BACHELOR THESIS

Jan Papesch

RISC-V support in MSIM

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Vojtěch Horký, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

I want to thank my supervisor Mgr. Vojtěch Horký, Ph.D., for his insightful comments and the guidance provided me with.

Title: RISC-V support in MSIM

Author: Jan Papesch

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Ph.D., Department of Distributed and Dependable Systems

Abstract: In this thesis we have added support for RISC-V into the MSIM emulator. MSIM supported only the MIPS R4000 as a CPU, and has been used for teaching the Operating systems course. This project redesigns MSIM to support multiple CPU architectures and adds an implementation of a RISC-V CPU. RISC-V as an architecture provides a basic instruction set as well as a wide variety of optional extensions. One chapter of this thesis is dedicated to giving an overview of the most important parts of the RISC-V architecture. The extensions that were thought to be useful or necessary for the Operating systems course have been implemented.

Keywords: RISC-V MSIM simulator instruction set

Contents

Introduction **3**

1 MSIM **5**
1.1 Goals . . . 5
1.2 Using MSIM . . . 5
1.3 MSIM architecture . . . 6
1.3.1 Shared memory space . . . 6
1.3.2 Devices . . . 6
1.3.3 MIPS CPU . . . 7

2 RISC-V **8**
2.1 RISC-V goals . . . 8
2.2 RISC-V terminology . . . 8
2.3 Unprivileged ISA . . . 9
2.3.1 RV32I . . . 9
2.3.2 RV64I . . . 10
2.3.3 Notable extensions . . . 10
2.4 Privileged ISA . . . 12
2.4.1 Control and Status Registers (CSRs) . . . 13
2.4.2 Machine-Level ISA . . . 13
2.4.3 Supervisor-Level ISA . . . 13
2.4.4 Exceptions, interrupts and traps . . . 14
2.4.5 Hypervisor extension . . . 15

3 Design Decisions **16**
3.1 Decoupling MIPS from MSIM . . . 16
3.1.1 General CPU . . . 16
3.1.2 Physical Memory (Physmem) . . . 17
3.1.3 Debug . . . 19
3.1.4 GDB support . . . 19
3.2 RISC-V support . . . 19
3.2.1 Architecture . . . 20
3.2.2 Extensions . . . 21

4 Implementation **23**
4.1 Notable features . . . 23
4.1.1 CSRs . . . 23
4.1.2 “A” Extension . . . 25
4.1.3 Memory model . . . 25
4.1.4 Custom instructions . . . 26
4.2 Deviation from specification . . . 26
4.2.1 EBREAK . . . 26
4.3 Testing . . . 26
4.3.1 Unit tests . . . 27
4.3.2 System tests . . . 27

4.4 Evaluation 29

Conclusion **30**

Bibliography **32**

A Attachments **34**

A.1 MSIM project 34

Introduction

MIPS is a reduced instruction set architecture, which is being used as the reference architecture in several courses at the Faculty of Mathematics and Physics of Charles University. One of these courses is the NSWI004 Operating Systems (OS) course. In the practical seminars of this course, students develop their own small kernels for MIPS and use the MSIM simulator to run and test their code.[1] MSIM is used, because it is a deterministic simulator, which eases debugging and allows the submitted kernels to be tested and graded based on the results of these tests. MSIM, first developed as part of Viliam Holub's Master Thesis [2], has been developed for this purpose and is being maintained on MFF.

In 2021 MIPS Technologies stopped designing the MIPS architecture. Instead, it has become a member of RISC-V International, the nonprofit group that coordinates the development of the RISC-V architecture, and has switched to developing the RISC-V architecture.[3] Meanwhile, RISC-V is actively being developed and is even being used by major companies, as shown by the articles on the RISC-V blog.[1] We think that when even the developers of MIPS have switched to RISC-V, current students should not be taught on MIPS, but on a more current architecture. In particular, we would like the OS course to switch to RISC-V. For this to happen, we would first need the tools used in this course to support RISC-V.

One option would be to switch from MSIM to an existing RISC-V simulator. According to the RISC-V Technical Wiki,[4] there are two most prominent RISC-V simulators, QEMU and Spike. The simulator rv8 is also mentioned in the specification of the RISC-V architecture,[5] which itself took inspiration from TinyEMU.

QEMU is a complex system that supports emulating whole systems, emulating user-level code on a different architecture, and virtualization.[6] This is a very powerful tool that has many use cases, but it is far too complex for the OS course.

Spike has, according to the wiki,[4] been the "proof-of-concept target for all RISC-V extensions", until it has been superseded by SAIL generated simulation. Spike emulates only the CPU and memory, no other devices, so even though it has rich debugging features, it would need to be highly modified to be used in the OS course. The fact that it simulates multiple processors in multiple threads also makes it non-deterministic.[7]

SAIL provides a formal specification of RISC-V and allows new extensions to be added. SAIL can then generate a simple emulator from this specification that can be linked with Spike in order to extend Spike by the defined extensions. SAIL provides a way to formally define new extensions and verify them using the formal definition and running custom tests on the generated emulators. These emulators would not be good teaching tools in our opinion.[8, 9]

[1]`https://riscv.org/news/risc-v-blog/`

rv8 is the closest fit from all the emulators listed here. It has support for more devices than MSIM, it has many tools for debugging, but similar to Spike, it simulates multiple processors in multiple threads, but when simulating whole systems, it allows only 1 CPU and simulates it in the main thread. rv8 also starts with the registers containing random values.[10] For these reasons, rv8 is also not a good fit for the OS course.

TinyEMU is the simplest of the emulators listed, as it supports only one processor, which it simulates deterministically. It also supports various devices that could be used in the OS course. There are two problems with this emulator, the code is written in a way that would be hard to extend, and there is no debugging support, which would be hard to add because of the first problem.[11]

MSIM has an extensible architecture that is made to support debugging, and most importantly is fully deterministic, even with multiprocessor systems. Compared to starting from scratch, MSIM provides a functional framework for device management, memory management, and a user interface. We think that it would be easier to extend MSIM than create a new emulator. Therefore, the main goal of this project is to add support for RISC-V to MSIM. We will design the system with regards to the needs of the OS course.

This thesis is divided into four chapters. The first two chapters focus on the technology of MSIM and RISC-V. The third chapter focuses on the design decisions made during the development process, while the fourth chapter describes some notable implementation details and tests.

1. MSIM

As said in the reference manual,[12, Introduction]

> MSIM is a light-weight computer simulator based on MIPS R4000. It is used for education and research purposes, mainly to teach the construction and implementation of operating systems.

The above clearly states the main purpose of MSIM and also shows us how deeply coupled is the MIPS CPU architecture with this simulator. Our main goal in this chapter will be to introduce MSIM from the point of view of a user, as well as to describe its architecture and inner workings. Most of the information in this chapter is taken from the reference manual [12] or is made by observation from the source code and its documentation.

1.1 Goals

MSIM is being developed with specific goals.[12] We will keep these goals in mind while refactoring and extending MSIM.

Determinism

MSIM aims to be a fully deterministic simulator. This allows the user to write tests for their software using MSIM as a runtime environment.

Simple device management

MSIM takes care to design devices in a straightforward manner and does not aim to make the devices realistic. This alleviates the problems of real hardware and allows the users to focus on the basic principles of device operation. All devices can be easily configured, mainly including the addresses of memory-mapped registers.

Speed

MSIM is not designed for speed of execution.

1.2 Using MSIM

MSIM is designed with a command line interface for device management, but also provides a script-like configuration file that can be used upon startup. The user can either let the system run on its own or run it in an interactive mode. When running in interactive mode, the user can make MSIM simulate only one step of the system, which is useful for debugging. The interactive mode allows the user to modify the environment of the system. More about the stepwise execution can be found in Subsection 1.3.2.

For a detailed overview of all the functionalities of MSIM, see the Reference Manual.[12]

1.3 MSIM architecture

MSIM works with a simplified model of a computer based on the von Neumann architecture.

1.3.1 Shared memory space

There are multiple devices interconnected through a shared memory space. The CPU is considered to be one of the devices and accesses the other devices by reading and writing to memory-mapped registers.

