CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Graphical RISC-V Architecture Simulator

Instructions Decode and Execution and OS Emulation

Max Hollmann
dev@maxholl.com

August 2021
https://github.com/cvut/qtrvsim
Supervisor: Ing. Pavel Píša PhD.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hollmann Max** Personal ID number: **483587**

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 - Instructions Decode and Execution and OS Emulation

Bachelor’s thesis title in Czech:

Grafický simulátor architektury RISC-V - dekodér, zpracování instrukcí a emulace systému

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 architecture 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. Design and implement solution (utility program) to transform instruction set description in QEMU or other source into tables required to decode RISC-V instructions.
3. Reimplement pipeline stages, ALU and other core components to match RISC-V architecture and referenced textbooks.
4. Implement basic subset of machine control registers required for system calls and exceptions processing.
5. Implement subset of Linux kernel system calls emulation.

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: **13.08.2021**

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 his continuous mentorship, endless information about computer architecture and sparking my interest in instruction sets in the first place.

I would then like to thank my colleague Jakub Dupák who worked on different parts of this simulator for discussing various design decisions with me and helping me with advanced `git` operations.

I would also like to acknowledge our entire study group, Jakub Dupák, Vojtěch Štěpančík, Matěj Kafka, and Jáchym Herynek, for bringing interesting information from various computer science fields to my attention and for endless technical discussions.

And lastly I would like to thank Sára Krinerová and other members of my family for their encouragement and loving support without which I would have likely given up somewhere along the way.

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 13.8.2021

.....

Abstrakt / Abstract

Instruční sada MIPS se na fakultě elektrotechnické ČVUT k výuce předmětů spojených s architekturou počítačů používá již řadu let. Jedná se o jednoduchou instrukční sadu a na jejím vývoji se podílel jeden z autorů populární učebnice architektury počítačů. Některé aspekty architektury MIPS se ale ukázaly být neefektivní a instrukční sada se potýká s licenčními problémy. Autoři výše jmenované učebnice spolu se svými studenty vyvinuli novou instrukční sadu, RISC-V, která byla navržena k výuce, aby byla jednoduchá k pochopení ale také jednoduchá na implementaci v hardware. Tato práce je součástí snahy přesunout výuku z MIPS na RISC-V. Tato práce se zaměřuje na vykonávání a překlad instrukcí a simulaci systémových volání.

Klíčová slova: RISC-V, architektura počítačů, CPU simulátor, dekódování instrukcí, emulace OS, QtRvSim, QtMips

The MIPS ISA is being used at the Faculty of Electrical Engineering for most computer architecture courses. It is simple in design and it was co-developed by one of the authors of a popular computer architecture textbook. However, some design decisions in MIPS have proven inefficient and the ISA is encumbered in licensing problems. The authors of the textbook along with their students developed a new architecture, RISC-V. It was designed for teaching purposes, making it simple to understand in theory, but also simple to implement in hardware. This thesis is part of the effort to switch our courses from MIPS to RISC-V, updating the QtMips simulator to RISC-V. It focuses on core execution, the internal assembler/disassembler and system call simulation.

Keywords: RISC-V, computer architecture, CPU simulator, instruction decode, OS emulation, QtRvSim, QtMips

/ Contents

Tables / Figures

Chapter 1
Introduction

Computer chips are slowly taking over all industry sectors. What used to be a carburetor and a distributor is now an electronically controlled fuel injection system and ignition coils. Many mechanical state machines are getting replaced with sensors and computers. Most washing machines and dishwashers sold today are controlled by small computers instead of mechanical systems. With the recent popularity of Internet of Things devices, this trend is moving onto ovens, refrigerators and more. Computerized appliances are often cheaper to manufacture yet provide more advanced features to their users. Cheap electronics also enabled new products which were not possible without them. Medical devices such as pacemakers and insulin pumps are saving lives daily, yet none would be possible without computer chips. However, computers are not magic and to properly utilise them, it is important to understand their inner workings and limitations. For this reason, the computer architectures course is part of informatics and robotics study programs at the Faculty of Electrical Engineering CTU. To teach the course, the MIPS architecture has been chosen in part due to its simplicity and its common use as a teaching tool on other universities across the world. The selected textbook[4] comes with a graphical simulator, MipsIt [1]. However, the simulator is old and closed source. With the latest supported OS being Windows XP and no option to update it, it was decided to choose a different simulator. Unfortunately no suitable replacement with cache and pipeline visualization was found. Therefore Ing. Karel Kočí developed a graphical MIPS simulator as his master's thesis project, QtMips[1].

