Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Ole Sivert Aarhaug

Implementing a hypervisor on RISC-V with Rust using the 1.0 hypervisor extension

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Bjørn B. Larsen
Co-supervisor: Michael Engel
June 2022

Ole Sivert Aarhaug

Implementing a hypervisor on RISC-V with Rust using the 1.0 hypervisor extension

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Bjørn B. Larsen
Co-supervisor: Michael Engel
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Contents

List of Figures

List of Tables

Listings

Abstract

Norwegian

Formålet med denne avhandlingen er å undersøke og implementere den nye hypervisorutvidelsen for RISC-V, som ble ratifisert i spesifikasjonen i november 2021. Den sentrale implementeringen oppnås ved å implementere en hypervisor og en gjestekjerne skrevet i Rust, for å undersøke dens levedyktighet som et systemprogrammeringsspråk. Den resulterende implementeringen kjører en hypervisor med QEMU og virtualiserer en supervisor-gjestekjerne, hvor det er et hypercall-grensesnitt for SBI klokke kall og virtuelt minne som kartlegger fysiske enheter som UART. Det er imidlertid noen begrensninger: hypervisoren kan ikke virtualisere mer enn én gjest, og virtuelt gjesteminne fungerer ikke som forventet. Og er dermed ikke i stand til å virtualisere mer komplekse gjestekjerner som f.eks. operativsystemer. På grunn av dette, og at den kjørte på en simulator, var det kun mulig å danne en sammenligning basert på en statisk analyse utført på eldre hypervisorer skrevet før spesifikasjonen for utvidelsen ble ratifisert, da det ikke var mulig å samle inn numeriske data. Det er imidlertid fortsatt mulig å si at den nye hypervisorutvidelsen er et verdifullt tillegg, som reduserer programvarekompleksiteten og gjør det lettere å utvikle hypervisorer på arkitekturen. Erfaring fra skrivingen av hypervisoren viser at Rust som programmeringsspråk på systemnivå har mange potensialer, men at det fortsatt er et stykke igjen før det kan bli en pålitelig erstatning for andre industristandarder som C.

English

This thesis aims to explore and implement the new hypervisor extension for RISC-V, ratified into the specification in November 2021. The core implementation is achieved by implementing a hypervisor and guest kernel written in Rust to explore its viability as a system programming language. The resulting implementation runs a hypervisor with QEMU, virtualizing a supervisor guest kernel where a hypercall interface exists for SBI timer calls and virtual memory mapping physical devices like UART. However, some limitations exist: the hypervisor cannot virtualize more than one guest, and guest virtual memory does not work as expected. And thus it is not able to virtualize more complex guest kernels like operating systems. Due to this, and it was running on a simulator, it was only possible to form a comparison based on a static analysis performed on older hypervisors written before the specification for the extension was ratified since it was not possible to gather numerical data. However, it is still possible to say that the new hypervisor extension is a valuable addition which reduces software complexity and makes hypervisors easier to develop on the architecture. Experiences from writing the hypervisor show that Rust as a system-level programming language has a lot of potentials but still has some way to go before it can become a reliable replacement for other industry standards like C.

Acknowledgement

A special thanks go to my advisors, Michael Engel and Bjørn B. Larsen, for helping me with the technical details and advice regarding writing this thesis.

Other thanks go to Takashi Yoneuchi for his unfinished hypervisor project rvvisor[18] and Stephen Marz for his RISC-V OS using Rust[12]. Both helped with the fundamental understanding of how to do a hypervisor implementation on RISC-V and provided a project skeleton for performing it myself.

1 Introduction

Virtualization is a widely used technology for more complex computer systems and architectures. Today most of what we think of as the cloud is a vast number of applications running isolated from each other in virtual machines on powerful servers hosted in data centres. The virtualized nature of these applications also enables flexibility by abstracting away hardware-specific dependencies so the software can run on many systems.

We can also find virtualization on a smaller scale where its concepts are being used to isolate applications on embedded platforms for enhancing platform security and reliability. One example is Siemens Jailhouse which is open source and publicly available.

The concept and technology exist for several decades already but are even more relevant in today's connected and cloud-based environments. We can find references dating to the '70s when Goldberg's and Popek's article "Formal requirements for virtualizable third generation architectures" [14]. So it already lays out the general requirements for a system to support virtualization.

1.1 Background and Motivation

RISC-V has matured increasingly in recent years, going from being an academic new instruction set architecture to being widely adopted and used in the industry. With broader adoption comes a wider interest in adapting technologies from other architectures to RISC-V. For example, ARM64 and X86_64 have their specification-defined way of virtualization with embedded hardware helping features. This was not the case with RISC-V initially, so hypervisors like RVirt's [11] rely on trap-and-emulate within the privileged system mode, where the software is solely responsible for doing the virtualization separation itself.

At the end of November 2021, the RISC-V hypervisor extension was ratified and formally adopted into the privileged architecture specification. It defines hardware features that can be implemented into a core to reduce virtualization overhead and simplify the implementation of a hypervisor. It was, therefore, fascinating to look at and implement software which uses the new hypervisor extension and document the process since there is little documentation on how to utilize the extension at the time of writing.

Traditionally, the classic go-to system development language has been C or C++ for low-level software development. However, in recent years new languages like Rust have started to appear, which claim to provide the same flexibility and speed C has but with modern language features and memory safety checks. It is interesting to see how Rust holds up, especially when writing software for a newer architecture like RISC-V.

1.2 Scope and objectives of this thesis

The objectives of this master thesis will be the following.

- Explore the steps needed to create a hypervisor with the new RISC-V hypervisor extension (H-extension).
- Evaluate the simplicity of the new extension compared to the previous methods.
- Evaluate the pros and cons of using Rust as a system development language.
- Design and implement a working hypervisor on a RISC-V platform that supports the hypervisor extension.

2 Theory

2.1 Virtualization

Part of this section is from a paper I wrote earlier about virtualization [1].

In this section, we will explain and show the differences between the two types of virtualization, namely full virtualization and paravirtualization. The idea here is to give a short overview so each concept is understandable and the suggested implementation can be understood. For a more comprehensive and detailed overview of these concepts, please see the relevant whitepaper [17]

Virtualization is the concept of running software semi or fully isolated from the host system while giving the running software the impression it is running on a system of its own. Depending on the implementation, the guest system (the virtualized environment) can also have direct or indirect access to the hardware. For example, the host can section off parts of its system resources and give the guest complete control over that hardware. It can alternatively direct enable access to hardware through a hypervisor which will map the whole relevant system memory or peripherals to the guest sees to the relevant allocated sections on the host's system. A hypervisor is a component that enables the possibility of virtualization within a system and handles and manages everything corresponding to it.

There are different ways virtualization is implemented on various platforms. On operating systems like Windows and Linux, embedded hypervisors are part of the operating system's kernel. They are enabling the possibility of virtualizing software while running the operating system. However, implementations also exist where the hypervisor is running alone as an operating system of the machine. This we can find in, for example, the Xen project, where the sole purpose of the system is to virtualize software in so-called virtual machines (VMs) [4].

There are also different types of virtualization support depending on the implementation. As an example, there are differences between hardware support and IO (input and output) support. For example, one enables essential support for the virtualization of the hardware, and the other allows for the use of general input and output devices. These IO devices can be, for example, disk or console input.

In terms of terminology, we often refer to guests and hosts when discussing virtualization. Here the guest is the piece of software being virtualized by the host. There can be only one host but multiple guests in a virtualization system.

2.1.1 Full virtualization

This type of virtualization is the most commonly used form, where all instructions executed on the guest go directly to the host's hardware. If the guest wishes to access hardware like IO, memory or disk, it will trigger a trap into the hypervisor that is running underneath it all. A context switch will happen, and the hypervisor handles the request of the guest system and returns what the guest would expect or signals/handles an error. This allows the hypervisor to store data that was supposed in the guest's mind and go to a physical hard disk in a file instead. The advantage of an approach like this is that you don't need to make any changes to the guest system to make this work. You only need a precompiled executable and can run it as long as the hypervisor can handle all the relevant hardware requests. The disadvantage of this approach is that it's a lot of overhead. Triggering a trap and context switch every time you need to access IO is very time-consuming, which is why full virtualization is a fair bit slower than running the software natively on the host.

2.1.2 Paravirtalization

In paravirtualization, we have a more practical approach to virtualization than full virtualization, although it is not without its downsides. Paravirtualization works similarly to full virtualization, where we still abstract away hardware calls to a hypervisor which then handles these calls respectively. The change is that instead of going through a trap handler, we recompile the respective guest system to call the respective hypervisor calls directly rather than making it think it's accessing real hardware. In practice, this might be implemented as syscalls to the hypervisor for the different types of hardware it wants to access. This advantage is that we get rid of the overhead by having a trap handler and needing to parse the respective hardware call in the hypervisor, making the hypervisor stage significantly faster. However, the disadvantage of this approach, which might be obvious, is that you need to recompile and change the running software. This can be time-consuming since it requires familiarity with the codebase to know which function needs to be patched. Sometimes the source code is also not always available if the plan is to virtualize any proprietary software where you only have access to the binary files. This makes the patching process even harder since you would need to reverse engineer and find the relevant function before patching them.

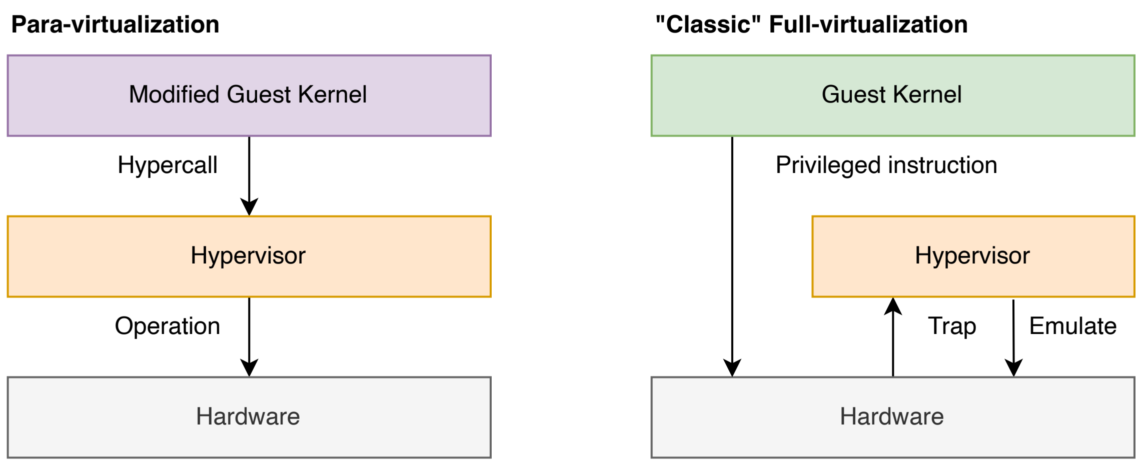

Figure 1: Full virtualization compered to paravirtualization

Source: RicoRico, CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0), via Wikimedia Commons

Ultimately, which of these two virtualization methods is best comes down to the problem you are trying to solve for running many different types of software. If you are trying to virtualize specialized software you are familiar with, you can gain a lot of performance by rewriting it to interact directly with the hypervisor.

2.2 RISC-V architecture

In recent years there have been efforts to make RISC-V an open ISA (Instruction Set Architecture) for academia and industry to provide a feature set on par with commercial and closed licensed instruction sets. Its features are free to use and extend as the user wishes without paying licensing fees or royalties [3].

2.2.1 Privilege modes

In RISC-V, as with other architectures, the concept of privilege modes limits what the processor has access to in the given moment. Different architectures can describe these modes as ”rings”, but on RISC-V, it is just referred to as privilege modes. Each hart (hardware thread which is the RISC-V term for a processor core) runs in its own privilege mode.

Table 1 shows a list of the possible privilege modes to be used on a RISC-V core. Of course, it's up to the implementer of the RISC-V core which privilege modes it includes, but it needs always to have machine mode and any of the privilege levels below.

Virtualization Mode (V)	Nominal Privilege	Abbreviation	Name	Two-Stage Translation
0	U	U-mode	User mode	Off
0	S	HS-mode	Hypervisor-extended supervisor mode	Off
0	M	M-mode	Machine mode	Off
1	U	VU-mode	Virtual user mode	On
1	S	VS-mode	Virtual supervisor mode	On

Table 1: Privilege modes with the hypervisor extension.

Source: RISC-V International, CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/), via Github

M-mode mode is the highest privileges and can be found on all RISC-V core implementations since the specification mandates it. Code running here is said to be trusted since it can access everything in the system. M-mode would usually be reserved for low-level firmware in a system with multiple privilege levels. An example is the opensbi, a platform-specific firmware developed based on the SBI specification to provide an interface to interact with M-mode to control features requiring M-mode privileges. User and supervisor hypervisor modes are then usually used for more conventional applications like operating systems. And thus have fewer privileges than software running in M-mode or S-mode respectively [2].

2.2.2 Control and Status Registers (CSRs)

Control and Status Registers, abbreviated to CSRs, are used as stated in the name to control and monitor the status of the processor. Each privilege level has its own CSRs to control and monitor the state of interrupts, exception delegation, address translation and more. By the privilege level, one hart running with lower privileges cannot access the CSRs of a higher privilege level, while the opposite is possible.

All CSR has a prefix with the privilege level they belonged to an overview of which can be found in section 2.2.1. So machine mode CSRs has prefix **"m"**, supervisor mode has **"s"** and so on.

For further details of the currently allocated CSRs in the RISC-V specification, please see the privileged architecture specification[2].

2.2.3 Hypervisor extension

In RISC-V there is a ratified hypervisor extension to the RISC-V specification [2]. It describes how a RISC-V implementation should handle and implement registers and modes corresponding to functions which makes the implementation of a hypervisor easier. Specifically, the hypervisor extension enables the possibility of running the processor in what is called VS (Virtual Supervisor) and VU (Virtual User)-mode. This is parallel to the normal supervisor mode but with fewer privileges than the normal supervisor mode. User mode mimics the same behaviour as normal user mode apart from sending traps and syscalls into VS-mode instead of HS-mode. When this extension is enabled, the normal S-mode privilege level becomes Hypervisor-extended supervisor mode which is abbreviated to HS-mode.

The main advantage the hypervisor implementation gives is that it automatically handles the translation of CSR reads and writes in the virtualized supervisor mode (VS-mode) to CSRs with the prefix **"vs"**. This enables simpler implementations and a hypervisor only needs to keep track of the state of these registers for each machine it is virtualizing.

Another part the hypervisor extension has added is dedicated trap codes for the different calls it might do like environmental calls (ecall instruction), a page fault and more. These trap codes are described in appendix B. In general, all the trap cause codes are in place to easily distinguish the virtualized user or supervisor environment from a none virtualized user and supervisor environment.

Specific interrupt registers can be controlled from the hypervisor, triggering a trap with the respective trap cause set in VS-mode. This can be used to create abstracted implementations for timers and other peripherals which the VM might expect.

An overview of the whole system implemented with a hypervisor using the extension features can be seen in figure 2.

