Port of Mimiker Operating System for RISC-V Architecture

(Port systemu operacyjnego Mimiker na architekturę RISC-V)

Michał Błaszczyk

Praca inżynierska

Promotorzy: mgr Krystian Bacławski
dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

14 lutego 2022

Abstract

In my thesis, I will describe the process of porting an operating system for a new architecture. The operating system of choice is Mimiker, whereas the target architecture is RISC-V. Preparing a port involves significant work around the most crucial components of the system and requires an in-depth understanding of the architecture along with introduced software and hardware environments. As the outcome of my work, Mimiker has been adopted to a wide range of hardware platforms with a fully open design.

W mojej pracy opiszę proces portowania systemu operacyjnego na nową architekturę. Wybranym systemem operacyjnym jest Mimiker, zaś docelowa architektura to RISC-V. Przygotowanie portu wiąże się z wykonaniem znaczącej pracy w obrębie najbardziej kluczowych komponentów systemu oraz wymaga dogłębnej wiedzy na temat architektury wraz z narzuconym środowiskiem programowym i sprzętowym. Jako rezultat mojej pracy Mimiker został przystosowany do szerokiego zakresu platform sprzętowych z w pełni otwartym projektem.

To my Mom and Dad, for their love, patience, hard work, and faith.
Because it wasn't easy and they were always there for me.

Contents

Chapter 1

Introduction

Wherever we go, whatever we do, we are surrounded by electronic devices. Integrated chips can be found in our phones, smartwatches, cars, kitchen appliances, just to name a few. We are constantly finding new ways to exploit electronics, and thereby digital devices are bound to become even more pervasive.

The majority of released digital devices are sophisticated enough to contain an on-chip processor. A processor (or central processing unit (CPU)) is the brain of a computer. The most basic characteristic of a CPU is its architecture. The two most dominant architectures are `x86` and `ARM`, but for several years, there has been a new alternative.

`RISC-V` has already made a great impact on the industry. As an open architecture with customization among its main objectives, `RISC-V` is the architecture of choice for many companies that design digital devices. The number of applications has already grown from thousands to millions and has finally reached billions.

1.1 With great power comes great responsibility

After the `RISC-V` specifications [7][8] had been ratified and released to the public, first implementations started to grow. But what is the usage of a processor if there is no software to run on it?

Whenever a new architecture is created, along with the effort concentrated on specifications, parallel work is needed to supply basic software tools in order to enable software development. Among the most crucial elements regarding the software are:

- compiler – a new backend needs to be created,
- linker – relocations defined by the architecture must be implemented,
- assembler – the new instruction set must be incorporated,

- libc – an implementation of the standard C library must be provided to permit basic software development (for the most basic solutions, it would be a port of the Newlib library).

The work on hardware and software specifications along with the development and maintenance of the software tools creates a wide and healthy `RISC-V` ecosystem. The ecosystem is powered by the `RISC-V` community which has grown over the years and includes a broad range of members, starting from enthusiasts and students, reaching one of the most skilled industry experts.

But the software development isn't constrained to the basic programming tools. With the arrival of HiFive Unleashed, the world saw the first `RISC-V` board equipped enough to run Linux. But how do you run an operating system on a board based on a new architecture?

1.2 Porting an operating system

Before even considering running an operating system on a `RISC-V` based board, we should wonder why would we want an operating system on a `RISC-V` board.

With the software development tools mentioned in the first section, we are able to write and compile code that can run on a `RISC-V` processor. Although it may not seem much, it is! There are plenty of simple embedded systems that don't need anything more. Instead of having a sophisticated operating system, such platforms are only equipped with simple firmware built solely from the basic tools. Such a runtime is basically restricted to controlling simple peripheral devices contained on the board. However, such a solution works only for very specific applications. If we deal with general-purpose boards and want them to run elaborate applications, we need an operating system.

A conventional operating system consists of three components:

- Kernel – controls the hardware and provides an execution environment for all user space processes.
- System libraries – basic libraries linked with user applications. Examples include the standard C library and C math library.
- User space programs – applications that run in the environment delivered by the kernel. User processes communicate with the kernel via system calls (syscalls).

An operating system kernel is a composition of hierarchical layers. For our needs, three main layers can be distinguished (bottom to top):

- Hardware – this layer is responsible for managing hardware resources. It operates directly on the underlying hardware and differs for each target platform. The main purpose of this layer is to implement the platform-independent interfaces exposed by the next layer.
- Abstraction – a thin layer that bridges the hardware layer with the core layer. It defines interfaces to abstract from a particular platform. Above this layer, every device is viewed as a model which supports some well-defined methods and provides some established attributes. Such models do not restrict to peripheral devices, a CPU is also perceived as a model.
- Core – this layer builds upon the abstraction layer. Provided with abstract models, it contains all machine-independent components of the kernel. Examples include memory management, scheduler, syscall implementation, process management, and much more.

The task of creating a port of an operating system focuses on expanding the hardware layer of the operating system so that it includes implementations of the interfaces fulfilled by devices accommodated in the target platform including the employed CPU.

This thesis will describe the process of porting the Mimiker operating system for the `RISC-V` architecture concentrating on the 32-bit variant.

1.3 Instruction set architecture (ISA)

An instruction set architecture (ISA) is an abstract model of a processor. It describes the components of the hardware which are seen and managed by the system programmer. Among the specified things are register description, available instructions, privilege levels, memory model, exception handling, and memory translation systems.

We can distinguish two main types of architectures:

- Reduced instruction set architecture (RISC) – a RISC ISA exhibits the Unix philosophy, that is, each instruction performs a single and rather a simple task. Most RISC ISAs are load/store architectures meaning that the memory accesses are possible only via load and store instructions. An exemplary RISC ISA is `Aarch64`.
- Complex instruction set architecture (CISC) – a CISC instruction can perform numerous tasks at once, for example, add two registers and store the result directly to designated memory location. CISC architectures usually present many sophisticated addressing modes. An example of a CISC architecture is `x86`.

RISC-V is a load/store RISC ISA.

1.4 Why RISC-V?

Although have justified that operating systems are needed and desirable, we haven't considered the other point of view. Why would an operating system want to support the RISC-V architecture?

1.4.1 Open hardware

Perhaps the most revolutionary aspect of RISC-V is its openness. Whereas most companies (e.g. Arm Ltd. and Intel) make their architectures proprietary and charge royalties from any company that wishes to make its own implementation of the ISA, RISC-V is an open ISA and can be implemented by anyone without charging any fees.

1.4.2 Open implementations

To have an open-source microarchitecture you need an open ISA. RISC-V is just it.

There is a number of open-source RISC-V CPU implementations. Most of such projects are FPGA friendly, thereby, all we need to have a running RISC-V CPU is an FPGA board capable of accommodating selected implementation.

Among the most popular open source implementations are VexRiscv [1] and PicoRV32 [2].

Besides open implementations made by individuals, there are open implementations made by professional companies. SiFive has already produced RISC-V CPUs with freely available design files.

1.4.3 Educational architecture

Open microarchitectures present a wide range of complexity levels, starting from single-cycle implementations, advancing to pipelined solutions, reaching sophisticated out-of-order designs with a number of heterogeneous mechanisms employed for branch prediction and prefetching.

Such versatility and accessibility make RISC-V the architecture of choice for enthusiasts and students.

A number of top worldwide known universities have already migrated their main digital logic design and computer architecture courses to RISC-V [9][10].

The University of Wrocław is not an exception. Digital logic lecture familiarizes students with RISC-V ISA and addresses single cycle and pipelined implementations [11]. Moreover, as a university, we are especially interested in RISC-V, as we are starting an effort to create a unique out-of-order implementation of the RISC-V architecture on our own. A project devoted to this task will be guided by Marek Materzok.

With Mimiker running on RISC-V, we would have a complete implementation of a computer system developed entirely at the University of Wrocław.

1.4.4 Architecture of the future

In 2014, the inventors of RISC-V announced their goal for RISC-V to become the standard ISA for all computing devices [3].

Considering the impact RISC-V has already made on the industry, the growing interest around it, more and more companies employing the architecture in their chips, RISC-V has a great chance of fulfilling the promise given by its inventors and becoming a dominant architecture of the future.

1.5 Mimiker

Mimiker is an open-source, research operating system inspired by the world of Unix, in particular by its *BSD flavor [4].

Mimiker is developed at the University of Wrocław in a project guided by Krystian Bacławski.

In the beginning, Mimiker was written for the Malta-R development platform [5] which contains a MIPS CPU. However, around one year ago, MIPS Technologies has announced discontinuation for the MIPS processor family. Currently, the company is moving to make chips based on RISC-V.

Even before the aforementioned news was published, a decision had been made to prepare a port for the Aarch64 architecture. The rationale was that the Malta board is impossible to acquire, while Aarch64 based boards are widely available for reasonable prices. The person responsible for the port was Paweł Jasiak, and as a result of his work Mimiker can be run on the Raspberry Pi 3 board [6].

The effort of my work is concentrated on running Mimiker on platforms designed using the LiteX SoC builder utilizing a 32-bit RISC-V softcore. Although LiteX is the main goal, other boards based on RISC-V can also be supported by adding required device drivers and platform description in the form of a device tree file.

Chapter 2

Abstractions

In general, to prepare a port of an operating system, a fair amount of knowledge is needed. As we will write low-level code to control the target CPU, we need to know the interface to the hardware, namely the RISC-V ISA. When we become familiar with the architecture, we will discover a typical hardware environment a RISC-V CPU operates in, this involves interrupt controllers and timer. Next, we will explore the characteristics of the target board. After covering these topics, we will move to the software components. The supervisor binary interface will be crucial. Besides that, we will discuss the application binary interface and device tree standard.

This chapter describes the most important abstractions defined by RISC-V, LiteX, SBI, ABI, and device tree.

2.1 RISC-V ISA

All knowledge regarding RISC-V is presented in the form of specifications maintained on GitHub by the RISC-V Foundation organization.

The specifications are partitioned into two groups:

- ISA – the primary goal of these specifications is the maintenance of the RISC-V ISA. They contain the base volumes of the manual, extension drafts, the formal specification of the ISA, ACLINT, PLIC, and more [7].
- Non-ISA – these are all remaining specifications relevant to RISC-V. Among the contained specifications the following can be found: SBI, ABI, ACPI, and RISC-V specific extensions to UEFI [8].

The ultimate source of knowledge regarding RISC-V ISA is the RISC-V Instruction Set Manual. The manual is structured into two volumes:

- Volume I: Unprivileged ISA – describes the core architecture along with un-

privileged extensions. The volume finishes with a detailed description of the applied memory model [12].

- Volume II: Privileged Architecture – covers privilege modes, privileged registers, basic privileged architectures, and optional extensions [13].

This section will only cover the most essential aspects of the manual from the perspective of a system programmer attempting to port an operating system. If the reader would like to delve into details, it is advised to reach for the actual `RISC-V` manual.

2.1.1 What is RISCV?

`RISC-V` isn't a typical ISA. In fact, `RISC-V` is a collection of ISAs.

The unprivileged architecture defines four base ISAs and a number of optional extensions. Each core ISA is called an integer ISA and defines the architectural state and available instructions (computational, memory, control flow, fence, trap). The core ISAs are distinguished by native pointer size (32, 64, 128) and the amount of implemented integer instructions (all or just a subset). The list of defined extensions is constantly growing and currently includes integer multiply/divide extension ("M"), floating-point extensions ("F", "D", and "Q"), atomic extension ("A"), and more.

The privileged architecture is composed of base machine mode ISA, optional supervisor and hypervisor base ISAs, and a few extensions to the core ISAs.

Besides the standard (unprivileged and privileged) extensions, custom extensions may be provided by each vendor, for instance, to accelerate a specific type of computations. Herein, we will restrict our reasoning to standard extensions.

Each implementation of `RISC-V` must support at least a single integer ISA and machine mode ISA. Besides this core, an implementation is likely to support some standard extensions. Most `RISC-V` CPUs provide at least "M" and "A" extensions.

2.1.2 Why do extensions matter?

So let's say we have a `RISC-V` CPU. Why would we even bother about the extensions it supports?

Performance

The truth is, in the case of unprivileged extensions, the awareness of all supported extensions isn't mandatory. If the operating system can function relying just on a handful of unprivileged extensions (e.g. "M" and "A"), then we can simply ignore

other extensions and the system will work. The penalty of such an approach is a potential loss in performance. If our CPU supports the double-precision floating-point extension and we don't utilize it, all floating-point operations will be emulated using integers. The compiler will replace each floating-point operation with a call to a function implementing the operation with integer computational instructions. This scheme is known as soft floating-point. This won't cause any incorrectness in our programs but can result in significant delays in execution.

Hardware control

While it is beneficial to be aware of supported unprivileged extensions, in general, we must be aware of privileged extensions. When porting an operating system, we need to know which privilege modes are supported, what are the available virtual memory translation modes, and which optional status and control registers are implemented.

2.1.3 How to utilize an extension?

Utilizing an extension differs depending on the privilege level it regards.

Unprivileged extensions

To employ an unprivileged extension, in most cases, all we need to do is provide the compiler with appropriate target architecture where we specify all supported extensions (the `-march` GCC option), and select the appropriate ABI to specify the argument passing method (the `-mabi` GCC option) [14]. The compiler should be smart enough to make use of the extensions and automatically generate efficient code. For example, in the case of compressed instructions extension ("C"), the compiler should detect instructions with a compressed counterpart and embrace them in the generated object file. Unfortunately, the compiler might not be sufficiently skilled or maybe simply unable, to put some extensions in use in all scenarios. Sometimes, the programmer needs to help the compiler by directly translating a snippet of code (e.g. using embedded assembly), or by explicitly highlighting the code that needs special attention (perhaps via compiler-specific attributes).

Privileged extensions

The process of employing a privileged extension depends on the exact extension. User mode extension provides essential code isolation which is a profound principle in operating system design, thereby is implicitly used in the sole organization of an operating system. Supervisor base ISA extension supplies primary mechanisms needed to implement the most basic features of the kernel (e.g. exception handling). Other extensions (e.g. optional CSRs and translation modes) may be utilized in the

implementation of an abstract CPU model (the hardware layer of the kernel) by explicit usage in the code (e.g. in physical memory map management module).

2.1.4 What is the kernel overhead of using an extension?

From a kernel developer's point of view, when utilizing an extension, there is usually some software overhead to incur.

Unprivileged extensions

This is the easier part. An unprivileged extension can optionally define an additional architectural state. If the extension doesn't introduce any visible state, then there is no software overhead in the kernel. However, if the extension does enhance the architectural state, then the machine context of a user thread grows and overhead occurs.

Privileged extensions

When it comes to privileged extensions, it is often the case, that a complete kernel module must be built to adequately utilize an extension. A privileged extension is often intended to implementing some known mechanism (e.g. virtual memory) that is crucial regarding the inner working of the kernel and thereby requires some thorough interface to be constructed upon it.

2.1.5 Keeping track of the architectural state of an unprivileged extension

From a kernel's perspective, we can distinguish two kinds of threads

- User threads – these threads periodically switch between user and kernel space and constitute user space processes.
- Kernel threads – these threads operate fully in kernel space. A good example of such a thread is an interrupt thread destined for servicing delegated handlers in an outright environment.

In most architectures, whenever a context switch from a user thread occurs, the kernel will save the whole architectural state of the thread (which mostly consists of integer and floating-point registers) to restore it while switching back to this thread. The process of saving the whole context is unconditional and occurs whenever a context switch is performed.

RISC-V introduces a mechanism meant for reducing the copying performed while saving the user thread's context. The supervisor level status register contains a state for each of the following:

- floating-point architectural context,
- vector extension architectural context,
- summarized state of all other extensions' architectural contexts.

Four state values are defined:

- Off
- Initial
- Clean
- Dirty

We will use the extension context state information for the floating-point unit to speed up the context switch implementation.

Currently, the only extension, besides floating-point extensions, that define some additional architectural state is the vector extension. As we don't support the vector extension, state information for this extension, as well as for the other extensions (which are not there yet) will be ignored.

The exact meaning along with some exemplary scenarios of the aforementioned values can be found in section 3.1.6.6 of the privileged volume [13].

2.1.6 Terminology

RISC-V defines some essential notions used throughout the specifications. This section serves to explain the most crucial ones from our point of view.

Hardware terminology

A RISC-V core is defined to be a hardware component with an independent instruction fetch unit.

A core can deploy multithreading, in which case it has its internal pipeline shared between multiple hardware threads. A hardware thread in RISC-V terminology is called a hart.

Software terminology

RISC-V introduces a rather unique software stack by presenting the notion of an execution environment and expands it to all privilege levels, except for the machine mode which has unrestricted access to the underlying hardware.

Execution environment Every software in the software stack runs in some execution environment. An execution environment interface (EEI) provides an interface between the software executing in the environment and the execution environment itself.

The most common example of an EEI is ABI in a typical operating system. User processes operate in userspace under the control of the kernel which is the application execution environment. ABI provides means for the user space to communicate with the kernel through syscalls (e.g. write request).

Besides the aforementioned example of EEI that is commonly applied in operating systems, RISC-V extends this notion to other, more privileged levels. The supervisor mode software executes in the supervisor execution environment which is the next privilege level (hypervisor or machine mode) and uses supervisor binary interface (SBI) to issue services from the lower layer. The same applies to hypervisor mode software if the hypervisor extension is present.

SBI plays important role in RISC-V operating system development. Such a solution makes the kernel much more portable and simplifies virtualization. SBI and implementations are discussed in 2.8.

Hart From the perspective of machine mode, a RISC-V hardware thread (hart) can be perceived as an entity consisting of some context and independent control flow. RISC-V broadens the notion of hart and defines it as an autonomous context and control flow, considered in a specific execution environment (see section 1.1 of [12]). Thereby, a hardware thread is a hart viewed by machine mode. Another great example of a RISC-V hart is a user thread operating in user mode.

Control and status registers (CSRs)

Each hart has a separate address space for control and status registers (CSRs).

Accesses to CSRs are made using dedicated instructions starting with the "CSR" prefix which atomically read-modify-write a selected register. The subset of visible CSRs differs depending on the privilege level the hart operates in. For example, if the only supported extension is "F", then the only read-write CSR visible to user mode hart is the floating-point control and status register `fcsr`.

It is important to be aware of writable CSRs of a hart in a specific execution

environment, as they contribute to the context of the hart and thereby become relevant when performing operations on the context in the kernel (e.g. save/load).

2.1.7 Endianness

In RISC-V endianness has to be considered separately for instructions and data:

- Instructions – always little-endian order,
- Data – modifiable independently for each privilege mode (in specific CSR).

Mimiker assumes little-endian order, thereby, from that point onward, we restrict our consideration to little-endian scenarios.