QtMips was first released in May 2018 but the computing landscape has changed significantly since. RISC-V has been gaining traction both in embedded applications as well as a teaching tool. Meanwhile the MIPS architecture has been officially abandoned in March 2021[14].

In 2014, the authors of RISC-V stated that *"While the first RISC-V beachhead may be IoTs or perhaps WSCs, our goal is grander: just as Linux has become the standard OS for most computing devices, we envision RISC-V becoming the standard ISA for all computing devices."* [16] Today we can observe that their vision is slowly coming to fruition:

- The Internet of things and embedded development may not require a lot of computing performance, but these use cases require a large amount of simple and cheap processor. A free and open specification allows for companies to compete with various implementations. This is in stark contrast with the current status quo where complete core licenses are cheaper than ISA architecture licenses which allow the licensee to design their own implementations.
- Domain-specific chips also form a large and growing market. RISC-V is well suited for this purpose. The modular design of RISC-V does not force implementations to include unnecessary instructions and the ISA reserves encoding space for vendor-specific instructions. This property of the ISA has opened a new market for RISC-

[1] `https://www.eit.lth.se/fileadmin/eit/courses/eit090/MipsIt/MipsITEnvRef.html`

V-based ASIC design and there are at least two companies which offer application-specific CPU design services in the Czech Republic alone, Codasip[1] and Espressif[2]. On the international scale Western Digital has openly embraced RISC-V[13] for use in controllers.

- RISC-V's modularity and extensibility opens up to new experimentation in high-performance computing. An interesting example is Esperanto Technologies' ET-SoC-1 A RISC-V-based AI accelerator with over 1000 cores using custom vector and matrix multiplication extensions. Art Swift, the CEO of Esperanto Technologies, stated that *"We expect to see 100 times better energy efficiency in terms of inferences per watt versus the incumbent solutions."*[15]

It was therefore decided that the computer architecture courses at the Faculty of Electrical Engineering should utilise RISC-V instead of MIPS going forward.

This thesis is part of an effort to switch the QtMips simulator to RISC-V. As stated in the formal assignment, my focus was on decoding and executing RISC-V instructions and emulating a subset of Linux system calls.

The thesis is structured to provide an overview of RISC-V followed by instruction encodings and their decoding. The internal assembler is discussed in the same chapter as instructions. The next chapter focuses on simulating RISC-V instruction execution, including the emulated system calls. The final chapter present other existing RISC-V simulator and their possible use in the computer architecture course.

[1] `https://codasip.com/`
[2] `https://www.espressif.com/`

Chapter 2
RISC-V Instruction Set

"Prof. Krste Asanovic and graduate students Yunsup Lee and Andrew Waterman started the RISC-V instruction set in May 2010 as part of the Parallel Computing Laboratory (Par Lab) at UC Berkeley, of which Prof. David Patterson was Director"[8] RISC-V was designed to be a modular instruction set first and foremost. The base instruction set consists of only 49 instructions with the 64-bit version adding 14 instructions for working with the longer values. The feature set of the base instructions is rather limited and does not include many functions considered necessary in modern CPUs. The RISC-V specification provides more advanced features in its varius extensions.

- M: Integer Multiplication And Division
- A: Atomic Instructions
- F: Single-Precision IEEE 754-2008 Floating-Point
- D: Double-Precision IEEE 754-2008 Floating-Point

The RISC-V specification also includes definitions which are not strictly instructions.

- RVWMO (RISC-V WMO): Memory consistency model. Specifies rules for reordering memory writes and reads to global memory. Does not include any instructions.
- Zicsr: Reads and writes to CPU's Control and Status Registers.
- Zifencei: Instruction-Fetch Fence
- G: Not an actual extension. This is a shorthand for the combination of the above mentioned extensions. It is intended to represent a standard general-purpose ISA.

The RISC-V specification also includes additional extensions, many of which are not ratified as of writing.

- C: Compressed Instructions represent some instructions from G using only 16 bits for their encoding.[17]
- B: Bit manipulation.
- J: Features to help JIT languages.
- V: Vector operations. Unlike SIMD allows for more variation in implementations.
- Ztso: Similar to RVWMO, but with different memory ordering semantics. Intended to easy porting software from TSO ISAs.
- E: Identical to the I base set except for register count. Intended for embedded applications. Has 16 registers instead of 32.

These extensions are out of scope of the computer architecture course and are therefore not implemented in this thesis.

2.1 Data Sizes and Memory

The RISC-V specification defines 5 data sizes[3, pg. 6]:

- *byte* is 8 bits

- *halfword* is 16 bits (2 bytes)
- *word* is 32 bits (4 bytes)
- *doubleword* is 64 bits (8 bytes)
- *quadword* is 128 bits (16 bytes)

Different versions of RISC-V have different native sizes. RV32 uses *word*, RV64 uses *doubleword* and RV128 uses *quadword*. Each version can also operate on smaller sizes than its respective native size.