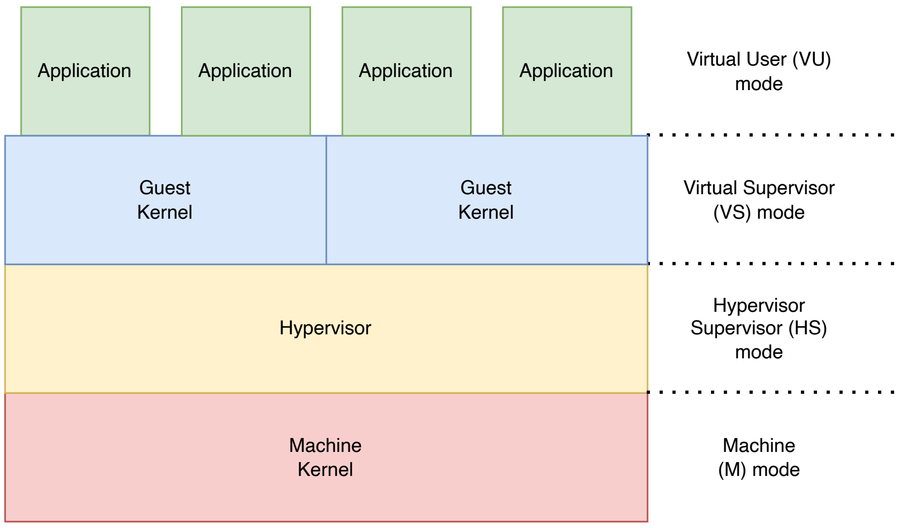

Figure 2: An overview of how the overall system would look

For more details please see the RISC-V privileged architecture specification [2].

2.3 Virtual Memory

To isolate separate memory properly between a guest and a host, there needs to be some way to protect against a program which runs as a guest to modify memory or read memory to which it is not intended to have access. This could be accomplished with the physical memory protection feature in RISC-V [2] which disallows a program that runs in a different privileged mode from accessing the memory of a higher privilege mode if configured correctly. Although this accomplishes the task of protecting memory that should not be accessed by a lower privileged mode, it does not allow us to run general software which expects access to these memory areas when they are compiled to be able to run. This is where virtual memory comes in. A concept of translating memory addresses through the memory calls through an MMU (Memory Management Unit). This allows the host system to map memory addresses that look like an ordinary program memory for the guest program to an arbitrary memory location which the host decides on. Virtual memory is an important feature which is used in many operating systems to enhance security and is essential when implementing a hypervisor for virtualization.

2.3.1 RISC-V implementation

On RISC-V, virtual memory setup is accomplished by setting the **sgatp** for supervisor mode virtual address translation or **hgatp** for use with the hypervisor extension to the root page table. In this status register, we also configure the address translation configuration. Currently, there is Sv32x4 for 32-bit and Sv39x4, Sv48x4 and Sv57x4 for 64-bit RISC-V systems as available configurations. The first number represents the number of bits used in the virtual address. The more bits that are available, the more virtual addresses we can have at the same time. The second number after the x represents translation for the hypervisor extension. The difference is that the hypervisor extension version adds more bits in the VPN (Virtual Page Number) field. A larger virtual address bit size potentially requires more storage if used at total capacity. Each page we allocate in our page table will be of size 4KiB. This is the minimum size the MMU can map. In the case of virtual memory translation for the hypervisor extension, the root page table needs to be 16KiB aligned instead of the usual 4KiB alignment, which you have in normal supervisor mode address translation [2].

Depending on which address translation configuration is chosen, the page table has a different number of levels starting from two levels with Sv32x4 to five with Sv57x4. The root page table is counted as the first level. Based on the number of levels, the virtual address is split up into sections that are used for indexes VPN for locating the physical address. Specific details for Sv39x4 will be discussed more, although more information for the other configurations is similar apart from bit width and number of page table levels. The virtual address consists of three VPN sections used to find the corresponding page table entry. The first 12 bits from LSB are called the offset, directly translated to the physical address. This is why the MMU can only map memory in 4KiB chunks.

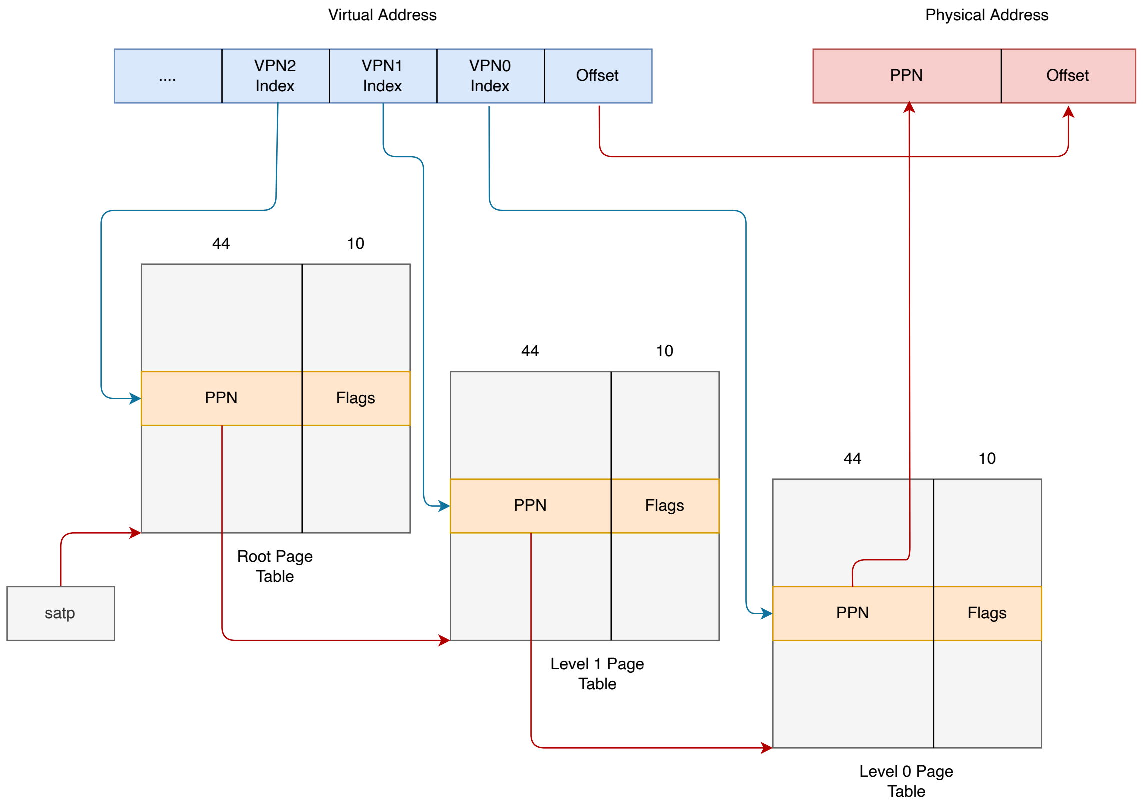

Figure 3: An overview of how virtual to physical memory translation works with Sv39x4.

The highest VPN field in the direction of MSB is used to find the first index in the root page table. This field holds the physical address of the following page table PPN and flags to indicate if the entry is valid or not. Next, the address of the PPN combined with the following VPN field in the original virtual address is used to find another index repeating the process just described until we reach our third lookup, which gives us a physical memory location the MMU is going to map in the PPN field. Here the flags also indicate permissions of the mapping like read or write access or if the lookup is accessible in user mode. If the action the program is trying to do does not match what is set in the corresponding flags, then a page fault is triggered. If not, the PPN entry is combined with the offset bits from the virtual address to create the physical memory location to which our virtual address is mapped. An overview of this process can be found in figure 3 which shows the translation steps for an Sv39x4 configuration. Red arrows symbolize physical memory addresses, while blue arrow indicates virtual addresses.

2.4 Timers

A timer is a hardware-implemented peripheral that counts up to a given number and then triggers an interrupt. They can be implemented differently depending on the platform. But they are an essential part of a system where multiple tasks must be accomplished. For example, in a hypervisor context, timers are the fundamental part of our scheduler, determining when we are switching between VMs or returning to our hypervisor to update some parameters.

2.4.1 RISC-V implementation

On RISC-V, timers do not have a specific implementation according to the specification, so implementation will differ depending on the target platform. For example QEMU the timer is based on a specific SiFive FU540-C000[16] core implementation.

2.5 Rust

Rust is a general-purpose system programming language that focuses on safety and performance. Especially safe concurrency is a critical trait that Rust prioritizes, which results in programs written in Rust being free of problems like race conditions by guaranteeing memory safety. Rust offers mechanisms for low-level memory management and high-level language features like built-in library support and a package manager. Syntax wise, it is inspired by C++, OCaml, Haskell, and Erlang [7]. Since its original release in 2010, Rust has received a wide adoption in the industry and is used by larger software companies like Amazon and Microsoft [5].

One feature that makes Rust ideal when it comes to low-level system programming is the advanced compile-time checks it does. In addition, since Rust guarantees memory safety, it does borrow checks[6] on all variables used to check for concurrency problems. This results in a program that is free from memory access faults which causes less time to be used on debugging these problems later down the road.

Rust also provides ways of skipping these checks through the use of the **unsafe** keyword. This is sometimes necessary to set up a hardware driver where you need to dereference pointers to hardcoded memory addresses. Then a safe wrapper can be created around this **unsafe** code segment, and if memory safety-related problems occur, we know they can be isolated to the **unsafe** sections.

3 Design

In this section, we will outline our hypervisor's high-level design. This allows for a more general description that does not rely on specific implementation details. The exact details surrounding the platform and the implementation details can be found in section 4.

The hypervisor will consist of two parts, our machine kernel running in machine mode and a hypervisor running in hypervisor supervisor mode. Additionally, we need software to test our hypervisor. Therefore we will also design a simple guest kernel that will act as our general supervisor mode software to be virtualised. See figure 2 for a general overview of a hypervisor architecture.

3.1 Machine Kernel (M-Mode)

When the RISC-V core does a system reset, the program counter is set to a known value and instructions at that memory location are fetched and executed. The system is now at an unknown state, and the Entrypoint mark in figure 4. Afterwards, we proceed to System Initialization, where we can configure our registers and set all the necessary CSRs RISC-V expects. All exceptions apart from the timer and environmental call from HS mode are delegated to the hypervisor. This is the stage where we also configure peripherals that will be used. In this design, we need a timer, which is also initialised. After system initialisation, we hand off execution to our hypervisor entry point and continue to run code in HS-mode.

The core needs to sometimes return to machine mode to handle tasks that require machine mode privileges. This includes handling timer interrupts and environmental calls from the hypervisor. The timer interrupt is dealt with and propagated to the hypervisor by triggering the respective CSR. For this hypervisor design, we only need an interface to disable and enable global and timer interrupts from the hypervisor. This is required to have the ability to disable these functions when the hypervisor code enters a critical section.

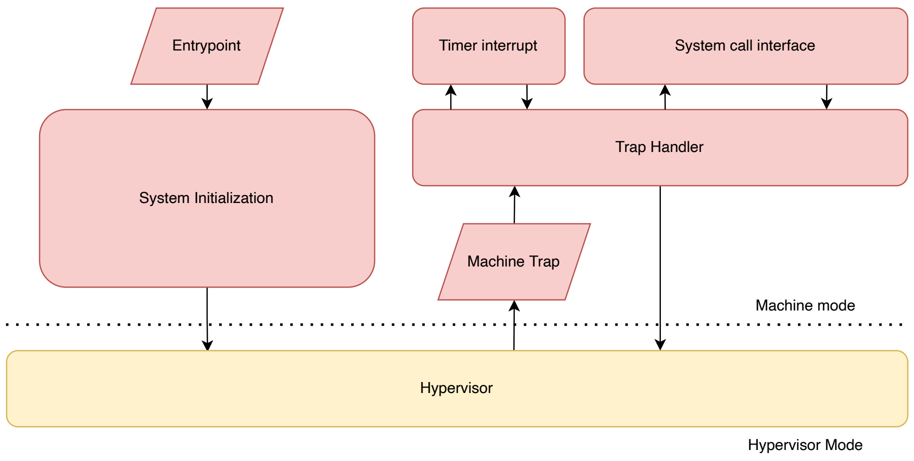

Figure 4: Overview of the general design of the machine mode kernel

3.2 Hypervisor (HS-Mode)

Picking up from where our machine kernel handed us off, the purpose of the hypervisor component is to manage and set up the virtualisation aspect of the system. Here the guest's memory is set up with the help of virtual memory, and necessary hardware interfaces are directly mapped. There also needs to be a guest setup stage where the essential structure for the virtual machine is set up, and the guest kernel is loaded into the virtual memory initialised for the guest. After the guest is fully set up, we hand off execution to the guest running in virtual supervisor mode. A high-level overview of the hypervisor design can be found in figure 5.

After setting up the guest, there still needs to be interactions with the hypervisor. This can be the guest trying to access memory it does not have access to and thus triggering a page fault. Alternatively, it can also be environment calls from the guest called hypercalls as described as paravirtualisation, see section 2.1.2. In this case, the hypervisor will have a hypercall interface for SBI timer calls as defined in the SBI specification [9]. Hardware timer interrupts are then regularly triggered on the hypervisor, which is combined with the parameters of the guest SBI timer call to initiate an emulated timer interrupt for the guest. This theoretically allows us to have as many timers interrupts for guests as we want. This hardware timer interrupt section can be extended further. In a more complex hypervisor, this would be the ideal place to implement a scheduler to switch between guests. However, that is outside the scope of this design. An overview of the hypervisor trap handler interface can be found in figure 6.

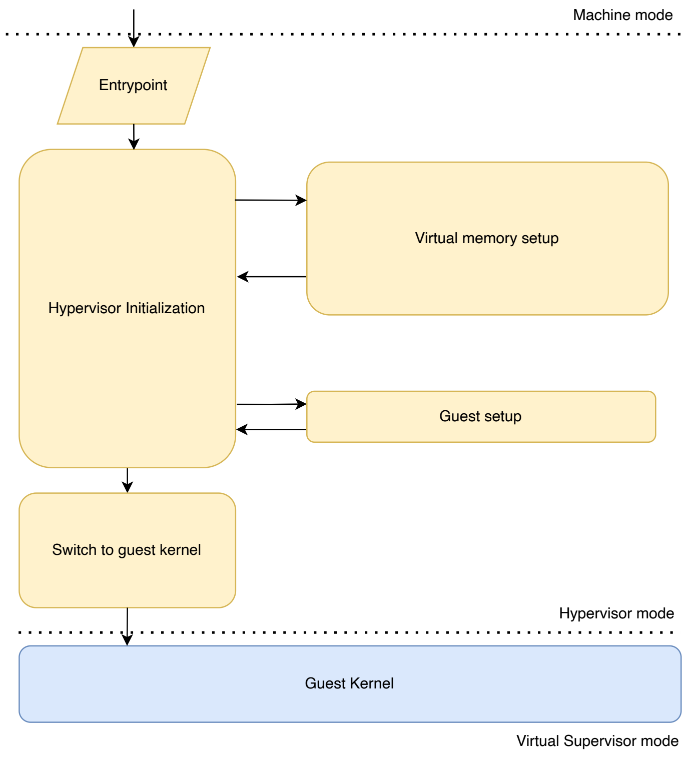

Figure 5: Overview of the initialization of the planned hypervisor

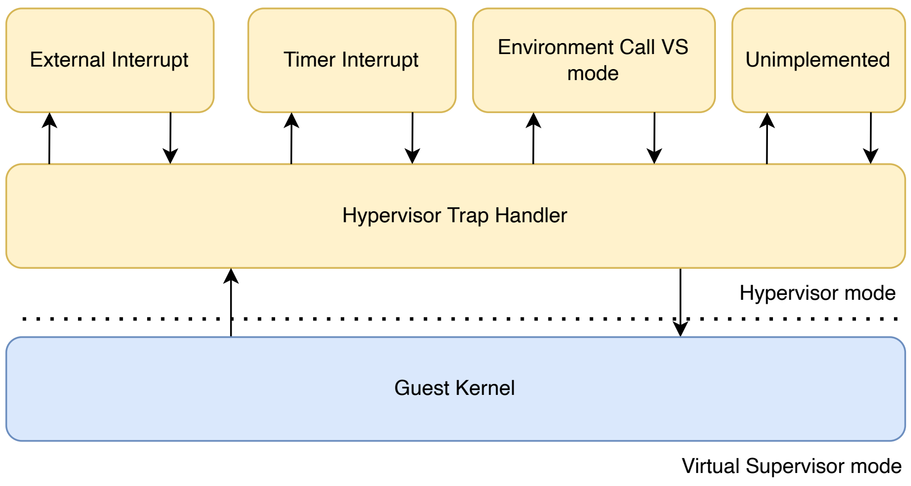

Figure 6: Overview of the trap handler for the hypervisor

3.3 Guest Kernel (VS-Mode)

Although not necessary for designing the hypervisor itself, we create a simple guest kernel to be able to test if our hypervisor is working as expected quickly. This design is generic and should assume that it runs in supervisor mode with a machine mode bootloader with an SBI interface to control interfaces like timers. This kernel follows the generic operating system design principles that one would think of creating for a low-level target.