2.1.8 Instruction length

Generally, RISC-V employs 32-bit, naturally aligned instructions. The default scheme may be changed by additional extensions as RISC-V instruction encoding supports any instruction length dividable by 16 (in bits). Currently, the only extension that modifies the basic instruction encoding is the compressed instructions extension which introduces short instructions encoded on just two bytes.

In order to avoid calculating instruction size in the kernel, we will assume instructions to be 32-bits long (i.e. no "C" extension).

2.1.9 Why is instruction length important?

Although instruction length may seem irrelevant at the first glance, it has an impact on the performance of a system and becomes crucial when venturing into the kernel.

Instruction cache utilization

Instruction length determines how many instructions can fit into an instruction cache line. If more instructions can be accommodated in the cache, the fetch unit can be kept busy which improves utilization of available instruction window slots and refines the overall performance. This is the reason why the compressed instructions extension is so appealing.

Syscall

A syscall causes a trap into the kernel. Upon encountering a trap in RISC-V, PC points at the current instruction (i.e. the faulting instruction). After the syscall is

handled (assuming no restart), the PC at which the thread will resume execution needs to be incremented to point to the instruction after the syscall. The applied offset is equal to the length of the syscall instruction.

Fork

When forking a new thread, the new thread's PC must be set to the instruction after the syscall which implements a fork request. This scenario is different from the syscall scenario described above, as the new thread will enter the userspace for the first time.

Kernel debugger

If a kernel supports debugging of its inner workings (by implementing a debug stub), it may be necessary to be aware of the instruction size, depending on the implementation of the step request.

2.1.10 Memory

The size of the memory address space of a hart is implied by supported base integer ISAs.

Memory is byte-addressable and its layout is stated by the execution environment. Some memory areas may be empty or forbidden in which case access will result in an exception (page fault or protection violation), other regions may contain data/instructions, or maybe marked as I/O regions.

`RISC-V` deploys the `RISC-V` weak memory ordering (RVWMO) memory consistency model. Weak memory ordering introduces some difficulties in SMP systems as different cores may observe memory operations in a different order.

Whether unaligned data accesses are supported depends on the actual implementation. They may be:

- not implemented (misaligned data access causes an exception to be raised),
- implemented in hardware,
- implemented in software (perhaps, by more privileged mode).

2.1.11 Assumed architectural state

The architectural state is part of the context of a user thread and a subset of that state composes the context of a kernel thread. To be able to manage thread contexts in the kernel, we need to establish the shape of the assumed architectural state.

As of this moment, there are three contributors to an architectural state:

- base integer ISA – defines 32 integer registers and the program counter (PC) register,
- floating-point extensions – define 32 floating-point registers and the floating-point control and status register (`fcsr`),
- vector extension – defines 32 vector registers and a number of additional CSRs.

In my solution, I provided support for three target architectural states:

- integer,
- integer + single-precision floating-point,
- integer + single-precision floating-point + double-precision floating-point.

2.1.12 Privilege levels

The main idea behind privilege levels is isolation. Distinguishing different privilege levels and delegating some software to more restricted environments provides natural protection for more sensitive components (e.g. operating system kernel).

`RISC-V` defines three privilege levels (less to more privileged):

- user mode – intended for user applications,
- supervisor mode – destined for operating system kernel,
- machine mode – usually hosts runtime firmware.

Although user and supervisor modes are not mandatory, they are needed for running a conventional operating system on a `RISC-V` CPU.

2.1.13 Employed software stack

`RISC-V` supports various software stacks depending on supported privilege levels, for example, simple embedded systems may be equipped in a CPU that provides only the base machine mode ISA, hence, the software stack consists of a single layer, and isolation doesn't exist.

When porting an operating system, the exact structure of the software stack is not important (and shouldn't be in order to provide easy virtualization), however, when the port is ready and we have to prepare an environment for running the result (either emulator, simulator, or hardware), details begin to matter.

We will assume the most typical software stack:

- user and supervisor privilege levels are supported,
- hypervisor extension isn't supported (i.e. the execution environment for the kernel is provided by runtime firmware running in machine mode).

2.1.14 Machine mode ISA

Although kernel operates entirely in supervisor mode, there are notions in machine mode ISA which impact some components of the kernel, for instance, physical memory protection and physical memory attributes. Besides, when considering an SBI implementation, it's valuable to understand some basic mechanisms provided by the base machine mode ISA, for example, interrupt and exception delegation. Finally, when investigating the ACLINT specification it's precious to be familiar with timer memory-mapped registers defined in the ISA.

Machine trap delegation

By default, whenever a processor encounters an exception or decides to handle an asserted interrupt, it suspends further instruction fetch and regardless of the current privilege level the hart is operating in, the control flow is redirected to machine mode trap handling routine. Although simple and straightforward, this is not a trap handling mechanism we would consider useful when it comes to operating systems. If an environment call from userspace resulted in a jump over the kernel directly to the machine mode runtime, this would be a complete disaster (as syscall handling is implemented in the kernel).

For an operating system to work, we would like most user mode traps to be handled in supervisor mode. But what about supervisor mode traps? On the one hand, we would like supervisor environment calls to be handled by machine mode SBI implementation (that's the whole point of SBI). On the other hand, this doesn't apply to all traps, for instance, if we expected runtime firmware to handle supervisor page faults, that would mean we anticipate it knows the virtual memory map of the kernel!

It's apparent that a mechanism is needed to delegate some user and supervisor traps for handling in supervisor mode while preserving the default behavior for others.

Exception delegation The `medeleg` CSR serves for exception delegation to supervisor mode. If an exception is marked to be delegated, whenever that exception occurs in user or supervisor mode, it will be handled by kernel exception handling routine.

Interrupt delegation The `mideleg` CSR is destined for interrupt delegation to supervisor mode and functions analogously to the exception delegation register.

Timer memory-mapped registers

Without a time measurement facility, the kernel would be unable to provide multitasking and all modules relying on the callout [32] mechanism wouldn't function as expected. If we want to have a functional operating system, we need to measure time. If we want to measure time, we need a device generating interrupts at constant intervals, namely a timer.

`RISC-V` platforms provide a real-time timer operating with constant frequency. The timer is exposed as a 64-bit memory-mapped register accessible for reading and writing in machine mode. Less privileged levels (optionally including user mode) may read the value of the register by reading the `time` CSR.

Beside the memory-mapped time register (referred to by the specification as `mtime`), platforms supply 64-bit time compare memory-mapped register for each hart (`mtimecmp`). It can be accessed exclusively by machine mode software and is used to generate timer interrupts. Whenever `mtime` contains a value greater or equal to `mtimecmp` a machine mode timer interrupt is asserted.

Physical memory attributes (PMA)

In many architectures, page table entries beside basic permission bits (readable, writable, and executable) contain attributes that describe some characteristics of a physical memory region (usually a single page). Among memory attributes configurable in such a way are cacheability (non-cached, write-through, or write back), memory ordering, access granularity, and atomicity. `RISC-V` terms such properties physical memory attributes (PMAs).

`RISC-V` made an insight that such attributes constitute primary properties of the underlying hardware and exposing them in a virtual memory system breaks the abstraction and may be error-prone as an attribute selected for a specific region may in fact conflict with the actual properties of that memory range. To enforce the abstraction and remove potential conflicts, `RISC-V` doesn't embed any properties beside the permission bits inside a page table entry. Instead, PMAs are maintained and checked in a hardware structure known as the PMA checker.

Although the applied solution is elegant and justified, it creates an issue. While in most cases attributes of a physical memory region are fixed and established at design time (thus a need to modify any of them never occurs), in some cases a modification might be needed. The specification suggests, that if it is apparent (or cannot be excluded) that dynamic configuration of some physical memory attributes may be needed, the design should provide a way for achieving this task, for example,

via memory-mapped registers. If some means for PMA modification were supplied, runtime firmware would have to be extended to handle them and some interface for supervisor mode would be eligible (perhaps an enhancement in SBI).

Mimiker and PMAs

Mimiker contains a machine-dependent kernel module for physical address map management (pmap) inspired by the corresponding module in FreeBSD [15].

The pmap module exposes an API to the remaining components of the kernel. The method employed for entering new mappings into a specified physical address map accepts flags that can specify desirable cacheability. However, the target platforms of the 32-bit `RISC-V` port have fixed PMAs, thereby, in the `RISC-V` implementation of the aforementioned module, the provided cacheability flags are simply discarded.

One could argue, that ignoring the supplied flags reduces the portability of the code. What if we wanted to run Mimker on a different `RISC-V` based board? It would be preferable to change as little code as possible, therefore, why not make an SBI call inside the pmap module to request a modification of PMAs to honor the new cacheability options? If it's not supported then it will be discarded and nothing will change. While it seems like a reasonable idea, there is a single problem. At the time of writing this thesis, there is no SBI call to address PMA modification. If an appropriate extension to SBI appears in the future, the code most certainly will require the described change.

Physical memory protection (PMP)

Whereas PMA checker serves for specifying and enforcing physical memory attributes. The physical memory protection (PMP) unit is destined for defining and checking physical memory access permissions.

The supported permissions are:

- readable – physical memory range is readable,
- writable – physical memory range is writable,
- executable – physical memory range supports instruction fetching.

While PMAs describe some profound properties of physical memory, PMP is configurable on a per hart basis and serves for protection and isolation in an SMP system.

The PMP unit is configured through a set of machine mode CSRs.

2.1.15 Supervisor mode ISA

`RISC-V` supervisor mode ISA defines the environment in which our kernel will operate. We will divide our discussion into three parts:

- CSRs – some most basic control and status registers, essential for implementing fundamental kernel mechanisms,
- virtual memory translation system – translation mode used by 32-bit `RISC-V` CPUs equipped with memory management unit (MMU),
- Memory management fence – an instruction regarding address translation control.

Supervisor CSRs

Supervisor mode CSRs constitute an interface used by the kernel to control the underlying hardware (e.g. memory translation and exception handling).

Status register (`sstatus`)

The `sstatus` register controls and reflects the current state of a supervisor hart. The register is divided into several fields, which can be classified into two categories:

- fields meaningful only when the hart has an active trap (i.e. is busy handling an exception or an interrupt),
- remaining fields.

Within each class, we will only explore fields crucial from our point of view.

To begin with, let us examine the first group:

- SPIE – contains the value of SIE before the trap was taken,
- SPP – identifies the privilege mode that encountered the trap.

The second group consists of the following members:

- SIE – globally enables or disables interrupts in supervisor mode,
- FS – keeps track of the floating-point architectural state,
- SUM – enables or disables supervisor access to user pages.

Trap vector register (`stvec`)

The `stvec` register serves for:

- specifying the address of trap handling routine,
- selecting the trap handling mode.

In `RISC-V`, all exceptions delegated for handling in supervisor mode, jump to a common trap vector specified in the `stvec` register.

The specified mode controls how interrupts are handled. Interrupts can either jump to the same location as exceptions (direct mode), or each exception can have a dedicated interrupt vector (vectored mode).

In Mimiker, we employ the direct trap handling mode.

Interrupt control registers

`RISC-V` defines three supervisor mode interrupts:

- software interrupt (SSI),
- timer interrupt (STI),
- external interrupt (SEI).

Besides the aforementioned interrupts, each `RISC-V` based platform can define some custom interrupts, for example, a UART interrupt in a single-core system.

The `sie` register controls which of the supervisor interrupts are enabled.

The `sip` register reflects asserted supervisor interrupts. From the three supervisor interrupts defined by the specification, the only one that can be cleared via this register is the software interrupt. The timer interrupt should be cleared by an SBI call, and the external interrupt should be cleared through the platform level interrupt controller (PLIC, described in a dedicated section) or some platform-specific memory-mapped registers.

Scratch register (`sscratch`)

The `sscratch` register is simply an additional register for supervisor mode usage. It isn't destined for any particular application, however, it is commonly used to support the trap handling mechanism.

When a kernel thread encounters a trap, before any processing can be done, we have to save the machine context of the thread on the kernel stack. It doesn't

present any problems, as all we have to do is decrement the stack pointer, fetch each register composing the context and write it to the allocated space (called trap frame). However, there is a problem regarding user threads. When a user thread encounters a trap, we can't just decrement the stack pointer to allocated space for the context as it points somewhere within the user-space stack (`RISC-V` doesn't automatically switch stacks as `Aarch64` does).

To solve this problem we will apply the following scheme for managing the `sscratch` register:

- When a user thread executes in userspace, then `sscratch` contains a pointer to the context save area on the kernel stack.
- When a user thread executes in kernel space, the `sscratch` holds zero.

Although `sscratch` seems useless in kernel mode when the described scheme is applied, there is one advantage. In trap handler, instead of extracting SPP bits to obtain whether the trap came from user or kernel, we can simply examine the value of the `sscratch` register. Zero means that the trap has occurred in kernel mode, while a non-zero value implies that the trap was caused when the thread was executing in user mode.

Exception program counter register (`sepc`)

The `sepc` register contains the address of the instruction that encountered the latest trap.

When the return from exception (`SRET`) is executed, the next instruction fetch is made from the address contained in `sepc`.

In case of interrupts, the trap is taken before executing the instruction at `sepc`, so there is no problem with re-executing the faulting instruction.

Cause register (`scause`)

The `scause` register identifies the latest trap occurred. It has a dedicated bit that serves for distinguishing interrupts from exceptions and provides a code that further describes the trap.

The full list of defined exception codes can be found in section 3.1.15 of the privileged ISA [13].

Trap value register (`stval`)

From a kernel's perspective, all we need to handle an interrupt is the code obtained from the `scause` register. However, in the case of an exception, the exception code may not be enough.

Whatever environment call a user hart does, the exception code is always the same, and claims purely and simply that a user-mode environment call has been made. The kernel must examine the trap frame of the calling hart to determine which syscall has been requested and to fetch all the parameters enforced by the signature of the syscall.

In the case of an environment call, all remaining information can be acquired from the trap frame, but some exceptions need even more. To reason about a page fault exception, the kernel needs to know the address that has caused the exception to occur. Such address is called the faulting address and can be read from the trap value register (`stval`).

The value of `stval` is dependent on a particular exception, but in most cases provides the faulting address (if relevant).

Address translation and protection register (`satp`)

The `satp` register selects and controls the virtual memory translation system. It consists of three components:

- MODE – specifies the translation mode to use,
- PPN – the physical page number of the root page directory of the translation hierarchy,
- ASID – current address space identifier.

In a 32-bit system, virtual memory can be either disabled (MODE = 0), or the Sv32 memory translation scheme can be applied (MODE = 1).

32-bit virtual memory translation system (Sv32)

When the Sv32 memory translation mode is selected, each user and supervisor mode virtual address is translated via two-level page table structure into a supervisor physical address, which is then examined by the PMP unit, and further translated into a machine physical address. Although it isn't specified where exactly the check should be applied, somewhere along the way, the PMA checker must be incorporated.

The page size in Sv32 is 4KiB.

Virtual addresses An Sv32 virtual address has the layout pictured in table 2.1. The indexes are used for navigation through the paging structure. In general, level 0 index selects an entry in the page directory located using the address found in `satp`, whereas level 1 index points to a leaf entry in the page table obtained in the first step.

bits [31:22]	bits [21:12]	bits [11:0]
Level 0 index	Level 1 index	Page offset

Table 2.1: Sv32 virtual address format

Physical addresses The format of an Sv32 physical address is shown in table 2.2. The physical page number is attained from a leaf page table entry found by traversing the page table structure for a given virtual address. The page offset is directly copied from the corresponding field in the corresponding virtual address.

bits [33:12]	bits [11:0]
Physical page number (PPN)	Page offset

Table 2.2: Sv32 physical address format

Although a physical address is 34-bit long, we will assume that only the first 32 bits are meaningful (thereby, we assume the upper two bits to be hardwired to zero).

Page table entries (PTEs) In the Sv32 memory translation scheme, each page table entry has format depicted in table 2.3.

bits[31:10]	bits [9:8]	bits [7:0]
Physical page number (PPN)	RSW	Access and permission bits

Table 2.3: Sv32 page table entry format

RSW stands for reserved for software usage. These bits are used by the kernel.

The access and permission bits field consists of eight single bit members. The following list presents the bits from least significant to most significant:

- Valid (V) – if a PTE is marked as invalid (V = 0), then all remaining bits in the entry are meaningless to the hardware, and the kernel is free to utilize them in any way.
- Readable (R) – states whether the pointed page is readable.
- Writable (W) – indicates whether the pointed page is writable.

- Executable (X) – implies if the destination page permits instruction fetching.
- User (U) – denotes a user mapping. A user-mode hart can only access user pages.
- Global (G) – designates that the mapping is visible for all address spaces (i.e. the ASID is negligible).
- Accessed (A) – indicates whether the page has already been accessed.
- Dirty (D) – tells if the contents of the page have been modified.

A PTE points to next level page table if and only if R = W = X = 0. If a PTE in a page directory doesn't point to the next level page table, then it creates a mapping for a super page which is a 4MiB page.

Table 2.4 shows all permissible settings of the access and permission bits (the global bit isn't shown as it's irrelevant). Any other configuration will result in a page fault when used in address translation.

Description	D	A	U	XWR	V
page table pointer	*	*	*	000	1
user readable	*	1	1	**1	1
user writable	1	1	1	*11	1
user executable	*	1	1	1**	1
kernel readable	*	1	0	**1	1
kernel writable	1	1	0	*11	1
kernel executable	*	1	0	1**	1

Table 2.4: Permissible settongs of the Sv32 PTE access and permission bits

In Mimiker, we assume access and dirty bits to be unsupported and emulate them in the software.

Memory management fence instruction

Whenever we modify a mapping in the memory translation structure, we need to ensure two conditions:

1. The modification is contributed to the physical memory before any subsequent implicit references read the mapping.
2. The obsolete mapping must be erased from the translation lookaside buffer (TLB).

The supervisor instruction `SFENCE.VMA` satisfies these requirements.

The instruction accepts two register arguments. The first provides the virtual address (if the zero register is used, then the instruction regards all virtual addresses), and the second specifies the address space identifier (if the zero register is used, then the operation concerns all address spaces).

On some implementations, the only implemented variant of the `SFENCE.VMA` instruction is the one with the zero register as both source arguments. On such implementations, other combinations may cause an invalid instruction exception. An example of such implementation is VexRiscv.

Mimiker relies on the aforementioned most general variant of the `SFENCE.VMA` instruction.

2.2 Hart-level interrupt controller (HLIC)

Each `RISC-V` hart contains a local interrupt controller which routes every interrupt before it finally reaches the hart. This interrupt controller is called hart-level interrupt controller (HLIC) [16].