2.1.1 Endianness

"RISC-V base ISAs have either little-endian or big-endian memory systems, with the privileged architecture further defining bi-endian operation"[3, pg. 9]. Little-endian systems store bytes in multi-byte sizes from least significant to most significant. Big-endian systems store bytes in multi-byte sizes from most significant to least significant. Instructions however are stored as a sequence of 16-bit little-endian parcels. This was done because instrucion lenght is encoded in the least significant bits of an instruction. This allows implementations to only observe the first few bits of the parcel to determine the lenght of the instruction.

The simulator developed as part of this thesis uses little-endian memory operations.

2.2 Registers

RISC-V has 31 general purpose registers and special register 0, which is hard-wired to the value 0. Any reads from register 0 will always return 0 and all writes to it must perform all side-effects, but the register's value remains 0. On RV32E there are only 16 registers 2.1. The size of these registers coresponds to the processor's native size 2.1.1. There is also a special register for the program counter. This register cannot be modified directly with regular instructions, but can be read from using AUIPC and modified using several jump and branch instructions. The program counter is also automatically incremented after each executed instruction by the instruction's lenght. Unlike most architectures, RISC-V's basic arithmetic operations do not utilize special-purpose registers.

- FLAGS on x86 for conditional instructions
- HI and LO on MIPS for multiplication and division
- switching zero and stack pointer registers on AArch64

2.3 Instruction Encoding

Most instructions in the RISC-V base set use one of 6 instruction formats. These are R, I, S, B, U and J2.1, but some instructions (FENCE, ECALL, EBREAK) use special single-purpose encodings.

"The RISC-V ISA keeps the source (rs1 and rs2), and destination (rd) registers at the same position in all formats to simplify decoding."[3, pg. 15] The RISC-V instruction encoding scheme has been optimised to minimize hardware logic required to implement its decoding.

The only field included in all instructions is *opcode*. It is encoded in the lowest 7 bits of the instruction 2.2. However, only 5 bits of *opcode* can be used. The lowest 2 bits are always `11` with other values reserved for 16-bit instructions.

7	5	5	3	5	7	bits
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20,10:1,11,19:12]				rd	opcode	J-type

Table 2.1. RISC-V instruction encodings. Copied from [3, pg. 130]

RISC-V instructions allow for up to two source and one destination register. As is tradition among RISC instruction sets, arithmetic operations operate on registers only. Memory operations are reserved for the *load* and *store* instructions. 2.7 All operands are optional. To minimize multiplexers required for register number decoding, both sources and destination registers are always encoded using the same bits. If an instruction does not use a register argument, the bits are reused to encode the immediate value.

funct3 and *funct7* blocks provide additional information to further specify some instructions. *funct3* is used to choose among related instructions. Implementations do not have to actually decode this part of the instruction but can instead pass it directly to the next stage.2.4, 2.5, 2.7, 3.3

funct7 is present only in R instructions which do not use an immediate value. It is mainly used to allow for encoding of more register-register instructions, but it is mostly unused in the base instruction set. Its only usecase in the base set is separating logical from artihemic right-shifts and addition from subtraction. Extensions can use the encoding space generated by *funct7* to encode new instructions without requiring new *opcode*. For example, the M extension uses *funct7* to distinguish itself from base arithmetic operations.

Immediate values on RISC-V are encoded in unused fields of the respective instruction formats. This however means that each instruction format has the immediate value encoded using different bits. To simplify immediate value decoding, the bits have been placed to maximise bit reuse and the most significant bit is always encoded on bit 31 to simplify sign extenstion. All immediate values are sign-extended.

To avoid executing uninitialized memory in case of a software error, the instructions encoded as all 0 and all 1 are reserved to be explicitly invalid.

2.4 ALU operations

RISC-V has 2 types of ALU instructions, register-register or register and immediate value. The register-register arithmetic instructions use the R-type encoding and read their input values from registers addressed by *rs1* and *rs2*. The register and immediate arithmetic instructions are encoded using the I-type encoding and do not utilise *rs2*. Instead, they use the immediate value encoded in the instruction. Both instruction types write their result to the register provided in *rd*. An interesting feature of RISC-V is that these operations are encoded using only 2 opcodes. One for the R-type and one for the I-type. The specific operation is encoded in the *funct3* field and is same for both instruction types. Adding an immediate value to a register is encoded using the same value in the *funct3* block as adding two registers. This detail allows implementations to not decode *funct3* but instead pass it to the ALU unit directly.