Execution starts by code starting the default expected entry point, which is usually a predefined memory location depending on the target architecture. Afterwards, we need to initialise our kernel with the necessary interfaces, specifically virtual memory and timer interrupt. The timer setup is done by relying on the SBI timer extension found in the specification [9]. Following the setup, our kernel goes into an infinite loop, waiting for the timer interrupt to happen. The kernel trap handler handles specifically the timer interrupt requested by the SBI call and notifies us that everything is working as expected. An overview of the guest kernel can be found in figure 7.

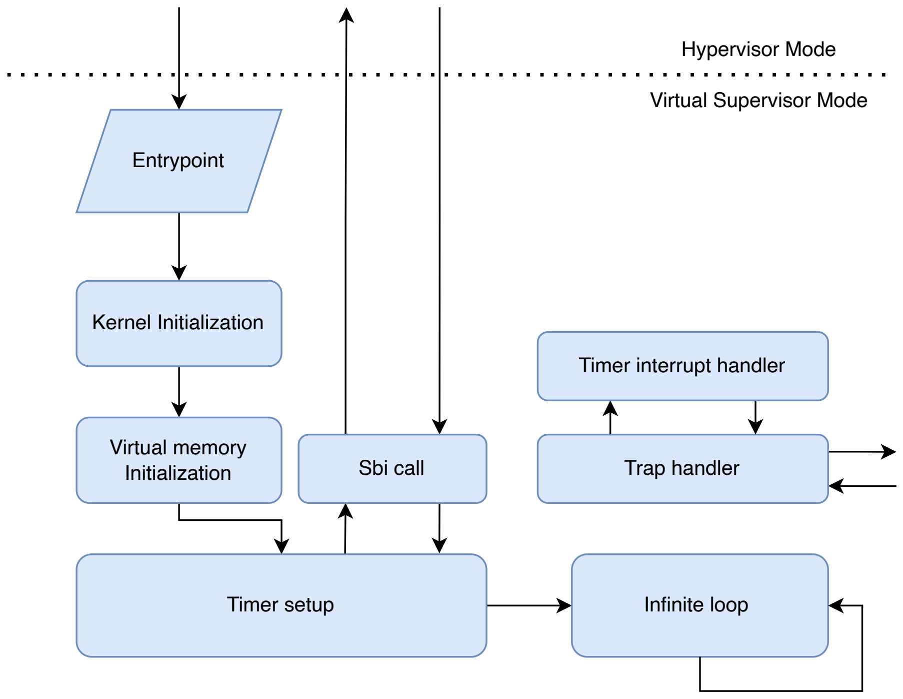

Figure 7: Overview of the general designed of the planed guest kernel.

4 Implementation

As stated in section 1, this master thesis aims to document, evaluate and implement a hypervisor with the newly ratified hypervisor extension for the RISC-V instruction set architecture. Additionally, we want to consider Rust as a system programming language on the RISC-V platform. As a basis, this implementation will take inspiration from Takashi Yoneuchi's unfinished rust hypervisor project rvvisor[18] which was based on an earlier draft of the hypervisor extension specification. The generic outline of the design of the hypervisor and guest kernel can be found in section 3.

As a target platform, we will use QEMU version 7.0.50 to emulate a single-core RISC-V system with 512MB memory and a standard Virt interface. Since we want to make implementation generic, we will not describe the detailed implementation of the drivers towards the VirtIO interface since this will change if the target system changes.

This section will use abbreviations defined under section 2.2. Especially the abbreviations for the different privilage modes in table 1.

There will also be relevant code snippets from the implementation code itself. The whole codebase is not going to be included here but can be found on the following GitHub repository[13]

The hypervisor will consist of the following components, and the implementation of each will be described in detail:

M-Mode:

- Bootloader to initialize the system and jump to HS-mode.
- Environmental call interface controls the machine timer and interrupts from HS-mode.

HS-Mode:

- Virtual memory controller to isolate virtual machine's memory from each other.
- SBI standardized environmental call for virtual timers to the virtualized machines.
- Setting up the guest's memory space and loading the guest kernel before switching to it.

4.1 Rust and RISC-V

Since Rust is still evolving as a programming language, we have two main branches of the language that can be used. One is stable, and the other is nightly. As can be inferred from the name itself, stability is standardized and not changing and nightly have features that the maintainer is subject to change or deprecate later. Depending on what you want to accomplish, using some of these nightly features might be needed when doing system programming.

Rust uses LLVM [8] as a compiler backend. That means the specific Rust compiler must only compile the rust code to LLVM IR, an intermittent representation highly portable to different architectures. Support for RISC-V in Rust is mainly depending on the LLVM backends RISC-V architecture support which is well supported.

An additional benefit of Rust is the built-in package manager cargo which makes managing dependencies and setting up build environments reasonably simple. This contributes to making iteration time faster, which makes the development process smoother.

4.1.1 Macros and assembly abstracting

In system programming, we need to do memory accesses or issue assembly instructions directly, which Rust deems to be an unsafe behaviour. Since we want to take advantage of the Rust borrow-

Listing 1: Control Status Register macro hypervisor/src/riscv/csr/macros.rs

```rust
macro_rules! define_read {
    ($csr_number:expr) => {
        pub fn read() -> usize {
            unsafe {
                let r: usize;
                asm!("csrrs␣{0},␣{csr},␣x0",
                out(reg) r,
                csr = const $csr_number,
                options(nostack)
                );
                r
            }
        }
    };
}

macro_rules! define_write {
    ($csr_number:expr) => {
        pub fn write(v: usize) {
            unsafe {
                asm!("csrrw␣x0,␣{csr},␣{rs}",
                rs = in(reg) v,
                csr = const $csr_number,
                options(nostack)
                );
            }
        }
    };
}
```

Listing 2: Controll Status Register definition example hypervisor/src/riscv/csr/misa.rs

```rust
define_read!(0x301);
define_write!(0x301);
pub const HV: usize = 1 << 7;
```

ing checks on compile, we need to wrap these unsafe calls in safe functions that do the necessary checks. Keeping these segments concise helps prevent memory-related bugs from happening.

Two instructions that are going to be used a lot are **csrrs** and **csrrw** to facilitate writes and reads to CSRs (Control Status Registers). Since there are a lot of different CSR registers and numbers we need to use we can wrap this unsafe call Rust macros **define_read** and **define_write** which can be seen in Listing 1. Defining these macros in a file will implement the functions read and write with respective CSR id. Note that the unsafe section here only deals with passing the input and output from the assembly instruction, and there is no possibility for undefined behaviour. It will always read and write to a CSR and return a value if expected.

In a separate file, we can then call the macros with the specific id of the CSR. In listing 2 we define the call for **misa** CSR. Afterwards, reading and writing to all the CSRs we have defined can be

Listing 3: Controll Status Register call example hypervisor/src/mkernel.rs:81

```rust
let misa_state = riscv::csr::misa::read();
riscv::csr::misa::write(misa_state | riscv::csr::misa::HV);
```

done safely, as shown in the example in listing 3 where we read and set a constant using OR we have defined in the file and then write back the original value.

4.2 Machine Kernel (M-Mode)

In this section, we will explain the detailed implementation of the machine kernel design found in section 3.1, which is the software running in the highest privilege mode. In this privilege mode, we are mainly concerned about doing the necessary setup of the system before handing it off to the hypervisor running in HS-mode. We also handle required hardware interrupts and system calls from the hypervisor mode to control these interrupts. An overview of the functions and program flow can be found in figure 8.

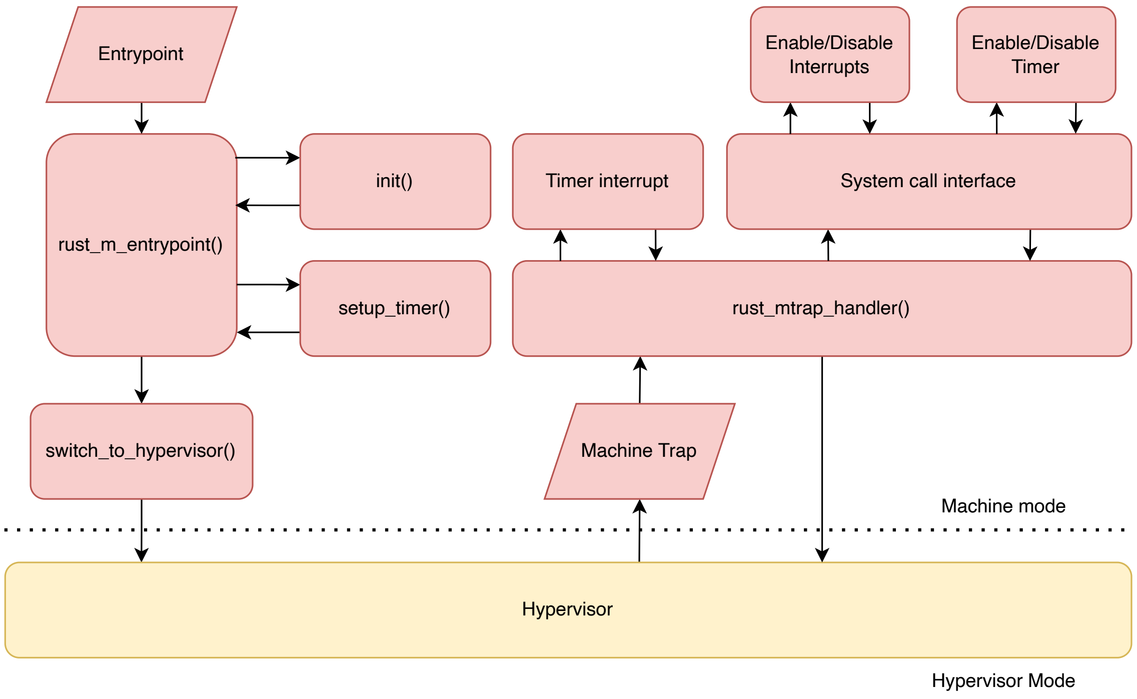

Figure 8: Overview of the machine mode kernel

4.2.1 Bootstrapping

As part of every bare-metal software implementation, we need some bootstrapping before we can run our Rust code. QEMU starts program execution on address 0x8000_0000. We, therefore, tell our linker script to include some assembly code at the start of our program section, so it is the first instructions that QEMU executes, which can be found in listing 4. The linker script can be found in appendix A.

Listing 4: m_entrypoint from hypervisor/src/boot.S

```
    la a0, _trapframe
    csrw mscratch, a0
    # load stack addr
    la     sp, _m_stack_end
    # jump to rust code
    tail       rust_m_entrypoint
```

We set up a trap frame and write this into **mscratch** in case we want to peek at what went wrong if we get an unexpected trap. Next, we need to set up our stack to store local variables. This is done by loading the address we have allocated to our stack into the sp register (stack pointer). The core is now ready to start executing our Rust code, and we jump to **rust_m_entrypoint** which is explicitly not mangled and exported as a C style function to increase compatibility with the linker. All of this is encapsulated into **Entrypoint** in figure 8

4.2.2 Initialization

The primary purpose of the machine kernel is to function as a simple bootloader for our hypervisor and an interface layer to change components that require machine mode privileges. We, therefore, only do the necessary setup before handing it off to our hypervisor.

Since we are working with bare-metal code, there is no pre-defined way our system should behave when it panics. We, therefore, need to define this, which can be seen in listing 5. We can see that when panic is called, we invoke **print** and **println**, which is another macro defined as part of the UART driver, which will be described later. We use these to print the information Rust makes available through its core panic library. Finally, at the end of the panic call **abort** is called, which forces the core to wait indefinitely for an interrupt, enabling us to attach and debug the core if we wish to get more information.

Listing 5: Panic handling definition hypervisor/src/debug.rs

```rust
#[panic_handler]
fn panic(info: &core::panic::PanicInfo) -> ! {
    print!("abort:␣");
    if let Some(p) = info.location() {
        println!(
            "line␣{},␣file␣{}:␣{}",
            p.line(),
            p.file(),
            info.message().unwrap()
        );
    } else {
        println!("no␣information␣available.");
    }
    abort();
}

#[no_mangle]
extern "C" fn abort() -> ! {
    loop {
        unsafe {
            asm!("wfi", options(nostack));
        }
    }
}
```

After the boot is done as described in section 4.2.1, the first part of our entry code **rust_m_entrypoint** calls **init** where the function is wrapped in and result checker, which is a Rust language feature to make error handling easier, which can be seen in listing 6. We can then have an error propagate from init, and if it is not handled, it will call the **panic** macro, which aborts execution in an expected way and prints which file and line it failed at. One important thing to note here is that **panic** cannot display any information to our screen before the output interface is set up, which is a UART interface. Therefore the UART interface initialization is one of the first functions called in **init**, see listing 7.

Listing 6: Code snippet to show the call and error handling of init hypervisor/src/mkernel.rs

```rust
pub extern "C" fn rust_m_entrypoint(hartid: usize, opqaue: usize) -> ! {
    if let Err(e) = init() {
        panic!("Failed␣to␣initialize.␣{:?}", e);
    };
    (...)
    if let Err(e) = setup_timer() {
        panic!("Failed␣to␣initialize␣timer.␣{:?}", e);
    };
    switch_to_hypervisor(hypervisor::entrypoint as unsafe extern "C" fn());
}
```

The UART interface is platform-specific for the QEMU Virt interface, so the specific implementation is not described but can be found in the corresponding source file in the repository[13]. Within this implementation, the macros **print** and **println** are defined as described in section 4.1.1 which gives us a way to print characters to the respective output interface. Following the initialization of our UART interface, we proceed to configure our control status registers (section 2.2.2) which have already been defined with their corresponding ids described in section 4.1.1. The first CSR we configure is **medeleg** which allows us to delegate exceptions to the hypervisor supervisor mode (section 2.2.1). Here all exceptions are delegated apart from environmental calls from the hypervisor mode so the hypervisor can interact with the machine kernel. The **mideleg** CSR is configured to forward all supervisor external timer and software interrupts to the supervisor mode, which is by default delegated to machine mode. One optional CSR here is **misa** which has an extension field that allows the software to turn off different RISC-V extensions implemented on the core. All of the supported fields should be enabled per the RISC-V supervisor specification[2], but it is enabled to be sure. To handle traps, we also need to tell our system where to jump when a trap is caused. This is done by setting the **mtvec** CSR to our trap handler. Lastly, the **satp** CSR is set to zero, which makes sure virtual address translation is turned off for the HS-mode since this would only cause an unnecessary performance impact on the code running our HS-mode since we don't want to isolate the hypervisor code from accessing the machine mode memory. If then no errors have occurred the **init** function returns **Ok** since our function expects either a **Ok** or **Err** type.