Sections of this memory space can be occupied by memory areas that function as main memory. These areas can be loaded with data from a specified file.

The whole memory space is managed by the MIPS CPU. This is one of the couplings, we will need to uncouple before we start implementing the support for RISC-V.

1.3.2 Devices

The MSIM devices work on a step-by-step basis. Each step means one clock tick for the CPU, but also for any other device. MSIM makes all devices take steps one at a time in a round-robin fashion, making the whole system deterministic, as is one of the goals of MSIM. Interactive commands are executed in-between each iteration through all of the devices. This main simulation loop is visualized in Figure 1.1.

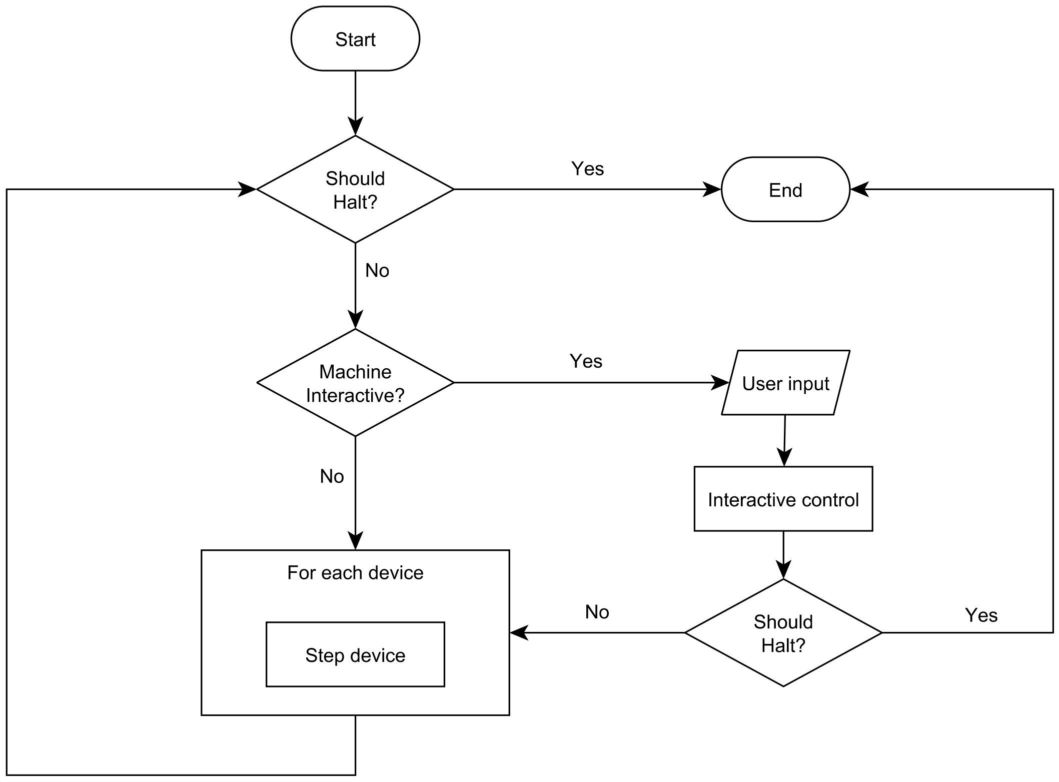


Figure 1.1: Flowchart of the main simulation loop

1.3.3 MIPS CPU

The CPU communicates with the devices by reading and writing to their memory-mapped registers. To simulate non-blocking IO, the devices can also raise interrupts.

The CPU simulates one instruction per step and in program order. The step includes the entire fetch-decode-execute cycle for this single instruction. Interrupt and exception handling is also included in this cycle. The whole process is illustrated in Figure 1.2.

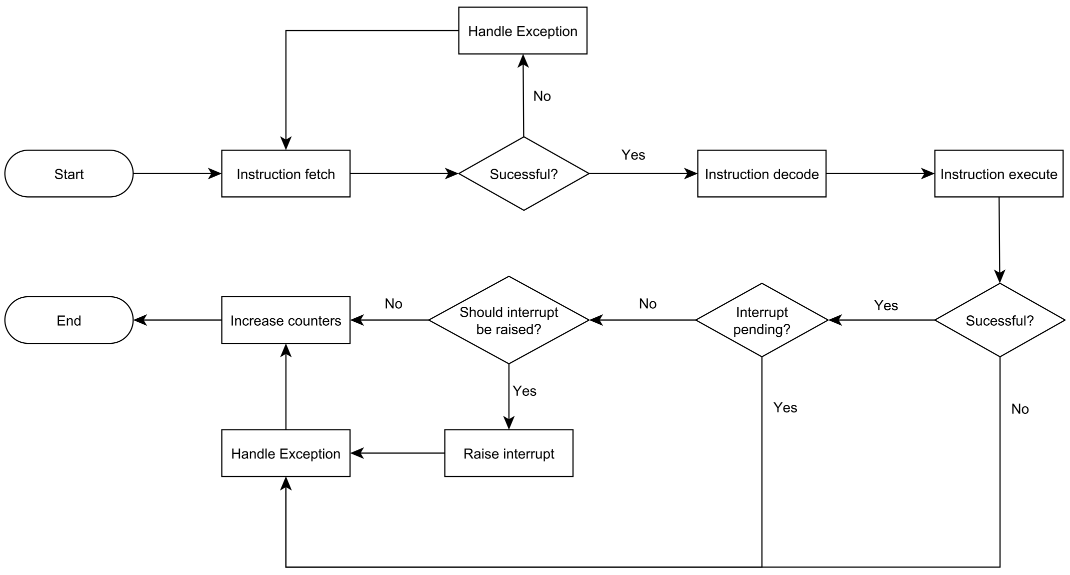


Figure 1.2: Flowchart of simulating one step of the MIPS CPU

If this process were implemented as is, the instructions would need to be decoded at every step. If we assume that the code we want to run does not change, this would be very inefficient. Because of this, MSIM employs its only performance optimization and caches pages of decoded instructions. More about how this affected the decoupling MIPS from MSIM and how we implement a similar mechanism for RISC-V can be found in Subsection 3.1.2.

2. RISC-V

This chapter describes the basics and provides an overview of the most important features of the RISC-V architecture. We do not focus on OS specific topics, but rather describe the architecture in its whole. We will describe some details that affected our implementation of RISC-V into MSIM in the later chapters.

2.1 RISC-V goals

RISC-V is an open standard instruction set architecture (ISA) [13] which was originally designed for computer architecture research and education.[5, Introduction] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA [5] describes the goals with which RISC-V is being designed. We would like to highlight some of these goals, which are particularly aligned with the goals of MSIM, stated in Section 1.1.

One of these goals has already been stated; RISC-V, as well as MSIM, is designed for education. RISC-V is also designed as a small base integer ISA, with support for many optional extensions. This allows us to choose only the functionality we need and nothing else. We will later describe which extensions RISC-V supports in Section 2.3.3 and which we have chosen to implement for MSIM in Section 3.2.2.

RISC-V is being designed to be suitable for hardware implementation, but is not being developed for a specific microarchitecture style[1] or technology.[2] This makes RISC-V very flexible and allows us to emulate it deterministically without adjusting the ISA too much.

2.2 RISC-V terminology

For consistency, we will use the same terminology as in the RISC-V manuals.[5, 14] This section serves as a glossary of the terms used in this text.

Execution Environment Interface The unprivileged manual [5] gives a precise definition of EEI (notes stripped).

> A RISC-V execution environment interface (EEI) defines the initial state of the program, the number and type of harts in the environment including the privilege modes supported by the harts, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on each hart, and the handling of any interrupts or exceptions raised during execution including environment calls.

For us, the execution environment will be MSIM together with its configuration.