The `RISC-V` specification [13] defines three interrupt sources. The sources are:

- software interrupts,
- timer interrupt,
- external interrupts.

From the supervisor mode software, HLIC is controlled by the supervisor mode interrupt CSRs described in the previous section (`sie` and `sip`), and the only interrupts visible to the kernel are the interrupts managed by these registers.

2.3 Interrupt handling at the platform level

Supervisor software interrupts are caused by SBI calls issued by other harts to implement inter-processor communication (e.g. to synchronize or exchange threads). Supervisor timer interrupts are asserted by the SBI implementation as a result of receiving a machine mode timer interrupt (as only machine mode has access to the timer). Finally, supervisor external interrupt occurs whenever there is at least one peripheral device requesting attention.

In contrast to software and timer interrupts where the fact that an interrupt has occurred is all we need to handle it, an external interrupt is merely an indication that some device requires special care. The kernel needs to investigate exactly which device is causing the interrupt.

The actions required to obtain the asserting device highly depend on the actual platform. We can distinguish two schemes employed by `RISC-V` based hardware platforms:

- the platform consists of a single hart without a peripheral-level interrupt controller (PLIC),
- the platform contains a PLIC.

In platforms without PLIC, the platform will expose some memory-mapped registers accessible from the supervisor mode. At least two registers should be provided:

- interrupt mask register – to enable fine-grained masking of external interrupts,
- interrupt pending register – to provide the kernel with information on which devices require urgent attention.

In Mimiker, we assume PLIC to be present, however, we plan to extend the code to support single-core platforms without PLIC.

2.4 Platform-level interrupt controller (PLIC)

The `RISC-V` platform-level interrupt controller specification was invented to standardize how peripheral interrupts are handled [17].

As PLIC interrupt priorities are optional and not used in my implementation, we will omit them in the following discussion.

2.4.1 A PLIC context

A PLIC is characterized by a number of interrupt sources (up to 1023), and several contexts. A PLIC context consists of the following components:

- enable bits – a single bit for each interrupt source,
- claim/complete register – used for identifying pending interrupts and for signaling end of interrupt (EOI).

Each PLIC context can be linked with one or more privileged modes (supervisor or machine) from one or more harts. Whenever there is a pending interrupt that is enabled in a given context, the external interrupt is asserted for each privilege mode linked with this context.

2.4.2 Exemplary setup

To better understand PLIC, let's examine a usage scenario.

Let's consider a hypothetical `RISC-V` based platform containing two harts. The platform incorporates the following elements:

- two harts, each implementing machine, and supervisor privilege modes,
- PLIC with 32 interrupt sources and 3 contexts,
- UART device capable of generating an interrupt (TX FIFO empty or RX FIFO non-empty).

The UART device is connected to PLIC interrupt source 1. PLIC context 0 and 1 are connected to machine and supervisor mode of hart 0, respectively. PLIC context 2 is connected to both machine and supervisor mode of hart 1. Contexts 1 and 2 enable interrupt source 1, while context 0 keeps it disabled.

Now, let's say that a byte has been queued into the receiver hardware queue, thereby PLIC source 1 becomes pending. Since context 0 ignores the interrupt, nothing happens from its perspective. At the same time, context 1 signals a supervisor external interrupt for hart 0, whereas context 2 signals machine and supervisor external interrupts for hart 1.

2.4.3 Interrupt handling process

When an external interrupt occurs, the following steps should be taken:

1. The claim/complete register should be read to obtain the interrupt source.
2. Proper operations should be performed on the asserting device to satisfy the interrupt.
3. The claim/complete register should be written with the interrupt source number to signal the end of the interrupt.

2.4.4 Why use PLIC in Mimiker?

As Mimiker is restricted to a single core, PLIC isn't mandatory. However, some benefits justify incorporating PLIC:

- Portability – PLIC is widely used in the `RISC-V` world. An outright PLIC driver makes an operating system much more portable than relying on some platform-specific solutions. If we were to port Mimiker on some of the SiFive boards, PLIC would be inevitable.

- Isolation – PLIC contexts can provide isolation between different privilege modes, for instance, machine mode runtime firmware may wish to operate on external devices independently of the supervisor mode kernel.

2.5 Advanced core local interruptor (ACLINT)

The `RISC-V` advanced core local interruptor specification [19] defines three memory-mapped devices:

- Machine-level timer device (MTIMER) – standardizes how machine timer memory-mapped registers are implemented.
- Machine-level software interrupt device (MSWI) – defines a standard interface for managing machine software interrupts.
- Supervisor-level software interrupt device (SSWI) – defines a standard interface for generating software interrupts.

2.5.1 MTIMER

The MTIMER device provides a single fixed-frequency 64-bit monotonic `mtime` register which is shared among all connected harts, and a single `mtimecmp` register for each connected hart.

2.5.2 MSWI

The MSWI device provides the `msip` register for each connected hart. Each register is 32-bits long with the upper 31 bits hardwired to 0. The least significant bit serves for setting and clearing the machine software interrupt on the corresponding hart.

2.5.3 SSWI

The SSWI device supplies the `setssip` register for each connected hart. The `setssip` registers work analogously to the `msip` registers, with two differences:

- `setssip` regards supervisor software interrupts,
- `setssip` only can generate an interrupt, the interrupt can be cleared via `mip` or `sip`.

2.6 SiFive core local interruptor (CLINT)

The SiFive core local interruptor (CLINT) is a compound of MTIMER and MSWI ACLINT compatible devices, thereby, providing an implementation for the machine memory-mapped timer registers along with means to control machine software interrupts for each connected hart [18].

CLINT is the most common implementation of ACLINT devices (excluding the SSWI device). CLINT was the inspiration for the ACLINT specification.

2.6.1 Why use CLINT in Mimiker?

Although CLINT (and ACLINT in general) is intended for deployment in systems with multiple `RISC-V` harts, it is still profitable to support CLINT in Mimiker. It can be argued as follows:

- Portability – as with PLIC, it is the most valuable benefit.
- Small overhead – while it may seem costly to provide a CLINT in a unicore system, in fact, we still need to implement the machine memory-mapped registers (which in CLINT is done by the MTIMER device). The only overhead is the requirement to implement an additional 4-byte `msip` register with the ability to set and clear machine mode software interrupt.

2.7 LiteX

LiteX is an open-source SoC builder [20]. It is hosted on GitHub and maintained by the Enjoy-Digital company.

LiteX supports many soft cores and provides some essential peripheral devices including PCIe and Ethernet cores.

The tool is very flexible and can be used in various ways depending on the user's background and needs. We utilize LiteX to accomplish the following goals:

- provide an implementation of a platform to run the prepared Mimiker port,
- generate a device tree conveying the target platform,
- generate a platform description used by our system emulator.

2.7.1 Mimiker RISC-V target platform

The Mimiker `RISC-V` port is destined to run on LiteX generated platforms incorporating a 32-bit `RISC-V` softcore. Additionally, the following assumptions have been

made:

- the softcore provides a CLINT device,
- the softcore provides a PLIC device,
- the LiteX UART (LiteUART) peripheral is employed and its interrupt is connected to PLIC.

CLINT, PLIC, and LiteUART devices are mandatory to run Mimiker. In the future, I would like to add support for soft cores which don't support CLINT or PLIC.

From that point onward, whenever we refer to a Mimiker `RISC-V` target platform we mean a LiteX platform fulfilling the aforementioned assumptions.

2.8 Supervisor binary interface (SBI)

The `RISC-V` supervisor binary interface (SBI) specification defines an interface between the supervisor software and supervisor execution environment (machine mode software in our case) [21].

SBI defines exactly two things:

- a set of available services (in the form of extensions and functions),
- a binary encoding (i.e. calling convention on the binary level).

2.8.1 Extensions and functions

Much like the `RISC-V` ISA, SBI defines a mandatory base called the base extension, along with a number of optional extensions intended for a particular application. Besides the standard extensions, each implementation may provide custom SBI extensions.

Each SBI extension (including the base extension) is identified by an extension identifier (EID) and defines a number of functions, each distinguished by a relative function identifier (FID) and providing some service for the calling hart. The only exception of that principle is legacy extensions which define only a single function and are identified solely by the EID.

2.8.2 Calling scheme

SBI calls resemble function calls with two differences:

- a call is made using the environment call instruction (ECALL) instead of the regular routine call instruction (CALL),
- argument registers `a7` and `a6` are destined for EID and FID, respectively.

Upon return, `a0` contains an error code, whereas `a1` contains a return value. The validity of the aforementioned registers depends on the signature of the specific function.

2.8.3 Timer extension

For now, besides the base extension, Mimiker only uses one additional extension - the timer extension.

The timer extension defines a single function called set timer. The set timer function accepts a single 64-bit argument specifying when the next timer interrupt should occur. Calling the function with a parameter greater than the current value of the `time` CSR will clear any pending timer interrupt.

2.8.4 Mimiker SBI library

In order to nimbly incorporate SBI calls to kernel, some handy wrappers should be provided to make the code more legible. Besides, some functions can be called once during kernel bootstrap and the results can be cached (e.g. get machine architecture ID). Furthermore, employing SBI requires some initial reasoning to be performed, for instance, to ensure that all functionalities the kernel relies on are supported. Such operations make a great candidate for an initialization function of a kernel module.

In Mimiker, I have provided an SBI kernel library. The library is ported from FreeBSD (with slight modifications).

The key function of the module is the `sbi_call` function which implements a generic SBI call. Each SBI function wrapper is implemented by calling this function with appropriate EID and FID.

2.8.5 SBI implementation

When preparing to run the code on an emulator or hardware, we must provide a supervisor execution environment (SEE) also known as SBI implementation, as someone needs to receive and fulfill requests issued by the kernel.

Considering the software stack we have chosen, an SBI implementation will take the form of software running in machine mode. Such software is known as runtime firmware.

```
static sbi_ret_t sbi_call(unsigned long ext, unsigned long func,
                          unsigned long arg0, unsigned long arg1,
                          unsigned long arg2, unsigned long arg3,
                          unsigned long arg4) {
  register register_t a0 __asm("a0") = (register_t)(arg0);
  register register_t a1 __asm("a1") = (register_t)(arg1);
  register register_t a2 __asm("a2") = (register_t)(arg2);
  register register_t a3 __asm("a3") = (register_t)(arg3);
  register register_t a4 __asm("a4") = (register_t)(arg4);
  register register_t a6 __asm("a6") = (register_t)(func);
  register register_t a7 __asm("a7") = (register_t)(ext);

  __asm __volatile("ecall"
                   : "+r"(a0), "+r"(a1)
                   : "r"(a2), "r"(a3), "r"(a4), "r"(a6), "r"(a7)
                   : "memory");

  return (sbi_ret_t){
    .error = a0,
    .value = a1,
  };
}
```

Listing 1: SBI call wrapper (`sbi.c`)

There are many SBI implementations with the most popular one being OpenSBI. We use OpenSBI to create the software stack for running Mimiker.

2.9 OpenSBI

OpenSBI is an open-source reference implementation of SBI destined to be used as runtime firmware [22].

OpenSBI isn't meant for any particular platform. The main component of OpenSBI is a platform-independent static library called `libsbi.a`. The library can be used in two ways:

- to build more sophisticated firmware, for example, custom SBI implementation or bootloader,
- it can be integrated with a platform-specific component to generate a compound library or runtime firmware.

2.9.1 Platform implementation

It is important to realize that although SBI provides some abstract models to the kernel, the models must be implemented in a platform-specific way.

Each target platform must provide an `sbi_platform` struct containing a full description of the target platform. The most vital member accommodated in the platform structure is an `sbi_platform_operations` struct. `sbi_platform_operations` defines an interface used by SBI to handle platform-specific operations. Most of the functions composing the interface are init/exit callbacks, for instance, `ipi_init`, `ipi_exit`, `timer_init`, and `timer_exit`.

In the platform-independent layer, OpenSBI contains drivers for the most common devices including all ACLINT devices and PLIC. In the majority of cases, to implement a platform in OpenSBI, we have to describe incorporated devices (e.g. MTIMER base address) and announce them inappropriate init functions exposed in the platform operations interface.

Besides the implementation of the platform structure, (in a simple scenario) each platform must specify an entry point address of the supervisor mode software.

OpenSBI provides some reference platform implementations (e.g. implementation used by QEMU) along with a template to simplify adding new targets. Unfortunately, OpenSBI doesn't contain implementation for LiteX generated platforms.

2.9.2 OpenSBI on LiteX

Enjoy-digital provides an open-source OpenSBI platform implementation for the LiteX VexRiscv platform [23], however, the code is seriously obsolete and doesn't include PLIC.

Using the supplied template combined with the aforementioned code, I prepared an updated implementation for the target LiteX VexRiscv platform. Although the code is destined for this particular platform, it should work correctly for all target platforms we support. The implementation includes the following functionalities:

- Console – I provided a primitive LiteUART driver used to implement the legacy console extensions and to print any OpenSBI logs (including the initial ASCII art).
- ACLINT – I utilized the supported OpenSBI drivers for MTIMER and MSWI devices.
- PLIC – I included a PLIC device (which is currently unused by the implementation, as the supplied LiteUART driver doesn't rely on interrupts).

The entry point for Mimiker is dependent on the exact physical memory layout of the target platform.

2.9.3 Supervisor mode initial environment

OpenSBI defines the following initial state for each hart upon entering the supervisor mode:

- `satp` register is set to 0 thereby the virtual memory translation is off and the hart operates on physical addresses,
- supervisor interrupts are globally disabled (i.e. SIE in `sstatus` if cleared),
- `a0` contains the hart identifier of the hart,
- `a1` contains the physical address of device tree blob (DTB),
- the state of remaining registers is undefined and the hart shouldn't assume any specific values.

Besides the initial setup of registers, OpenSBI delegates supervisor interrupts (SSI, STI, and SEI) and page fault exceptions for handling in supervisor mode.

Although only the three page fault exceptions are delegated, the supervisor should still expect to encounter a broader range of exceptions as OpenSBI may redirect exceptions that are initially processed in machine mode, down to supervisor mode. A good example of such a scenario is the illegal instruction exception. When an illegal instruction is encountered, control is redirected to the machine mode trap handler. If the instruction is emulated by OpenSBI, then the handler returns in the standard fashion, and instruction fetch resumes. However, if the instruction isn't emulated, OpenSBI will set supervisor CSRs related to exception handling and return from exception (`MRET`) to supervisor trap handler. From the perspective of supervisor mode, it appears exactly as if the exception was handled directly by the supervisor mode.

2.10 Application binary interface (ABI)

Whereas supervisor binary interface defines available functions exposed by the supervisor execution environment along with calling convention (i.e. SBI is just an environment execution interface (EEI)), application binary interface (ABI) is different.

Application binary interface defines communication between two binary modules at each privilege level. Every two binary modules making up a user, supervisor, or machine mode program communicate with each other using ABI.

`RISC-V` defines a number of ABIs which differ regarding argument and return value passing [24].

In contrast to SBI, a generic ABI doesn't contain a definition of the interface between user-mode applications and supervisor mode kernel. The syscall interface is OS-dependent and thereby not included.

The processor-specific ABI is composed of the following components:

- calling convention,
- `RISC-V`-specific ELF aspects,
- `RISC-V`-specific DWARF aspects.

2.10.1 Calling convention

The `RISC-V` calling convention defines:

- Register convention – for each register set (integer, floating-point, and vector) determines which registers should be preserved across procedure calls, and what is the destination of each register (e.g. which one is used to store return address).
- Argument and result passing – how are integer and floating-point arguments passed within procedure calls, how aggregated types are handled, variadic functions, stack alignment, and more.
- defined ABIs – The choice of specific ABI depends on supported extensions.
- C/C++ basic type sizes and alignments – for each ABI defines sizes and alignment of primitive data types (e.g. `char`, `int`, `void *`, and `double`).

2.10.2 ELF

The ELF component of the `RISC-V` psABI defines all ELF-related `RISC-V` notions and definitions. This includes the following:

- available code models (small and medium),
- new definitions regarding the ELF format itself (e.g. `RISC-V` machine target),
- linker relocations.

2.10.3 DWARF

The chapter dedicted to DWARF debugging format defines a mapping between DWARF register numbers and `RISC-V` registers (integer, floating-point, vector, and CSRs).

2.11 Device tree

Device tree provides a standardized way of describing the organization of a computer [25]. A device tree is organized in a set of nodes. A node usually represents a single device which in turn defines a set of properties and potentially a number of subnodes representing its child devices.

Device tree is used mainly in embedded systems which don't provide a way for discovering attached devices (e.g. PCI device enumeration).

A typical device tree contains the following components:

- kernel command line,
- initial ramdisk boundaries,
- a compound node (usually called "cpus") that contains a node for each provided CPU,
- a memory node,
- a reserved memory node,
- a compound node (usually called "soc") that accommodates a node for each peripheral device.

2.11.1 Device tree formats

Device tree defines two file formats:

- device tree source (DTS) – a human-readable device tree format,
- device tree blob (DTB) – a compiled DTS.

Additionally, a compiled device tree source is sometimes referred to as flattened device tree (FDT).

Chapter 3

Mimiker port

Mimiker machine-dependent components can be divided into elements constituting kernel space and elements composing user space.

Kernel machine-dependent components can be further divided into constituents reliant on the CPU architecture and elements reliant on the target platform.

The target architecture introduces:

- a CPU model,
- architecture-specific devices, for instance, interrupt controllers (HLIC and PLIC) and timer (MTIMER).

The target platform defines:

- physical memory map (ROM, SRAM, DRAM, I/O regions),
- peripheral devices (e.g. UART and eMMC controller).

Userspace machine-dependent components confine to system libraries.

We will structure our discussion as follows. First, we will examine the physical memory map and virtual memory layout. Afterward, we will elaborate on the implementation of the architecture-dependent components, excluding `RISC-V` implied device drivers. Thereafter, we will discuss device drivers required by the architecture and target platform. Finally, we will move to user space and explore machine-dependent aspects of system libraries.

3.1 Memory map

In order to prepare a port of an operating system, we need to become familiar with the physical address space presented by the target platform and establish a layout

of the virtual address space in which the kernel and user processes will operate.

3.1.1 Physical address space

The layout of physical memory is dependent on the target platform. A typical physical memory map of a LiteX generated platform looks like this (small addresses to high addresses):

- a small address range corresponding to an on-chip ROM memory,
- a range mapped to an SRAM memory of similar size as the ROM region,
- a significant address span representing DRAM (called the main memory),
- irregular range for memory-mapped devices.

3.1.2 Why is the physical memory map important?

From our perspective, the most important properties of the physical address space are:

- the boundaries of the DRAM backed address range,
- the beginning of the I/O region, and which portions of it are inhabited (i.e. the boundaries of each memory-mapped device).