Listing 7: Code snippet mkernel init hypervisor/src/mkernel.rs

```
pub fn init() -> Result<(), Error> {
    // init UART
    uart::Uart::new(memlayout::UART_BASE).init();

    // medeleg: delegate synchoronous exceptions
    //  except for ecall from HS-mode (bit 9)
    riscv::csr::medeleg::write(
        0xffffff ^ riscv::csr::medeleg::HYPERVISOR_ECALL );

    // mideleg: delegate all interruptions
    riscv::csr::mideleg::write(
        riscv::csr::mideleg::SEIP |
        riscv::csr::mideleg::STIP |
        riscv::csr::mideleg::SSIP);
    // enable hypervisor extension
    let misa_state = riscv::csr::misa::read();
    riscv::csr::misa::write(misa_state | riscv::csr::misa::HV);
    assert_eq!(
        (riscv::csr::misa::read()) & riscv::csr::misa::HV,
        riscv::csr::misa::HV
    );

    // mtvec: set M-mode trap handler
    riscv::csr::mtvec::set(&(trap as unsafe extern "C" fn()));
    assert_eq!(
        riscv::csr::mtvec::read(),
        (trap as unsafe extern "C" fn()) as usize
    );
    riscv::csr::satp::write(0x0); // satp: disable paging
    Ok(()) // Return no error
}
```

The next initialization step, seen in listing 6 is to initialize our hardware timer, which relies on a platform-specific CLINT implementation described in section 2.4.1 so as with UART, the details will not be described. The only generic step apart from the platform-specific implementation is to set the flag in the **mie** CSR to enable our machine timer to interrupt.

4.2.3 Switching to hypervisor supervisor mode

Following the initialization of our general core and timer, the core is ready to jump to the hypervisor by calling the function **switch_to_hypervisor** the code of which can be seen in listing 8. To do a proper switch to hypervisor mode, the **mpp** CSR to CPU mode supervisor and **mpv** CSR to Virtualization Mode Host. Since the hypervisor mode is just an extension of the normal supervisor mode, the value controls the distinction between virtualized supervisor mode and hypervisor supervisor mode in the **mpv** CSR. Please see table 1 for all the available modes. The CPU also needs to know where to start to execute after we invoke the context switch. This is achieved by setting **mepc**, which is the machine exception program counter, to our hypervisor entry address. The last thing we need to configure before we can jump into the hypervisor supervisor is to configure the physical memory protection (PMP) to allow our hypervisor to access the program memory. In this implementation, we give the hypervisor access to all memory and then disable PMP since we don't need to segregate the memory. This is achieved by configuring the CSRs **pmpcfg0** and **pmpaddr0** with the assembly code seen in listing 8 from line 10 to 13. Finally, we invoke the trap return instruction for this mode **mret** which sets the program counter and privilege mode correctly based on what we configured earlier.

Listing 8: Code for jumping to the hypervisor hypervisor/src/mkernel.rs

```
1  pub fn switch_to_hypervisor<T: util::jump::Target + Copy>(target: T) -> ! {
2          riscv::csr::mstatus::set_mpp(riscv::csr::CpuMode::S);
3          riscv::csr::mstatus::set_mpv(riscv::csr::VirtualzationMode::Host);
4          riscv::csr::mepc::set(target);
5          assert_eq!(
6              riscv::csr::mepc::read(),
7              target.convert_to_fn_address()
8          );
9          unsafe{
10             asm!("li␣t4,␣31");
11             asm!("csrw␣pmpcfg0,␣t4");
12             asm!("li␣t5,␣(1␣<<␣55)␣-␣1");
13             asm!("csrw␣pmpaddr0,␣t5");
14         }
15         riscv::instruction::mret();
16     }
17
```

4.2.4 Trap handling

Even though we have delegated most of the exceptions to our hypervisor mode trap handler, there are still cases that the machine kernel trap handler needs to handle. As described in section 3.1 and figure 8, we want the hypervisor to be able to control the hardware timer and interrupt from its privilege mode. We, therefore, need a system call interface that handles environmental calls from the hypervisor. Additionally, on our platform, the hardware timer can only trigger timer interrupts in machine mode. If our hypervisor receives these interrupts, we also need to propagate them manually from machine mode.

In section 4.2.2 the **mtvec** CSR was set to point to **trap**. This implementation points to the following assembly code in listing 9. There needs to be some assembly before we can call **rust_mtrap_handler** because we need to save the state of registers so we can put the CPU into the same state before we return from the trap. Using the assembly macro **save_gp**, we save the current register values to our local context retrieved from **mscratch**. After the trap frame is saved, we prepare a stack for the trap handler and load CSRs we need as function arguments. When the Rust trap handler returns, we expect a return value which will be used to set **mepc** which will dictate where the code continues executing after we return from this trap. Lastly, we restore the saved trap frame with the modifications we might have done and exit our trap with **mret**.

Listing 9: trap from hypervisor/src/mkernel.S

```
.macro load_gp i, base
    ld x\i, ((\i)*8)(\base)
.endm

.macro save_gp i, base
    sd x\i, ((\i)*8)(\base)
.endm
trap:
    csrrw t6, mscratch, t6
    .set i, 0
    .rept 31
        save_gp %i, t6
        .set i, i+1
    .endr
    mv t5, t6
    csrr t6, mscratch
    save_gp 31, t5
    csrw mscratch, t5

    csrr    a0, mepc
    csrr    a1, mtval
    csrr    a2, mcause
    csrr    a3, mstatus
    csrr    a4, mscratch
    la      sp, _mintr_stack_end
    call    rust_mtrap_handler
    csrw    mepc, a0
    csrr    t6, mscratch

    # restore GPRs
    .set i, 1
    .rept 31
        load_gp %i, t6
        .set i, i+1
    .endr
    mret
```

The main logic of the trap handler happens in the Rust part of the handler, snippets of which can be found in listing 10. Here the arguments prepared in listing 9 are parsed into the function, and as long as we return a valid program counter value, we can do the rest of the trap handling through Rust. Using the value of **mcause**, we can figure out what type of trap is called and handle it accordingly. Here we first match if the trap is an interrupt or not and then look at the exception code. A complete list of the trap cause codes can be found in appendix B. Suppose the trap cause is neither a machine timer interrupt nor an environment call from HS-mode. In that case, we call the **unimplemented** macro, which causes a **panic** so we can implement a handler for any unknown trap cause we might find.

For our timer interrupt, we propagate this to HS-mode by setting the HS-mode timer interrupt pending bit in **mip** and enabling the hs-mode timer interrupts in **mie**. This will trigger a timer interrupt trap properly in HS-mode. We also do the platform-specific timer configuration to set the time for when our next timer interrupt is triggered. The other exception handling we have is our environmental call interface, which is to allow the code running in HS-mode to either disable or enable all interrupts or enable or disable timer interrupts. We use the saved trap frame from our hypervisor to retrieve the argument stored in register a0. As long as it's a recognized argument, it will return with a value of zero in a0, and if it's an unknown environmental call, it returns with a return of one. Reading and writing to the trap frame is, as one can see, wrapped in unsafe brackets. This is due to random memory access where Rust cannot ensure memory safety; therefore, we must be cautious in inspecting the edge cases in how we access our trap frame. When we are done handling our environmental call, we return **mepc** + 0x4. This is because we need to skip one instruction ahead. If not, we would endlessly do an environmental call. Otherwise, we return the same value of **mepc** as we received into **rust_mtrap_handler**.

Listing 10: Code for trap handler hypervisor/src/mkernel.rs

```rust
#[repr(C)]
#[derive(Clone, Copy, Debug)]
pub struct TrapFrame {
    pub regs: [usize; 32],  // 0 - 255
    pub pc: usize,          // 256
}
#[no_mangle]
pub extern "C" fn rust_mtrap_handler(
    mepc: usize,            /* a0 */
    mtval: usize,           /* a1 */
    mcause: usize,          /* a2 */
    mstatus: usize,         /* a3 */
    frame: *mut TrapFrame, /* a4 */) -> usize {
    let is_async = ((mcause >> 63) & 1) == 1;
    let cause_code = mcause & 0xfff;
    if is_async {
        match cause_code {
            7 => {
                riscv::csr::mip::set_stimer();
                riscv::csr::mie::enable_s_mode_hardware_timer();
                let timer = clint::Clint::new(0x200_0000 as *mut u8);
                timer.set_timer(0,
                    timer.get_mtime() + M_MODE_TIMER_VALUE
                );
            }
            _ => {
                unimplemented!("Unknown␣M-mode␣interrupt␣id:␣{}"
                , cause_code);
            }
        }
    } else {
        match cause_code {
            9 => {
                let hypervisor_frame = unsafe{*frame.clone()};
                let a0 = hypervisor_frame.regs[10];
                let mut result = 0;
                match a0 {
                    m_mode_calls::ENABLE_ALL_INTERRUPTS => {
                        unsafe{riscv::interrupt::enable();}
                    }
                    m_mode_calls::DISABLE_ALL_INTERRUPTS => {
                        unsafe{riscv::interrupt::disable();}
                    }
                    m_mode_calls::ENABLE_ALL_TIMERS => {
                        riscv::csr::mie::enable_m_mode_hardware_timer();
                    }
                    m_mode_calls::DISABLE_ALL_TIMERS => {
                        riscv::csr::mie::clear_m_mode_hardware_timer();
                    }
                    _ => {
                        result = 1;
                    }
                }
                unsafe {(*frame).regs[10] = result;}
                return mepc + 0x4;
            }
            _ => {
                unimplemented!("Unknown␣M-mode␣Exception␣id:␣{}"
                , cause_code);
            }
        }
    }
    return mepc;
}
```

4.3 Hypervisor (HS-Mode)

In this section, we will describe the detailed implementation of the hypervisor kernel described in the design section 3.2 which runs in hypervisor supervisor mode also known as HS-mode. This is where everything in regards to virtualization is handled. This section will have two main parts, the initialization part and the trap handling part, an overview of which can be found in figure 9 and 10 respectively.

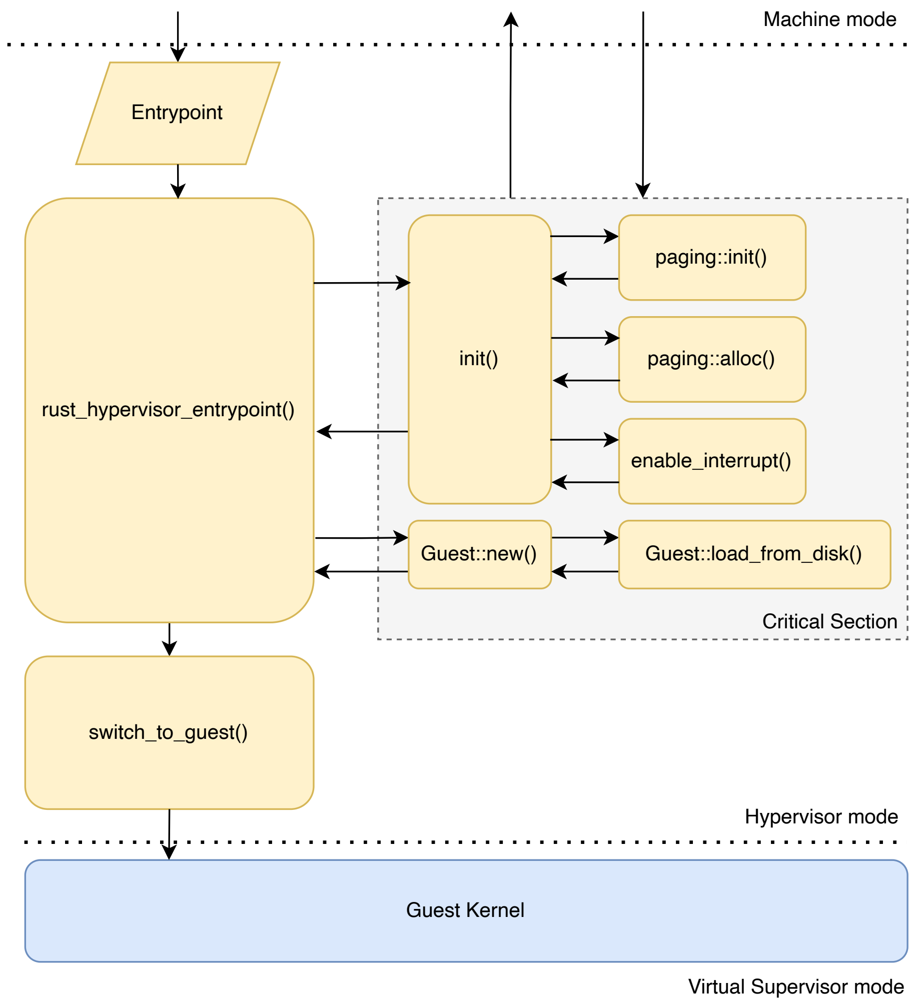

Figure 9: Overview of the initialization of the hypervisor

4.3.1 Initialization

After the system initialization in M-mode described in section 4.2, the function we enter is **rust_hypervisor_entrypoint** which can be found in listing 11. As with the mkernel initialization in section 4.2.2, we use the built in result type in rust to do error handling on our **init** function. Notice here that the **init** function is wrapped in a **riscv::interrupt::free** found in listing 12 which wraps our critical section function by disabling timers before calling the respective function and then reenabling them afterwards. The functions **disable_timers** and **enable_timers**, the code of which can be found in listing 13, are wrappers for environmental calls. The handling of those is described in section 4.2.4.