[1]E.g. in-order, microcoded, out-of-order

[2]E.g. custom microcontroller, FPGA, software emulation

Hart Hart is an abstraction of a hardware thread. To software running inside an execution environment, the harts function as hardware threads, but they do not explicitly need to be hardware threads. One such case is MSIM, where multiple harts will be simulated by one software thread, which will most likely run on a single physical hardware thread. The execution environment must guarantee that every hart will eventually progress in its execution.[3]

Word Word is a unit of memory consisting of 32 bits. Doubleword consists of two words, and hence is 64 bits long. Halfword is 16 bits long.

2.3 Unprivileged ISA

The Unprivileged ISA defines the unprivileged instructions which are the instructions usable in all privilege modes and all privilege architectures. As stated in Section 2.1, the ISA is split into a base integer ISA and optional extensions. There are multiple base ISAs, that differ mainly by the number of registers, the width of the processor registers, denoted XLEN, and by the size of the address space. The size of the address space matches XLEN in all of the defined cases. The XLEN and the number of registers of the four defined base ISAs can be found in Table 2.1.

Base ISA	XLEN	#regs
RV32I	32	32
RV32E	32	16
RV64I	64	32
RV128I	128	32

Table 2.1: XLEN and number of registers of the RISC-V Base ISAs

RV32E is designed for small embedded microcontrollers and the only change to RV32I is the reduced number of registers. RV128I has been defined as a way to future-proof RISC-V if a need for 128 bit address spaces arises. We will pay closer attention to RV32I and RV64I, as these are defined for general-purpose use.

2.3.1 RV32I

RV32I uses 32 general purpose registers that are 32 bits long, denoted `x0-x31`. The register `x0` is hardwired to the value of 0 and all writes to this register are ignored. In addition to these registers, the ISA defines `pc` to hold the address of the current instruction.

There is no dedicated stack pointer or return address register, but the standard software calling conventions use registers `x2` and `x1` for these purposes, respectively. For a full description of the usage of all registers in the standard calling conventions, see Table 25.1 in the Unprivileged Manual [5, Chapter 25].

[3]Retire an instruction or trap. This mechanism is suspended while the hart waits for some event or is terminated, as stated in the manual.[5, Introduction]

Instructions

RV32I defines 40 instructions, which are split into these 5 categories

Integer Computational Register-Register or Register-Immediate operations. Do not cause arithmetic exceptions. Do not allow memory operands.

Control Transfer Unconditional jumps and conditional branches. Do not have a branch delay slot. Branch instruction compare two registers instead of using condition codes.[4]

Load and Store Memory accesses. Memory transfers are done only between memory and registers. Defined for byte, halfword, and word width. Loads of shorter widths than a word can be sign- or zero-extended to fill the whole target register.

Memory Ordering FENCE instruction. Used to order memory and I/O accesses by other harts or devices.

Environment Call and Breakpoints SYSTEM instructions. ECALL is used to make a service request to the execution environment. EBREAK is used to return control to a debugging environment.

For a complete list of instructions and their encoding, see Table 24.2 in the Unprivileged Manual.[5, Chapter 24]

2.3.2 RV64I

RV64I is based on RV32I. It widens the registers to 64 bits and uses 64 bit address space. Integer computational instructions now operate on 64 bits. Additional variants of these instructions are added that operate on the lower 32 bits and are indicated by 'W' suffix. 64-bit load and store instructions are added.

2.3.3 Notable extensions

In addition to the base ISAs, the Unprivileged Manual [5] defines several extensions that provide additional functionality or allow a more efficient execution of some operations. Some of the extensions are fully specified in the manual, while others are still being designed, and only placeholders for them can be found. We would like to briefly describe the fully specified extensions.

"C" Extension for Compressed Instructions

The "C" extension introduces short 16-bit encoding for the most common instructions. It can be added to any of the base ISAs, and the term RVC is used to signify any base ISA with the "C" extension added.

According to manual:[5, Chapter 16] "Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.".

[4]Condition codes are used for example in x86 and ARM

“M” Extension for Integer Multiplication and Division

The “M” extension adds instructions for multiplying and dividing. Multiplication of two XLEN bit registers produces a 2×XLEN-bit result. The higher XLEN bits are accessed by a different instruction than the lower XLEN bits.

Division by zero does not raise an exception, but rather produces a defined result that can be used for a branch if this should produce an error. The remainder operation works in a similar manner.

“A” Extension for Atomic Instructions

The “A” extension specifies instructions that atomically read-modify-write memory and defines the RISC-V Weak Memory Ordering (RVWMO) memory model. This allows multiple harts to run in a shared memory space and synchronize between themselves.

This extension defines two types of instructions: the load-reserved/store-conditional pair[5] and the fetch-and-op memory instructions.

Load-reserved (LR) loads a word or doubleword from memory and makes a reservation on some set of addresses. Store conditional (SC) tries to write to memory, but checks if the target is in the reserved set and if no writes to this set have taken place between the last LR and this SC.

The fetch-and-op instructions atomically load a value from memory, perform some operation on it, and then write the result back to memory.

Both types of atomic instructions support memory consistency orderings, including unordered, acquire, release, and sequentially consistent semantics.

“Zicsr” Control and Status Register (CSR) Instructions

The privileged manual [14, Section 2.1] defines a separate address space of 4096 Control and Status registers (CSRs) for each hart. These are mainly used by privileged software, but there are several unprivileged uses for CSRs including counters, timers, and floating-point status.

This extension defines six instructions used for operating on CSRs. These instructions atomically read-modify-write a single CSR, whose number is encoded into the instruction. They work either with an immediate value or a value stored in a register and support 3 write operations: full write and setting or clearing only specified bits.

Counters

RISC-V optionally provides a set of up to 32×64-bit performance counters and timers, regardless of the base ISA.

Three of these have dedicated functions, while the others can be configured to count platform-defined events. The three predefined counters are CYCLE, TIME, and INSTRET, counting clock cycles, real-time, and instructions retired, respectively.

[5]As opposed to Compare and Swap used by x86.

“F”, “D” and “Q” Extensions for Single, Double and Quad Precision Floating-Point

These extensions describe support for 32, 64 and 128 bit floating point numbers compliant with the IEEE 754-2008 standard.[15] They depend on the “‘Zicsr” extension. These extensions define 32 new floating-point registers and one CSR `fcsr`, which contains the operating mode and the exception status.

“Zifencei” Instruction-Fetch Fence

This extension defines a single new instruction, the FENCE.I, which provides synchronization between writes to memory and instruction fetches.

“Zam” Extension for Misaligned Atomics

The “Zam” extension standardizes the support for misaligned atomic memory operations made by the instructions defined by the “A” extension.

“Ztso” Extension for Total Store Ordering

The “Ztso” extension defines the RISC-V Total Store Ordering (RVTSO) memory model, which is a stronger variant of RVWMO.[6] This extension is made to support easier porting of code written for the x86 or SPARC architectures, which use Total Store Ordering (TSO) by default.

2.4 Privileged ISA

The privileged manual [14] describes the RISC-V privileged architecture. The architecture is best summarized in the manual itself.[14, Chapter 1]

> [The RISC-V privileged architecture] covers all aspects of RISC-V systems beyond the unprivileged ISA, including privileged instructions as well as additional functionality required for running operating systems and attaching external devices.

There are multiple privilege modes defined; only one of which, the machine mode (M-mode), is mandatory. The other modes are user mode (U-mode) and supervisor mode (S-mode).

M-mode has the highest privileges and code running in M-mode is considered to be secure, and can be used to provide Execution Environment to the lower privilege modes.

S-mode is intended for conventional operating system usage and adds virtual memory support.

U-mode is used by generic applications and has the lowest privileges.

Each privilege mode has a core set of functionalities that can be modified or expanded by additional extensions. For example, M-mode can support physical memory protection, described in Section 2.4.2, and S-mode can be extended to support virtualization by the Hypervisor Extension[7].

[6]RISC-V Weak Memory Ordering

[7]Described in Section 2.4.5

2.4.1 Control and Status Registers (CSRs)

In order to support privileged operation, the RISC-V ISA defines a 12-bit address space of up to 4096 control and status registers. Each of these registers is XLEN-bit wide and is typically associated with some privilege level. The CSR is accessible to this and any higher privilege levels.