Main memory boundaries

At the very beginning of the kernel bootstrap process, before memory translation is turned on, the kernel operates on physical addresses. The portion of the kernel that operates on bare memory is called the `boot` segment. As Mimiker is linked as a static binary, we need to specify the physical address of the `boot` segment in the kernel linker script in order to appropriately resolve any relocations.

Moreover, the main memory range is crucial to construct a direct map (dmap) needed to manage memory translation structures.

Besides, main memory boundaries are required by the physical memory management module (physmem) [33]. physmem must be aware of every unreserved main memory region, as it maintains information about each physical page in the system.

I/O space

The root bus device contains a manager meant for resource allocation on behalf of child devices and buses. The manager must be given the boundaries of the I/O space of the platform.

3.1.3 Virtual address space

Except for the `boot` segment, all remaining segments constituting the kernel image operate fully in virtual address space.

Whereas physical memory layout is defined by the target platform, virtual memory layout is wholly programmable. It is up to the kernel developer to establish a layout of the virtual memory.

In `RISC-V`, there is a single translation structure for both kernel and userspace. The majority of operating systems ported to architectures with this feature employ one of the following virtual memory layouts:

- fifty-fifty – the bottom half of the virtual address space is devoted to user programs, while the top half is meant for the kernel.
- three-quarters to one-quarter – the bottom three-quarters is intended for userspace and the remaining part is used by the kernel.

For this port, I chose the fifty-fifty variant. The kernel macro defining the start of the virtual address space of the kernel is called `KERNEL_SPACE_BEGIN` and is defined to 0x80000000.

3.2 RISC-V kernel

This section will describe the implementation of architecture-dependent components of the Mimiker kernel with the exception of device drivers related to the CPU which will be discussed in the next section.

3.2.1 Kernel linker script

Linker scripts are used to specify the final layout of the generated binary. They describe the output format of the binary, program entry point, how input sections are mapped to output sections, what is the address (virtual and load) of each output section, and how program headers are constructed.

Preprocessed linker scripts

In Mimimker, each architecture has a dedicated linker script. Nevertheless, we would like to have a single linker script for as many platforms as possible. However, there is a problem. As described in 3.1, main memory boundaries are platform-specific and the physical address of the `boot` segment must be specified in the linker script. Having said that, there is also good news. The address of the `boot` segment seems

to be the only platform-specific aspect that has to be contained in the `RISC-V` kernel linker script for Mimiker.

The above reasoning was my motivation for introducing preprocessed linker scripts. Preprocessed linker scripts are processed using the C preprocessor (cpp) to obtain an actual linker script used by the linker to produce kernel executable binary.

Equipped with preprocessed linker scripts, we can establish the following convention:

- each `RISC-V` target platform defines a macro `KERNEL_PHYS` which expresses the physical address of the `boot` segment (not to confuse with the start address of the main memory),
- the kernel linker script for `RISC-V` utilizes the aforementioned macro instead of using any explicit value.

Kernel output format

The linker generates a standard 32-bit little-endian ELF executable. Although some emulators and bootloaders are capable of processing ELF binaries, our emulator for LiteX platforms (Renode) anticipates a raw binary image. The generated ELF is used for debugging (as it accommodates debug info), and the final raw binary is produced using the objcopy tool from our toolchain.

It is worth mentioning, that using a raw binary has some benefits, for instance:

- a raw binary is usually much smaller than the corresponding ELF,
- using a raw binary enables us to use a simpler bootloader as all it has to do is to copy the binary image to the specified address.

Kernel entry point

The entry point of the kernel is defined to be the `_start` function defined in `start.S`.

Output sections

The provided kernel linker script employs the following output sections:

- `.boot` – composes the `boot` segment. It is loaded at `KERNEL_PHYS` and operates on bare physical memory.
- `.text` – contains the code of all kernel modules and libraries.
- `.eh_frame` – contains metadata including CFI related contents.

- `.rodata` – read-only kernel data.
- `.data` – initialized data.
- `.sdata` – small data and read-only data input sections. Global pointer is defined to point around the middle of this section.
- `.bss` – uninitialized data. This in a nobits section meaning that it occupies no space in the file (ELF as well as raw binary).

Besides the listed sections, there are many sections corresponding to linker sets.

The `start.o` file needs to be the first file included in the `.boot` section ❶ as it contains kernel entry point and our output format is raw binary.

Virtual memory address (VMA) of the `.text` section ❷ is defined to be at the start of the virtual address space plus the offset of the `.text` load memory address (LMA) from `KERNEL_PHYS`. This enables us to convert a virtual address within the image to the corresponding physical address by clearing bits set in `KERNEL_SPACE_BEGIN` and adding `KERNEL_PHYS`.

```
1  .boot KERNEL_PHYS : AT(KERNEL_PHYS) ALIGN(4096)
2  {
3    __boot = ABSOLUTE(.);
4  ❶KEEP(riscv/riscv.ka:start.o)
5    *(.boot .boot.*)
6    . = ALIGN(4096);
7    __eboot = ABSOLUTE(.);
8  } : boot
9
10 HIDDEN(_boot_size = __eboot - __boot);
11
12 ❷.text KERNEL_SPACE_BEGIN + _boot_size : AT(__eboot) ALIGN(4096)
13 {
14   __kernel_start = ABSOLUTE(.);
15   __text = ABSOLUTE(.);
16   *(.text .text.*)
17   __etext = ABSOLUTE(.);
18 } : text
```

Listing 2: `.boot` and `.text` sections (`riscv.ld.in`)

Kernel segments

A loadable program header is called a segment. Mimiker kernel image consists of four segments that aggregate selected output sections.

Beside the segments corresponding to program headers, the `.bss` section constitutes the `bss` segment.

Kernel linker symbols

The linker script defines a few linker symbols which have external linkage within the kernel and are used during bootstrap, for instance, to map segments to virtual memory and zero the `bss` segment.

3.2.2 Direct map

A direct map is a contiguous kernel virtual memory area that is mapped one to one to the entire main memory. It is used to manipulate physical pages, for instance, to modify mappings in memory management structures, or to zero or copy a designated physical page.

I divided the virtual address space of the kernel into two parts:

- the first half contains allocatable virtual memory and kernel image (excluding the `boot` segment),
- the second half is devoted to the direct map.

For the 32-bit variant of the architecture, the above scheme assumes that the size of the main memory doesn't exceed 1GiB.

3.2.3 Libkern

Each user space program can be linked against the standard C library to benefit from a wide range of handy functions, for example, string manipulation functions. But if we wanted to call `memcpy` within a kernel module?

The libkern library is a small kernel library implementing a selected subset of functions defined by the standard library.

Mimiker builds libkern from the same source files as libc for user programs using its flexible build system. If the programmer needs a not yet included functionality provided by libc, makefiles can be modified to include the desirable feature.

How is libkern machine-dependent?

Some of the functions composing libkern are implemented separately for each architecture as they are written directly in assembly to gain better performance.

3.2.4 Generic assembly

Although my work is focused on the 32-bit variant of `RISC-V`, I strongly believe that we will expand to include the 64-bit counterpart in the range of supported architectures. This encouraged me to employ a generic `RISC-V` assembly introduced by NetBSD [26].

Generic assembly introduces a C preprocessor macro for each assembly instruction that has a different mnemonic in 32- and 64-bit variants. All other instructions remain in the usual form.

```
1  #if __riscv_xlen == 64
2  #define PTR_L ld
3  #define PTR_S sd
4  #define PTR_LR lr.d
5  #define PTR_SC sc.d
6  #define PTR_WORD .dword
7  #define PTR_SCALESHIFT 3
8  #else
9  #define PTR_L lw
10 #define PTR_S sw
11 #define PTR_LR lr.w
12 #define PTR_SC sc.w
13 #define PTR_WORD .word
14 #define PTR_SCALESHIFT 2
15 #endif
```

Listing 3: Generic integer assembly (`asm.h`)

The same abstraction is applied to floating-point operations dependent on the precision of supported extension.

```
1  #ifdef __riscv_d
2  #define FP_L fld
3  #define FP_S fsd
4  #elif defined(__riscv_f)
5  #define FP_L flw
6  #define FP_S fsw
7  #endif
```

Listing 4: Generic floating-point assembly (`asm.h`)

Besides the described features, generic assembly supplies some aliases for common assembly instructions which usage is optional.

3.2.5 thread0

Although it is not a machine-dependent aspect, it is worthwhile to wonder which software entity is responsible for performing the bootstrap process.

The control flow that performs the bootstrap of the kernel is known as thread0. thread0 is a statically allocated thread constituting a statically allocated process termed proc0. Although it is attached to a process, it never enters user space. In fact, the only reason why it has a parent process is to unify the fork logic as thread0 issues a fork to spawn the init process after the bootstrap is done.

After the init process is created, thread0 turns into the idle thread of the hart it executes on.

3.2.6 Bare memory boot

The `RISC-V` kernel boot process is divided into two stages:

- bare memory boot – operates directly in physical memory,
- virtual memory boot – runs in virtual memory in the environment prepared by the former phase.

The bare memory boot is contained in the `boot` segment and its sole goal is to bring the kernel into virtual address space.

Kernel entry point

The first boot stage begins with `_start` which is the entry point of the kernel.

As we employ linker relaxations [39], the first thing we should do is set up the global pointer for the kernel ❶.

If we were to run the kernel on a multihart system, we would have to distinguish one hart to perform the global initialization and redirect the remaining harts to execute the hart-local initialization. For the time being, Mimiker utilizes only a single hart with hart identifier equal to 0 ❷. If there is more, they will stay idle.

As stack pointer is among the registers that remain in an undefined state upon entry, we have to set an initial stack before calling any functions ❸.

At the end, we move to `riscv_init` and pass the DTB address as the argument ❹.

```
1 _ENTRY(_start)
2       ❶LOAD_GP()
3
```

```
4         ❷bnez a0, halt
5
6           /* Move to the initial stack. */
7         ❸PTR_LA sp, initstack_end
8
9           mv a0, a1
10        ❹tail riscv_init
11  halt:
12          wfi
13          j halt
14  _END(_start)
```

Listing 5: Kernel entry point (`start.S`)

RISC-V init

The remaining part of the bare memory boot is performed within `riscv_init`. The purpose of this function is to set up an environment for the second stage boot and handle the control over to `riscv_boot` which implements the virtual memory boot phase.

To prepare the environment, we have to create a kernel page directory along with a few page tables. These constructs require some additional physical memory to be allocated. The boot memory allocator was designed for this purpose. The allocator accepts a requested size, rounds it up to page size multiple, and extends the end of the kernel image in physical memory. The initialization function ❶ sets the kernel's physical brk pointer to the end of the kernel's image and sets the top boundary that the brk pointer can reach.

After we allocate memory for the page directory, we map the kernel image (i.e. text, rodata, data, and bss segments) into virtual memory.

The virtual memory boot phase has to have access to the DTB as it contains kernel command line, ramdisk address, and information regarding CPU and all peripheral devices. However, there is an issue. When we turn on memory translation, we can no longer access the DTB using its physical address. Moreover, we cannot use the direct map as to construct the direct map we need to know the main memory boundaries which are contained in the DTB itself. To solve this problem, we will temporarily map the DTB and the kernel page directory into the virtual memory area destined for the direct map. When the mappings are no longer needed, they will be overwritten by the direct map.

Before we will move to the second stage, there is still one more thing to do. While performing kernel machine-independent initialization, we will initialize the physmem module. The physmem initialization function will allocate a structure for each physical page and insert the structure into a free list used by the buddy system

(i.e. physical page allocation algorithm). The allocated structures will be mapped right after the kernel image in virtual memory by the pmap module. However, if during that process, pmap will encounter an invalid page directory entry, it will try to allocate a physical page for a page table using the physical page allocator which is not initialized yet. This will result in a kernel panic. To avoid this scenario, I introduced the `vm_page_ensure_pts` function which serves the purpose of ensuring that some established amount of page directory entries mapping virtual addresses right after the kernel image are valid (i.e. they point to allocated page tables).

Afterward, it is time to enable address translation and move to the virtual memory boot phase implemented by `riscv_boot`. To accomplish that, we use the following scheme:

1. We don't map the `boot` segment into virtual memory.
2. We ensure that boot physical addresses don't overlap with virtual addresses of the mapped kernel image ❶. This requisite is crucial as we assume that the first instruction fetch after enabling memory translation ❻ causes an instruction fetch page fault.
3. We temporarily set the supervisor trap vector `stvec` ❷.
4. We set the DTB physical address and the kernel page directory physical address as the arguments for `riscv_boot` ❸.
5. We set the stack pointer to the virtual memory boot stack pointer ❹.
6. Finally, we enable the virtual memory translation system ❺.

```
1  __boot_text __noreturn void riscv_init(paddr_t dtb) {
2  ❶if (!(__eboot < __kernel_start || __kernel_end < __boot))
3      halt();
4
5    bootmem_init();
6
7    /* Create kernel page directory. */
8    pd_entry_t *pde = bootmem_alloc(PAGESIZE);
9
10   map_kernel_image(pde);
11   map_dtb(dtb, pde);
12   map_pd(pde);
13   vm_page_ensure_pts(pde);
14
15   /* Temporarily set the trap vector. */
16 ❷csr_write(stvec, riscv_boot);
17
18   /*
19    * Move to VM boot stage.
20    */
```

```
21    const paddr_t satp = SATP_MODE_SV32 | ((paddr_t)pde >> PAGE_SHIFT);
22    void *boot_sp = &boot_stack[PAGESIZE];
23
24    __sfence_vma();
25
26  ❸__asm __volatile("mv a0, %0\n\t"
27                     "mv a1, %1\n\t"
28                   ❹"mv sp, %2\n\t"
29                   ❺"csrw satp, %3\n\t"
30                   ❻"nop" /* triggers instruction fetch page fault */
31                     :
32                     : "r"(dtb), "r"(pde), "r"(boot_sp), "r"(satp)
33                     : "a0", "a1");
34
35    __unreachable();
36  }
```

Listing 6: RISC-V init (boot.c)

3.2.7 Virtual memory boot

The purpose of the virtual memory boot stage is to bring the kernel to the state where the board initialization process can be performed.

The first thing to do is setting registers to obey the kernel register usage convention ❶:

- tp – the thread pointer register always points to the PCPU structure of the hart,
- gp – the global pointer should be set close to the middle of the small data output section (.sdata),
- sscratch – the use scheme of this register was described in 2.1.15.

The stvec register must be set to the address of the trap handling routine (cpu_exception_handler) ❷. The trap handling mode of choice is the direct mode, that is, all exceptions and interrupts trap to the same trap vector.

Although the bss segment has already been mapped, the physical pages backing that segment contain undefined contents. Therefore, before we can advance, the bss virtual memory area has to be zeroed.

The first kernel mode that gets initialized is the klog module which is used for logging kernel messages [34]. It is essential to initialize it as soon as possible as it is used throughout the entire kernel and implements such major functions as assert and panic.

Now it's time to process the DTB. The data retrieved from the DTB is maintained in a dedicated kernel module called kernel environment (kenv) [35]. The module is used across the bootstrap process including the board initialization phase. The `board_stack` function (described in the next subsection) handles this task and returns a new stack pointer that points at the bottom of stack of thread0.

Before we can depart to board initialization, we need to supply the pmap module with the physical address of the kernel page directory and build the direct map required for further operation of the pmap module. These tasks are accomplished by `pmap_bootstrap`. The direct map may be created as the DTB has been processed and main memory boundaries can be read from the kernel environment.

Eventually, we switch to the thread0's stack and proceed to board initialization ❸.

```
1  static __noreturn void riscv_boot(paddr_t dtb, paddr_t pde) {
2    /*
3     * Set initial register values.
4     */
5  ❶__set_tp();
6    csr_write(sscratch, 0);
7
8    /*
9     * Set trap vector base address:
10    *  - MODE = Direct - All exceptions set PC to BASE
11    */
12 ❷csr_write(stvec, cpu_exception_handler);
13
14   clear_bss();
15
16   init_klog();
17
18   void *sp = board_stack(dtb, BOOT_DTB_VADDR);
19
20   pmap_bootstrap(pde, BOOT_PD_VADDR);
21
22   /*
23    * Switch to thread0's stack and perform `board_init`.
24    */
25 ❸__asm __volatile("mv sp, %0\n\t"
26                    "tail board_init" ::"r"(sp));
27   __unreachable();
28 }
```

Listing 7: `RISC-V` boot (`boot.c`)

3.2.8 Board stack

As building the kernel environment requires extensive usage of the DTB module, we need to initialize that module first ❶.

When forking the init process, the new thread's user machine context is copied from the context of thread0. However, the statically allocated thread0 doesn't have a statically assigned user machine context. To avoid a kernel oops, we set the user machine context pointer of thread0 to a buffer allocated on its stack ❷.

Kernel environment consists of the following members:

- main memory start address and size,
- reserved memory start address and size (we assume a single reserved range for OpenSBI),
- ramdisk start address and size,
- tokens specified in the kernel command line.

The kernel command line is further divided into kernel arguments and init program arguments. The kernel arguments group includes the absolute path to the init program. The two groups of arguments are separated by "–".

All the elements described above are delivered as properties in the DTB. They will be retrieved and copied into the thread0's stack.

Along with copying the properties into the stack, two argument vectors will be constructed. The first one composes the kernel environment and contains pointers to all the above elements excluding the tokens corresponding to init arguments. The second one constitutes the argument vector for the init program.

❸ shows the allocation of both vectors. They are contained within a single buffer and the two additional pointers serve for terminating the vectors.

After the vectors are populated and the properties are copied to the stack. The vectors are handed to the kenv module.

Since the virtual memory boot has started, thread0 has been executing on a temporary virtual memory stack. `board_stack` returns a pointer to the bottom of the thread0's stack so that the virtual memory boot process can switch to it.

```
1 void *board_stack(paddr_t dtb_pa, vaddr_t dtb_va) {
2 ❶dtb_early_init(dtb_pa, dtb_va);
3
4   kstack_t *stk = &thread0.td_kstack;
5
6 ❷thread0.td_uctx = kstack_alloc_s(stk, mcontext_t);
7
```

```
8   ❸const size_t nptrs = count_args() + 2;
9     char **kenvp = kstack_alloc(stk, nptrs * sizeof(char *));
10
11    process_dtb(kenvp, stk);
12    kstack_fix_bottom(stk);
13
14    init_kenv(kenvp);
15
16  ❹return stk->stk_ptr;
17  }
```

Listing 8: `RISC-V` board stack (`board.c`)

3.2.9 Board initialization

Board initialization is the last machine-dependent phase of the bootstrap. The purpose of this stage is to perform any remaining operations that weren't performed earlier, for instance, due to the lack of the kernel environment.