2.4.1 Arithmetic

RISC-V provides a total of 3 arithmetic operations: `ADD`, `SLT` and `SLTU`. `ADD` performs a binary addition of its arguments. If an overflow occurs, it is ignored. The R-type variation of this operation uses an additional bit in *funct7*, if set, two's complement of the argument in *rs2* is used instead, resulting in a `SUB` instruction. Because all immediate values are considered signed, this modification is not present in the I-type version of the instruction, where a negative value can be used directly.

`SLT` and `STLU` perform singed and unsigned comparison respectively. If the first operand is less than the second operand, the result is 1, othwerwise the result is 0.

2.4.2 Logical

A set of logical operations is included. These consist of `AND`, `OR` and `XOR`, performing conjunction, disjunction and exclusive disjunction respectively. The unary operation `NOT` is substituted using `XOR` with an immediate argument of -1.

All operations are performed bitwise. The N-th bit in the result is equal to applying the logical operation to the N-th bit of input 1 and N-th bit of input 2.

2.4.3 Shift

Bit shifts are provided in 3 forms: `SLL`, `SRL` and `SRA`. `SLL` performs a left-shift, filling in the empty bits with 0. `SRL` performas a right-shift, filling in the empty bits with 0. `SRA` performs a right-shift, but fills in the empty bits with the source values sign bit.

The immediate versions of shifts are encoded as a specialization of the I-type. The shift amount is encoded in the lower 5 bits (6 bits for 64-bit operands) of the immediate field. Bit 30 is used to encode the type of right shift. This is the same bit used in `ADD` to create `SUB`.

2.5 Branch and Jump Instructions

Branch and jump instructions are the only instructions in RISC-V for manipulating the program counter. Branch instructions perform a conditional jump and jump instructions perform a jump unconditionally.

Unlike MIPS or SPARC, RISC-V does not have architecturally visible delay slots.

2.5.1 Branches

Branch instructions are the only instructions which use 2 source registers and an immediate value. They are encoded using the B-type encoding. All branch instructions use the same *opcode*.

A branch instruction performs a comparison of the values stored in *rs1* and *rs2*. If the comparison is true, the value encoded in *immediate* is added to the current program counter and used instead.

The comparison is performed by subtracting the value in *rs2* from *rs1*. The *funct3* bits encode whether the zero bit, signed oveflow bit or unsigned overflow bit should be used and if the result should be negated. This approach generates the set of branching instructions: `BEQ`, `BNE`, `BLT`, `BGE`, `BLTU` and `BGEU`. The missing conditions such as *branch if less or equal* and *branch if greater* are supported using pseudo-instructions and are encoded by swapping *rs1* and *rs2* and comparing using `BGE` and `BLT` respectively..

2.5.2 Jumps

Jump instructions come in two forms: `JAL` and `JALR`.

`JAL` adds the immediate value to the current program counter and jumps to it. Compared to branch instructions with an always true condition, `JAL` uses the J-type encoding a provides a larger range for the destination of the jump. `JAL` is the only instruction using J-type encoding.

`JALR` is the only control transfer instruction which takes a register argument. It adds the offset encoded in the immediate to the address located in *rs1* and jumps to the result. In combination with `LUI` 2.6, `JALR` can be used to jump to any 32-bit address.

Both jump instructions write the address of the following instruction to *rd*. A later `JALR` can jump to the address saved by the previous jump and continue execution. This combination is used by the `CALL` and `RET` pseudoinstructions to implement function calls[3, pg. 140].

2.6 Upper Immediate

Arithmetic and memory instructions are limited to a 12-bit immediate 2.1. When a larger value is needed as a constant or an address of a static variable, building it from basic ALU operations would be inefficient. The `LUI` instruction has a 20-bit immediate field and loads its immediate into the upper bits of *rd*. The lower 12 bits are zeroed out. If a full 32-bit value is desired, an `ADDI` instruction can be used to fill in the remaining bits.

In modern position-independent code, variables are not stored at a fixed address but instead at a known offset. In such cases, the `AUIPC` instruction can be used. `AUIPC` is similar to `LUI` except that after unpacking its immediate value, the current program counter is added. The combination of `AUIPC` and `ADDI` can be used to construct any PC-relative address in signed 32-bit range[1].

2.7 Load and Store

All of the instructions listed above operate only on the general-purpose registers and the program counter. To operate on values in main memory, the value has to first be loaded into a general-purpose register. The modified value can then be stored back into the main memory. Both load and store instructions are provided in all sizes 2.1.1 supported by the specific variation of RISC-V. Load instructions for smaller-than-native sizes support extending the value both as a signed and unsigned integer. The full list of load and store instructions on RV32 is: `LB`, `LH`, `LW`, `LBU`, `LHU`, `SB`, `SH`, `SW`.