Listing 11: Entry point code for hypervisor hypervisor/src/hypervisor.rs

```
#[no_mangle]
pub fn rust_hypervisor_entrypoint() -> ! {
    if let Err(e) = riscv::interrupt::free(|_| init()) {
        panic!("Failed␣to␣init␣hypervisor.␣{:?}", e)
    }
    let mut guest = riscv::interrupt::free(|_| Guest::new("guest01"));
    riscv::interrupt::free(|_| guest.load_from_disk());
    switch_to_guest(&guest);
}
```

Listing 12: Critical section handler hypervisor/src/riscv/interrupt.rs

```
pub fn free<F, R>(f: F) -> R where F: FnOnce(&CriticalSection) -> R, {
    m_mode_calls::disable_timers();
    let r = f(unsafe { &CriticalSection::new() });
    m_mode_calls::enable_timers();
    return r; }
```

Listing 13: Custom environmental calls to M-mode hypervisor/src/m_mode_calls.rs

```
pub const DISABLE_ALL_INTERRUPTS: usize = 0x01;
pub const ENABLE_ALL_INTERRUPTS: usize  = 0x02;
pub const DISABLE_ALL_TIMERS: usize     = 0x03;
pub const ENABLE_ALL_TIMERS: usize      = 0x04;
pub fn disable_interrupts() {
    riscv::instruction::ecall_with_args(DISABLE_ALL_INTERRUPTS,0x0,0x0,0x0);
}
pub fn enable_interrupts() {
    riscv::instruction::ecall_with_args(ENABLE_ALL_INTERRUPTS,0x0,0x0,0x0);
}
pub fn disable_timers() {
    riscv::instruction::ecall_with_args(DISABLE_ALL_TIMERS,0x0,0x0,0x0);
}
pub fn enable_timers() {
    riscv::instruction::ecall_with_args(ENABLE_ALL_TIMERS,0x0,0x0,0x0);
}
```

After timer interrupts are disabled, we enter into the **init** function found in listing 14 which does the necessary configuration of CSRs and initializes modules the hypervisor depends on. The first modules we initialize are **paging**, which manages our virtual memory for our guest, and **virtio**, the platform-specific interface we use to load the guest kernel into memory. The virtual memory and page implementation is described in more detail in section 4.3.2. The CSR **hedeleg** is configured to propagate exceptions to the guest like environmental calls from virtual user mode, breakpoints and instruction address misalignment. Additionally, instruction, load and store page faults are also propagated to the guest. For interrupts, we configure the **hideleg** CSR to propagate external timer and software interrupt to the guest. **hvip** is set to zero to make sure we have no interrupts pending for the guest. We configure **stvec** to our trap handler in the hypervisor so if a trap occurs, the CPU knows where to execute. Next, we allocate a page where the address is set to the **sscratch** CSR to save the trap frame when it occurs. The function **enable_interrupt** enables timer and external interrupts by setting the relevant bits in the **sie** CSR. Further global HS-mode interrupts are enabled by setting the relevant bit in **sstatus**.

Listing 14: Initialization function for hypervisor hypervisor/src/hypervisor.rs

```
pub fn init() -> Result<(), Error> {
    paging::init();
    virtio::init();
    riscv::csr::hedeleg::write(riscv::csr::hedeleg::INST_ADDR_MISALIGN
        | riscv::csr::hedeleg::BREAKPOINT
        | riscv::csr::hedeleg::ENV_CALL_FROM_U_MODE_OR_VU_MODE
        | riscv::csr::hedeleg::INST_PAGE_FAULT
        | riscv::csr::hedeleg::LOAD_PAGE_FAULT
        | riscv::csr::hedeleg::STORE_AMO_PAGE_FAULT);
    riscv::csr::hideleg::write(riscv::csr::hideleg::VSEIP
        | riscv::csr::hideleg::VSTIP
        | riscv::csr::hideleg::VSSIP
    );
    riscv::csr::hvip::write(0);
    riscv::csr::stvec::set(&(trap as unsafe extern "C" fn()));
    let trap_frame = paging::alloc();
    riscv::csr::sscratch::write(trap_frame.address().to_usize());
    enable_interrupt();
    Ok(())
```

4.3.2 Heap and Virtual Memory

As mentioned in 4.3.1, part of the initialization step was to set up our paging memory. The init function can be found in listing 15. This creates the basis for a simple heap where we intend to be able to allocate pages. The base address for the simple heap is therefore made sure to be page-aligned according to the requirements for the virtual memory, see section 2.3.1.

Listing 15: Initialization of paging hypervisor/src/paging.rs

```rust
pub const HEAP_SIZE: usize = 64 * 1024; // 64KiB
pub const PAGE_SIZE: u16 = 4096;
pub unsafe fn elf_start() -> usize {
    unsafe { &_elf_start as *const usize as usize }
}
pub unsafe fn elf_end() -> usize {
    unsafe { &_elf_end as *const usize as usize }
}
pub unsafe fn heap_start() -> usize {
    (elf_end() & !(0xfff as usize)) + 4096
}
pub unsafe fn heap_end() -> usize {
    heap_start() + HEAP_SIZE
}
static mut base_addr: usize = 0;
static mut last_index: usize = 0;
static mut initialized: bool = false;
pub fn init() {
    unsafe {
        base_addr = (heap_end() & !(0xfff as usize)) + 4096;
        last_index = 0;
        initialized = true;
    }
}
```

We can then begin to set up a page table according to section 2.3.1. This will map our guest's program memory to an allocated section on this simple page heap. A root page table can then be created by allocating a 16KiB page with the function **alloc_16**. This in turn calls the function **alloc** and makes sure to return a 16KiB page which is also 16KiB aligned to fit with the specification requirements. The **alloc** in turn makes sure to allocate a 4KiB size page which is zeroed out. The code of which can be found in listing 16

Listing 16: Allocation of 4KB and 16KB pages hypervisor/src/paging.rs

```rust
pub fn alloc() -> Page {
    unsafe {
        if !initialized {
            panic!("page␣manager␣was␣used␣but␣not␣initialized");
        }

        last_index += 1;
        let addr = base_addr + (PAGE_SIZE as usize) * (last_index - 1);
        if addr > DRAM_END {
            panic!("memory␣exhausted;␣0x{:016x}", addr)
        }
        let p = Page::from_address(PhysicalAddress::new(addr));
        p.clear();
        p
    }
}

/// Makes sure the root page follows a 16KiB boundry
pub fn alloc_16() -> Page {
    let mut root_page = alloc();
    while root_page.address().to_usize()&(0b11_1111_1111_1111 as usize) > 0 {
        root_page = alloc();
    }
    alloc();
    alloc();
    alloc();
    root_page
}
```

Next, we want to map virtual addresses to physical memory in our page table. This is done by the **map** function. Which creates an Sv39x4 page table entry and inserts it into our page table. The specific details on how the page table entry is made and inserted into a page table are generic and the same in most implementations. Therefore the details will not be discussed here, only the details which are specifically relevant to this implementation, but the code can be found in appendix C.

4.3.3 Guest Setup

Following the initialization, in 4.3.1 we proceed to set up our guest so we can properly virtualize it. As we can see in figure 9 and listing 11 **Guest::new** and **Guest::load_from_disk** are still being executed from our critical section and we are thus still able to perform memory operations without being interrupted by a timer interrupt. The first step in **Guest::new** (code in listing 17)is to create a root page table for the guest in question, which allows us to switch between virtual memory mapping for different guests easily. **prepare_gpat_pt** takes care of the initial creation of the root page table and directly maps the UART device and allocates a memory region for the guest's DRAM with the respective mapping. For details the of **prepare_gpat_pt** can be found in appendix D.

Listing 17: Creating of a new guest hypervisor/src/guest.rs

```
pub fn new(name: &'static str) -> Guest {
    let root_pt = prepare_gpat_pt().unwrap();
    let hgatp = riscv::csr::hgatp::Setting::new(
        riscv::csr::hgatp::Mode::Sv39x4,
        0,
        root_pt.page.address().to_ppn(),
    );

    Guest {
        name: name,
        hgatp: hgatp,
        sepc: memlayout::GUEST_DRAM_START
    }
}
```

Following the creation of the guest, we can now load the guest kernel into the allocated memory through **load_from_disk**. This implementation will not be described since it is platform-specific to QEMU and the VirtIO interface. From a high-level view, it takes an ELF of a kernel we have provided to QEMU, parses it and proceeds to load it into the allocated guest memory so it can be executed.

4.3.4 Guest Switching

After preparing in section 4.3.3 the guest is now ready to be switched to by calling the function **switch_to_guest**. The code is found in listing 18 with reference to our guest struct. With the root page table in the provided guest struct, we configure the **hgatp** CSR. This will cause the MMU to automatically map memory calls through our root page table when the CPU is set to VS-mode. Ultimately, we create just a generic struct to manage our guests. The **hfence_gvma** instruction is then called to flush the cache, so no old page table entries reside there. Next, we set the virtualization mode to guest in the **hstatus** CSR and make sure supervisor mode is still selected as our privilege level in the **sstatus** CSR. The current program counter value stored in our guest struct is loaded into the **sepc** CSR to tell the CPU which address to jump to after a return is called. Finally, we call the **sret** instruction to do the return. If everything is done correctly, the CPU should now be executing code the guest's program code in an isolated environment virtualized from the rest of the system.

Listing 18: Switch to guest hypervisor/src/hypervisor.rs

```rust
pub fn switch_to_guest(target: &Guest) -> ! {
    // hgatp: set page table for guest physical address translation
    riscv::csr::hgatp::set(&target.hgatp);
    riscv::instruction::hfence_gvma();;

    // hstatus: handle SPV change the virtualization mode to 0 after sret
    riscv::csr::hstatus::set_spv(riscv::csr::VirtualzationMode::Guest);

    // sstatus: handle SPP to 1 to change
    //          the privilege level to S-Mode after sret
    riscv::csr::sstatus::set_spp(riscv::csr::CpuMode::S);

    // sepc: set the addr to jump
    riscv::csr::sepc::set(&target.sepc);

    // jump!
    riscv::instruction::sret();
}
```

4.3.5 Trap handling

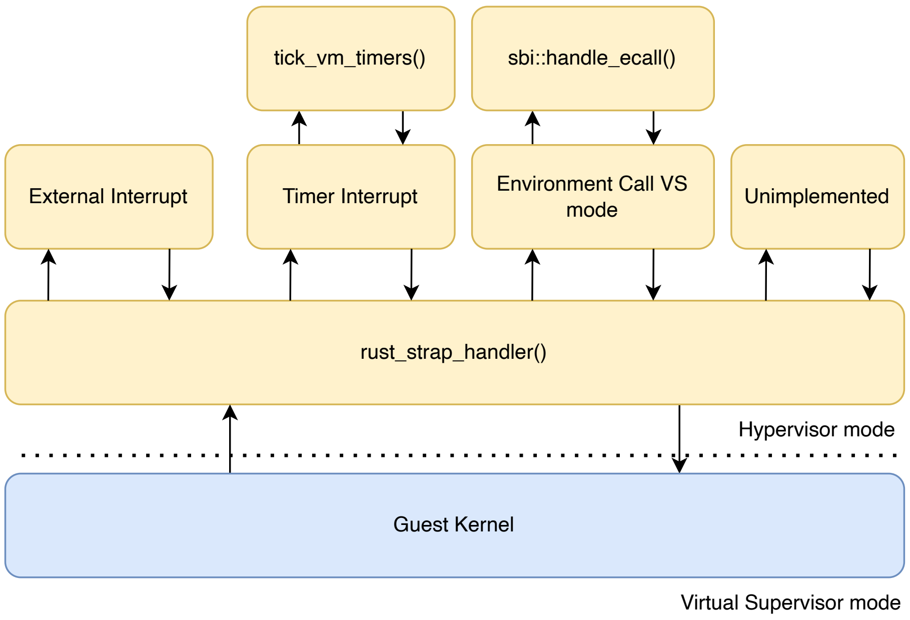

Figure 10: Overview of the traphandler of the hypervisor

As with the trap handler in our machine kernel found in section 4.3.5, there also needs to be a trap handler in the hypervisor which can handle exceptions which might occur in HS- or VS-mode. The general design of this implementation was discussed in the design section 3.2. An overview of the implementation can be found in figure 10. Initially, the handling is the same as trap handling in machine mode (section 4.3.5) apart from accessing the equivalent status registers for HS-mode which can be seen in listing 19 for a more detailed description of seeing the relevant section in the machine kernel implementation.

Listing 19: trap from hypervisor/src/hypervisor.S

```
    .macro load_gp i, base
        ld x\i, ((\i)*8)(\base)
    .endm

    .macro save_gp i, base
        sd x\i, ((\i)*8)(\base)
    .endm
trap_to_hypervisor:
    csrrw t6, sscratch, t6

    # save GPRs
    .set  i, 1
    .rept 30
        save_gp %i, t6
        .set i, i+1
    .endr

    mv  t5, t6
    csrr t6, sscratch
    save_gp 31, t5

    csrw sscratch, t5

    csrr a0, sepc
    csrr a1, stval
    csrr a2, scause
    csrr a3, sstatus
    csrr a4, sscratch
    la  sp, _intr_stack_end
    call rust_strap_handler
    csrw sepc, a0
    csrr t6, sscratch

    # restore GPRs
    .set i, 1
    .rept 31
        load_gp %i, t6
        .set i, i+1
    .endr

    sret
```

As with the machine kernel trap implementation, the main logic part of the logic is handled in **rust_strap_handler** which is part of our Rust code found in listing 20. Here the initial logic is identical since we need to check if our trap is an exception or an interrupt and then match the value of the **scause** CSR accordingly. The different trap cause values can be found in appendix B. We have two interrupts we need to deal with. One is external interrupts and timer interrupts. If the interrupt is unrecognized, we trigger the **unimplemented** macro. The external interrupts are not that relevant to the general implementation since they are dependent on the platform-specific PLIC implementation. But in this instance, we use it to handle VirtIO and UART interrupts. The timer interrupts, on the other hand, clears the respective timer interrupt pending bit in the **sip** CSR and enables bit in **sie** CSR since this is enabled when the hardware timer interrupt is triggered in machine mode see section 4.2.4. With this timer interrupt, we increment all of our virtual timers where one should exist for each guest. This is implemented to be scalable for multiple guests even though we only virtualize one guest in this implementation. This timer interface will be described in more detail in section 4.3.6.

Regarding exceptions, we need to provide an interface for the guest to send hypercalls to enable paravirtualization functionality (section **??**). This is done by handling environmental calls from the guest, as with the environmental call interface in the machine mode trap handler in section 4.2.4 we need to access the trap frame, which is all of the values of the register before the trap was caused. This was saved as part of our assembly code in listing 19. Also, this is an unsafe operation since we need to dereference a pointer to a fixed memory address. So Rust cannot check for us if this is a memory-safe operation or not on compile time. Therefore we need to ensure this pointer value is valid to avoid a memory fault. In this case, we take the values from the registers a0-7, which we then use to call the SBI interface handler, described more in section 4.3.6. When the trap returns, the result is put back into the registers a0 and a1. Lastly, before returning from an environmental call trap, the program counter value in **sepc** needs to be incremented one instruction further to avoid calling the same environmental call again.

Listing 20: Switch to guest hypervisor/src/hypervisor.rs

```rust
#[no_mangle]
pub extern "C" fn rust_strap_handler(
    sepc: usize,  /* a0 */  stval: usize, /* a1 */
    scause: usize, /* a2 */ sstatus: usize, /* a3 */
    frame: *mut TrapFrame, /* a4 */ ) -> usize {
    let is_async = scause >> 63 & 1 == 1;
    let cause_code = scause & 0xfff;
    if is_async {
        match cause_code {
            9 => { // external interrupt
                if let Some(interrupt) = plic::get_claim() {
                    match interrupt {
                        1..=8 => {
                            virtio::handle_interrupt(interrupt);
                        }
                        10 => {
                            uart::handle_interrupt();
                        }
                        _ => {
                            unimplemented!()
                        }
                    }
                    plic::complete(interrupt);
                } else {
                    panic!("invalid␣state")
                }
            }
            5 => { //timer interrupt
                riscv::csr::sip::clear_stimer();
                riscv::csr::sie::clear_hardware_timer();
                if let Some(mut timer) = timer::TIMER.try_lock() {
                    timer.tick_vm_timers(HYPERVISOR_TIMER_TICK);
                    let timer_trigger_list = timer.check_timers();
                    let guest0_timer_intr_trigger = timer_trigger_list[0];
                    if guest0_timer_intr_trigger {
                        riscv::csr::hvip::trigger_timing_interrupt();
                    }
                }
            }
            _ => {
                unimplemented!("Unknown␣interrupt␣id:␣{}", cause_code);
            }
        }
    } else {
        match cause_code {
            10 => { // Environment call
                let user_frame = unsafe{*frame.clone()};
                let guest_number = 0;
                let a7 = user_frame.regs[17];
                let a6 = user_frame.regs[16];
                let a1 = user_frame.regs[11];
                let a0 = user_frame.regs[10];
                let params = [user_frame.regs[10], user_frame.regs[11],
                    user_frame.regs[12], user_frame.regs[13],
                    user_frame.regs[14], user_frame.regs[15]];
                let sbi_result = sbi::handle_ecall(
                    a7, a6, params, guest_number);
                unsafe {
                    (*frame).regs[10] = sbi_result.error;
                    (*frame).regs[11] = sbi_result.value;
                }
                return sepc + 0x4; // Skips to the next instruction in guest
            }
            _ => {
                unimplemented!("Unknown␣Exception␣id:␣{}", cause_code);
            }
        }
    }
    sepc
```

```
}
```

4.3.6 SBI Timer Interface

As mentioned in our design section 3.2, we want a hypercall interface which implements the SBI timer interface for a guest to request timer interrupts as per the SBI (Supervisor Binary Interface) specification[9]. It is meant as a generic interface to enable supervisor mode based software to execute some privileged operations by doing environment calls. Usually, this is implemented by having a piece of software like OpenSBI[10] running in machine mode. In this case, we can use this generic interface to create a hypercall interface for our hypervisor since a guest is running in VS-mode. Many SBI calls can be implemented, but here we will only implement the timer interface.