While we have already initialized the klog module, the kernel command line could have contained a customized klog mask (which is used for filtering kernel logs). Thereby, we need to update the mask which is achieved by `klog_config`.

As was claimed in 2.8.4, the SBI module requires some initial processing to be done before the kernel can rely on SBI services.

Before physical memory management module will be initialized, we must provide it with the information of available main memory regions. The unusable main memory regions are:

- kernel image (including pages allocated by boot memory allocator),
- device tree blob,
- OpenSBI,
- initial ramdisk.

All remaining regions are usable and should be reported to physmem. `physmem_regions` takes care of it.

Until now, interrupts have been globally disabled (as SBI ensures that `sstatus.SIE` is 0 upon entry to the supervisor mode). Before we will advance, we will mask each individual supervisor interrupt and enable interrupts globally ❶. Each specific interrupt will be enabled when some device will register to utilize it.

The remaining part of the bootstrap is performed by machine-independent `kernel_init`.

```
1  void __noreturn board_init(void) {
2    init_kasan();
3    klog_config();
4    init_sbi();
5    physmem_regions();
6    /* Disable each supervisor interrupt. */
7  ❶csr_clear(sie, SIE_SEIE | SIE_STIE | SIE_SSIE);
8    intr_enable();
9    kernel_init();
10 }
```

Listing 9: `RISC-V` board initialization (`board.c`)

3.2.10 Trap handling

As we deploy the direct trap handling mode, the first thing we do after a trap occurs is obtain whether the trap has been caused in supervisor mode or rather in user mode as both cases are handled differently.

Regardless of the privilege mode that has caused the trap, before a more sophisticated C handler can be called, a trap frame must be created. A trap frame consists of:

- integer registers,
- program counter,
- status register,
- trap value (`stval`, usually the faulting address),
- trap cause (`scause`).

In Mimiker, the trap frame has the same layout as the CPU context, thereby there is no need to introduce a new construct and the `ctx_t` structure is used instead.

If the trap came from the supervisor, the trap frame can be allocated directly on the stack. Otherwise, the trap frame is saved at the address retrieved from the `sscratch` register which points to the user machine context save area allocated on the kernel stack of the current thread.

Beside composing a trap frame, if the trap came from user mode, then we need to set the registers to obey the kernel register usage convention (`tp`, `gp`, and `sscratch`).

After these tasks are accomplished, the `trap_handler` C function is called which examines the cause of the trap and calls a destined handler. `intr_root_handler` han-

dles interrupts, `user_trap_handler` handles user exceptions, and `kern_trap_handler` covers kernel exceptions.

Kernel exception handler

The only permissible kernel exceptions are page fault exceptions. In general, kernel page fault exceptions should happen rarely as almost all kernel mappings are wired (i.e. non-pageable). Page faults are handled by the `page_fault_handler` function described below.

As we don't provide a kernel debug stub, all remaining exceptions result in an error log and kernel panic.

User exception handler

In general, user mode exceptions are translated into signals:

- Access and misalignment exceptions result in `SIGBUS` being delivered to the process.
- If a floating-point unit (FPU) is implemented, the first illegal instruction exception caused by a floating-point instruction results in the FPU being enabled for the current thread. All remaining illegal instruction exceptions are translated into `SIGILL`.
- Breakpoint exception triggers `SIGTRAP` delivery.

User page fault exceptions are handled in the same fashion as kernel page faults.

The user environment exception is used in the syscall calling convention. The syscall implementation is described in 3.2.15.

Page fault handler

The `page_fault_handler` function plays an essential role in to primary kernel mechanisms:

- access bit emulation,
- demand paging.

First, we need to establish which physical address map manages the faulting address ❶. If the address lies within kernel virtual address space, then it's the kernel pmap. Otherwise, the user pmap is used.

pmap access bit emulation mechanism must be given the access permissions of the faulting access, which can be determined based on the exception code ❷.

`pmap_emulate_bits` emulates the referenced and modified permission bits and is discussed in 3.2.13. The return value of the function should be interpreted as follows:

1. `EFAULT` – there's no mapping for the faulting address.
2. `EACCES` – the faulting access violates the permissions of the faulting address mapping.
3. `EINVAL` – an invalid condition has been found. This can only occur in kernel mode and will most likely result in a kernel panic.

If there is no mapping for the faulting address, we will try to page-in the corresponding physical page.

To begin with, we have to establish the virtual map that manages the faulting address (similar to determining pmap, either kernel's or user's).

`vm_page_fault` is a machine-independent function that will try to find a virtual memory area corresponding to the faulting address, and if permissions haven't been violated, will call the pager of the object corresponding to the obtained virtual memory area. Afterward, the mapping will be entered into the physical address map.

Error handling depends on privilege mode and whether the fault has occurred during data copying between kernel and userspace (i.e. when on fault routine is set). The on fault mechanism is described in 3.2.14.

```
1  static void page_fault_handler(ctx_t *ctx) {
2    thread_t *td = thread_self();
3
4    unsigned code = ctx_code(ctx);
5    void *epc = (void *)_REG(ctx, PC);
6    vaddr_t vaddr = _REG(ctx, TVAL);
7
8    klog("%s at %p, caused by reference to %lx!",
9         exceptions[code], epc, vaddr);
10
11 ❶pmap_t *pmap = pmap_lookup(vaddr);
12   if (!pmap) {
13     klog("No physical map defined for %lx address!", vaddr);
14     goto fault;
15   }
16
17   vm_prot_t access;
18 ❷if (code == SCAUSE_INST_PAGE_FAULT)
19     access = VM_PROT_EXEC;
```

```
20    else if (code == SCAUSE_LOAD_PAGE_FAULT)
21      access = VM_PROT_READ;
22    else
23      access = VM_PROT_WRITE;
24
25    int error = pmap_emulate_bits(pmap, vaddr, access);
26    if (!error)
27      return;
28
29    if (error == EACCES || error == EINVAL)
30      goto fault;
31
32    vm_map_t *vmap = vm_map_lookup(vaddr);
33    if (!vmap) {
34      klog("No virtual address space defined for %lx!", vaddr);
35      goto fault;
36    }
37
38    if (!vm_page_fault(vmap, vaddr, access))
39      return;
40
41  fault:
42    if (td->td_onfault) {
43      /* Handle copyin/copyout faults. */
44      _REG(ctx, PC) = td->td_onfault;
45      td->td_onfault = 0;
46    } else if (user_mode_p(ctx)) {
47      /* Send a segmentation fault signal to the user program. */
48      sig_trap(ctx, SIGSEGV);
49    } else {
50      /* Panic when kernel-mode thread uses wrong pointer. */
51      kernel_oops(ctx);
52    }
53  }
```

Listing 10: `RISC-V` page fault handler (`board.c`)

Interrupt handler

`intr_root_handler` is a machine-independent function that calls a platform-dependent root filter routine. The root filter routine is an interrupt handler registered by the root bus device of the target platform which is the root interrupt controller. The handling flow will travel from the root interrupt controller, down through child interrupt controllers, to finally reach a leaf interrupt controller which will service all asserting devices attached to the specified interrupt line.

Returning from a trap

If the trap was an interrupt, then we have to check whether a context switch is needed. Preemption may be necessary because for example:

- The trap was a timer interrupt and the time slice of the current thread has finished.
- The filter routine of a device that has caused the interrupt has delegated some work to the interrupt thread. As a result, the dedicated interrupt thread has been awakened and happens to have higher priority than the current thread.

The check and potentially switch out, are performed by machine-independent `on_exc_leave`.

In case of a supervisor mode trap, all that's left to do is load the saved context from the trap frame (under disabled interrupts) and issue a supervisor return from trap (`SRET`) instruction.

Returning from a user-mode trap

The first observation is that when returning from a user-mode exception, we have to call the `on_exc_leave` function as the exception could have been a syscall that, for instance, could have created a higher priority thread. Moreover, if the trap was a syscall, we need to deliver the result of the call to the thread.

As we mentioned earlier, most user-mode exceptions are translated to signals, thereby, we need to deliver any pending and unmasked signals. This is described in detail in 3.2.16.

Finally, we can move to the low-level user trap return routine called `user_exc_leave`.

On entrance, we globally disable interrupts as the following sequence shouldn't be interrupted.

If FPU isn't implemented (i.e. floating-point exceptions are not supported), then the scheme is very similar to what is done for the supervisor mode trap return. However, if FPU is implemented then we potentially might need to restore the floating-point context.

If the current thread employs FPU and it has been switched out for a while, then the kernel must have saved its floating-point context. Since the only way out from the kernel back to user space is through `user_exc_leave`, this is the place where we need to restore the saved floating-point context.

After handling the floating-point context, we restore the context saved in the trap frame and return to the aborted userspace instruction via `SRET`.

```
user_exc_leave:
        /* Disable interrupts. */
        li t0, SSTATUS_SIE
        csrc sstatus, t0

#if FPU
        /* Read private thread flags. */
        PTR_L t0, PCPU_CURTHREAD(tp)
        INT_L t1, TD_PFLAGS(t0)

        /* Skip FPU restoring if FPE context is not used. */
        li t2, TDP_FPUINUSE
        and t2, t1, t2
        beqz t2, skip_fpu_restore

        /* Restore FPE context iff a context has been saved. */
        li t2, TDP_FPUCTXSAVED
        and t2, t1, t2
        beqz t2, skip_fpu_restore

        /* Clear TDP_FPUCTXSAVED flag. */
        li t2, ~TDP_FPUCTXSAVED
        and t1, t1, t2
        INT_S t1, TD_PFLAGS(t0)

        /* Restore FPE context. */
        PTR_L t0, TD_UCTX(t0)
        load_fpu_ctx t0, t1

skip_fpu_restore:
#endif
        load_ctx 0
        csrrw sp, sscratch, sp
        sret
```

Listing 11: `RISC-V` user trap return (`exception.S`)

3.2.11 Thread entry setup

The `thread_entry_setup` function implements machine-dependent part of the thread creation process. It is destined for setting the entry point for a thread. `target` is the destination entry point and `arg` is an argument for the entry function.

To achieve the desirable goal, we utilize three different contexts:

- `kctx` – kernel context saved and restored during context switch.
- `kframe` – kernel trap frame. This context is populated upon supervisor trap

entry and restored while returning from a supervisor trap (in `kern_exc_leave`).

- `uctx` – User machine context. Saved when entering user trap and restored while returning from user trap (in `user_exc_leave`).

As the first instruction of the thread is executed after a context switch, the initial context of the thread will be the context restored during context switch, that is, `kctx`. Although we could set the PC in `kctx` to the target function, we cannot set the argument in this way as the argument register is caller-saved and thereby isn't saved nor restored by the context switch routine. However, a kernel trap frame contains PC as well as all integer registers, hence `kframe` can be used to accommodate the desirable values. To utilize `kframe`, we set the PC of the `kctx` context to `kern_exc_leave` and the corresponding stack pointer to `kframe` ❶ as `kern_exc_leave` restores context pointed by stack pointer. In addition to setting the entry point and the argument in `kframe`, we also set the return address to `thread_exit` and the stack pointer to `uctx`. Setting the stack pointer to `uctx` is justified by the fact that in the case of creating a user thread (i.e. forking), the entry point will be set to `user_exc_leave` which restores the user machine context pointed by the stack pointer.

In case of forking a new thread, the `uctx` context will be filled in the implementation of the fork syscall using the contents of the requesting thread. In this scenario, the entry point argument and `kframe` return address are discarded.

```
1  void thread_entry_setup(thread_t *td, entry_fn_t target, void *arg) {
2    kstack_t *stk = &td->td_kstack;
3
4    kstack_reset(stk);
5
6    mcontext_t *uctx = kstack_alloc_s(stk, mcontext_t);
7    ctx_t *kframe = kstack_alloc_s(stk, ctx_t);
8    ctx_t *kctx = kstack_alloc_s(stk, ctx_t);
9
10   td->td_uctx = uctx;
11   td->td_kframe = NULL;
12   td->td_kctx = kctx;
13
14   /* Initialize registers in order to switch to kframe context. */
15 ❶ctx_init(kctx, kern_exc_leave, kframe);
16
17   /* This is the context that kern_exc_leave will restore. */
18   ctx_init(kframe, target, uctx);
19   ctx_setup_call(kframe, (register_t)thread_exit, (register_t)arg);
20 }
```

Listing 12: `RISC-V` context switch (`thread.c`)

3.2.12 Context switch

Context switch transfers control over the hart from one thread (`from`) to another (`to`). The calling thread (which must be pointed by `from`) will be suspended until it is switched back on the CPU.

The context switch process is accomplished by the machine-dependent `ctx_switch` funtion.

As `ctx_switch` is a regular function, it has to obey the calling convention. Additionally, we save the stack pointer register along with the status register.

Although the destination thread will resume its execution in the kernel (after the call to `ctx_switch`), it may potentially move to user space (e.g. if the destination thread is a user thread interrupted by a timer interrupt) which would clobber the floating-point state of the source thread. To avoid losing the FPU context, we must save it before switching out. The saved FPU context is the one that will be restored while executing `user_exc_leave`.

The actual saving of the floating-point context is optional. The context will be saved only if:

- the source thread utilizes the floating-point unit,
- the context hasn't been saved yet,
- the floating-point context state is marked as dirty.

Please note that the context will be marked as saved regardless of the third condition ❶.

After the dirty context is saved, we mark the FPU context as clean in the status register of the source thread ❷.

Thereafter, we switch to the kernel stack of the destination thread, update the current thread pointer (in PCPU), and switch the virtual memory map if necessary.

Finally, we restore the context of the target thread.

```
1  /*
2   * long ctx_switch(thread_t *from, thread_t *to)
3   */
4  ENTRY(ctx_switch)
5          /* `ctx_switch` must be called with interrupts disabled. */
6          csrr t0, sstatus
7          li t1, SSTATUS_SIE
8          and t1, t0, t1
9          bnez t1, halt
10
11         /* Save context of `from` thread. */
```

```
12          save_ctx t0
13          PTR_S sp, TD_KCTX(a0)
14
15  #if FPU
16          /* Read private thread flags. */
17          INT_L t1, TD_PFLAGS(a0)
18
19          /* If FPU isn't used or FPE context has already been saved,
20           * then skip the saving. */
21          li t2, TDP_FPUINUSE|TDP_FPUCTXSAVED
22          and t2, t1, t2
23          li t3, TDP_FPUINUSE
24          bne t2, t3, skip_fpu_save
25
26          /* Get UCTX pointer for `from` thread. */
27          PTR_L t4, TD_UCTX(a0)
28
29          /* Save the FPE context only if it's dirty. */
30          li t2, SSTATUS_FS_MASK
31          and t3, t0, t2
32          li t2, SSTATUS_FS_DIRTY
33        ❶ bne t3, t2, set_ctxsaved
34
35          /* Mark FPE state clean. */
36          li t2, ~SSTATUS_FS_MASK
37          and t0, t0, t2
38        ❷ li t2, SSTATUS_FS_CLEAN
39          or t0, t0, t2
40          INT_S t0, CTX_SR(t4)
41
42  set_ctxsaved:
43          /* Set TDP_FPUCTXSAVED flag. */
44          li t0, TDP_FPUCTXSAVED
45          or t1, t1, t0
46          INT_S t1, TD_PFLAGS(a0)
47
48          /* Save FP regs. */
49          save_fpu_ctx t4, t0
50
51  skip_fpu_save:
52  #endif
53          /* Switch stack pointer to `to` thread. */
54          PTR_L sp, TD_KCTX(a1)
55
56          /* Update `curthread` pointer to reference `to` thread. */
57          PTR_S a1, PCPU_CURTHREAD(tp)
58
59          /* Switch address space if necessary. */
60          mv a0, a1
61          call vm_map_switch
62
```

```
63          /* Restore `to` thread context. */
64          load_ctx t0
65
66          ret
67  halt:
68          wfi
69          j halt
70  END(ctx_switch)
```

Listing 13: `RISC-V` context switch (`switch.S`)

3.2.13 Physical address map (pmap) management

The physical address map management (pmap) module is a machine-dependent manager of memory translation structures for both kernel and each user-space process.

The main usage of pmap is to enter and remove mappings of virtual pages to physical ones. As an example, whenever the kernel malloc module [36] runs out of memory areas, it calls the kmem module [37] which, in turn, allocates a requested number of kernel virtual memory pages along with the corresponding number of physical memory pages and maps each virtual page into a physical page using the pmap module. A good example regarding user space is the demand paging mentioned in 3.2.10. After the page is paged in, pmap is used to enter the mapping into the physical map of the process that has caused the page fault.

To begin with, we will have a look at the machine-independent pmap API to gain familiarity with what the module does. Next, we will explore some of the most crucial aspects of the implementation.

The pmap API can be divided into three components:

- generic – contains generic functions applicable regardless of the type of mapping,
- pageable – functions related to pageable mappings,
- wired – functions regarding wired kernel mappings.

Wired mappings may only be used by the kernel. A wired physical page must never be paged out.

Pageable mappings are the only type of mappings used to populate physical maps of user processes. A physical page mapped using a pageable mapping may be paged out by the demand paging mechanism.

The wired and pageable interfaces should not be mixed.

The `pmap_t` structure used throughout the API is a machine-dependent structure representing a physical map. The API operates solely on pointers to these structures.

If the operation fulfilled by a pmap API function requires flushing the translation lookaside buffer (TLB), then an appropriate flush is performed inside the module and no further action is required from the user.