A complete listing of CSRs can be found in Tables 2.2-2.6 of the Privileged Manual.[14, Section 2.2]

The privileged architecture depends on the "Zicsr" extension described in Section 2.3.3, which defines the instruction used for reading and writing to the CSRs.

2.4.2 Machine-Level ISA

The machine-level ISA defines the operations supported by M-mode. It defines them in terms of CSRs and privileged instructions.

These operations contain:

Machine Information Defines CSRs dedicated to providing information about the processor itself (e.g., Vendor ID or Hart ID).

Trap Setup and Handling Describes how exceptions and interrupts are used in RISC-V. Also defines privileged instructions used to yield control to a less privileged mode. We have decided to dedicate Section 2.4.4 to this matter.

Configuration Describe some general configuration CSRs.

Counters and their configuration The counters described in Section 2.3.3 are read-only views of the M-mode counters, which M-mode can write to. M-mode can also disable the counters, making them unavailable to lower privilege modes, and configure the general performance counters.

Physical memory protection

The machine-level ISA defines an optional extension that restricts access to some parts of the memory space. In a computer system, some parts of the physical memory space will be occupied by memory regions, others will be occupied by memory-mapped registers of devices, and some space will be left vacant. Hence, different memory areas will allow or disallow reading, writing, or execution. Some parts of the memory space might not support atomic operations, while others will. These properties are termed *physical memory attributes* (PMAs) by the Privileged Manual.[14, Section 3.6] The manual then defines a way to describe how different memory areas have different PMAs and how it affects memory accesses.

2.4.3 Supervisor-Level ISA

The supervisor-level ISA defines a set of operations that support the implementation of modern operating systems. This ISA deliberately restricts some access to physical memory or interrupts, in order to ease virtualization. Some of the operations are restricted forms of operations performed by M-mode. In addition to these, the Supervisor level ISA defines memory translation and protection mechanisms, together with a memory fence instruction that orders memory accesses with changes to the translation data structures.

Virtual Memory

RISC-V defines several modes of virtual memory translation. Only one is supported on RV32, Sv32. Sv39, Sv48, and Sv57 are available for RV64. The numbers in the names signify the effective length of the virtual address. All of these modes work on 4 KiB pages and traverse a tree-like page table to find the physical address. The difference between the modes is the depth of this tree and the size of one entry. In addition to the 4 KiB pages, the translation mechanism allows one to mark a higher-level page table entry as the leaf entry, which contains the physical address, thus translating into larger pages.[8] This model also allows U-mode software to control memory access. It can restrict which pages can be used for reading, writing, or executing and can set access to U-mode.

The "Svnapot", "Svpbmt", and "Svinval" further extend the functionality of the memory translation and protection. "Svnapot" defines a denser representation of a translation in a TLB, "Svbmt" allows the translation mechanism to override the PMA mechanism, and "Svinval" provides more fine-grained memory fences to be used for memory translation.

2.4.4 Exceptions, interrupts and traps

One of the largest responsibilities of any privileged architecture is exception and interrupt handling, which is defined in the machine-level and supervisor-level chapters of the Privileged Manual,[14] but we have decided to separate it into its own section.

RISC-V describes an exception as an "unusual condition at run time associated with an instruction", interrupt as an "external asynchronous event that may cause a transfer of control" and a trap as a "transfer of control to a trap handler caused by an interrupt or an exception".[5, Section 1.6]

The full list of exceptions and interrupts can be found in Table 3.6 in the Privileged Manual[14].

The privileged manual describes three types of interrupt, each with two variants based on the privilege level for which the interrupt is intended. Each privilege can choose which interrupts it wants to enable, or it can decide to disable interrupts altogether.

External External interrupts are meant to be used by external devices. The Supervisor external interrupts can be raised by devices as well as M-mode software. S-mode is presented that the interrupt is pending if either of these two sources raised the interrupt (logical or basis).

[8]4 MiB megapages for Sv32, 2 MiB megapages for Sv39, 2 MiB megapages or 1 GiB gigapages for Sv48 and 2 MiB megapages, 1 GiB gigapages or 512 GiB terapages for Sv57

Timer Machine timer interrupt is raised by the real-time counter (`mtime` and `mtimecmp` CSRs). Software timer interrupt is raised and reset by M-mode for S-mode.

Software Software interrupts are meant to be used for interprocessor communication. Again, Supervisor software interrupt may be raised by M-mode software and external means.

Traps

When a RISC-V processor traps, it transfers control to a trap handler. There are two modes in which this transfer of control can happen. Let A be the address in `mtvec` or `stvec` CSRs Direct mode allows only a single trap handler; it is located at A. Vectored mode defines one handler for all exceptions, and then one additional handler per interrupt. If the trap is caused by an exception, then the handler will be again located at A. If the trap is caused by an exception, let B be the number of the interrupt, then the trap handler will be located at $A + 4 \times B$.

All traps are by default handled by M-mode, but can be delegated to lower privilege levels. This is configured by the `medeleg` and `mideleg` CSRs for exceptions and interrupts, respectively. Traps never transition from a higher privilege level to a lower privilege level. If this were to happen, the exceptions would be handled by the current privilege level and the interrupts would be masked.

2.4.5 Hypervisor extension

The hypervisor extension extends the supervisor-level architecture to support virtualization of operating systems. It changes supervisor mode to hypervisor-extended supervisor mode (HS-mode), where a hypervisor runs. HS-mode defines additional CSRs and instructions that facilitate a new stage of address translation and support virtualization of a guest operating system that runs in virtual S-mode (VS-mode). Software written for S-mode can be executed in HS-mode or VS-mode without any modifications. Virtualized user-level applications run in VU-mode.

This extension changes the virtual address translation process for the virtualized modes. The translation first works as before, but we translate the result again, using the page-table of the hypervisor.

3. Design Decisions

In this chapter, we will describe the design decisions that have been made during the development process. Namely, it includes the architectural changes required for MSIM to support multiple CPU architectures, the architecture of the RISC-V implementation, and the decision process behind choosing the supported RISC-V extensions.

MSIM is written in pure C. Because of this, we will make decisions and design new interfaces to work with the programming paradigms of C. We will take advantage of the static typing of C and try to simplify dynamic memory management.

3.1 Decoupling MIPS from MSIM

Since MSIM has been developed with only one architecture in mind and because the CPU is the central part of every computer system, MIPS is tightly coupled with most modules of the system. Every time a component needs to communicate with the CPU, it communicates with MIPS directly. If we want to support multiple CPU architectures, first we need to decouple MIPS from the rest of the system.

One of the ways this could be done is to redefine the interface in terms of MSIM devices, adjusting the device abstraction to support the operations needed for a CPU. This would extend the device interface by new functions that would not be defined for most devices. The operations desired from a CPU can be found in Section 3.1.1. We can see that some of the functions could be useful for other devices, like dumping registers, which could be broadened to printing status of the device, but most of the functions are CPU specific. For example, setting the program counter to some location does not make sense for any device that is not a CPU.

We have decided to decouple MIPS by introducing a level of abstraction we call the *General CPU*, which will function in the same way as implementing an interface in classical Object-Oriented Programming. We think this is the best way to model this behavior because we make the system depend on the abstraction of a CPU and not on the implementation itself, which will allow us to support multiple different implementations of a CPU.

3.1.1 General CPU

Before we start to describe the structure of the *General CPU*, we will have to solve one question; that of identifying the CPUs themselves. As it stands, the CPU identifies itself by the pointer to itself in the code. If we wanted to change the identification to work with the *General CPU* structure, we would need the CPU to be aware that it is contained within a *General CPU*. Instead, we decided to identify the CPU by its ID (in MIPS terminology Processor ID, in RISC-V terminology Hart ID) and to support the lookup of the *General CPU* structure based by this ID. This requires the address space of these IDs to be shared across architectures, but this would be required despite this because of inter-processor interrupts.