The generic types defined in the SBI specification can be found in listing 21, which we need to incorporate into the design. The SBI command is done through an environment call instruction where register a7 is the extension type which is the type of SBI call being performed. In this case, this is the constant **EXTENSION_TIMER** in listing 21. Register a6 should contain the sub-function of the extension. In the timer extensions case, there is only one, so it is not relevant what this value is set to. The register a0 is used for the function argument, and the function with the argument has the following purpose "Programs the clock for the next event after stime_value time. stime_value is in absolute time. This function must clear the pending timer interrupt bit as well."[9]. After executing the request, we need to return a pair of values described in the **sbiret** struct. The error code is located in register a0 and the value is in register a1. We then return the appropriate values if we encounter any errors or are successful.

Listing 21: Generic c struct and function for the sbi timer

```c
    struct sbiret {
        long error;
        long value;
    };
    enum SBI_ERROR{
        SBI_SUCCESS=0,
        SBI_ERR_FAILED=-1,
        SBI_ERR_NOT_SUPPORTED=-2,
        SBI_ERR_INVALID_PARAM=-3,
        SBI_ERR_DENIED=-4,
        SBI_ERR_INVALID_ADDRESS=-5,
        SBI_ERR_ALREADY_AVAILABLE=-6,
        SBI_ERR_ALREADY_STARTED=-7,
        SBI_ERR_ALREADY_STOPPED=-8
    }
    const int EXTENSION_TIMER = 0x54494D45; // "TIME"
    struct sbiret sbi_set_timer(uint64_t stime_value)
```

When the SBI timer function is called, we run the following function **set_time** which, with the argument, takes in the guest number we infer from the trap context. The timer struct is encapsulated in a mutex made with the **lazy_static** macro, which is an external library that allows for the declaration of statics that is only initialized at runtime. After the lock is acquired, the timer is set in the virtual timer, the implementation of which can be found in appendix E. Lastly, we clear any waiting timer interrupts for the guest with the **hvip** CSR, which is as per the SBI specification.

Listing 22: set_timer function in SBI timer interface hypervisor/src/sbi/timer.rs

```rust
lazy_static::lazy_static! {
    pub static ref TIMER: spin::Mutex<VmTimers> =
                    spin::Mutex::new(VmTimers::new());
}

```

```
fn set_timer(arg0: usize, guest_number: usize) -> SbiRet {
    let time_value = arg0 as u64;
    if set_timer_value(time_value, guest_number) {
        SbiRet::ok(0)
    } else {
        // should be probed with probe_extension
        SbiRet::not_supported()
    }

}
pub fn set_timer_value(time_value: u64, guest_number: usize) -> bool {
    let mut timer = TIMER.lock();
    timer.set_timer(time_value, guest_number);
    riscv::csr::hvip::clear_timing_interrupt();
    true
}
```

As we mentioned in section 4.3.5, we trigger a timer tick in our virtual timer every time a timer interrupt is triggered, which can be seen in listing 20. This is protected by a mutex required by Rust, even though we know there can only be one exception handling done simultaneously. Therefore we need a mutex to make Rusts borrow checking happy. After acquiring the mutex lock, we access the virtual timer struct built-in functions found in appendix E to determine if any guest timers need to be triggered. If so, the timer interrupts pending is set in the respective guests **hvip** CSR.

4.4 Guest Kernel (VS-Mode)

When we have a hypervisor implemented, we need a guest kernel to test it with. This section will describe the implementation of the guest kernel design outlined in section 3.3. The purpose here is to make a guest kernel that is generic and made to be run on top of an SBI firmware like OpenSBI running in M-mode.

The guest's boot code and linking are very similar to how the hypervisor mode is booted due to the HS-mode and S-mode being the same privilege level in practice. If of interest, the boot code and linker file can be found in appendix F.

4.4.1 Initialization

As with the basic initialization of the hypervisor running in HS-mode described in section 4.3.1, we set the following: **stvec** CSR to our trap handler, the **sstatus** CSR to enable interrupts and the **sie** CSR where we specifically enable timer interrupts.

Next, we configure a page table with a virtual memory mapping to be able to test if the two-stage address translation is working. The details of this implementation are the same as with the hypervisor's virtual memory implementation described in section 4.3.2. We map our UART peripheral and program memory to have the virtual address match precisely with the physical address, so we can still access these. After we have set up our root page table, we configure the **satp** CSR accordingly and call the **sfence.vma** assembly instruction which synchronizes the TLB (translation lookaside buffer) and ensures the in-memory memory-management data structures are up to date.

4.4.2 SBI Timer

As described in section 4.3.6 we have an SBI interface as a guest to execute some privileged operations. We can execute an **ecall** instruction with the appropriate parameters. Listing 23 shows a function wrapper for the SBI call and an example call which is the one used in our guest code. It takes in arguments defining the extension, function and two args which is what we need to configure the underlying timer interface.

Listing 23: sbi_call wrapping a ecall instruction with example guest/src/kernel.rs

```rust
fn sbi_call(extension: usize, function: usize,
            arg0: usize, arg1: usize)
    -> SbiRet {
    let (error, value);
    unsafe { asm!(
            "ecall",
            in("a0") arg0, in("a1") arg1,
            in("a6") function, in("a7") extension,
            lateout("a0") error, lateout("a1") value,
    )};
    SbiRet { error, value }
}

let response = sbi_call(EXTENSION_TIMER, 0x0, 0xdead, 0xbeef);
```

4.4.3 Trap Handling

As with the initialization section 4.4.1, the trap handler is also identical to the hypervisor implementation (section 4.3.5) when it comes to assembly code and handling. The only difference is that we don't handle any exceptions, and when a timer interrupt is called, we print to our screen and call the SBI timer interface described in section 4.4.2 again.

5 Results

After implementing our hypervisor and guest kernel described in the implementation section 4, we can compile and run the code in QEMU. This is the following console output we get.

Listing 24: Resulting console output with virtual memory enabled in guest

```
 -----------------------
 rustyvisor
-----------------------
[INFO] logger was initialized
[INFO] processor is in m-mode running with hartid: 2147483676
[INFO] Initing heap implementation: 0x0000000082325000 ->
         0x0000000082335000 size: 0x0000000000010000
[INFO] jump to hypervisor while changing CPU mode from M to HS
[INFO] Current mepc addr 0x80100ea0
[INFO] hypervisor started
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] virtio0 addr: 0x0000000010001000
[INFO] a block device found
[INFO] -> allocated query object: 0x0000000082336000
[INFO] sscratch: 0000000082338000
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] succeeded in initializing the hypervisor
[INFO] a new guest instance: guest01
[INFO] -> create metadata set
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] a page 0x000000008233c000 was allocated for a
         guest page address translation page table
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] -> load a tiny kernel image
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] -> entrypoint: 0x0000000080000000
[INFO] -> section found: name=.text.entrypoint,
         address:0x0000000080000000, offset=0x0000000000000010
[INFO] -> section found: name=.text,
         address:0x0000000080000010, offset=0x00000000000048f8
[INFO] -> section found: name=.rodata,
         address:0x0000000080004910, offset=0x0000000000001395
[INFO] -> section found: name=.eh_frame,
         address:0x0000000080005ca8, offset=0x00000000000003bc
[INFO] -> section found: name=.data,
         address:0x0000000080007000, offset=0x0000000000000010
[INFO] -> the ELF was extracted into the guest memory
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] switch to guest
trap set to: 0x80000220
stvec is set to: 0x0000000080000220
hello world from a guest
a page 0x0000000080208000 was allocated
     for a guest page address translation page table
satp to be written: 0x8000000000080208

PAGE ALLOCATION TABLE
ALLOCATED: 0x80208000 -> 0x80211000
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Virt: 0x10000000 => Phys: 0x10000000
Virt: 0x80000000 => Phys: 0x80000000
...
```

```
Virt: 0x801ff000 => Phys: 0x801ff000
Num pages after each other: 511

Virt: 0x80200000 => Phys: 0x80200000
...
Virt: 0x80207000 => Phys: 0x80207000
Num pages after each other: 7

Virt: 0x82010000 => Phys: 0x82010000
...
Virt: 0x82020000 => Phys: 0x82020000
Num pages after each other: 16
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Allocated:    544 pages (   2228224 bytes).

[INFO] <--------- trap --------->
[INFO] sepc: 0x00000000800007dc
[INFO] stval: 0x00000000800007dc
[INFO] scause: 0x000000000000000c
[INFO] sstatus: 0x0000000200000120
[INFO] ------- trapframe --------
     x0  = 0x0000000000000000 | ra  = 0x00000000800007d4
     sp  = 0x0000000080106c00 | gp  = 0x0000000000000000
     tp  = 0x0000000000000000 | t0  = 0x0000000000000064
     t1  = 0x0000000080106841 | t2  = 0x346dc5d63886594b
     s0  = 0x0000000000000000 | s1  = 0x0000000000000000
     a0  = 0x8000000000080208 | a1  = 0x0000000000000000
     a2  = 0x000000000000000a | a3  = 0x0000000080002c36
     a4  = 0x0000000000000000 | a5  = 0x0000000000000000
     a6  = 0x0000000080106880 | a7  = 0x0000000080005b1a
     s2  = 0x0000000000000000 | s3  = 0x0000000000000000
     s4  = 0x0000000000000000 | s5  = 0x0000000000000000
     s6  = 0x0000000000000000 | s8  = 0x0000000000000000
     s8  = 0x0000000000000000 | s9  = 0x0000000000000000
     s10 = 0x0000000000000000 | s11 = 0x0000000000000000
     t3  = 0x0000000000002710 | t4  = 0x000000000000147b
     t5  = 0x0000000005f5e0ff | t6  = 0x0000000000000038
[INFO] ------- registers --------
     x0  = 0x0000000000000000 | ra  = 0x000000008010e97e
     sp  = 0x0000000081221640 | gp  = 0x0000000000000000
     tp  = 0x0000000000000000 | t0  = 0x000000000000000a
     t1  = 0x000000000000000f | t2  = 0x0000000081221728
     s0  = 0x0000000000000000 | s1  = 0x0000000000000000
     a0  = 0x000000008010e854 | a1  = 0x000000008011ea78
     a2  = 0x000000008011ecb8 | a3  = 0x000000008010ba80
     a4  = 0x0000000000000000 | a5  = 0x0000000000000000
     a6  = 0x0000000081221590 | a7  = 0x0000000081221508
     s2  = 0x0000000000000000 | s3  = 0x0000000000000000
     s4  = 0x0000000000000000 | s5  = 0x0000000000000000
     s6  = 0x0000000000000000 | s8  = 0x0000000000000000
     s8  = 0x0000000000000000 | s9  = 0x0000000000000000
     s10 = 0x0000000000000000 | s11 = 0x0000000000000000
     t3  = 0x0000000000002710 | t4  = 0x000000000000147b
     t5  = 0x0000000082338000 | t6  = 0x0000000000000038
[INFO] ---------  S csr ---------
     satp       = 0x0000000000000000 | sepc       = 0x00000000800007dc
     sie        = 0x0000000000000200 | sscratch   = 0x0000000082338000
     sstatus    = 0x0000000200000120 | stvec      = 0x0000000080100eb0
```

```
        scounteren = 0x0000000000000000 | scause     = 0x000000000000000c
        stval      = 0x00000000800007dc | sip        = 0x0000000000000020
[INFO] ---------  H csr ---------
        hedeleg    = 0x000000000000a109 | hcounteren = 0x0000000000000000
        hgatp      = 0x800000000008233c | hgeie      = 0x0000000000000000
        hgeip      = 0x0000000000000000 | hideleg    = 0x0000000000000444
        hie        = 0x0000000000000000 | hip        = 0x0000000000000000
        hstatus    = 0x00000002000001c0 | htval      = 0x0000000000000000
        hvip       = 0x0000000000000000 | htimedelta = 0x0000000000000000
[INFO] --------- VS csr ---------
        vsatp      = 0x8000000000080208 | vscause    = 0x0000000000000000
        vsepc      = 0x0000000000000000 | vsie       = 0x0000000000000000
        vsip       = 0x0000000000000000 | vsscratch  = 0x0000000000000000
        vsstatus   = 0x0000000200000000 | vstval     = 0x0000000000000000
        vstvec     = 0x0000000080000220
[INFO] -------- Prev Mode -------
[INFO] Previous Mode before trap: Virtual Supervisor Mode (VS)
abort: line 319, file src/hypervisor.rs:
          not implemented: Unknown Exception id: 12
```

From listing 24 you can see the full console output of both the hypervisor and the guest. Everything that is prefixed with [**INFO**] is from the hypervisor, and the text with no prefix or indent is from the guests. We can see that an unexpected exception is happening with the number 12. Referring to table 2 in appendix B we can see this is a **Instruction page fault**. Commenting out the code in the guest setting the **satp** CSR and executes **sfence.vma** yields the following result shown in listing 25.