Generic API

pmap predicators verify whether a single virtual address or a virtual memory range are managed by a pointed physical map.

```
1 bool pmap_address_p(pmap_t *pmap, vaddr_t va);
2 bool pmap_contains_p(pmap_t *pmap, vaddr_t start, vaddr_t end);
```

Listing 14: pmap perdicators (`pmap.h`)

Determine the boundaries of virtual memory range managed by a physical map.

```
1 vaddr_t pmap_start(pmap_t *pmap);
2 vaddr_t pmap_end(pmap_t *pmap);
```

Listing 15: pmap boundaries (`pmap.h`)

Returns a handle to the physical map that manages a specified virtual address.

```
1 pmap_t *pmap_lookup(vaddr_t va);
```

Listing 16: pmap lookup (`pmap.h`)

Called at the very beginning of machine-independent bootstrap. Used to initialize the kernel physical address map.

```
1 void init_pmap(void);
```

Listing 17: pmap module initialization (`pmap.h`)

Return a handle to kernel or user physical map.

```
pmap_t *pmap_kernel(void);
pmap_t *pmap_user(void);
```

Listing 18: Privilege mode's pmap (`pmap.h`)

Physical address map creation and destruction.

```
pmap_t *pmap_new(void);
void pmap_delete(pmap_t *pmap);
```

Listing 19: pmap creation and destruction (`pmap.h`)

Clears specified physical page.

```
void pmap_zero_page(vm_page_t *pg);
```

Listing 20: pmap zero page (`pmap.h`)

Copies the contents of the source physical page to the destination physical page.

```
void pmap_copy_page(vm_page_t *src, vm_page_t *dst);
```

Listing 21: pmap copy page (`pmap.h`)

Activates pointed physical map.

```
void pmap_activate(pmap_t *pmap);
```

Listing 22: pmap activate (`pmap.h`)

Raises the kernel brk pointer to at least `maxvaddr`.

```
void pmap_growkernel(vaddr_t maxkvaddr);
```

Listing 23: pmap activate (`pmap.h`)

Pageable API

Creates a mapping for a specified virtual page to a pointed physical page with provided protection and cacheability. This function may also be used to alter an existing mapping.

```
1 void pmap_enter(pmap_t *pmap, vaddr_t va, vm_page_t *pg, vm_prot_t prot,
2                 unsigned flags);
```

Listing 24: Enter pageable mapping (`pmap.h`)

Returns the physical address corresponding to a given virtual address within a specified physical map.

```
1 bool pmap_extract(pmap_t *pmap, vaddr_t va, paddr_t *pap);
```

Listing 25: pmap extract (`pmap.h`)

Removes specified virtual memory range from supplied physical address map.

```
1 void pmap_remove(pmap_t *pmap, vaddr_t start, vaddr_t end);
```

Listing 26: Remove pageable mapping (`pmap.h`)

Alters protection of given virtual memory range.

```
1 void pmap_protect(pmap_t *pmap, vaddr_t start, vaddr_t end, vm_prot_t prot);
```

Listing 27: pmap protect (`pmap.h`)

Removes a given physical page from all physical maps that contain a mapping pointing at this page.

```
1 void pmap_page_remove(vm_page_t *pg);
```

Listing 28: pmap remove page (`pmap.h`)

Manage the referenced and modified bits. A physical page is said to be referenced or modified if at least one virtual page that is mapped to the physical page has been referenced or modified, respectively.

```
void pmap_set_referenced(vm_page_t *pg);
void pmap_set_modified(vm_page_t *pg);
bool pmap_is_modified(vm_page_t *pg);
bool pmap_is_referenced(vm_page_t *pg);
bool pmap_clear_modified(vm_page_t *pg);
bool pmap_clear_referenced(vm_page_t *pg);
```

Listing 29: pmap remove page (`pmap.h`)

Verifies if access with provided permission to specified virtual address would succeed and emulates the referenced and modified bits.

```
int pmap_emulate_bits(pmap_t *pmap, vaddr_t va, vm_prot_t prot);
```

Listing 30: pmap remove page (`pmap.h`)

Wired API

This is a counterpart of `pmap_enter` used for kernel wired mappings.

```
void pmap_kenter(vaddr_t va, paddr_t pa, vm_prot_t prot, unsigned flags);
```

Listing 31: Enter wired mapping (`pmap.h`)

Returns a physical address corresponding to a given kernel virtual address.

```
bool pmap_kextract(vaddr_t va, paddr_t *pap);
```

Listing 32: pmap kernel extract (`pmap.h`)

This is a counterpart of `pmap_remove` used for kernel wired mappings. Instead of specifying range boundaries, it accepts a starting address and size.

```
void pmap_kremove(vaddr_t va, size_t size);
```

Listing 33: Remove wired mapping (`pmap.h`)

State structure

The pmap module maintains a `pmap_t` structure for the kernel and each user-space process.

The most underlying members of the structure are address space identifier (`asid`) and physical address of the page directory (`pde`).

The supervisor address translation and protection (`satp`) member caches the value that is written to the `satp` register over physical address map activation. This value never changes during the existence of a physical map.

Additionally, `pmap_t` contains a list of all allocated pages used for the memory management structure (`pte_pages`), a list of physical pages that are mapped by this physical map (`pv_list`), and a link on the list of all physical maps corresponding to user processes (`pmap_link`).

```
1 typedef struct pmap {
2   mtx_t mtx;          /* protects all fields in this structure */
3   asid_t asid;        /* address space identifier */
4   paddr_t pde;        /* directory page table physical address */
5   paddr_t satp;       /* supervisor address translation and protection */
6   vm_pagelist_t pte_pages; /* pages we allocate in page table */
7   LIST_ENTRY(pmap) pmap_link;     /* link on `user_pmaps` */
8   TAILQ_HEAD(, pv_entry) pv_list; /* all mapped pages */
9 } pmap_t;
```

Listing 34: `RISC-V` pmap state (`pmap.c`)

Bootstrap

The `pmap_bootstrap` module is called by the virtual memory boot phase even before `init_pmap` gets called.

The main goal of this function is to build a direct map. After the main memory boundaries are retrieved from the kernel environment and some basic assumptions are verified, the direct map is built using 4MiB super pages ❶.

Besides building the direct map, the function stores the provided physical address of the kernel page directory which is used by the kernel physical map.

```
1 void pmap_bootstrap(paddr_t pd_pa, vaddr_t pd_va) {
2   uint32_t dmap_size = kenv_get_ulong("mem_size");
3
4   /* Obtain basic parameters. */
5   dmap_paddr_base = kenv_get_ulong("mem_start");
6   dmap_paddr_end = dmap_paddr_base + dmap_size;
7   kernel_pde = pd_pa;
```

```
8
9   /* Assume physical memory starts at the beginning of L0 region. */
10  assert(is_aligned(dmap_paddr_base, L0_SIZE));
11
12  /* We must have enough virtual addresses. */
13  assert(dmap_size <= DMAP_MAX_SIZE);
14
15  /* We assume 32-bit physical address space. */
16  assert(dmap_paddr_base < dmap_paddr_end);
17
18  klog("Physical memory range: %p - %p",
19       dmap_paddr_base, dmap_paddr_end - 1);
20
21  klog("dmap range: %p - %p", DMAP_VADDR_BASE,
22       DMAP_VADDR_BASE + dmap_size - 1);
23
24  /* Build direct map using 4MiB superpages. */
25  pd_entry_t *pde = (void *)pd_va;
26❶size_t idx = L0_INDEX(DMAP_VADDR_BASE);
27  for (paddr_t pa = dmap_paddr_base; pa < dmap_paddr_end;
28       pa += L0_SIZE, idx++) {
29    pde[idx] = PA_TO_PTE(pa) | PTE_KERN;
30  }
31}
```

Listing 35: `RISC-V` pmap bootstrap (`pmap.c`)

Physical map creation

The pmap module has a dedicated memory pool for `pmap_t` structures. Whenever a new physical map is needed it is allocated from the pool. Similarly, when a physical map gets destroyed, the corresponding structure is released and goes back to the pool.

First, a physical page for the page directory must be allocated. The address of the page will be reflected in the `pde` field ❶.

The setup function allocates a new address space identifier, initializes the guarding mutex and contained lists, and calculates the value of the `satp` register for the new physical map.

Contrary to `Aarch64`, `RISC-V` uses a single structure to translate both user and supervisor virtual addresses. Whenever we switch to a new address space, the physical map has to be switched as well. However, this creates a problem as kernel mappings should always be accessible regardless of the current user process. To solve this issue, we copy all kernel page directory entries to the user page directory while creating a new physical map ❷. There are no conflicts as kernel virtual addresses are mapped by the upper half of a page directory, whereas user virtual addresses

are mapped by the bottom half. To make it work, we have to broadcast any change in the kernel page directory to all user physical maps. That is why each `pmap_t` structure contains a link on a list of all user physical maps. Fortunately, new kernel page tables aren't created very often thereby the overhead is not significant.

```
1  pmap_t *pmap_new(void) {
2    pmap_t *pmap = pool_alloc(P_PMAP, M_ZERO);
3    vm_page_t *pg = pmap_pagealloc();
4  ❶pmap->pde = pg->paddr;
5    pmap_setup(pmap);
6
7    /* Install kernel pagetables. */
8    const size_t off = PAGESIZE / 2;
9    WITH_MTX_LOCK (&kernel_pmap.mtx) {
10   ❷memcpy((void *)phys_to_dmap(pmap->pde) + off,
11           (void *)phys_to_dmap(kernel_pde) + off, PAGESIZE / 2);
12   }
13
14   TAILQ_INSERT_TAIL(&pmap->pte_pages, pg, pageq);
15   klog("Page directory table allocated at %p", pmap->pde);
16
17   WITH_MTX_LOCK (&user_pmaps_lock) {
18     LIST_INSERT_HEAD(&user_pmaps, pmap, pmap_link);
19   }
20
21   return pmap;
22 }
```

Listing 36: `RISC-V` pmap creation (`pmap.c`)

Physical map activation

Whenever we switch between threads operating in different virtual address spaces (e.g. a user process thread and a kernel thread), we have to switch to the address space of the target thread. The activation of virtual address space is fulfilled by the activation of the corresponding physical map.

In `RISC-V`, all we have to do to switch the memory translation structure is change the pointer to the page directory. For correctness, the address space identifier also needs to be changed ❶.

```
1  void pmap_activate(pmap_t *pmap) {
2    SCOPED_NO_PREEMPTION();
3
4    pmap_t *old = PCPU_GET(curpmap);
5    if (pmap == old)
6      return;
7
```

```
8   ❶csr_write(satp, pmap->satp);
9     PCPU_SET(curpmap, pmap);
10
11    __sfence_vma();
12  }
```

Listing 37: RISC-V pmap activation (pmap.c)

Page table walk

Each of the following two functions performs a walk-through two-level memory management structure.

The lookup function serves for locating the page table entry corresponding to a given virtual address. The direct map is used to access the physical pages composing the physical map. Although the only relevant application of super pages is the direct map mapping, we still have to consider them when traversing the page directory.

pmap_ensure_pte must ensure that a page table entry for specified virtual address exists. If the corresponding page directory entry is invalid, then a page table will be allocated and the modification will be distributed to all user physical maps ❶.

```
1   static pt_entry_t *pmap_lookup_pte(pmap_t *pmap, vaddr_t va) {
2     pd_entry_t *pdep;
3     paddr_t pa = pmap->pde;
4
5     /* Level 0 */
6     pdep = (pd_entry_t *)phys_to_dmap(pa) + L0_INDEX(va);
7     pd_entry_t pde = *pdep;
8     if (!is_valid_pde(pde))
9       return NULL;
10
11    /* A direct map superpage? */
12    if (is_leaf_pte(pde))
13      return (pt_entry_t *)pdep;
14
15    pa = PTE_TO_PA(pde);
16
17    /* Level 1 */
18    return (pt_entry_t *)phys_to_dmap(pa) + L1_INDEX(va);
19  }
20
21  static pt_entry_t *pmap_ensure_pte(pmap_t *pmap, vaddr_t va) {
22    assert(mtx_owned(&pmap->mtx));
23
24    pd_entry_t *pdep;
25    paddr_t pa = pmap->pde;
```

```
26
27    /* Level 0 */
28    pdep = (pd_entry_t *)phys_to_dmap(pa) + L0_INDEX(va);
29    if (!is_valid_pde(*pdep)) {
30      pa = pmap_alloc_pde(pmap, va);
31      *pdep = make_pde(pa);
32    ❶pmap_distribute_l0(pmap, va, *pdep);
33    } else {
34      pa = PTE_TO_PA(*pdep);
35    }
36
37    /* Level 1 */
38    return (pt_entry_t *)phys_to_dmap(pa) + L1_INDEX(va);
39  }
```

Listing 38: `RISC-V` page table walk (`pmap.c`)

Entering a mapping

Creating a kernel wired mapping is a straightforward operation. We simply form an appropriate PTE, ensure that a PTE exists, and apply the new PTE.

`pmap_write_pte` will perform a flush TLB.

While entering a pageable mapping is similar, there are some differences:

- The permission bits in the final PTE differ depending on the privilege level. This is done to implement the access bit emulation described in 3.2.13.
- Each physical page contains a list of so-called physical to virtual entries (`pv_entry_t`) that states which physical maps contain a mapping pointing to the page. If an appropriate mapping doesn't already exist (i.e. this is not a modification of the mapping), it must be created.
- Page flags regarding permission bit emulation must be initialized.

```
1   void pmap_kenter(vaddr_t va, paddr_t pa, vm_prot_t prot, unsigned flags) {
2     pmap_t *pmap = pmap_kernel();
3
4     assert(page_aligned_p(pa) && page_aligned_p(va));
5     assert(pmap_address_p(pmap, va));
6
7     klog("Enter unmanaged mapping from %p to %p", va, pa);
8
9     pt_entry_t pte = make_pte(pa, prot, flags, true);
10
11    WITH_MTX_LOCK (&pmap->mtx) {
12      pt_entry_t *ptep = pmap_ensure_pte(pmap, va);
```

```
13      pmap_write_pte(pmap, ptep, pte, va);
14    }
15  }
16
17  void pmap_enter(pmap_t *pmap, vaddr_t va, vm_page_t *pg, vm_prot_t prot,
18                  unsigned flags) {
19    paddr_t pa = pg->paddr;
20
21    assert(page_aligned_p(va));
22    assert(pmap_address_p(pmap, va));
23
24    klog("Enter virtual mapping %p for frame %p", va, pa);
25
26    bool kern_mapping = (pmap == pmap_kernel());
27    pt_entry_t pte = make_pte(pa, prot, flags, kern_mapping);
28
29    WITH_MTX_LOCK (&pv_list_lock) {
30      WITH_MTX_LOCK (&pmap->mtx) {
31        pv_entry_t *pv = pv_find(pmap, va, pg);
32        if (!pv)
33          pv_add(pmap, va, pg);
34        pmap_set_init_flags(pg, kern_mapping);
35        pt_entry_t *ptep = pmap_ensure_pte(pmap, va);
36        pmap_write_pte(pmap, ptep, pte, va);
37      }
38    }
39  }
```

Listing 39: `RISC-V` virtual memory mapping (`pmap.c`)

Permission bit emulation

In Mimiker, we assume that the referenced and modified access bits are emulated in software. The structure that represents a physical page keeps track of the value of these bits in the form of flags in a bit vector. Although `RISC-V` specification claims that the corresponding accessed and dirty bits must be supported, we cannot utilize them as they may be managed directly in hardware which wouldn't be reflected in the page structure flags.

Permission bit tracking is only performed for physical pages mapped using pageable mappings on behalf of user processes.

To emulate the referenced bit, we initially clear the valid bit for each user pageable mapping. When a virtual page is accessed, a page fault will be triggered. If the access is valid, the corresponding physical page will be marked as referenced and the mapping will be marked as valid ❶.

The modified bit is emulated by initially clearing the writable permission bit.

Similarly, upon a store page fault, the corresponding physical page will be marked as modified and the writable bit in the mapping will be set ❷. However, an issue arises. If we have cleared the writable bit, how do we know upon the page fault that the mapping allows writing? As was mentioned in 2.1.15, we have two spare bits reserved for software in each PTE. To solve our problem, we can encode the writable permission in one of these bits.

```
1  int pmap_emulate_bits(pmap_t *pmap, vaddr_t va, vm_prot_t prot) {
2    paddr_t pa;
3
4    WITH_MTX_LOCK (&pmap->mtx) {
5      if (!pmap_extract_nolock(pmap, va, &pa))
6        return EFAULT;
7
8      pt_entry_t pte = *pmap_lookup_pte(pmap, va);
9
10     if ((prot & VM_PROT_READ) && !(pte & PTE_SW_READ))
11       return EACCES;
12
13     if ((prot & VM_PROT_WRITE) && !(pte & PTE_SW_WRITE))
14       return EACCES;
15
16     if ((prot & VM_PROT_EXEC) && !(pte & PTE_X))
17       return EACCES;
18   }
19
20   vm_page_t *pg = vm_page_find(pa);
21   assert(pg);
22
23   WITH_MTX_LOCK (&pv_list_lock) {
24     /* Kernel non-pageable memory? */
25     if (TAILQ_EMPTY(&pg->pv_list))
26       return EINVAL;
27   }
28
29 ❶pmap_set_referenced(pg);
30 ❷if (prot & VM_PROT_WRITE)
31     pmap_set_modified(pg);
32
33   return 0;
34 }
```

Listing 40: `RISC-V` virtual memory mapping (`pmap.c`)

3.2.14 Communication with user space

Many syscall signatures include a pointer as an argument. A pointer can be used either to specify an input buffer (e.g. read syscall) or to provide a destination buffer

(e.g. write syscall). Regardless of the exact interpretation of the pointer, some data will be copied between user space and kernel space. However, if the supplied pointer is faulty, we run into trouble. Since the copying is performed in the kernel, any fault will eventually result in a kernel panic. Instead of halting the whole kernel, we would like to intercept any faults caused by data copying between user and kernel and return an error code to indicate that the operation has failed. This functionality is provided by the data copying kernel module.

The data copying module introduces three machine-dependent functions with a machine-independent API.

API

Copies specified amount of data from user space into kernel space.

```
1 int copyin(const void *restrict udaddr, void *restrict kaddr, size_t len);
```

Copies specified amount of data from kernel space to userspace.

```
1 int copyout(const void *restrict kaddr, void *restrict udaddr, size_t len);
```

Copies at most `len` bytes of a null-terminated string from user space into kernel space. `lencopied` is used to return the actual number of copied bytes.

```
1 int copyinstr(const void *restrict udaddr, void *restrict kaddr, size_t len,
2               size_t *restrict lencopied);
```

Each of the defined functions returns 0 in case of success and a non-zero error code if any problem is encountered, including any fatal page fault exceptions.

Implementation

All three functions are implemented similarly. Although we will only examine the implementation of the `copyin` function, the same ideas are employed to realize the remaining functions.

In the first place, the function performs some basic validation of provided arguments, for instance, does the userspace pointer point within the user address space ❶.

The fatal page fault exception interception is implemented as follows:

- The `thread_t` structure accommodates the `td_onfault` pointer.
- Whenever a fatal page fault occurs, `page_fault_handler` checks if the on fault pointer is set. If the answer is positive, the trap frame is modified to return control to the address indicated by the on fault pointer, and the on fault pointer is cleared.
- Before we begin the actual copying, we set the on fault pointer to the address of the `copyerr` routine ❷.
- If the copping is successful, we clear the pointer ❸ on our own and return zero to indicate success.