Interface

To implement the *General CPU* abstraction, we need to define its interface, in other words, the functionality we need a CPU to support. We want this interface to include only the functionality we require from a CPU, but which is not already included in the device abstraction made by MSIM, as this would introduce duplicity. We will also define the interface so that if the caller does not need a specific CPU but only some CPU, the function can be called with a `NULLPTR` argument and a deterministic fallback will be chosen. This fallback will be CPU with ID 0. Through observation of the implementation of MIPS and from where its functions are called, we have deduced that this abstraction should support:

Raising and Clearing Interrupts The different devices raise interrupts to signal some event to the CPU, and clear them to signal that the event has ended.

Store Conditional Check In order to support the LL(LR)/SC atomic instructions, the different CPUs must guarantee the failure of SC when any write takes place to some set of addresses. Different architectures might implement this mechanism in different ways. For example, RISC-V allows the reservation set of an LR instruction to be arbitrarily large but must contain all the read data. To support this range of behaviors, we define the interface to allow the CPU to be notified that a write to an address has taken place, making the architecture itself decide whether this affects its reservations.

Converting Addresses from Virtual to Physical Convert a virtual address to physical based on the current state of the CPU.

Adding and Removing Breakpoints We want to support debugging breakpoints that trigger when the computation reaches a given address.

Register dump Dump the content of the general purpose registers to the standard output.

Setting the Program Counter Set the program counter, effectively jumping to an arbitrary location.

We can split these into two categories. The first two types serve a purpose in the system itself, while the rest are used to support debugging.

3.1.2 Physical Memory (Physmem)

In MSIM, the physical memory space is implemented in the same file as the MIPS CPU, even though it is conceptually a separate module. We will separate it into its own file and redefine its interface, to work with processor ID. We do not use the *General CPU* structure because, in most cases, we do not need the information which CPU accesses the memory, and because we try to hide the abstraction from the CPUs. In some cases, physical memory needs to be accessed by the devices (e.g., for devices supporting DMA, like ddisk). A special processor ID of -1 is reserved for such occasions.

In addition to accessing main memory and memory mapped registers of devices, the Physmem module also had two other responsibilities in MSIM: notifying the CPUs of writes for LL/SC, and caching decoded instructions. The next two sections will describe how we will change these.

LR/SC

We will keep this responsibility in Physmem, but we will adjust it to work with the *General CPU* abstraction. We track the processor IDs of the CPUs that have an active registration. The CPU should register itself for this tracking when executing the LL(LR) instruction and unregister itself on successful SC. When writes are made to main memory, all registered CPUs are notified of the physical address and size of the write, and they themselves decide whether the write invalidates the reservation. For MIPS, the invalidation is done if the write would be to any byte of the cache line (64B), and since MIPS allows only aligned memory operations, the address of the write has to reside inside the same cache line as the reserved address.[16, Pages 26, 289]

Cached translation pages

The Physmem frames hold a reference to an array of decoded MIPS instructions (function pointers, one per instruction in the frame, on the proper offsets) and a valid flag signifying the correctness of the cache. If we wanted to support a similar mechanism for both MIPS and RISC-V, we would need two such caches per frame. If we restricted ourselves to the case that only one architecture can have cached instructions in one frame, then we would need to store the data in a weakly typed form (as a `void*`) and store the type information separately. We believe that either of these approaches is problematic because they cannot be easily modified to support an additional architecture. We think that the caching of the decoded instructions is an implementation detail of the architecture itself and should not be exposed to Physmem.

Physmem will still hold the valid flag used for these translations. This flag will be set when any architecture caches the instructions and will be reset on writes to these pages. This mechanism functions correctly only when there are instructions from at most one architecture that implements caching present in one frame, but this constraint is easily fulfilled by loading the code for different architectures separately into different memory areas and taking care not to jump into the code for another architecture, running only a single architecture, and not writing to code pages (making them read-only by MSIM also works).

The worst case is that MIPS CPU M will execute code from some frame F, then jump elsewhere. After that, some write to F changes the MIPS instructions stored there. Then a RISC-V CPU R jumps to F, caches the instructions, and sets the valid flag, after which M jumps back to F, thinks that its cached translation is still valid, and executes based on its cache.

This situation is very rare and can happen as a result of a bug in a program or in an environment with dynamic code changes and heterogeneous processing. Because of this, we highly recommend setting the code memory areas and read-only, especially on systems with multiple architectures running at the same time. Dynamic code generation is discouraged on systems with multiple architectures.

If we were to solve this problem, we would need to store whether a translation is valid on a frame for an architecture. We could have valid flags on the frame itself (per architecture), but as we said earlier, we think this mechanism should be hidden from Physmem. One way of implementing it would be to have the flags as bit fields in one single integer field, which would be reset to 0 on writes. But then we would need to manage the uniqueness of these fields based on the architecture. The other way around would be to store this information together with the cached data, but then Physmem would need to notify the architectures about all the writes, which again exposes this system to Physmem, which we do not want.

All in all, we think that the solution with a global valid flag is the solution that separates this mechanism from Physmem the best, while adding only small constraints, which have no effect on regular use (1 architecture running at a time).

3.1.3 Debug

We have decided to support some of the debug functionality supported by MIPS. We will support the printing of general-purpose registers and CSRs. We will also support the tracing functionality by printing the current instruction and its parameters. We have decided to support comments with additional information about the traced instructions, like the absolute targets of relative jumps, different representations of numbers or more arithmetical notation of the computation done by the instruction. We have decided not to implement physical memory breakpoints, but they could be easily added.

3.1.4 GDB support

The GDB functionality is experimental and is deeply coupled with the MIPS implementation details. We have added some of the required functionality to the *General CPU* abstraction, but we have not reimplemented the GDB code to work with this abstraction. We think that proper GDB support would be beneficial for MSIM, but we recommend reimplementing it from scratch using the abstractions defined for this work.

3.2 RISC-V support

We have decided to choose the RV32I base ISA because it is the simplest base ISA. RV64I and RV128I are unnecessary for the Operating Systems course since it already uses only 32-bit MIPS operations and the additional instructions or address translation schemes would only complicate the kernels of students. We also do not restrict ourselves to the reduced number of 16 registers in RV32E, because that would not benefit us in any way and would only hinder the possible implementation of the “H” extension.

We have decided to follow the steps of MIPS and allow only memory accesses aligned to their widths. We also simulate in program order, meaning that we do not reorder the instructions or perform out-of-order execution. This is done to fulfill the goal of MSIM to be deterministic and to make debugging easier.

3.2.1 Architecture

We have decided to split the RISC-V implementation into three modules. The *CSR* module is responsible for the implementation of the CSRs and the operations on them. The *Instr* module defines the instructions and their encoding. This module is itself divided into several submodules, one per instruction type and one extra for decoding instructions. Lastly,the *CPU* module defines the operations of the whole CPU.

We have chosen to implement the support for RISC-V in a manner similar to that of MIPS, meaning that we define one function per instruction. This approach supports instructions to be decoded from the data ahead of time and to be cached.

CPU Step

The simulation of one step of the CPU is similar to the way it is done for MIPS, as described in Figure 1.2, but with some slight modifications. We have removed the instruction fetch exception loop. In the original implementation, if the instruction fetch of a trap handler caused an exception, the simulation would enter an infinite loop, without giving the user any information or control. We have decided to remove this loop. In this way, if the fetch of the trap handler itself caused an exception, the repeated fetch caused by the exception would be attempted again in the next step. Between these steps, the user can have interactive control of the simulator and can get feedback on what is happening. The MIPS CPU has been reimplemented in this way as well. The adjusted flowchart is shown in Figure 3.1.

