will work on any Linux distribution, no matter how minor it is.

To achieve this, the Nix package manager installs and manages **all** dependencies (including the required version of the libc, Qt, etc.) in addition to those managed by the system package manager. The obvious disadvantage of wasting disk space is negligible given the current capacities of hard disks and their prices. Also, thanks to Nix design, all of these files are isolated in specialized directories and do not pollute the system.

6.3 Implementation

All package-related files have been moved from the project root to `extras/packaging`. They are stored in the form of CMake templates[12]. In the configuration phase (the initial call to CMake), up-to-date information is injected into placeholders within

[1] `https://flatpak.org/`
[2] `https://appimage.org/`
[3] `https://nixos.wiki/wiki/Nix`
[4] Nix comes with Linux distribution NixOS, but given its isolative properties, it is well usable on other systems as well. I use it on Manjaro alongside the Pacman.
[5] `https://guix.gnu.org/en/`
[6] GNU package manager based on Nix
[7] `https://brew.sh/`
[8] `https://docs.microsoft.com/en-us/windows/uwp/`
[9] `http://chocolatey.com/`
[10] `https://docs.github.com/en/actions/using-github-hosted-runners/customizing-github-hosted-runners`
[11] `https://docs.microsoft.com/en-us/windows/package-manager/winget/`
[12] Text files with CMake variables like this: `@A_VARIABLE@`

the templates. After that, an `open_build_service_bundle` target makes a directory with all files necessary to build all supported packages. To distribute a new release, it is now sufficient to update the changelog and upload the files produced by the `open_build_service_bundle` target to OBS. This procedure can be performed on any operating system with CMake, Git, and XZ. The only limitation is that the file system where the files are produced must support file permission modification. Debian builds rely on correct file permission setting. Correct permissions are set automatically during the build whenever the file system supports it.

I have switched the Debian package to the *quilt*[11] format, which keeps the source archive intact and ships Debian-specific files in another tar.

After that, I have created Arch Linux and Nix packages. The Arch Linux package is now also built by openSUSE Build Service.

Binaries for Windows and macOS can easily be obtained from GitHub actions as they are built as part of the automatic testing CI procedure. At this point, automatic release publishing was not a priority, but it can be achieved by extending the CI script according to this example[1].

6.3.1 Fallbacks

To make the code compile against older versions of Qt, I had to devise several fallbacks in the polyfill library. For versions older than 5.10 (Ubuntu 18), `QStringView` is replaced by `QString`. `QStringLiteral` macro is necessary for literals to work with both versions. Prior to version 5.13, Qt objects cannot be stored in an STL container. However, I need to store references in the GUI controller. That is not possible in Qt containers, where all values need to be default-constructible. To overcome this issue, I implement `std::hash` for `QString` and `QStringView` manually for older Qt versions.

6.3.2 Distributions Excluded From Support

The distributions listed below (previously supported by QtMips) are no longer supported for not providing required versions of dependencies and build tools:

- Debian 9
- Ubuntu 16 (used minimally, zero downloads of the 2020 version, see B.2)
- openSUSE Leap 15.1
- default OBS AppImage (replaced by AppImage based on openSUSE Leap 15.2)

[1] `https://cristianadam.eu/20191222/using-github-actions-with-c-plus-plus-and-cmake/`

Chapter 7
Conclusion

QtRVSim, a graphical simulator of the RISC-V computer architecture, was released as a result of this thesis and the thesis of Max Hollmann, who worked on the CPU core simulation. The simulator is based on QtMips, MIPS CPU simulator, created by Ing. Karel Kočí in his master's thesis[1] and extended by Ing. Pavel Píša. The creation of QtRVSim is the crucial step to switch teaching computer architecture lectures to the RISC-V ISA. QtRVSim provides users with the same user interface, providing the same level of introspection as QtMips but with improved internals and simulation options on the RISC-V CPU. Unlike QtMips, QtRVSim can simulate both, little- and big-endian CPUs, and internally it is ready for 64bit CPUs. The memory model supports variable size, unaligned accesses, which will be needed for compressed instruction extension.