However, one problem remains. Contrary to `Aarch64`, in our implementation of libc, `RISC-V` uses a generic implementation of the machine-dependent functions. Since the generic implementation is written in C, we cannot make any assumptions regarding its treatment of registers. Especially, it may utilize the callee-saved registers, stack pointer, and return address. To preserve these registers, we have to save them before the call and restore them afterward (potentially in the error routine). The registers are saved on the stack and the pointer to the location is saved in the machine-dependent fields of PCPU (to make it accessible from the on fault function).

```
1  ENTRY(copyin)
2          /* len > 0 */
3          beqz a2, 1f
4
5          /* (uintptr_t)udaddr < (uintptr_t)(udaddr + len) */
6          PTR_ADD t0, a0, a2
7          bgeu a0, t0, reterr
8
9          /* (uintptr_t)(udaddr + len) <= USER_SPACE_END */
10       ❶REG_LI t1, USER_SPACE_END
11         bgtu t0, t1, reterr
12
13       ❷onfault_set_and_save t0, t1, copyerr
14         call bcopy
15       ❸onfault_clr_and_load t0, t1
16
17 1:
18         mv a0, zero
19         ret
20 END(copyin)
21
22 ENTRY(copyerr)
23         ctx_load
```

```
24 reterr:
25         REG_LI a0, EFAULT
26         ret
27 END(copyerr)
```

Listing 41: `RISC-V` copyin (`copy.S`)

3.2.15 Syscalls

The only machine-dependent aspect regarding syscalls is the calling convention introduced in the OS-specific ABI.

In Mimiker, I applied the standard convention deployed in Linux:

- register `a7` contains the code identifying a particular request,
- registers `a0-a5` are used as arguments for the syscall.

We can distinguish two components of the machine-dependent code related to syscalls:

- user environment call handling – this is part of the kernel trap handling module and it discussed in this section,
- libc syscall wrappers – this is part of the libc system library and is discussed in 3.4.4.

User environment call handling

After retrieving the code ❶, we have to gather all arguments for the syscall. This is done by simply copying register values from the trap frame ❷.

After adjusting the code if needed ❸, we fetch the structure that describes the machine-independent implementation of the syscall. Subsequently, we call the implementation and form a syscall result structure based on the outcome of the call.

```
1  static void syscall_handler(mcontext_t *uctx, syscall_result_t *result) {
2    register_t args[SYS_MAXSYSARGS];
3  ❶register_t code = _REG(uctx, A7);
4
5  ❷memcpy(args, &_REG(uctx, A0), sizeof(args));
6
7  ❸if (code > SYS_MAXSYSCALL) {
8      args[0] = code;
9      code = 0;
10   }
```

```
11
12  ❹sysent_t *se = &sysent[code];
13    size_t nargs = se->nargs;
14
15    assert(nargs <= SYS_MAXSYSCALL);
16
17    thread_t *td = thread_self();
18    register_t retval = 0;
19
20    assert(td->td_proc);
21
22    int error = se->call(td->td_proc, (void *)args, &retval);
23
24    result->retval = error ? -1 : retval;
25    result->error = error;
26  }
```

Listing 42: `RISC-V` user environment call handling (`trap.c`)

3.2.16 Signals

A signal is an asynchronous message sent to a process or a user thread. Signals can be sent either by the kernel or by a process (a process can even send a signal to itself). Signals sent directly by the kernel are user-level counterparts of exceptions encountered by a thread composing the process. Signals issued by other processes are used for interprocess communication.

API

The Mimiker signal API defines two machine-dependent functions.

Prepares the user context of the calling thread for the execution of the signal handler indicated by provided `sigaction_t` structure. After the handler is executed, the kernel should apply a specified signal mask for the thread.

```
1  int sig_send(signo_t sig, sigset_t *mask, sigaction_t *sa,
2               ksiginfo_t *ksi);
```

Signal trampoline. Used to restore the thread's context after the signal is handled. `sigcode` points at the start of the signal trampoline in kernel virtual address space, whereas `esigcode` points at the end.

```
extern char sigcode[];
extern char esigcode[];
```

Implementation

`sig_send` copies the following components to the stack of the target thread (using `copyout`):

- the signal trampoline function ❶,
- the corresponding `siginfo_t` structure containing an extensive description of the signal ❷,
- the user context to resume after the signal is handled ❸.

Afterward, we modify the context saved in the trap frame so that:

- the thread will resume at the beginning of the signal handler ❹,
- registers `a0-a2` contain signal number, pointer to the provided signal info structure, and pointer to the user context that encountered the signal (and will be restored), respectively ❺,
- the return address points at the beginning of the signal trampoline copied to thread's stack ❻.

```
int sig_send(signo_t sig, sigset_t *mask, sigaction_t *sa,
             ksiginfo_t *ksi) {
  thread_t *td = thread_self();
  mcontext_t *uctx = td->td_uctx;

  ucontext_t uc;
  mcontext_copy(&uc.uc_mcontext, uctx);
  uc.uc_sigmask = *mask;

❶register_t sc_code = sig_stack_push(uctx, sigcode, esigcode - sigcode);
❷register_t sc_info = sig_stack_push(uctx, &ksi->ksi_info,
                                                  sizeof(siginfo_t));
❸register_t sc_uctx = sig_stack_push(uctx, &uc, sizeof(ucontext_t));

❹_REG(uctx, PC) = (register_t)sa->sa_handler;
❺_REG(uctx, A0) = (register_t)sig;
  _REG(uctx, A1) = sc_info;
  _REG(uctx, A2) = sc_uctx;
❻_REG(uctx, RA) = sc_code;

```

```
21    return 0;
22  }
```

Listing 43: RISC-V signal sending (signal.c)

The signal trampoline employs the sigreturn syscall to restore the machine context of the thread and apply the signal mask set by sig_send.

```
1  ENTRY(sigcode)
2          mv a0, sp       /* address of ucontext to restore */
3          li a7, SYS_sigreturn
4          ecall
5
6          /* Just in case `sigreturn` fails. */
7          ebreak
8
9  /* `esigcode` is used just to compute size of the following code. */
10 EXPORT(esigcode)
11 END(sigcode)
```

Listing 44: RISC-V signal trampoline (sigcode.S)

3.3 Device drivers

This section will describe device drivers for the essential devices we assume to be present in a target LiteX RISC-V platform.

Interfaces

An interface is an abstract device model. It presents a set of functionalities that must be implemented by an actual device driver. A device may implement more than one interface, for instance, a system controller device can implement the bus interface along with the interrupt controller interface.

Interfaces are platform-independent. Implementations of the interfaces are highly platform-dependent.

3.3.1 Bus interface

Each function composing the bus interface accepts only a single device pointer that specifies a requesting device. This is an obsolete design and newer interfaces take two device arguments: one for requesting device and one for the device that implements the interface.

Allocates a resource of a specified type on behalf of a pointed device. The calling device must specify the acceptable range within which the resource of provided size will be allocated. `rid` supplies a resource identifier that will be assigned to the resource, whereas `flags` guides the process of allocation.

```
resource_t *(*alloc_resource)(device_t *dev, res_type_t type, int rid,
                              rman_addr_t start, rman_addr_t end,
                              size_t size, rman_flags_t flags);
```

Releases an allocated resource.

```
void (*release_resource)(device_t *dev, resource_t *r);
```

Activates a given resource. Usually, activation means mapping the resource into kernel virtual memory.

```
int (*activate_resource)(device_t *dev, resource_t *r);
```

Deactivates an active resource.

```
void (*deactivate_resource)(device_t *dev, resource_t *r);
```

Besides the bus interface. It is the responsibility of a bus device to create a software representation for each of its child devices and attach it to the kernel's device tree structure.

3.3.2 Interrupt controller interface

Initially, the interrupt controller interface was merged with the bus controller. I distinguished the two interfaces for the needs of my PLIC device driver.

Allocates an interrupt identified by `irq`. Usually, an interrupt may be allocated by more than one device.

```
resource_t *(*intr_alloc)(device_t *ic, device_t *dev, int rid,
                          unsigned irq);
```

Releases an interrupt resource.

```
1 void (*intr_release)(device_t *ic, device_t *dev, resource_t *r);
```

Registers a new interrupt source for interrupt identified by provided resource. Whenever the interrupt occurs, a specified filter function will be called. If a service function is provided, the filter function can delegate processing to the service function which will be executed in an interrupt thread environment. A provided argument is used as the parameter for the filter and service routines. `name` describes the interrupt source.

```
1 void (*intr_setup)(device_t *ic, device_t *dev, resource_t *r,
2                    ih_filter_t *filter, ih_service_t *service,
3                    void *arg, const char *name);
```

Removes the interrupt source registered by a specified device (`dev`) for interrupt identified by pointed resource.

```
1 void (*intr_teardown)(device_t *ic, device_t *dev, resource_t *r);
```

3.3.3 Timer interface

Starts pointed timer. Sets the starting timer to a specified value. The timer will trigger a timer interrupt with the frequency established by provided period. `flags` are used to further configure the timer.

```
1 int (*start)(timer_t *tm, unsigned flags, const bintime_t start,
2              const bintime_t period);
```

Stops a timer (i.e. interrupt will no longer be generated).

```
1 int (*stop)(timer_t *tm);
```

Retrieves the time measured by a specified timer.

```
bintime_t (*gettime)(timer_t *tm);
```

3.3.4 UART interface

In the following functions, `state` is a pointer to the implementation-specific state, provided while registering a UART device.

Returns true if the receiver hardware queue is non-empty.

```
bool (*rx_ready)(void *state);
```

Retrieves a single byte from the corresponding UART device. The function will only be called when the receiver queue is known to be non-empty.

```
uint8_t (*getc)(void *state);
```

Returns true if the transmitter hardware queue is not full.

```
bool (*tx_ready_t)(void *state);
```

Sends a single byte via the corresponding UART device. The function will only be called when the transmitter queue is known to be not full.

```
void (*putc)(void *state, uint8_t byte);
```

Enables transmitter interrupt.

```
void (*tx_enable)(void *state);
```

Disables transmitter interrupt.

```
void (*tx_disable)(void *state);
```

3.3.5 Interrupt events

Interrupt controllers drivers employ the machine-independent interrupt event construct (`intr_event_t`) which is a software representation of an interrupt line.

While creating an interrupt event for each interrupt line managed by a controller, the controller must provide the event with two functions of `ie_action_t` type.

```
1 typedef void ie_action_t(intr_event_t *);
```

One function is used to enable the corresponding interrupt, whereas the other one is applied to disable the interrupt.

`intr_event_run_handlers` will invoke the filter function for each registered interrupt source.

3.3.6 Root bus device

For each architecture, the root bus device (rootdev) is a compound device accommodating two components:

- core local interrupt controller,
- root bus which is an ancestor of all devices.

For `RISC-V` rootdev consists of:

- HLIC,
- bus corresponding to the soc device tree node.

HLIC

HLIC implements the interrupt controller interface 3.3.2.

The following two functions are passed to interrupt events to manage the underlying interrupt.

```
1 static void hlic_intr_disable(intr_event_t *ie) {
2   unsigned irq = ie->ie_irq;
3   csr_clear(sie, 1 << irq);
4 }
5
```

```
6  static void hlic_intr_enable(intr_event_t *ie) {
7    unsigned irq = ie->ie_irq;
8    csr_set(sie, 1 << irq);
9  }
```

Listing 45: RISC-V HLIC interrupt control (litex_riscv_rootdev.c)

HLIC is the root interrupt controller. Whenever an interrupt occurs, the trap handling module will call the HLIC interrupt handling routine (hlic_intr_handler) which is registered as the interrupt root filter.

```
1  static void hlic_intr_handler(ctx_t *ctx, device_t *bus) {
2    rootdev_t *rd = bus->state;
3    unsigned long cause = _REG(ctx, CAUSE) & SCAUSE_CODE;
4    assert(cause < HLIC_NIRQS);
5
6    intr_event_t *ie = rd->intr_event[cause];
7    if (!ie)
8      panic("Unknown HLIC interrupt %u!", cause);
9
10   intr_event_run_handlers(ie);
11
12   if (cause != HLIC_IRQ_TIMER_SUPERVISOR &&
13       cause != HLIC_IRQ_EXTERNAL_SUPERVISOR) {
14     csr_clear(sip, ~(1 << cause));
15   }
16 }
```

Listing 46: RISC-V HLIC interrupt handler (litex_riscv_rootdev.c)

Root bus

rootdev implements the bus interface 3.3.1.

rootdev manages the entire I/O region of the platform.

3.3.7 CLINT

CLINT implements the MTIMER ACLINT device and thus implements the timer interface 3.3.3.

During an invocation of the timer start routine, we convert provided period into MTIMER ticks ❶, register as a source for the supervisor timer interrupt ❷ and schedule the first timer interrupt using the timer SBI extension ❸.

```
1  static int mtimer_start(timer_t *tm, unsigned flags, const bintime_t start,
2                          const bintime_t period) {
3    device_t *dev = tm->tm_priv;
4    clint_state_t *clint = dev->state;
5
6  ❶clint->mtimer_step = bintime_mul(period, tm->tm_frequency).sec;
7
8  ❷intr_setup(dev, clint->mtimer_irq, mtimer_intr, NULL, clint, "MTIMER");
9
10   WITH_INTR_DISABLED {
11     uint64_t count = rdtime();
12   ❸sbi_set_timer(count + clint->mtimer_step);
13   }
14
15   return 0;
16 }
```

Listing 47: `RISC-V` MTIMER start (`clint.c`)

Whenever a timer interrupt handler is invoked, we must call a callback function registered for this timer by the kernel module that utilizes the timer. This is accomplished by a call to the `tm_trigger` function ❶.

Scheduling the next timer interrupt will increase the `mtimecmp` register and thereby clear the pending interrupt.

```
1  static intr_filter_t mtimer_intr(void *data) {
2    clint_state_t *clint = data;
3    register_t sip = csr_read(sip);
4
5    if (sip & SIP_STIP) {
6    ❶tm_trigger(&clint->mtimer);
7
8      uint64_t prev = rdtime();
9      sbi_set_timer(prev + clint->mtimer_step);
10
11     return IF_FILTERED;
12   }
13
14   return IF_STRAY;
15 }
```

Listing 48: `RISC-V` MTIMER interrupt handler (`clint.c`)

The `gettime` implementation reads the value of the `timer` CSR and converts the number of ticks to number of seconds.

```
static bintime_t mtimer_gettime(timer_t *tm) {
  uint64_t count = rdtime();
  bintime_t res = bintime_mul(tm->tm_min_period, (uint32_t)count);
  bintime_t high_bits = bintime_mul(tm->tm_min_period,
                                    (uint32_t)(count >> 32));
  bintime_add_frac(&res, high_bits.frac << 32);
  res.sec += (high_bits.sec << 32) + (high_bits.frac >> 32);
  return res;
}
```

Listing 49: RISC-V MTIMER gettime (clint.c)

3.3.8 PLIC

PLIC implements the interrupt controller interface 3.3.2.

Whereas HLIC has a constant number of interrupts, the number of interrupts controlled by PLIC may differ depending on the hardware implementation. Thereby, the actual number of interrupts is specified as a device tree property of the PLIC's node. The property is called "riscv,ndev" and is retrieved from the DTB during the attachment of the driver ❶.

As we don't utilize the priorities which are optionally provided by PLIC, we set the priority of each interrupt to 1 ❷ and the threshold register of the supervisor PLIC context to 0 ❸.

PLIC is routed through HLIC to the supervisor external interrupt and registers as an interrupt source just like any other peripheral device ❹.

```
static int plic_attach(device_t *ic) {
  plic_state_t *plic = ic->state;

  /* Obtain the number of sources. */
❶plic->nirqs = dtb_dev_cell(ic, "riscv,ndev");

  /* We'll need interrupt event for each interrupt source. */
  plic->intr_event =
    kmalloc(M_DEV, plic->nirqs * sizeof(intr_event_t *),
            M_WAITOK | M_ZERO);
  if (!plic->intr_event)
    return ENXIO;

  rman_init(&plic->rm, "PLIC interrupt sources");
  rman_manage_region(&plic->rm, 1, plic->nirqs);

  plic->mem = device_take_memory(ic, 0, RF_ACTIVE);
  assert(plic->mem);

```

```
20    /*
21     * In case PLIC supports priorities, set each priority to 1
22     * and the threshold to 0.
23     */
24  ❷ for (unsigned irq = 0; irq < plic->nirqs; irq++) {
25      out4(PLIC_PRIORITY(irq), 1);
26    }
27  ❸ out4(PLIC_THRESHOLD_SV, 0);
28
29    plic->irq = device_take_irq(ic, 0, RF_ACTIVE);
30    assert(plic->irq);
31
32  ❹ intr_setup(ic, plic->irq, plic_intr_handler, NULL, plic, "PLIC");
33
34    return 0;
35  }
```

Listing 50: `RISC-V` PLIC attachment (`plic.c`)

PLIC interrupt enabling and disabling is performed via the enable bits of the supervisor PLIC context.

```
1   static void plic_intr_disable(intr_event_t *ie) {
2     plic_state_t *plic = ie->ie_source;
3     unsigned irq = ie->ie_irq;
4
5     uint32_t en = in4(PLIC_ENABLE_SV(irq));
6     en &= ~(1 << (irq % 32));
7     out4(PLIC_ENABLE_SV(irq), en);
8   }
9
10  static void plic_intr_enable(intr_event_t *ie) {
11    plic_state_t *plic = ie->ie_source;
12    unsigned irq = ie->ie_irq;
13
14    uint32_t en = in4(PLIC_ENABLE_SV(irq));
15    en |= 1 << (irq % 32);
16    out4(PLIC_ENABLE_SV(irq), en);
17  }
```

Listing 51: `RISC-V` PLIC interrupt control (`plic.c`)

The PLIC interrupt handling scheme was described in 2.4.3.

```
static intr_filter_t plic_intr_handler(void *arg) {
  plic_state_t *plic = arg;

  /* Claim any pending interrupt. */
  uint32_t irq = in4(PLIC_CLAIM_SV);

  if (irq) {
    intr_event_run_handlers(plic->intr_event[irq]);
    /* Complete the interrupt. */
    out4(PLIC_CLAIM_SV, irq);
    return IF_FILTERED;
  }

  return IF_STRAY;
}
```

Listing 52: `RISC-V` PLIC interrupt handler (`plic.c`)

3.3.9 LiteUART

LiteUART implements the UART interface 3.3.4.