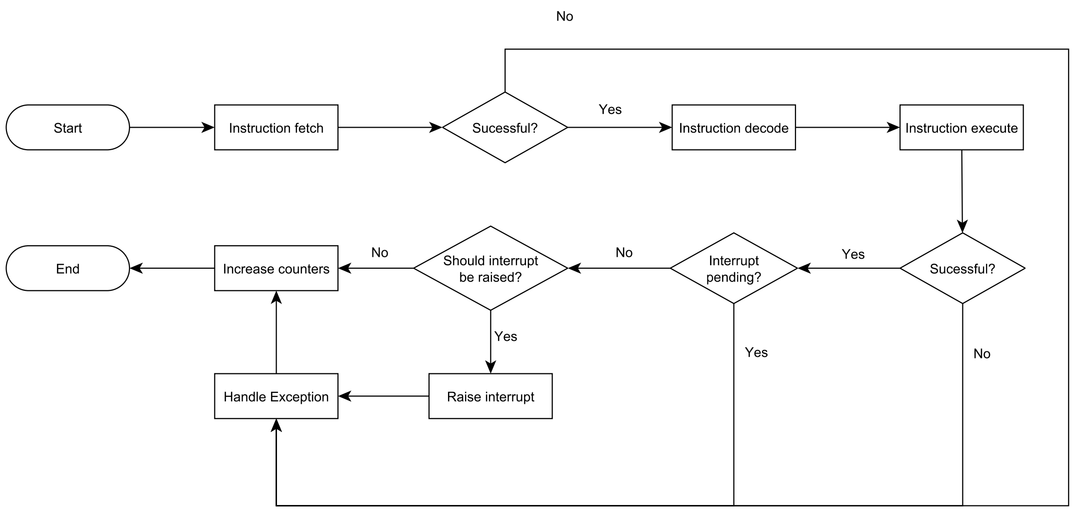


Figure 3.1: Flowchart of simulating one step of the RISC-V CPU

3.2.2 Extensions

For RISC-V to be useful in the Operating Systems course, it has to come with some extensions. First and foremost, the support for S-mode and U-mode is absolutely necessary, which depend on the "Zicsr" extension. The support for counters is also required because a timer interrupt is required for an implementation of a preemptive scheduler. One problem arises, the RISC-V only has support for a real-time based timer interrupts, which are inherently nondeterministic. Thus, we have defined a new non-standard extension which adds the support for `cycle`-based timer interrupt. This is closely described in Section 4.1.1.

In addition to the necessary extensions, we have also decided to support the "A" and "M" extensions. As a result of the way we simulate the instructions in order, we also support the "Ztso" extension as a side effect.

"A"

The atomic instructions defined in the "A" extension are not necessary for the Operating Systems course. The kernels implemented in the practicals run on a single thread (hart). Because of that, simply disabling interrupts is sufficient to support the implementation of synchronization primitives.[1]

Even if atomicity can be achieved by different means, some students might want to try to implement the synchronization primitives in a more realistic way, using atomic instructions. We think this should be encouraged, and MSIM should support this functionality for RISC-V.

"M"

The support for separate multiplication, division, and remainder instructions makes reading assembly code with these instructions more readable. We think that because these operations can be used in kernel development, we would benefit from the added readability.

Unsupported Extensions

"C" We do not need to restrict the space our code takes, as it will not be much in the first place.

"F", "D" and "Q" We believe these extensions are not very useful for the Operating Systems course. If MSIM were used in different circumstances, the support for these could be justified.

"Zifencei" We have decided not to support this extension because we deem it unnecessary because of the restrictions we have put on the instruction fetches. These restrictions can be found in Section 4.1.3.

"Zam" We have decided to support only aligned memory accesses, regardless of whether they come from general memory operations or from atomics. Adding support for misaligned atomics would thus go against this decision.

[1]Like mutexes and semaphores

“H” The hypervisor extension increases the complexity of S-mode, and virtualization support is not required for the Operating Systems course.

4. Implementation

This chapter describes the development process behind the implementation of RISC-V. We describe how we implemented some of the features, describe when we deviated from the RISC-V specification, and lastly describe how we tested and evaluated the implementation.

4.1 Notable features

In this section, we want to express our reasoning behind some notable implementation features and how they affect the system. We do not write about every module, because implementation of some of them was straightforward; for example, the implementation of most of the instructions is very simple.

4.1.1 CSRs

There are two questions that we have to solve while implementing the CSRs; how do we represent them in memory and how do we implement the instructions for their access.

There is a 12-bit address space dedicated to the CSRs, which are 32-bit wide in RV32I. A trivial representation would be to allocate the full 16 KiB array that would be needed for them. This approach has the advantage of direct indexing into such representation by the CSR number. But the rather large disadvantage of this representation is how much space is wasted. There are only 294 CSRs defined by the privileged standard,[5, Tables 2.2-2.5] which would take only 1.176 KiB, which is almost 16 times less space than the full representation. The representation we have chosen is a sparse representation in a structure. This representation adds the difficulty of accessing the data based on the CSR number, which is not an issue, as the way we access the CSRs allows us to make the translation naturally.

Many of the CSRs define access rules to them, sometimes even on the level of individual bits. For example, the `mcause` CSR can only hold values that are a valid exception code, and a write that would change the CSR to an invalid value must throw an exception, while `mstatus` has many individual bits, each having their own semantics, and some unwritable bit-fields. Having one universal function that would handle all of the different cases would result in a code that could not be modified easily. We have decided to separate the functionality between several functions. Special CSRs (such as `mcause` or `mstatus`) have their own functions for reading, writing, setting, and clearing bits, while more general CSRs (such as the 29 HPM counters) can share the same functions as long as their access logic is the same. The four operations reflect the operations defined by the instructions defined in the "Zicsr" extension, described in Section 2.3.3. The only question left is how to dispatch these functions based on the CSR number, which is implemented as a switch statement. As promised earlier, this approach solves the problem of finding the right member of the structure based on the CSR number; we access the CSRs from a function that works on a defined set of CSRs (mostly only one) and can access them directly in the structure.

Custom CSRs

In order to support deterministic timer interrupt, we have decided to define a nonstandard extension that contains one new CSR `scyclecmp` (`0x5C0`). This 32-bit register can be accessed from S-mode and any more privileged modes. It can be written with any value. The number for this CSR has been chosen from the range dedicated to custom read-write Supervisor-Level CSRs. When the value of the `cycle` CSR is greater than or equal to the value in `scyclecmp` we say that the Supervisor Timer interrupt is requested.

In the privileged specification,[14, Pages 32-34] the supervisor timer interrupt (STI) is considered pending when the STIP bit in `mip` CSR is set. We will now redefine how the STI should be made pending. STI is considered pending when the STIP bit is set **or** when STI is requested by `scyclecmp`. When `mip` is read with a CSR instruction, the value of the STIP bit should also be the logical or of these values, but writes to `mip` should only affect the stored value.

This behavior is based on the way the Supervisor External interrupt is made pending, which can also be set by M-mode or raised by a platform-specific interrupt controller.

Timer interrupts

As illustrated in Figure 3.1, the incrementing of counters is done as the last operation in simulating one step. The simulation is structured in this way, because some of the counters should not increase when a trap is taken. After the counters are increased, the check whether an interrupt should be made pending (in case of the machine timer interrupt) or requested (in case of STI) is made (we will refer to this situation as requesting an interrupt for simplicity). The same checks happen when the related CSRs or memory-mapped registers are changed by writes from the CPU.

Due to this detail, we first check for pending interrupts and only after that increase the counters and request interrupts. This results in the next instruction executing without being interrupted, even if there is a pending interrupt. The CPU traps right after executing this instruction. This situation is shown in Table 4.1.

Instruction	`cycle`	`scyclecmp`	*comment*
`ins1`	0	1	*STIP is set at the end of this step*
`ins2`	1	1	`ins2` *still executes, STIP is set*
			Trap
`tins1`	2	1	`tins1` *is the first instruction of the trap handler*

Table 4.1: Raising Timer Interrupt Example

This behavior is allowed by the specification of the way interrupt traps occur. The privileged manual prescribes this behavior as follows:[14, Page 32]

> [The conditions] for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in `mip`, and must also be evaluated immediately following

> the execution of an xRET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend.

We evaluate this condition right after the execution of each instruction, which satisfies this definition.

4.1.2 "A" Extension

LR/SC

Our implementation of defines the set of bytes that is reserved by the LR instruction to be the set consisting of the 4 bytes that were read exactly. We implemented the SC check to test if the write we are notified about overlaps these four bytes.