I have extended the *svgscene* library, which builds Qt scene from an SVG file, and used it to visualize the CPU core. More complex diagrams can now be created easily without extra effort. Except for moving to RISC-V, this project did not aim to extend the simulator capabilities in a way observable by the user but to improve the quality and capabilities of its internal components and the project's technical quality in general.

The development of the simulator will continue to provide even deeper insight and simulate more complex CPU functions. There are many options for extension: branch predictors, memory management unit, multicore simulation with cache coherence protocol visualization, etc. I believe that my work will prove valuable to those further extending the QtRVSim.

Appendix A
Source Code

The **QtMips** and **QtRvSim** projects are developed as open-source, and therefore the most up-to-date version of the source code is to be found publicly available on GitHub.com.

A.1 QtRvSim (CTU official)

The new official repository for the RISV-V edition of the simulator.

```
https://github.com/cvut/QtRvSim
```

A.2 QtMips (CTU official)

The original repository of the MIPS version.

```
https://github.com/cvut/QtMips
```

A.3 Development Repository

A fork containing all immediate work.

```
https://github.com/hollmmax/QtMips/
```

Appendix B
QtMips Download Statistics

release type	v0.7.5	v0.7.3	earlier version	TOTAL
AppImage	132	42	0	175
Linux x86_64 ZIP	125	189	173	487
macOS app	39	40	15	84
Win ZIP	282	180	235	697

Table B.1. Release download statistics on GitHub. [2019-2020]

release type	v0.7.5	v0.7.3	earlier version	TOTAL
Ubuntu 21.04	1	0	0	1
Ubuntu 20.10	14	0	0	14
Ubuntu 20.04	113	29	0	142
Ubuntu 19.10	0	33	0	33
Ubuntu 19.04	0	16	9	25
Ubuntu 18.10	0	0	21	21
Ubuntu 18.04	118	177	51	346
Ubuntu 16.04	0	14	13	27

Table B.2. Release download statistics on Launchpad (Ubuntu). [2019-2020].

Appendix C
Glossary

- ALU — arithmetic logic unit
- API — application programming interface
- ASIC — an application-specific integrated circuit
- C — the C programming language
- C++ — the C++ programming language
- CI — continuous integration
- CPU — central processing unit
- CSR — control and status register
- CSS — cascading style sheet
- CTU — Czech Technical University
- endian — adjective from endian
- endianness — order of bytes of integer value in memory
- FEE — Faculty of Electrical Engineering
- FIT — Faculty of Information Technology
- GNU — the GNU Project`http://www.gnu.org`
- GUI — graphical user interface
- IDE — integrated development environment
- IEEE — Institute of Electrical and Electronics Engineers
- IEEE 754-2008 — IEEE floating-point standard
- ISA — instruction set architecture
- JIT — just-in-time compilation
- libc — the C standard library
- MVSC — Microsoft Visual C++
- OBS — the Open Build Service (formerly called openSUSE Build Service)
- POSIX — the Portable Operating System Interface
- Qt — a widget toolkit for creating graphical user interfaces
- Qt4 — Version 4 of Qt
- Qt5 — Version 5 of Qt
- STL — the C++ Standard Template Library
- SVG — Scalable Vector Graphics
- TLB — translation lookaside buffer
- UWP — Universal Windows Platform `https://docs.microsoft.com/en-us/windows/uwp/`
- WASM — WebAssembly
- XLEN — for 64bit system XLEN in 64
- XML — Extensible Markup Language
- XZ — a LZMA compression algorithm

References

[1] Karel Kočí. *Graphical CPU Simulator with Cache Visualization*. Master's Thesis, CTU Prague. 2018.

[2] Andrew Waterman, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. 2017. `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`. (Accessed on 2021-02-28).