Both load and store instructions use 12-bit immediate offset from *rs1* for addressing. Store instructions read the value to store from *rs2* and load instructions write the result into *rd*. Therefore the immediate encoding is different for load and store instructions. I-type and S-type respectively. 2.1

funct3 is used to select the size of the memory operation. Values used to encode the operation's size are the same for load and store instructions.

[1] This covers the entire address space on RV32 and is enough for basically any usecase on RV64

Chapter 3
Instruction Encoding and Decoding

The main goal of this thesis was to switch the existing MIPS-specific parts of the simulator to support execution of RISC-V instructions instead. This chapter discusses rewriting instruction definitions and modifying the internal assembler.

3.1 Instruction Definitions

The QtMIPS simulator uses hierarchical indexed tables to internally represent the MIPS instructions. In order to decode an instruction, some bits of the instruction are masked out and used as an index into an array of child nodes of the current node. If the child node is a leaf node, the instruction is decoded and values for control signals stored in the node are used. Many instructions however require multiple layers of decoding. In such cases, the child node carries information on which bits should be used to determine the next node and an array of its descendants.

I have rewritten this data structure to match the RISC-V specification[3]. Due to different encodnigs of various instructions and a few architectural changes, some signals were removed and some new were added. Control signals for ALU and memory units use the same value as is encoded in *funct3* 2.1. I have also added a method for decoding immediate values from RISC-V instructions.

3.2 Assembler

One of the education-oriented features of QtMIPS is its embedded assembler. It is designed to be a simple assembler for students to start writing assembly without having to install an external assembler.

The assembler uses the same instruction database as the simulator core. However, an additional arguments field is used to describe the operands used by the instructions. The RISC-V operand descriptors were heavily inspired from `opcodes/riscv-opc.c` in GNU binutils[18].

```
  ('d', 'g', 0,         0x1f,     {{{5, 7}}, 0})
  ('s', 'g', 0,         0x1f,     {{{5, 15}}, 0})
  ('t', 'g', 0,         0x1f,     {{{5, 20}}, 0})
  ('j', 'n', -0x800,    0x7ff,    {{{12, 20}}, 0})
  ('>', 'n', 0,         0x1f,     {{{5, 20}}, 0})
  ('a', 'a', -0x80000, 0x7ffff, {{{10, 21}, {1, 20}, {8, 12}, {1, 31}}, 1})
  ('u', 'n', 0,         0xfffff000, {{{20, 12}}, 12})
  ('p', 'p', -0x800,    0x7ff,    {{{4, 8}, {6, 25}, {1, 7}, {1, 31}}, 1})
  ('o', 'o', -0x800,    0x7ff,    {{{12, 20}}, 0})
  ('q', 'o', -0x800,    0x7ff,    {{{5, 7}, {7, 25}}, 0})
```

Example. *Mappings of operands to fields in instruction.*

Mapping operands to instruction fields is done using the table above. The MIPS code for mapping operands assumed that each operand is mapped to a contiguous chunk of bits. On RISC-V, this is often not true for immediate values. To solve this, I have implemented a structure which encodes operands as *shift amount* and a list of pairs of *bit counts* and *offset*. The `BitArg` structure is the last field in the argument descriptions in the Example above. *shift amount* is first used to shift arguments which do not utilise the lowest bits of the value. The list of *bit counts* and *offsets* then describes at what *offset* the next *bit count* bits are located.

3.2.1 Examples

Several assembly examples are included with the simulator. The goal of these examples is to provide students with a starting point when writing their first assembly code. The first example[1] shows a simple combination of reading and writing data from/to memory. Two hello-world examples are alse provided. One example[2] writes "hello, world" to a simulated serial port and the other[3] to a terminal using a system call. The serial port example also includes a list of available peripherals and their locations in the memory address space and the system call example includes a list of supported system calls, their numbers and arguments. I have updated all examples to RISC-V assembly and all information provided in comments.

3.3 Automated table generation

The original plan for this project included generating the instruction tree automatically from an external source. However as the project progressed, it made more sense to fill in the tree manually instruction by instruction, because it provided valuable information about which control signals were no longer necessary and which had to be added.

Writing a tool to create the instruction listings automatically would be possible now that the necessary control signals are added, but would provide no additional value, as the instruction tree already exists.