We also have to address the restrictions on the eventual success of SC imposed by the specification of the "A" extension.[14, Section 8.3] The specification defines a *constrained LR/SC loop* and makes guarantees on the behavior of the SC instructions in such loops. This is done to prevent livelocking of LR/SC sequences in real systems, where two different harts could ping-pong one cache line between each other without ever succeeding on any SC. Our implementation does not have this problem, and because we simulate the steps of different harts sequentially, the first write on a particular address would either be a successful SC, or invalidate any reservations on the relevant bytes.

4.1.3 Memory model

We simulate in program order, which means that we execute the instructions in the order that they were fetched. Also, we simulate multiple harts step-by-step in a round-robin fashion. This stricter ordering satisfies all the restrictions that RVWMO and RVTSO put on memory orderings. We can even go as far as to implement the FENCE instruction as an empty operation.

Caching the instructions and FENCE.I

Regarding this caching mechanism, the RISC-V memory model allows us to reorder implicit memory reads as back as the nearest memory fence of the proper type. Instruction fetches are implicit reads, and the unprivileged manual [5, Section 1.4] says:

> Except when specified otherwise, implicit reads that do not raise an exception and that have no side effects may occur arbitrarily early and speculatively, even before the machine could possibly prove that the read will be needed. For instance, a valid implementation could attempt to read all of main memory at the earliest opportunity, cache as many fetchable (executable) bytes as possible for later instruction fetches, and avoid reading main memory for instruction fetches ever again.

This means that we could decode all instructions once and then never worry about cache invalidation. This would obviously not work with dynamically generated instructions. The "Zifencei" extension solves this problem by defining an instruction fence that would invalidate our caches. We think that if MSIM behaved in this way, it would only confuse the users, who have probably not read the RISC-V standard in its entirety and who would expect that a write to memory should be seen by the CPU on the next clock cycle (including instruction fetches). That is why we have decided to make the fetch implicit reads behave as if they were done at the same cycle as the instruction itself.

4.1.4 Custom instructions

MSIM defines several custom instructions for MIPS, most of which are designed as debugging tools, while DHLT which halts the simulation is necessary to stop the simulator from the code. We have also defined custom instructions for RISC-V. We have chosen to make all the custom instructions use the SYSTEM opcode and use the encoding dedicated for custom system instructions.

EHALT

EHALT stops the simulation. This instruction is unprivileged, even if we think that this functionality should be restricted to M-mode in real systems. We have decided to behave as such, to simplify its usage.

EDUMP

EDUMP prints the contents of the general purpose registers to standard output. This functionality is useful for debugging and we will be using it in our automated tests of MSIM described in Section 4.3.

4.2 Deviation from specification

4.2.1 EBREAK

In the unprivileged manual,[5, Page 27] EBREAK is said to return control to a debugging environment. This could be understood as turning on interactive control in MSIM. But in the privileged manual,[14, Page 46] it specifies that EBREAK only throws the breakpoint exception. This desire is unpractical in MSIM, so we have decided to deviate from the specification in this regard, and we begin interactive control in executing EBREAK and do not throw any exception.

MSIM can be made to adhere to the specification by making EBREAK do the defined behavior and defining a new custom instruction, which would enter the interactive control. We have decided to go against this decision, because we want to make placing a breakpoint into code easy for students from C, which can be achieved by simply writing `asm("ebreak")`, and not `asm(".word 0x8C400073")`.

4.3 Testing

This section describes how we have approached testing the RISC-V implementation. We have split testing into two categories: unit tests and system tests. Unit tests check the behavior of MSIM functions, are written in C, and linked with

MSIM. System tests verify the behavior of MSIM itself. Each system test runs the simulation of the provided code from start to finish. These tests are written in RISC-V assembly.

We have tried to cover as much functionality as possible, while not duplicating tests, choosing the type of test that fits the case the best in our opinion.

4.3.1 Unit tests

We have implemented 95 unit tests divided into 3 test suites.

Decoding of instructions Decoding of the instructions is in our context the translation from the 4 B that have been fetched to the function that implements this instruction. We set the data of the instruction up and then check if the decoded function is the one we would expect.

Exceptions We call an instruction function and see if it returns the exception that we would expect.

Immediate value decoding When an operand of an instruction is a constant in some range, the value itself can be encoded in the instruction data. This value is then called an immediate value. The unprivileged specification [5, Figure 2.4] defines several formats of the immediate values. We test if our decoding is done correctly.

PCUT

All unit tests are written in the Plain C Unit Test Framework (PCUT).[17] We have decided on this framework, because it is a fairly simple library, without any other dependencies.

4.3.2 System tests

Most functionality is tested by running the whole program and comparing the output with a predefined expected result. In some tests, we use the printer device to print characters to a file. In others, we dump the contents of the registers or CSRs. In this way, we can easily test the behavior of traps or of the address translation.

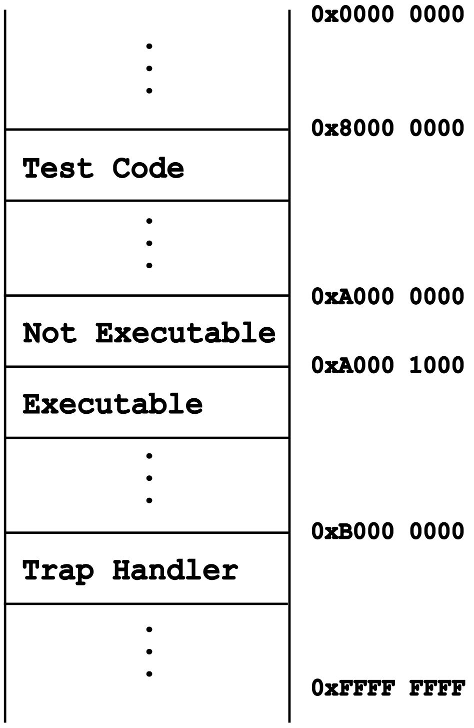
These tests are written as their assembly source code together with their compiled form and disassembly. A simple build script is also provided, but is not needed. The compiled code is present because we do not want to force everyone who wants to run the tests to install a RISC-V assembler.

We have defined 21 system tests, most of which consist of multiple test cases. There are nine tests that test unprivileged instructions, including the “M” and “A” extensions. These tests are split logically, based on their category, and further divided by their mode of operation (for example, we test conditional branches separately from unconditional jumps). We have also defined one simple test which loads data from memory, does simple arithmetic on it, and then stores the result back to memory. Then we have defined 10 tests that cover exceptions and interrupts. All these tests cause a trap and check if it was handled correctly.

The last test is the largest and covers virtual memory translation. In this test, we first set up the page table, then do different memory accesses with virtual address translation in effect. As some of the accesses cause exceptions, this test also consists of a simple trap handler, which notifies us of the trap by writing to the console, and then continues the execution right after the instruction that caused the trap.

One problem arises with testing attempted execution from a non-executable page. We want to jump from our test code to the non-executable page, which needs to cause a trap. Our trap handler would, after it ends, resume the execution on the next instruction after the one we jumped to, but we want to return the execution to the test code. We achieve this by jumping to the last instruction of the non-executable page. Then our trap handler resumes the execution on the first instruction of the next page, which we make executable, and we make the first instruction an unconditional jump to our test code.

The memory layout and pseudo-assembly code for this test is illustrated in Figure 4.1. For brevity, the instructions in the figure are not RISC-V assembly instructions. `MEPC` is the exception program counter CSR and holds the address of the instruction that caused the exception. The `JAL` instruction jumps to the specified offset and saves the address of the instruction after itself into the specified register (in our case, `ra`). `JR ra` jumps to the address in the `ra` register. `MRET` jumps to the specified address and lowers the privilege level.



```
//Test code
...
0x8000 0124   JAL ra, 0xA000 0FFC
...

//Trap Handler (MEPC=0xA000 0FFC)
0xB000 0000   Print 'T'
0xB000 0004   MRET MEPC + 4

//Executable Page
0xA000 1000   JR ra
```

Figure 4.1: Illustration of the Virtual Memory Test

4.4 Evaluation

We have prepared a demonstration of MSIM in a more realistic situation than in the tests. This demonstration is a simple program written in C, which is then compiled into RISC-V machine code, which is executed by MSIM. We have prepared the loader, which jumps into the C code together with the build scripts for compilation. The source code is compiled by the *riscv-gnu-toolchain*[1] with flags set to compile without the standard library and to strictly align data (`-nostdlib`, `-nostdinc`, and `-mstrict-align`).