[3] David A. Patterson, John L. Hennessy, and Alexander Perry. *Computer organization and design: the hardware/software interface*. 5th edition. Amsterdam: Morgan Kaufmann, 2014. ISBN 9780124077263.

[4] Jörg Bornemann. *Qt and CMake: The Past, the Present and the Future*. 2021. `https://www.qt.io/blog/qt-and-cmake-the-past-the-present-and-the-future`. (Accessed on 2021-03-29).

[5] David A. Patterson, and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Elsevier Science, 2017. ISBN 9780128122761.

[6] David A. Patterson, and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Elsevier Science, 2020. ISBN 9780128203316.

[7] Krste Asanovic. *RISC-V Summit 2020 The Next Ten Years*. 2021. `https://www.youtube.com/watch?v=lg33UqZ_en0`. (Accessed on 2021-04-02).

[8] *History - RISC-V International*. 2019. `https://riscv.org/about/history/`. (Accessed on 2021-04-19).

[9] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Techinal report. EECS Department, University of California, Berkeley.

[10] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. *Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline out-of-Order 64-Bit High Performance RISC-V Processor with Vector Extension.* In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE Press, 2020. 52–64. ISBN 9781728146614.

[11] Josip Rodin, and Osamu Aoki. *Debian New Maintainers' Guide*. 2020. `https://www.debian.org/doc/manuals/maint-guide/`. (Accessed on 2021-05-2).

[12] *News and feature lists of Linux and BSD distributions*. `https://distrowatch.com/dwres.php?resource=popularity`. (Accessed on 2021-05-08).

[13] Patrick Dengler, Chris Lilley, Jonathan Watt, Dean Jackson, Anthony Grasso, Doug Schepers, Cameron McCormack, Jon Ferraiolo, Erik Dahlström, and Jun

Fujisawa. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Techinal report. W3C.

[14] MDN Contributors. *SVG: Scalable Vector Graphics*. 2021. `https://developer.mozilla.org/en-US/docs/Web/SVG`. (Accessed on 2021-04-16).

[15] Philippe Le Hégaret, Lauren Wood, Arnaud Le Hors, Gavin Nicol, Mike Champion, Jonathan Robie, and Steven B Byrne. *Document Object Model (DOM) Level 3 Core Specification*. Techinal report. W3C.

[16] Erik Dahlström, Andreas Neumann, Vincent Hardy, Dean Jackson, Antoine Quint, Chris Lilley, Ola Andersson, Scott Hayman, Andrew Shellshear, Nandini Ramani, Andrew Emmons, Cameron McCormack, Doug Schepers, Anthony Grasso, Robin Berjon, Craig Northway, and Jon Ferraiolo. *Scalable Vector Graphics (SVG) Tiny 1.2 Specification*. Techinal report. W3C.

[17] News and Announcements. *IEEE 7th World Forum on Internet of Things*. 2021,

[18] Zhiguo Shi, Jingxiong Liang, Jun Pan, and Jiming Chen. How IoT and Blockchain Protect Direct-Drinking Water in Schools. *IEEE Internet of Things Magazine*. 2019, 2 (4), 2-4. DOI 10.1109/MIOT.2019.8982735.

[19] Andrei Frumusanu. *The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test*. 2020. `https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/`. (Accessed on 2021-05-03).

[20] *C++ exceptions support*. `https://emscripten.org/docs/porting/exceptions.html`. (Accessed on 2021-05-17).

[21] *GitHub Actions*. `https://docs.github.com/en/actions/`. (Accessed on 2021-05-17).

[22] Tedhudek. *Windows Documentation*. `https://docs.microsoft.com/en-us/windows/`. (Accessed on 2021-05-18).

[23] John L. Hennessy, and David A. Patterson. A New Golden Age for Computer Architecture. *Commun. ACM*. 2019, 62 (2), 48–60. DOI 10.1145/3282307.