[1] `simple-lw-sw-ia.S`
[2] `template.S`
[3] `template-os.S`

Chapter 4
Execution

Decoding RISC-V instructions into control signals is not enough to become a RISC-V CPU. The most significant difference between MIPS and RISC-V is in the absence of delay slots and the implications of their removal. Furthermore the behaviour of both the memory unit and ALU are slightly different on RISC-V and some operations can be done in different stages of the CPU than on MIPS.

The QtMIPS simulator provides two core variants. Both divide the execution of an instruction into 5 phases. The simple core executes the phases in order and simulates a single-cycle CPU. This CPU is used for demonstrating how a CPU executes instructions and what operations are performed in each phase. The pipelined core adds interstage buffers and executes the stages in reverse order, simulating a pipelined execution. It is used to demonstrate performance advantages of pipelined execution, but also the engineering challenges of resolving data dependencies. The phases in order of execution are:

- Fetch
- Decode
- Execute
- Memory
- Writeback

4.1 ALU, Memory and Interstage

ALU on RISC-V does not raise exceptions on overflow and therefore does not need to separate signed and unsigned operations. The ALU control signal is directly encoded in *funct3* 2.1. The ALU was updated to use control signal values from the respective instructions.

The memory unit also uses the value in *funct3* 2.1 directly and was updated accordingly. More importantly, the QtMIPS simulatorr used a big-endinan architecture, but little-endian is preferred on RISC-V. All changes required to support changing the endianness of memory are part of Jakub Dupák's thesis [2, Chapter 3].

Finally, all interstage buffers have been relocated to `machine/pipeline.h`. This change reduced the use of slow Qt signal slots and simplified extracting data from core for display in GUI [2, Chapter 4].

4.2 Branches

MIPS branch instructions were designed to be evaluated in the CPU's pipeline as early as possible. Only simple branching logic is provided: equality and inequality between registers and comparisons with zero. Furthermore, a branch delay slot is provided by the ISA. Evaluating the branch in the decode stage allows the implementation to

forward the branch target to the fetch stage. One instruction would be fetched between the branch and its target, filling the unconditional delay slot.

RISC-V branches allow for full comparisons between registers and do not require a delay slot 2.5. The chosen design for the RISC-V simulator uses the ALU to perform the comparison and writes the resulting program counter in the memory stage. Moving jumps from decode to memory stage does not cause any problems in the single-cycle core, because all phases have to be completed before the next instruction. To ease learning of MIPS assembly programming for students of the computer architecture course, it was possible to disable the delay slot. On the pipelined core, there are now 3 cycles after fetching a branch or jump instruction before the target is written to the program counter. I have decided to implement a solution similar to modern CPUs. When an instruction is fetched, a predictor guesses the address of the next instruction[1]. Once the memory stage is reached and the target is known, the target address is compared to the guessed address of the instruction immediately following in the pipeline. If the addresses differ, the pipeline is flushed before any of the instructions fetched due to an incorrect prediction can take effect.

4.2.1 Exceptions

The design of jumps and branches on MIPS has another implication: If an instruction is fetched, it will be executed. As a result, QtMIPS can stop execution and evaluate an exception before the exception reaches the memory stage. This is done whenever an invalid instruction is fetched. However, we cannot stop execution upon fetching an invalid instruction on RISC-V. The predictor mentioned in 4.2 can cause the CPU to fetch invalid instructions from a wrongly guessed address. These instructions will never take any effect, because they will be flushed once the mispredicted branch reaches memory stage and its true target is known.

To solve this issue, no stage before the memory stage can perform any side-effect. If an invalid instruction is fetched, the decoder has to pass the information onto the execute stage, which then passes the information to the memory stage. If an exception reaches the memory stage, it is acted upon. If the pipeline is flushed while an exception is getting passed through the pipeline, the exception is also flushed. The writeback stage cannot cause an exception and can be safely ignored.

4.3 System Calls

Another feature of QtMIPS is system call simulation. Simulating a subset of Linux system calls is used for demonstrating how a user programs interact with the kernel.

System calls are numbered differently on different platforms. However, not all system calls are supported on all platforms.

- `set_thread_area` is present only on MIPS, m68k and x86. It was removed from QtRvSim.
- `mmap2` was introduced to enable indexing into larger files on 32-bit systems, but is not necessary on 64-bit systems. RISC-V Linux supports `mmap` only.
- `open` was superseded by `openat`, which is functionally a superset of `open`. RISC-V supports only the newer `openat`. Only the `open` subset of `openat` is supported.

To map the system call implementations to their RISC-V numbers, I have modified the system call table from strace[19] to suit QtRvSim needs and format.

[1] The present predictor always adds 4 to the current program counter.