The code runs in M-mode for simplicity, but could be easily changed to run in U-mode instead.

The program consists of formatting functions that are used to display the result of the computing functions and the computing functions themselves. The formatting functions use the `dprinter` device, which allows us to write one character at a time by writing to its memory-mapped register. We have implemented printing of entire strings, integers, and matrices. These functions demonstrate how to work with MSIM devices and show a more complicated control flow.

We have also implemented calculation of the Fibonacci numbers by a loop, this function demonstrates that the basic arithmetic and control-flow work as expected. We have also implemented a recursive calculation of the factorial and Fibonacci numbers; these functions demonstrate that MSIM handles the standard function calling convention (mostly system stack management) well even under stress.

The last calculation is matrix multiplication, which requires more memory accesses than the previous calculations, and cannot be optimized to have all variables in registers. We have chosen to use 32×32 integer matrices (4 KiB each). We also test whether we can manipulate global variables.

We have prepared a simple demonstration that runs the calculations and outputs their result. We have chosen to run these calculations for relatively small inputs, to make the program terminate in reasonable time.

We have also prepared a more calculation intensive demonstration, which puts more stress on MSIM. This test runs the recursive (and exponentially slow) Fibonacci number calculation and many matrix multiplications. We have decided to calculate the 39-th Fibonacci number and execute 3500 matrix multiplications, as this resulted in run-time of around 22 minutes. We have not made any more extensive measurements, because MSIM does not aim for performance. This stress test is intended to give us a rough idea of the speed of the simulation and to demonstrate that even longer programs can run on MSIM without problems.

Both of these demonstrations run without problem and output the expected result.

[1]`https://github.com/riscv-collab/riscv-gnu-toolchain`

Conclusion

In this thesis, we have described how we changed MSIM to support RISC-V as well as MIPS. We have also summarized our reasoning behind the decisions we have made during the process of designing and implementing these changes. We have verified the correctness of our implementation with several test suites and even provided a way of compiling simple C programs into RISC-V machine code and then simulating the program. The source code is attached to the electronic version of this thesis; the latest version is also available online at `https://github.com/HanyzPAPU/msim`.

We will elaborate on further expansions of the functionality of MSIM in the rest of this chapter.

Further work

Advanced Operating System Features

The current state of MSIM is sufficient for the introductory Operating Systems course, but might not be sufficient for more advanced lectures. To support operating system topics not covered in the OS course, new devices and features would need to be added. MSIM supports a disk device that could be used for implementing a file system, but lacks a network card, or a GPU. An even more advanced topic would be virtualization, which could be supported by implementing the RISC-V "H" extension.

GDB

As discussed in Section 3.1.4, the current GDB implementation is an experimental one, and we think reimplementing it using the *General CPU* abstraction would benefit MSIM the best.

Better debugging support for RISC-V

As it stands, the MIPS CPU has more debugging functionality than RISC-V. We would like to point out these missing features, as well as describe some new ideas for debugging functionality.

More custom instructions

MIPS supports a wider range of custom instructions, for example, ones turning on or off the tracing functionality, or dumping the CP0 registers. These custom instructions could be implemented for RISC-V in the same way as the EHALT and EDUMP instructions.

Better selection of CSRs when dumping

We have implemented dumping of CSRs based on their number or name, as well as dumping all CSRs. A wider variety of options could be added, for example, dumping by the privilege level needed to access these CSRs or by the categories that the CSRs are split into in the privileged manual.[14, Tables 2.2-2.4]

Memory breakpoints

MSIM supports adding physical memory breakpoints, that cause the simulator to enter interactive mode when execution in either of the MIPS CPUs fetches from an address with a breakpoint. This mechanism can also be supported by RISC-V.

Debug Specification

There exists a not yet ratified specification of debugging support for RISC-V processors.[18] This specification provides a standardized set of debugging features that can be utilized by an external tool, such as GDB. It would be beneficial if MSIM could provide the specified debugging interface to such tools.

Floating point arithmetic

Support for the “F”, “D” or “Q” extensions could be added. MSIM would allow more general-purpose applications to run this way.

Physical Memory Protection

RISC-V defines a way to restrict the operations supported on some memory areas. The specification of these areas can even be made dynamic. For simplicity, our implementation does not restrict the operations in any way. The support for this could be useful for disallowing atomic access to memory-mapped registers of devices and other similar situations.

RV64I

The RISC-V implementation could be extended to support RV64I base ISA, in order to emulate more realistic systems.

Implementing other CPU architectures

We have added an abstraction layer to MSIM which allows us to emulate multiple CPU architectures. Now we can easily add support for another architecture to MSIM. We do not see the benefits of adding another architecture in this moment, but there could be in the future.

Bibliography

[1] Vlastimil Babka, Lubomír Bulej, Martin Děcký, Viliam Holub, and Petr Tůma. Teaching operating systems: student assignments and the software engineering perspective. pages 71–78, 05 2008.

[2] Viliam Holub. Educational OS simulator, 2003.

[3] Jim Turley. Wait, What? MIPS becomes RISC-V. Available at `https://www.eejournal.com/article/wait-what-mips-becomes-risc-v/`.

[4] Christoph Müllner. Emulators and Simulators. `https://wiki.riscv.org/display/HOME/Emulators+and+Simulators`. RISC-V Technical Wiki.

[5] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213.* RISC-V Foundation, December 2019.

[6] *QEMU, a fast and portable dynamic translator.* USENIX Association, 2005.

[7] Andrew Waterman, Yunsup Lee, Scott Johnson, Tim Newsome, and Chih-Min Chao. Spike RISC-V ISA Simulator. `https://github.com/riscv-software-src/riscv-isa-sim`.

[8] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[9] Prashanth Mundkur, Rishiyur S. Nikhil, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, Peter Sewell, Alexander Richardson, Hesham Almatary, Jessica Clarke, Nathaniel Wesley Filardo, Peter Rugg, and Scott Johnson. RISCV Sail Model, 2022. `https://github.com/riscv/sail-riscv`.

[10] Michael Clark. rv8: RISC-V simulator for x86-64, 2018. `https://github.com/michaeljclark/rv8`.

[11] Fabrice Bellard. TinyEMU, 2019. `https://bellard.org/tinyemu/`.

[12] V. Holub, M. Děcký, T. Martinec, V. Horký, and P. Tůma. MSIM Version 1.4.0 Reference Manual, 2020. Available at `https://github.com/d-iii-s/msim/blob/master/doc/reference.html`.

[13] RISC-V International. About RISC-V. `https://riscv.org/about/`, 2021.

[14] Andrew Waterman, Krste Asanović, and John Hauser, editors. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203.* RISC-V International, December 2021.

[15] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[16] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Inc., 1994.

[17] Vojtěch Horký. PCUT: Plain C Unit Testing mini-framework, November 2019. `https://github.com/vhotspur/pcut`.

[18] Paul Donauhe and Tim Newsome, editors. *RISC-V External Debug Support Version 1.0-STABLE*. RISC-V Debug task group, 2022. Available at `https://github.com/riscv/riscv-debug-spec`.

A. Attachments

A.1 MSIM project

The project is attached to the electronic version of this thesis, but the latest version can also be found online at `https://github.com/HanyzPAPU/msim`.

The source code can be found in the `src/` directory. The *General CPU* structure is defined in the header file `src/device/cpu/general_cpu.h` and implemented in `src/device/cpu/general_cpu.c`. The RISC-V implementation is located in the directory `src/device/cpu/riscv_rv32ima/`.

All RISC-V tests can be found in the directory `tests/rvtests/`.

The demonstration of running code compiled from C can be found in the `tests/rvtests/execute_c/` directory.

The reference documentation of MSIM resides in the `doc/reference.html`, while the development manual can be generated by doxygen from the directory `doc/`.