Listing 25: Resulting console output with virtual memory disabled in guest

```
 -----------------------
 rustyvisor
-----------------------
[INFO] logger was initialized
[INFO] processor is in m-mode running with hartid: 2147483676
[INFO] Initing heap implementation: 0x0000000082325000 ->
    0x0000000082335000 size: 0x0000000000010000
[INFO] jump to hypervisor while changing CPU mode from M to HS
[INFO] Current mepc addr 0x80100ea0
[INFO] hypervisor started
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] virtio0 addr: 0x0000000010001000
[INFO] a block device found
[INFO] -> allocated query object: 0x0000000082336000
[INFO] sscratch: 0000000082338000
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] succeeded in initializing the hypervisor
[INFO] a new guest instance: guest01
[INFO] -> create metadata set
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] a page 0x000000008233c000 was allocated for a
         guest page address translation page table
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] -> load a tiny kernel image
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] -> entrypoint: 0x0000000080000000
[INFO] -> section found: name=.text.entrypoint,
    address:0x0000000080000000, offset=0x0000000000000010
[INFO] -> section found: name=.text,
    address:0x0000000080000010, offset=0x00000000000048ea
```

```
[INFO] -> section found: name=.rodata,
     address:0x0000000080004900, offset=0x0000000000001395
[INFO] -> section found: name=.eh_frame,
     address:0x0000000080005c98, offset=0x00000000000003bc
[INFO] -> section found: name=.data,
     address:0x0000000080007000, offset=0x0000000000000010
[INFO] -> the ELF was extracted into the guest memory
[INFO] environment call from HS-mode at 0x000000008010c734
[INFO] switch to guest
trap set to: 0x80000220
stvec is set to: 0x0000000080000220
hello world from a guest
[INFO] environment call from VS-mode at 0x0000000080000832
[INFO] a0: 0xdead, a1: 0xbeef, a6: 0x0, a7: 0x54494d45
[INFO] ecall: SBI Extension Timer: extension: 0x54494d45,
     function: 0x0, param: [57005, 48879, 57005, 48879, 0, 0]
[INFO] Setting timer mtimecmp 57005 for guest0
[INFO] SBI result SBI_SUCCESS
[INFO] SBI result SbiRet { error: 0, value: 0 }
Sbi call error: 0,  value: 0
Testing timer
[INFO] triggering timer interrupt on guest0
<--------- trap --------->
sepc: 0x00000000800003ec
stval: 0x0000000000000000
scause: 0x8000000000000005
sstatus: 0x0000000200000120
vm timer interrupt triggered
[INFO] environment call from VS-mode at 0x0000000080000832
[INFO] a0: 0xdead, a1: 0xbeef, a6: 0x0, a7: 0x54494d45
[INFO] ecall: SBI Extension Timer: extension: 0x54494d45,
     function: 0x0, param: [57005, 48879, 57005, 48879, 0, 0]
[INFO] Setting timer mtimecmp 57005 for guest0
[INFO] SBI result SBI_SUCCESS
[INFO] SBI result SbiRet { error: 0, value: 0 }
Sbi call error: 0,  value: 0
[INFO] triggering timer interrupt on guest0
<--------- trap --------->
sepc: 0x00000000800003ec
stval: 0x0000000000000000
scause: 0x8000000000000005
sstatus: 0x0000000200000120
vm timer interrupt triggered
[INFO] environment call from VS-mode at 0x0000000080000832
[INFO] a0: 0xdead, a1: 0xbeef, a6: 0x0, a7: 0x54494d45
[INFO] ecall: SBI Extension Timer: extension: 0x54494d45,
     function: 0x0, param: [57005, 48879, 57005, 48879, 0, 0]
[INFO] Setting timer mtimecmp 57005 for guest0
[INFO] SBI result SBI_SUCCESS
[INFO] SBI result SbiRet { error: 0, value: 0 }
Sbi call error: 0,  value: 0
```

From listing 25, we can see the hypervisor launching successfully. This is because the guest kernel can use its SBI interface, and timer interrupts are triggered periodically in the guest.

6 Discussion

6.1 Intent of thesis

This master thesis aimed to explore the new hypervisor extension, ratified into the RISC-V specification seven months (December 2021) before writing this thesis. Part of this exploration was to try and implement a hypervisor with this extension and compare it to existing RISC-V hypervisors made before the H-extensions release. Another goal was to provide a detailed written down explanation of how one proceeds to create a hypervisor with this new specification since there are not that many in-depth explanations of the process at the time of writing.

This thesis also aimed to look at the Rust programming language and its viability in its current form as a system-level language for programming hypervisors. This is because Rust offers many modern languages and attractive features like memory concurrency checks on compile-time and a built-in package manager, which would make developing system-level software easier.

6.2 Summary of results

By coding the implementation of the Rust based hypervisor described in the implementation section 4, which in turn is based on the design in section 3, we get the console output results as shown in section 5. The results shown have two different outcomes, listing 24 successfully initializes the hypervisor and transfers to the guest. However, the kernel fails to initialize the guest's virtual memory and triggers an instruction page fault, which would signal a weakness with the implementation or a problem elsewhere. Listing 25 on the other hand, it shows the hypervisor booting up properly, setting up the guest and can request and handle virtual timer interrupt calls sent from the hypervisor and apart from the guest, virtual memory seems to be working as expected.

As inferred from the results, it was also possible to create a hypervisor using the Rust nightly toolchain where both the hypervisor code and guest kernel were written in Rust.

6.3 Interpretation of results

As seen in the result section 5, there was an issue with using two-stage address translation in the guest, which caused an instruction page fault as seen in listing 24. This trap is usually caused when the page tables are misconfigured, not allowing the CPU to read the program memory where the instructions are stored. After investigating further, that does not seem to be the issue because the page table configuration is identical to the hypervisor's apart from the different memory regions being mapped. It is also very deterministic when the **sfence.vma** instruction is called and misconfiguring. Memory mappings in the page table cause other faults before we even reach that instruction. Another possibility considered is a bug within the emulation software being used QEMU. Due to the newness of the extension, it might be possible that this implementation hits a corner case which causes a trapped bug like this.

If the two-stage address translation in the guest is disabled, we get a different result, as can be seen in listing 25. Here the behaviour is more in-line with what we are excepting, and the guest can initialize properly. Furthermore, it can be confirmed that the SBI interface and virtual timer interface also work as expected since the guest can set and get timer interrupts triggered. Unfortunately, due to the bug with the two-stage address translation, the hypervisor is limited to not being able to virtualize more complex software like operating systems before this is resolved.

Another aspect that was one of the objectives of this thesis was to evaluate the advantages of using the hypervisor extension compared to what has been done before. Since the RISC-V architecture has always been within the rules outlined by Goldberg's and Popek's article "Formal requirements for virtualizable third generation architectures" [14] people have been implementing hypervisors before the extension existed. One of these is RVirt [11] where the implementation relies on trap-and-emulate in S-mode where the software is solely responsible for doing the virtualization separation itself. In contrast, the hypervisor extension adds, for example, CSRs, which automatically switch between the hypervisor and virtualized environment, removing the need to change all of these when there is a context switch. The extension thus comes with simplicity since the programmer does not need to handle this CSR register themselves. It reduces the program's complexity and size, which is positive from a development point of view. Other papers have also evaluated earlier drafts of the hypervisor like Bruno, José and Sandro's "A First Look at RISC-V Virtualization from an Embedded Systems Perspective"[15] which states that the hypervisor extension reduces performance penalty.

Finally, the last goal of the thesis was to evaluate Rust as a system programming language in the context of creating a hypervisor on RISC-V. There are several advantages and disadvantages I discovered in the process of implementing the hypervisor. One of the significant advantages is compiling checks for memory concurrency and borrowing, which makes it very pleasant to do system programming. If you have discipline and follow those checks, your code will be exception and race condition free. This is a considerable advantage since developers spend many hours debugging issues like these with older languages like C. There are unsafe sections in Rust that are required for system programming, like dereferencing a pointer to a peripheral that can cause these bugs. However, these sections can be wrapped, so we ensure safe handling of data entering and exiting our unsafe areas, and if there are bugs, they are isolated to these sections. There are, however, some downsides. Since Rust is still an evolving language, we rely on using features like **asm_const** which is an unstable feature that might be removed or drastically changed in the future. The need to rely on unstable features is a downside of system programming on Rust since many examples and codebases of system programming which you find online might also be broken and not compile today if they rely on unstable features (which most do). Another aspect is outdated packages Rust calls crates, which I encountered while developing the hypervisor. Most RISC-V development on Rust currently relies on many official crates that wrap different assembly instructions and CSR access. However, these were not updated to reflect the new hypervisor extension, so making a new wrapper from the bottom was necessary. This is not a gripe with the language itself but more a comment on overreliant use, and trust in packages might not always be the best if you want complete control and understanding of the system.

6.4 Limitations of this thesis

This thesis has several limitations, limiting which conclusions can be drawn from the results and exploration.

Since we encountered an error with two-stage address translation, we could not progress further to implement more advanced features, allowing us to collect some numerical results that could be used to draw some conclusions. Therefore this thesis cannot provide any conclusive numerical results to say if there are any performance benefits by running a hypervisor with the hypervisor extension as opposed to not using it. We are only based on opinions on the resulting codebase from a static analysis of other projects. Another aspect is that the emulator which was used, QEMU, does not necessary emulate hardware at a low enough level. If one was to, for example, try to measure MMU speeds, the measurement might give an inaccurate result due to QEMU taking shortcuts to speed up emulation.

Another limitation was that we were not able to test more advanced pieces of software like operating systems with the hypervisor. This impedes our ability to run standardized benchmarks, which would help check the hypervisor's stability and edge case handling. It is, therefore, likely that the provided implementation contains edge case faults in what appears to currently works.

Another limitation is that this implementation is only tested on an emulated platform. Although we have an implementation which appears to run in an emulated QEMU environment, this might not be the case on another platform. This also limits our ability to identify if the two-stage address translation problem we had earlier is an emulator bug or not. Making the case harder, there is currently no physical hardware with the implemented hypervisor extension. Hence, we cannot check whether it is a platform-specific bug. The only alternative to emulation at the time of writing is cores written in a hardware description language, which requires a lot of time and resources to simulate and test our code on it.

6.5 Practical application and oppertunity for further work

Part of the motivation for writing this thesis was to provide a good overview of how the fundamentals of the hypervisor extension work and how it can be implemented. This is because not many sources describe step by step and in detail how the details in the RISC-V specification can be implemented into code. Many of the resources one can find online are either a short description of how the hypervisor works or codebases without description or documentation on how the code works. Some of these even have code incompatible with the current specification since it was based on a draft before the extension was finalized. Therefore, I hope this thesis might provide the documentation and explanations I felt were lacking while implementing this hypervisor.

Additionally, since this project was implemented in Rust, it is possible to use it as a baseline for implementing bare metal RISC-V in the same language since this thesis contains a lot of fundamental handlers and structures, which would be helpful in any bare-metal application, not just one facing hypervisors.

Due to the newness of the hypervisor extensions on RISC-V, there is still much ground to cover in assessing how it scales with larger hypervisors, which holds up to other virtualization extensions on different architectures and general performance. Since this thesis forms a foundation for RISC-V implementation hypervisors, a continuation of what it outlines might be an excellent opportunity to explore the hypervisor extension further. When physical hardware supporting the hypervisor extension is made available, the theory can be easily tested if there is a bug with the two-stage address translation. Additionally, it would be interesting to implement a scheduler for the hypervisor to enable the virtualization of multiple guests.

6.6 Takeaways

To sum it up, this thesis has some advantages and disadvantages in regards to what it was able to accomplish. We can implement a single guest hypervisor written in Rust running QEMU, providing a virtual timer interface and a virtual memory mapped program memory for the guest. However, we cannot proceed further in implementing a two-stage address translation for the guest due to a bug which we cannot determine if it is human error mode on our part with the hypervisor or guest implementation or if it is an emulation bug in QEMU. Furthermore, due to the hypervisor extension's newness, there is no hardware that we can use to test our implementation. Thus we are limited with the results we can collect from it.

An evaluation of the Rust programing language and its advantages and disadvantages, when used as a low-level system language. High-level language features are a welcome addition when one is used to programming in C. For example, the compile-time checker ensures that the code is memory safe, and we cannot generate race conditions unless we explicitly allow it. Unfortunately, this is also one of the disadvantages that we cannot take advantage of on compile checks since part of writing low-level software is that we need to dereference memory locations which Rust cannot ensure is safe. Furthermore, since Rust is still a developing language, we need to use the nightly branch and language features which are not yet stabilized and can be deprecated in the future. However, all that taken into consideration, with the speed the language is being developed, many of these shortcomings might be solved soon, making Rust a desirable language with much potential for future implementations.

7 Conclusion

In this master thesis, we have explored the steps needed to create a hypervisor with the new RISC-V hypervisor extension, ratified into the specification at the end of November 2021. This exploration was done by designing and implementing a hypervisor with the Rust programming language, where the resulting implementation was run in a QEMU RISC-V emulator. The main contribution of this thesis is to provide a detailed process description of how a hypervisor with the new extension is implemented due to the current lack of documentation at the time of writing. Another contribution aim was to evaluate the state of Rust's ability to perform this task.

The resulting hypervisor was able to virtualize a single guest kernel where it had its program memory-mapped through virtual memory created by the hypervisor. Additionally, the guest had a UART peripheral directly mapped and a virtual timer interface through the SBI abstraction layer, which shows that the hypervisor works on a fundamental level. Attempts were made to make two-stage address translation work, but a hard to solve bug was encountered, which stopped further implementation. Also, due to the current newness of the hypervisor extension, it is hard to determine if a human error in the implementation causes an edge case in the emulation software.

An evaluation of the hypervisor extension compared to none extension approaches was also done. We can see that the hypervisor extension simplifies the complexity needed by the hypervisor software, improving readability and making it less likely for more bugs to be created. However, we could not collect any numerical results for this evaluation, so no comment can be made on performance. Another assessment was made of the Rust programing language, which shows it has considerable potential for becoming a system-level language widely used due to its language features like memory safety. However, the language is still developing, and many needed features are still experimental language features that might be changed or deprecated in the future. Thus making Rust very capable, but it still has some ways to go to be a stable alternative to industry standards as C for system-level programming.

In the end, the takeaway of this thesis is there is a lot of potential in both Rust and the new hypervisor extension for RISC-V. There are many opportunities for future work in both areas concerning further development of the Rust language and more evaluation of the hypervisor extension. Hopefully, this thesis gave a fundamental understanding of how the hypervisor extension can be implemented and thus be used for further implementations or related research.

References

[1] Ole Sivert Aarhaug. *Virtualization of xv6*. URL: https://github.com/stemnic/tdt09_report/releases/download/v1.0.0/main.pdf (visited on 1st June 2022).

[2] John Hauser Andrew Waterman Krste Asanovi. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. URL: http://riscv.com (visited on 7th Feb. 2022).

[3] Krste Asanovic Andrew Waterman. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (visited on 4th Dec. 2021).

[4] The Linux Foundation. *THE HYPERVISOR (X86 & ARM)*. URL: https://xenproject.org/developers/teams/xen-hypervisor/ (visited on 26th Apr. 2022).

[5] The Rust Foundation. *FAQ*. URL: https://prev.rust-lang.org/en-US/faq.html (visited on 26th Apr. 2022).

[6] The Rust Foundation. *MIR borrow check*. URL: https://rustc-dev-guide.rust-lang.org/borrow_check.html (visited on 4th June 2022).

[7] The Rust Foundation. *Rust Influences*. URL: https://doc.rust-lang.org/reference/influences.html#influences (visited on 26th Apr. 2022).

[8] LLVM Developer Group. *The LLVM Compiler Infrastructure*. URL: https://llvm.org (visited on 26th Apr. 2022).

[9] RISC-V Platform Specification Task Group. *RISC-V Supervisor Binary Interface Specification*. URL: https://github.com/riscv-non-isa/riscv-sbi-doc/releases/tag/v1.0.0 (visited on 16th May 2022).

[10] RISC-V International. *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*. https://github.com/riscv-software-src/opensbi. 2022.

[11] Samuel Ortiz Jonathan Behrens Cel Skeggs and Frans Kaashoek. *RVirt*. https://github.com/mit-pdos/RVirt. 2019.

[12] Stephen Marz. *RISC-V OS using Rust*. URL: https://osblog.stephenmarz.com (visited on 26th Apr. 2022).

[13] Takashi Yoneuchi Ole Sivert Aarhaug. *RustyVisor*. https://github.com/stemnic/rvvisor. 2022.