QtMIPS also supported SPIM[1] system services. These were implemented to support running examples from SPIM in QtMIPS. With the switch to RISC-V, these services are not useful anymore and were removed.

[1] `https://sourceforge.net/projects/spimsimulator/`

Chapter 5
Existing RISC-V Simulators

Before starting work on QtRvSim, existing RISC-V simulators were evaluated for their possible use in the computer architecture course. This chapter lists the most notable simulators and why they were not used.

5.1 Ripes

Ripes[1] is a visual computer architecture simulator. It provides great core visualization, cache simulation and more.

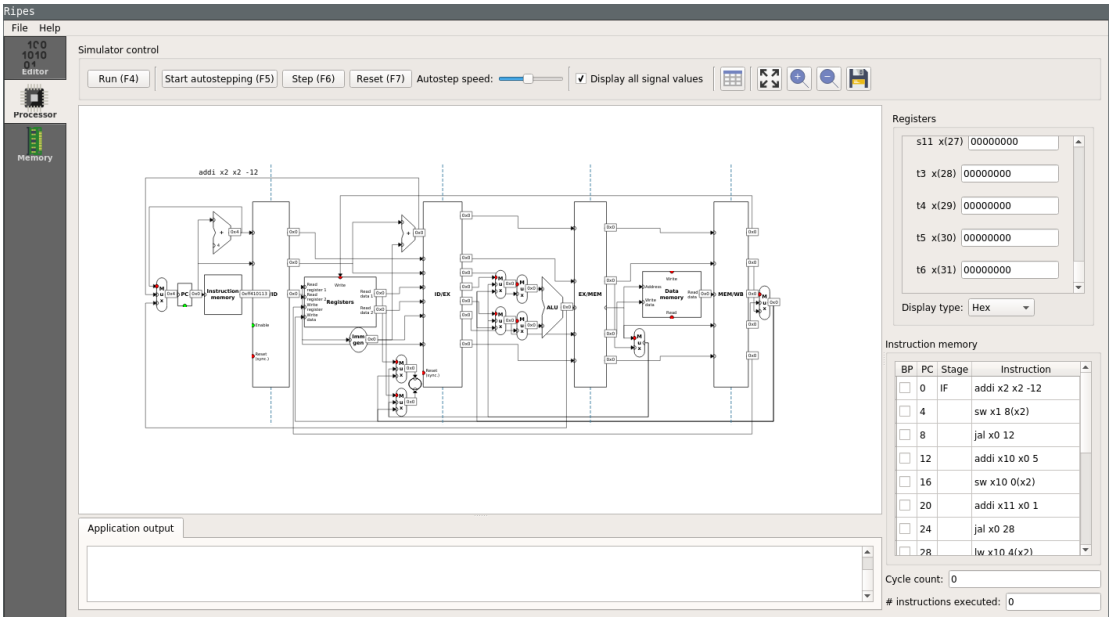

Figure 5.1. Ripes core view

Ripes supports almost all features required for the computer architecture course. Unfortunately Ripes does not support simulating system call execution and its internal design would have made it harder to add them to Ripes than switching QtMIPS to RISC-V. For this reason the decision was made to develop QtRvSim instead of modifying Ripes.

5.2 RARS

RARS[2] is a RISC-V continuation of MARS[3], a popular MIPS simulator. The goal of RARS is to assemble and simulate RISC-V programs. Unfortunately RARS does

[1] `https://github.com/mortbopet/Ripes`
[2] `https://github.com/TheThirdOne/rars`
[3] `http://courses.missouristate.edu/KenVollmar/MARS/`

not simulate the full execution pipeline and does not have a core view like Ripes and QtRvSim. RARS has a good assembly editor with instruction auto-completion, which would help computer architecture students when writing assembly.

5.3 Brisc-v

Brisc-v[1] is a RISC-V simulator developed by Adaptive & Secure Computing Systems Laboratory at Boston University. It is an online simulator. It includes a disassembler, register and memory view and instruction breakdown, which displays how instruction arguments are encoded 2.1. However, it does not simulate individual operations performed inside a CPU core. It also does not simulate system calls, cache or a serial port. Adding these features to Brisc-v would have been significantly more effort than switching QtMIPS to RISC-V.

5.4 Verilator

Unlike the other simulators listed, Verilator[2] is not a RISC-V simulator, but instead a Verilog simulator. Verilog is a hardware description language, which is beyond the scope of the computer architeture course.

Verilator is however well suited for use in the advanced computer architecture course where students design their own CPU cores.

5.5 Qemu

Qemu[3] is a generic machine emulator and virtualizer. Qemu can emulate a variety of target architectures on a variety of host architectures. However, Qemu's primary goal is speed rather than introspection. Qemu is a great tool for cross-developing to other platforms, but is not suitable for the computer architecture course because it does not simulate internal units inside of a CPU core.

[1] https://ascslab.org/research/briscv/simulator/simulator.html
[2] https://github.com/verilator/verilator
[3] https://www.qemu.org/

Chapter 6
Conclusion and Future Work

As a result of this thesis and the thesis of Jakub Dupák[2], QtRvSim is being released. QtRvSim is a RISC-V update to the QtMips MIPS simulator previously developed by Karel Kočí in his master's thesis[1] and maintained by Ing. Pavel Píša. QtRvSim supports the required features for use in courses and will be used in the upcoming computer architecture course on FEE CTU. However, I see multiple multiple options for improving on the work presented here in the future.

6.1 Predictor

The current predictor implementation 4.2 is very simple and the API does not provide necessary information for implementing a more advanced predictor. Future work could replace the current API and provide more advanced predictors. Switchable predictors in QtRvSim would improve the currently lacking tools for presenting branch predictors to students.

6.2 Pseudoinstructions

The RISC-V specification includes pseudoinstruction listings for assembly programmers [3, pg. 139]. Pseudoinstructions implement additional instructions by translating them into normal instructions with special arguments or a sequence of instructions. Currently the only supported pseudoinstruction is `nop`. Future work could investigate possible implementations of instruction rewriting to implement pseudoinstruction.

6.3 `QFile`

As my semestral project last semester, I rewrote file operations in QtMIPS to utilise Qt's file abstractions instead of assuming a POSIX-like host. This change has improved code readability and made memory-mapped files a possibility. Unfortunately, the Qt features required to use `QFile` are not supported on Ubuntu 18.04 LTS, the most popular Ubuntu version for QtMIPS [2, Appendix B].

When our users move to newer versions of Ubuntu which does support the required features, the `QFile` changes should be merged into the release version of the simulator.

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
Glossary

ALU	arithmetic logic unit
API	application programming interface
CPU	central processing unit
CTU	Czech Technical University
endian	adjective from endian
endianness	order of bytes of integer value in memory
FEE	Faculty of Electrical Engineering
GNU	the GNU Project`http://www.gnu.org`
GUI	graphical user interface
IEEE	Institute of Electrical and Electronics Engineers
IEEE 754-2008	IEEE floating-point standard
ISA	instruction set architecture
JIT	just-in-time compilation
LTS	long-term support
MIPS	microprocessor without interlocked pipeline stages, a RISC ISA developed by MIPS Technologies
m68k	Motorola 68000 family of microprocessors
OS	operating system
POSIX	the Portable Operating System Interface
Qt	a widget toolkit for creating graphical user interfaces
RISC	reduced instruction set computer
SPARC	scalable processor architecture, a RISC ISA developed by Sun Microsystems
TSO	total store order
WMO	weak memory order
x86	family of microprocessors based on Intel 8086. `x86` is usually used to refer to the 32-bit generation of the family

References

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

[2] Jakub Dupák. *Graphical RISC-V Architecture Simulator - Memory Model and Project Management*. 2021.

[3] 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).

[4] 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.

[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 2021-08-09).

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

[11] 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.

[12] 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.

[13] Anton Shilov. *Western Digital to Use RISC-V for Controllers, Processors, Purpose-Built Platforms*. 2017. `https://www.anandtech.com/show/12133/western-digital-to-develop-and-use-risc-v-for-controllers`. (Accessed on 2021-08-09).

[14] Jim Turley. *Wait, What? MIPS Becomes RISC-V*. 2021. `https://www.eejournal.com/article/wait-what-mips-becomes-risc-v/`. (Accessed on 2021-08-09).

[15] David Schor. *A Look At The ET-SoC-1, Esperanto's Massively Multi-Core RISC-V Approach To AI*. 2021.

https://fuse.wikichip.org/news/4911/a-look-at-the-et-soc-1-esperantos-massively-multi-core-risc-v-approach-to-ai/. (Accessed on 2021-08-09).

[16] Krste Asanovic, and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Techinal report. EECS Department, University of California, Berkeley.

[17] Andrew Waterman. *Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed*. Master's Thesis, EECS Department, University of California, Berkeley. 2011.
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-63.html.

[18] Free Software Foundation. *GNU binutils*. 2021.
https://www.gnu.org/software/binutils/. (Accessed on 2021-08-10).

[19] strace developers. *strace*. 2021.
https://strace.io. (Accessed on 2021-08-10).

[20] David Harris, Sarah Harris. *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.