[14] Gerald J. Popek and Robert P. Goldberg. 'Formal Requirements for Virtualizable Third Generation Architectures'. In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[15] Bruno Sá, José Martins and Sandro Pinto. 'A First Look at RISC-V Virtualization from an Embedded Systems Perspective'. In: (Mar. 2021).

[16] SiFive. *SiFive FE310-G000 Manual*. URL: https://static.dev.sifive.com/FE310-G000.pdf (visited on 7th Dec. 2021).

[17] VMWare. *Virtualization from VMWare*. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf (visited on 22nd Mar. 2022).

[18] Takashi Yoneuchi. *rvvisor*. https://github.com/lmt-swallow/rvvisor. 2020.

A Hypervisor linker file

Listing 26: Linker file for hypervisor hypervisor/scripts/linker.ld

```
OUTPUT_ARCH("riscv")

ENTRY(m_entrypoint)

SECTIONS
{
    . = 0x80000000;
    .text.entrypoint :
    {
        PROVIDE(_elf_start = .);
        *(.text.entrypoint);
    }

    .text :
    {
        *(.text) *(.text.*);
    }

    .rodata :
    {
        *(.rdata .rodata. .rodata.*);
    }

    . = ALIGN(4096);
    .data :
    {
        *(.data .data.*);
    }

    _bss_start = .;
    .bss :
    {
        *(.bss .bss.*);
        PROVIDE(_elf_end = .);
    }
}
```

B Trap Cause Codes

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	Virtual supervisor software interrupt
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	Virtual supervisor timer interrupt
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	Virtual supervisor external interrupt
1	11	Machine external interrupt
1	12	Supervisor guest external interrupt
1	13–15	*Reserved*
1	≥16	*Designated for platform or custom use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode or VU-mode
0	9	Environment call from HS-mode
0	10	Environment call from VS-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–19	*Reserved*
0	20	Instruction guest-page fault
0	21	Load guest-page fault
0	22	Virtual instruction
0	23	Store/AMO guest-page fault
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Table 2: Machine and supervisor cause register (`mcause` and `scause`) values when the hypervisor extension is implemented.

Source: RISC-V International, CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/), via Github

C Page table implementation code

Listing 27: Parts the relevant structs for the generic page table implementation hypervisor/src/paging.rs

```
#[derive(Debug)]
pub struct VirtualAddress {
    addr: usize,
}

impl VirtualAddress {
    pub fn new(addr: usize) -> VirtualAddress {
        VirtualAddress { addr: addr }
    }

    pub fn new_from_vpn(vpn : [usize; 3]) -> VirtualAddress {
        let addr =
            (vpn[2]) << 30 |
            (vpn[1]) << 21 |
            (vpn[0]) << 12
        ;
        VirtualAddress { addr: addr }
    }

    pub fn to_vpn(&self) -> [usize; 3] {
        [
            (self.addr >> 12) & 0x1ff, //L0 9bit
            (self.addr >> 21) & 0x1ff, //L1 9bit
            (self.addr >> 30) & 0x3ff, //L2 11bit
        ]
    }

    pub fn to_offset(&self) -> usize {
        self.addr & 0x3ff //Offsett 12bit
    }

    pub fn to_usize(&self) -> usize {
        self.addr
    }

    pub fn as_pointer(&self) -> *mut usize {
        self.addr as *mut usize
    }
}

// PhysicalAddress
/////

#[derive(Copy, Clone, Debug)]
pub struct PhysicalAddress {
    addr: usize,
}

impl PhysicalAddress {
    pub fn new(addr: usize) -> PhysicalAddress {
        PhysicalAddress { addr: addr }
    }

    pub fn to_ppn(&self) -> usize {
        self.addr >> 12 //ppn 44bit
    }

    pub fn to_ppn_array(&self) -> [usize; 3] {
        [
            (self.addr >> 12) & 0x1ff,      //L0 9bit
            (self.addr >> 21) & 0x1ff,      //L1 9bit
            (self.addr >> 30) & 0x3ff_ffff, //L2 26bit
        ]
    }

    pub fn to_usize(&self) -> usize {
        self.addr
    }

    pub fn as_pointer(&self) -> *mut usize {
        self.addr as *mut usize
    }
}

// Page
/////

#[derive(Copy, Clone, Debug)]
pub struct Page {
    addr: PhysicalAddress,
}

impl Page {
    pub fn from_address(addr: PhysicalAddress) -> Page {
        Page { addr: addr }
    }

    pub fn address(&self) -> PhysicalAddress {
        self.addr
    }
    /// Clears allocated memory for page
    pub fn clear(&self) {
        unsafe {
            let ptr = self.addr.as_pointer();
            for i in 0..512 {
                ptr.add(i).write(0)
            }
        }
    }
}
#[derive(Debug)]
struct PageTableEntry {
    pub ppn: [usize; 3],
    pub flags: u16,
}

pub enum PageTableEntryFlag {
    Valid = 1 << 0,
    Read = 1 << 1,
    Write = 1 << 2,
    Execute = 1 << 3,
    User = 1 << 4,
    Global = 1 << 5,
    Access = 1 << 6,
    Dirty = 1 << 7,
    // TODO (enhancement): RSW
}

impl PageTableEntry {
    pub fn from_value(v: usize) -> PageTableEntry {
        let ppn = [   (v >> 10) & 0x1ff,       // PPN[0] 9 bit
                                (v >> 19) & 0x1ff,       // PPN[1] 9 bit
                                (v >> 28) & 0x3ff_ffff]; // PPN[2] 26 bit
        PageTableEntry {
            ppn: ppn,
            flags: (v & (0x1ff as usize)) as u16,
        }
    }

    pub unsafe fn from_memory(paddr: PhysicalAddress) -> PageTableEntry {
        let ptr = paddr.as_pointer();
        let entry = *ptr;
        PageTableEntry::from_value(entry)
    }

    pub fn to_usize(&self) -> usize {
        (if (self.ppn[2] >> 25) & 1 > 0 {
            0x3ff << 54
        } else {
            0
        }) | ((self.ppn[2] as usize) << 28)
            | ((self.ppn[1] as usize) << 19)
            | ((self.ppn[0] as usize) << 10)
            | (self.flags as usize)
    }

    pub fn next_page(&self) -> Page {
        Page::from_address(PhysicalAddress::new(
            (self.ppn[2] << 30)
            | (self.ppn[1] << 21)
            | (self.ppn[0] << 12),
        ))
    }

    pub fn set_flag(&mut self, flag: PageTableEntryFlag) {
        self.flags |= flag as u16;
    }

    pub fn is_valid(&self) -> bool {
        self.flags & (PageTableEntryFlag::Valid as u16) != 0
    }
}

pub struct PageTable {
    pub page: Page,
}

impl PageTable {
    fn set_entry(&self, i: usize, entry: PageTableEntry) {
        let ptr = self.page.address().as_pointer() as *mut usize;
        unsafe { ptr.add(i).write(entry.to_usize()) }
    }

    fn get_entry(&self, i: usize) -> PageTableEntry {
        let ptr = self.page.address().as_pointer() as *mut usize;
        unsafe { PageTableEntry::from_value(ptr.add(i).read()) }
    }

    pub fn from_page(page: Page) -> PageTable {
        PageTable { page: page }
    }

    pub fn resolve(&self, vaddr: &VirtualAddress) -> PhysicalAddress {
        self.resolve_intl(vaddr, self, 2)
    }

    fn resolve_intl(
        &self,
        vaddr: &VirtualAddress,
        pt: &PageTable,
        level: usize,
    ) -> PhysicalAddress {
        let vpn = vaddr.to_vpn();

        let entry = pt.get_entry(vpn[level]);
        if !entry.is_valid() {
            panic!("failed␣to␣resolve␣vaddr:␣0x{:016x}", vaddr.addr)
        }

        if level == 0 {
            let addr_base = entry.next_page().address().to_usize();
            PhysicalAddress::new(addr_base | vaddr.to_offset())
        } else {
            let next_page = entry.next_page();
            let new_pt = PageTable::from_page(next_page);
            self.resolve_intl(vaddr, &new_pt, level - 1)
        }
    }

    pub fn map(&self, vaddr: VirtualAddress, dest: &Page, perm: u16) {
        self.map_intl(vaddr, dest, self, perm, 2)
    }

    fn map_intl(
        &self,
        vaddr: VirtualAddress,
        dest: &Page,
        pt: &PageTable,
        perm: u16,
        level: usize,
    ) {
        let vpn = vaddr.to_vpn();

        if level == 0 {
            // register `dest`  addr
            let new_entry = PageTableEntry::from_value(
                ((dest.address().to_usize() as i64 >> 2) as usize)
                    | (PageTableEntryFlag::Valid as usize)
                    | (PageTableEntryFlag::Dirty as usize)
                    | (PageTableEntryFlag::Access as usize)
                    | (perm as usize),
            );
            pt.set_entry(vpn[0], new_entry);
        } else {
            // walk the page table
            let entry = pt.get_entry(vpn[level]);
            if !entry.is_valid() {
                // if no entry found, create new page and assign it.
                let new_page = alloc();
                let new_entry = PageTableEntry::from_value(
                    ((new_page.address().to_usize() as i64 >> 2) as usize)
                        | (PageTableEntryFlag::Valid as usize),
                );
                pt.set_entry(vpn[level], new_entry);
                let new_pt = PageTable::from_page(new_page);
                self.map_intl(vaddr, dest, &new_pt, perm, level - 1);
            } else {
                let next_page = entry.next_page();
                let new_pt = PageTable::from_page(next_page);
                self.map_intl(vaddr, dest, &new_pt, perm, level - 1);
            };
        }
    }
```

D Guest prepare_gpat_pt

Listing 28: Creating a page table mapping for guest hypervisor/src/guest.rs

```
fn prepare_gpat_pt() -> Result<paging::PageTable, Error> {
    let root_page = paging::alloc_16();
    let root_pt = paging::PageTable::from_page(root_page);

    // create an identity map for UART MMIO
    let vaddr = memlayout::GUEST_UART_BASE;
    let page = paging::Page::from_address(
        paging::PhysicalAddress::new(vaddr)
        );
    root_pt.map(
        paging::VirtualAddress::new(vaddr),
        &page,
        (paging::PageTableEntryFlag::Read as u16)
            | (paging::PageTableEntryFlag::Write as u16)
            | (paging::PageTableEntryFlag::Execute as u16)
            | (paging::PageTableEntryFlag::User as u16), // required!
    );

    // Mapping VIRTIO memory to virtual machine
    for i in 0..8 {
        let vaddr = memlayout::VIRTIO0_BASE + (0x1000 * i);
        let page = paging::Page::from_address(
            paging::PhysicalAddress::new(vaddr)
            );
        root_pt.map(
            paging::VirtualAddress::new(vaddr),
            &page,
            (paging::PageTableEntryFlag::Read as u16)
                | (paging::PageTableEntryFlag::Write as u16)
                | (paging::PageTableEntryFlag::Execute as u16)
                | (paging::PageTableEntryFlag::User as u16), // required!
        );
    }

    // allocating new pages and map GUEST_DRAM_START ~ GUEST_DRAM_END
    // into those pages for guest kernel
    let map_page_num = (memlayout::GUEST_DRAM_END
        - memlayout::GUEST_DRAM_START)
        / (memlayout::PAGE_SIZE as usize)
        + 1;
    for i in 0..map_page_num {
        let vaddr = memlayout::GUEST_DRAM_START + i
            * (memlayout::PAGE_SIZE as usize);
        let page = paging::alloc();
        root_pt.map(
            paging::VirtualAddress::new(vaddr),
            &page,
            (paging::PageTableEntryFlag::Read as u16)
                | (paging::PageTableEntryFlag::Write as u16)
                | (paging::PageTableEntryFlag::Execute as u16)
                | (paging::PageTableEntryFlag::User as u16), // required!
        )
    }

    let map_page_num = (memlayout::GUEST_TEST_AREA_END
        - memlayout::GUEST_TEST_AREA_START)
        / (memlayout::PAGE_SIZE as usize)
        + 1;
    for i in 0..map_page_num {
        let vaddr = memlayout::GUEST_TEST_AREA_START + i
            * (memlayout::PAGE_SIZE as usize);
        let page = paging::alloc();
        root_pt.map(
            paging::VirtualAddress::new(vaddr),
            &page,
            (paging::PageTableEntryFlag::Read as u16)
                | (paging::PageTableEntryFlag::Write as u16)
                | (paging::PageTableEntryFlag::Execute as u16)
                | (paging::PageTableEntryFlag::User as u16), // required!
        )
    }

    Ok(root_pt)
}
```

E Virtual timer implementation

Listing 29: The struct for our virtual timer implementation hypervisor/src/timer.rs

```rust
#[derive(Debug, Copy, Clone)]
pub struct VmTimers {
    timers : [VmTimer; MAX_NUMBER_OF_GUESTS]
}

impl VmTimers {
    pub fn new() -> VmTimers {
        VmTimers{
            timers: [VmTimer::new() ; MAX_NUMBER_OF_GUESTS]
        }
    }
    pub fn tick_vm_timers(&mut self, amount: usize ){
        let mut i = 0;
        while i < MAX_NUMBER_OF_GUESTS-1 {
            self.timers[i].tick(amount as u64);
            i += 1;
        }
    }
    pub fn check_timers(&self) -> [bool; MAX_NUMBER_OF_GUESTS] {
        let mut vm_timer_list = [false ; MAX_NUMBER_OF_GUESTS];
        let mut i = 0;
        while i < MAX_NUMBER_OF_GUESTS-1 {
            let vmtimer = self.timers[i];
            if vmtimer.enabled {
                if vmtimer.mtime >= vmtimer.mtimecmp {
                    vm_timer_list[i] = true;
                }
            }
            i += 1;
        }
        return vm_timer_list
    }
}

#[derive(Debug, Copy, Clone)]
pub struct VmTimer {
    enabled: bool,
    mtime: u64,
    mtimecmp: u64
}

impl VmTimer {
    pub fn new() -> VmTimer {
        VmTimer{
            enabled: false,
            mtime: 0,
            mtimecmp: 0
        }
    }

    pub fn tick(&mut self, amount: u64){
        if self.enabled {
            self.mtime += amount;
        }
    }

    pub fn set_timer(&mut self, amount: u64){
        self.enabled = true;
        self.mtimecmp = amount;
        self.mtime = 0;
    }
}

impl Timer for VmTimers {
    #[inline]
    fn set_timer(&mut self, time_value: u64, guest_id: usize) {
            self.timers[guest_id].set_timer(time_value);
    }
}
```

F Guest kernel linker and boot code

Listing 30: Linker file for guest kernel guest/scripts/linker.ld

```
OUTPUT_ARCH("riscv")

ENTRY(entrypoint)

SECTIONS
{
    . = 0x80000000;
    .text.entrypoint :
    {
        PROVIDE(_elf_start = .);
        *(.text.entrypoint);
    }

    .text :
    {
        *(.text) *(.text.*);
    }

    .rodata :
    {
        *(.rdata .rodata. .rodata.*);
    }

    . = ALIGN(4096);
    .data :
    {
        *(.data .data.*);
    }

    _bss_start = .;
    .bss :
    {
        *(.bss .bss.*);
        PROVIDE(_elf_end = .);
    }
}
```

Listing 31: boot code from guest/src/boot.S

```
    entrypoint:
    # load stack addr
    la     sp, _stack_end
    # jump to rust code
    tail       rust_entrypoint
```