LiteUART is controlled via LiteX CSRs.

```
static bool liteuart_rx_ready(void *state) {
  liteuart_state_t *liteuart = state;
  return csr_read(LITEUART_CSR_RXEMPTY) == 0;
}

static uint8_t liteuart_getc(void *state) {
  liteuart_state_t *liteuart = state;
  return csr_read(LITEUART_CSR_RXTX);
}

static bool liteuart_tx_ready(void *state) {
  liteuart_state_t *liteuart = state;
  return csr_read(LITEUART_CSR_TXFULL) == 0;
}

static void liteuart_putc(void *state, uint8_t c) {
  liteuart_state_t *liteuart = state;
  csr_write(LITEUART_CSR_RXTX, c);
}

static void liteuart_tx_enable(void *state) {
  liteuart_state_t *liteuart = state;
  csr_set(LITEUART_CSR_EV_ENABLE, LITEUART_EV_TX);
}

```

```
26 static void liteuart_tx_disable(void *state) {
27   liteuart_state_t *liteuart = state;
28   csr_clr(LITEUART_CSR_EV_ENABLE, LITEUART_EV_TX);
29 }
```

Listing 53: LiteUART interface implementation (`liteuart.c`)

The interrupt handler calls a platform-independent UART interrupt handler (`uart_intr` ❶). However, `uart_intr` cannot be directly used as the interrupt handler since LiteUART requires an asserted interrupt to be explicitly cleared via the event pending LiteX CSR ❷.

```
1  static intr_filter_t liteuart_intr(void *data) {
2    device_t *dev = data;
3    uart_state_t *uart = dev->state;
4    liteuart_state_t *liteuart = uart->u_state;
5
6    uint32_t ev_pending = csr_read(LITEUART_CSR_EV_PENDING);
7
8    if (!(ev_pending & (LITEUART_EV_TX | LITEUART_EV_RX)))
9      return IF_STRAY;
10
11 ❶intr_filter_t res = uart_intr(data);
12
13 ❷csr_write(LITEUART_CSR_EV_PENDING, ev_pending);
14   return res;
15 }
```

Listing 54: LiteUART interrupt handler (`liteuart.c`)

3.4 System libraries

Mimiker provides two system libraries with machine-dependent components: csu and libc.

This section will describe machine-dependent components of the aforementioned libraries.

3.4.1 Linker script

The csu library contains a machine-dependent default linker script used while linking user programs.

The user linker script is much simpler than the sophisticated kernel linker script.

The layout of the script is the same regardless of the target architecture. The only `RISC-V`-specific aspects in the script are:

- setting of the global pointer,
- small input sections.

3.4.2 crt0

The csu library contains two modules:

- crt0 – machine-dependent implementation of assembly startup routine (`_start`),
- crt0-common – machine-independent implementation of C startup routine (`___start`).

`_start` is the default entry point of user programs. Its sole responsibility is to set the global pointer and compute the input parameters for machine-independent `___start`.

While performing an exec syscall, the kernel will put the following elements on the stack of the new process (stack bottom to top):

- environment strings,
- agrument strings,
- environment vector (`envp`),
- argument vector (`argv`),
- argument counter.

```
1  ENTRY(_start)
2          PTR_ADDI sp, sp, -CALLFRAME_SIZ
3          PTR_S zero, CALLFRAME_RA(sp)
4          PTR_S sp, CALLFRAME_SP(sp)
5
6          /* Prepare global pointer. */
7          LOAD_GP()
8
9          /* Grab argc from below stack. */
10         LONG_L a0, CALLFRAME_SIZ(sp)
11
12         /* Prepare argv pointing at argument vector below stack. */
13         PTR_ADDI a1, sp, CALLFRAME_SIZ + SZREG
14
15         /* Prepare envp, it starts at argv + argc + 1. */
16         LONG_ADDI t0, a0, 1        /* argv is NULL terminated */
17         LONG_SLLI t0, t0, LONG_SCALESHIFT
18         PTR_ADD a2, a1, t0
19
```

```
20          /* Jump to start in crt0-common.c. */
21          j _C_LABEL(___start)
22  END(_start)
```

Listing 55: `RISC-V` crt0 (`crt0.S`)

3.4.3 String functions

Some of the libc string functions (`bcopy`, `memcpy`, `memmove`, and `strlen`) are written directly in assembly for performance reasons.

Mimiker deploys the standard C library implementation from NetBSD. Other architectures simply utilize the corresponding implementation found in the original codebase. However, NetBSD doesn't provide an architecture-specific implementation of the string functions for `RISC-V`. Thereby, I ported the generic implementation of the functions. From now on, every new architecture can employ the generic implementation instead of providing an architecture-specific one.

3.4.4 Syscalls

Each architecture must provide a set of macros used by libc in a machine-independent fashion to define wrappers for syscalls.

Syscall macros

Defines a function with a specified name that implements a missing syscall.

```
1  SYSCALL_MISSING(name)
```

Defines a function with a specified name that performs a syscall identified by code provided as `num`. Used for syscalls that cannot fail.

```
1  SYSCALL_NOERROR(name, num)
```

Defines a function with a specified name that performs a syscall identified by code provided as `num`.

```
1  SYSCALL(name, num)
```

Implementation

```
#define SYSCALL_MISSING(name)                                   \
  ENTRY(name);                                                  \
  j _C_LABEL(__sc_missing);                                     \
  END(name)

#define SYSCALL_NOERROR(name, num)                              \
  ENTRY(name);                                                  \
  REG_LI a7, num;                                               \
  ecall;                                                        \
  ret;                                                          \
  END(name)

#define SYSCALL(name, num)                                      \
  ENTRY(name);                                                  \
  REG_LI a7, num;                                               \
  ecall;                                                        \
  bnez a1, 0f;                                                  \
  ret;                                                          \
  0 : j _C_LABEL(__sc_error);                                   \
  END(name)
```

Listing 56: `RISC-V` syscall macros (`syscall.h`)

Auxiliary functions

The implementation of `RISC-V` syscall macros embraces two auxiliary functions: `__sc_error` and `__sc_missing`. These functions are machine-dependent and lie within libc.

`__sc_error` handles a syscall error by setting the `errno` variable to reflect the syscall error code.

```
ENTRY(__sc_error)
        /* Set call frame and save syscall error. */
        PTR_ADDI sp, sp, -CALLFRAME_SIZ
        PTR_S ra, CALLFRAME_RA(sp)
        REG_S s0, CALLFRAME_S0(sp)
        mv s0, a1

        /* Obtain errno pointer. */
        call __errno

        /* Set errno to syscall error. */
        INT_S s0, (a0)

        /* Return indicating error. */
        REG_LI a0, -1
```

```
16          REG_L s0, CALLFRAME_S0(sp)
17          PTR_L ra, CALLFRAME_RA(sp)
18          PTR_ADDI sp, sp, CALLFRAME_SIZ
19          ret
20  END(__sc_error)
```

Listing 57: RISC-V syscall error (sc_error.S)

__sc_missing handles unimplemented syscalls. If a syscall is unimplemented, then each call behaves as if the syscall were implemented and the implementation returned ENOSYS error code.

```
1   ENTRY(__sc_missing)
2           /*
3            * Behave as if the syscall returned ENOSYS.
4            */
5           REG_LI a0, -1
6           REG_LI a1, ENOSYS
7           j _C_LABEL(__sc_error)
8   END(__sc_missing)
```

Listing 58: RISC-V missing syscall (sc_error.S)

3.4.5 Nonlocal goto

Nonlocal goto functions are used for transferring control flow from one function to an established location within another function.

In Mimiker, we have six different nonlocal goto functions grouped in pairs [27]:

- _setjmp and _longjmp – don't manipulate the signal mask,
- setjmp and longjmp – restore the signal mask,
- sigsetjmp and siglongjmp – signal mask handling depends on provided arguments.

The implementation of each nonlocal goto function is machine-dependent.

In Mimiker, the jmpbuf type is defined to be an alias for ucontext_t.

_setjmp and _longjmp

_setjmp stores the following elements:

- all callee-saved integer and floating-point registers (if hard floating-point is employed),

- PC (it points right after the call ❶),
- thread pointer (`tp`) and global pointer (`gp`),
- floating-point control and status register (`fcsr`).

`_longjmp` restores the saved context and adjusts the return value to distinguish the first return from `_setjmp` from any other returns.

```
1  ENTRY(_setjmp)
2        ❶REG_S ra, UC_GREGS_PC(a0)
3          REG_S sp, UC_GREGS_SP(a0)
4          REG_S gp, UC_GREGS_GP(a0)
5          REG_S tp, UC_GREGS_TP(a0)
6          REG_S s0, UC_GREGS_S0(a0)
7          REG_S s1, UC_GREGS_S1(a0)
8          REG_S s2, UC_GREGS_S2(a0)
9          REG_S s3, UC_GREGS_S3(a0)
10         REG_S s4, UC_GREGS_S4(a0)
11         REG_S s5, UC_GREGS_S5(a0)
12         REG_S s6, UC_GREGS_S6(a0)
13         REG_S s7, UC_GREGS_S7(a0)
14         REG_S s8, UC_GREGS_S8(a0)
15         REG_S s9, UC_GREGS_S9(a0)
16         REG_S s10, UC_GREGS_S10(a0)
17         REG_S s11, UC_GREGS_S11(a0)
18
19         INT_L t0, UC_FLAGS(a0)
20         li t1, _UC_CPU
21         or t0, t0, t1
22
23 #ifndef __riscv_float_abi_soft
24         frcsr t1
25         REG_S t1, UC_FPREGS_FCSR(a0)
26         FP_S fs0, UC_FPREGS_FS0(a0)
27         FP_S fs1, UC_FPREGS_FS1(a0)
28         FP_S fs2, UC_FPREGS_FS2(a0)
29         FP_S fs3, UC_FPREGS_FS3(a0)
30         FP_S fs4, UC_FPREGS_FS4(a0)
31         FP_S fs5, UC_FPREGS_FS5(a0)
32         FP_S fs6, UC_FPREGS_FS6(a0)
33         FP_S fs7, UC_FPREGS_FS7(a0)
34         FP_S fs8, UC_FPREGS_FS8(a0)
35         FP_S fs9, UC_FPREGS_FS9(a0)
36         FP_S fs10, UC_FPREGS_FS10(a0)
37         FP_S fs11, UC_FPREGS_FS11(a0)
38
39         li t1, _UC_FPU
40         or t0, t0, t1
41 #endif
42         INT_S t0, UC_FLAGS(a0)
```

```
43
44          mv a0, zero
45          ret
46  END(_setjmp)
```

Listing 59: RISC-V _setjmp (_setjmp.S)

setjmp and longjmp

setjmp performs the same operations as _setjmp and additionally saves the signal mask.

longjmp extracts and validates relevant fields from supplied jump environment, restores the saved signal mask, and applies the saved machine context using the setcontext syscall.

```
1   void longjmp(jmp_buf env, int val) {
2     ucontext_t *sc_uc = (ucontext_t *)env;
3     ucontext_t uc;
4
5     bzero(&uc, sizeof(ucontext_t));
6
7     /* Ensure non-zero SP. */
8     if (!_REG(sc_uc, SP))
9       goto err;
10
11    /* Ensure non-zero return vaule. */
12    val = val ? val : 1;
13
14    uc.uc_flags =
15      _UC_CPU | ((sc_uc->uc_flags & _UC_STACK) ? _UC_SETSTACK :
16                                                 _UC_CLRSTACK);
17
18    /* Restore sigmask. */
19    sigprocmask(SIG_SETMASK, &sc_uc->uc_sigmask, NULL);
20
21    /* Save return value in context. */
22    _REG(&uc, RV) = val;
23
24    /* Copy saved registers. */
25    _REG(&uc, RA) = _REG(sc_uc, RA);
26    _REG(&uc, SP) = _REG(sc_uc, SP);
27    _REG(&uc, GP) = _REG(sc_uc, GP);
28    _REG(&uc, TP) = _REG(sc_uc, TP);
29    _REG(&uc, S0) = _REG(sc_uc, S0);
30    _REG(&uc, S1) = _REG(sc_uc, S1);
31    _REG(&uc, S2) = _REG(sc_uc, S2);
32    _REG(&uc, S3) = _REG(sc_uc, S3);
33    _REG(&uc, S4) = _REG(sc_uc, S4);
```

```
34    _REG(&uc, S5) = _REG(sc_uc, S5);
35    _REG(&uc, S6) = _REG(sc_uc, S6);
36    _REG(&uc, S7) = _REG(sc_uc, S7);
37    _REG(&uc, S8) = _REG(sc_uc, S8);
38    _REG(&uc, S9) = _REG(sc_uc, S9);
39    _REG(&uc, S10) = _REG(sc_uc, S10);
40    _REG(&uc, S11) = _REG(sc_uc, S11);
41    _REG(&uc, PC) = _REG(sc_uc, PC);
42
43  #ifndef __riscv_float_abi_soft
44    /* Copy FPE state. */
45    if (sc_uc->uc_flags & _UC_FPU) {
46      /* FP callee saved registers are: f8-9, f18-27. */
47      memcpy(&_FPREG(&uc, 8), &_FPREG(sc_uc, 8),
48             (10 - 8) * sizeof(__fpreg_t));
49      memcpy(&_FPREG(&uc, 18), &_FPREG(sc_uc, 18),
50             (28 - 18) * sizeof(__fpreg_t));
51      _FPCSR(&uc) = _FPCSR(sc_uc);
52      uc.uc_flags |= _UC_FPU;
53    }
54  #endif
55
56    setcontext(&uc);
57  err:
58    longjmperror();
59    abort();
60    /* NOTREACHED */
61  }
```

Listing 60: `RISC-V` longjmp (`longjmp.c`)

sigsetjmp and siglongjmp

`sigsetjmp` calls either `_setjmp` or `setjmp` depending on the `savesigs` argument.

`siglongjmp` calls either `_longjmp` or `longjmp` depending on the arguments provided to the corresponding `sigsetjmp` call.

Chapter 4

Tools and usage

When developing an operating system kernel, the two most essential tools are:

- toolchain – a basic set of tools used for compilation, inspecting, and transforming generated binaries, and more,
- emulator – a system emulator provides an abstract platform compatible with the target platform (e.g. QEMU).

Besides an emulator, it is advantageous to have a cycle-accurate simulator of the target platform since it almost perfectly conveys the target hardware.

Furthermore, an additional set of tools is needed when moving to work with actual hardware.

This chapter focuses on practical applications of the Mimiker `RISC-V` port. First of all, we will describe the toolchain and explain the building process. Then, we will show how to compile the whole operating system and analyze generated items. Subsequently, we will depict the Mimiker `RISC-V` hardware repository. Thereafter, we will examine the Renode system emulator and a simulator generated using Verilator. Finally, we will show how to run Mimiker on an FPGA board.

4.1 Toolchain

Mimiker employs a customized GNU toolchain. The toolchain consists of the following elements:

- GCC,
- GNU Binutils,
- GNU Debugger.

Each of the components is built for each target architecture, thereby, whenever a new architecture is introduced, the building scripts must be expanded to include the target architecture.

The GNU toolchain is responsible for building the entire kernel, system libraries, and user space programs.

4.1.1 Building the toolchain

The GNU toolchain building directory is located at `toolchain/gnu`.

The contained `README` provides a list of prerequisites required to build the toolchain.

The build process is controlled through the `config.mk` makefile. For instance, by default, a build is performed for each target architecture. If the user wishes to only build the toolchain for a selected architecture, the `TARGETS` variable should be customized.

The building is performed by simply executing `make`. In the build directory.

The result of the building process is several Debian packages, each containing the toolchain built for a specific architecture.

4.2 Building Mimiker

The Mimiker build system is described in a `README` located at `build`.

Mimiker provides a few build parameters with the most essential one being `BOARD` which specifies the target board of the build. The full list of provided options can be found by inspecting `config.mk`. The most relevant options are responsible for enabling optional kernel modules, for instance, kernel address sanitizer (KASAN). However, not all configurations are supported by the `RISC-V` port yet.

By default, the target platform is set to the Malta board and all optional kernel features are disabled.

The board introduced by this port is called `litex-riscv`, thereby, the simplest way of compiling Mimiker for the `RISC-V` port is by executing the following command in the root directory of the codebase:

```
make BOARD=litex-riscv
```

The kernel is compiled to the form of a static ELF executable (`sys/mimiker.elf`) and is further converted to produce a raw binary image (`sys/mimiker.img`). The ELF is used to provide GDB with debug info, whereas the binary image will be handed over to the bootloader. The compiled kernel contains all essential device drivers thus the user doesn't have to bother about it.

All user-space programs and system libraries are compiled to produce static binaries and are placed in a typical directory hierarchy that composes an initial ramdisk (`initrd.cpio`) which is employed as the rootfs in Mimiker.

4.3 Mimiker RISC-V hardware repository

The Mimiker `RISC-V` hardware repository [28] maintains hardware-related components of the Mimiker `RISC-V` project.

A typical supported `RISC-V` platform will consists of the following elements:

- device tree – a description of the target platform,
- OpenSBI port – implementation of the platform-specific OpenSBI components,
- Renode platform description (repl) and script (resc) – provided to the Renode system emulator,
- simulator – cycle-accurate behavioral simulator generated using Verilator,
- platform implementation – for instance, full HDL code, or scripts used to guide a platform building tool.

For the time being, the only provided platform is LiteX VexRiscv.

4.3.1 LiteX VexRiscv

The `litex` directory contains scripts that:

- create desirable SoC along with any required firmware (e.g. LiteX BIOS),
- generate a DTB,
- provide a Renode platform description,
- run a Verilator generated simulator,
- load bitstream to FPGA and boot LiteX on the target board.

To utilize any of the provided functionalities, the user must install LiteX first. Instructions on how LiteX is installed can be found in the official repository [20].

Besides LiteX, there are some other prerequisites, however, in the majority of cases, an absence of any of the tools will be signaled to the user and most of the tools can be installed from standard packages.

4.3.2 Supported FPGA boards

For now, the supplied scripts support two target FPGA boards:

- Icesugar-pro,
- Arty A7.

Whenever building a SoC, the target FPGA board must be specified.

4.3.3 Basic build

A basic build for Arty A7 is performed by executing
`./make.py --board arty_a7`

This will generate some basic hardware related files (e.g. LiteX CSR map), LiteX BIOS image along with some libraries (e.g. libc), DTB (`/litex/images/rv32.dtb`), and repl (`litex/build/arty_a7`).

4.4 Building OpenSBI

Regardless of whether we are running the `RISC-V` port on a system emulator, simulator, or real hardware, we must provide an OpenSBI image that will serve as runtime firmware in the software stack.

The OpenSBI building process is described in detail in the main `README` of the official repository [21].

To build OpenSBI for our target platform, the following steps must be performed:

1. Clone the official OpenSBI repository.
2. Copy `patches/opensbi/litex` from the Mimiker `RISC-V` repository to `platforms` in the cloned repository.
3. Expose the toolchain by executing
   `export CROSS_COMPILE=riscv32-mimiker-elf-`.
4. Issue a build with
   `make PLATFORM=litex/vexriscv`.

After the build is done, a binary image of produced firmware (`fw_jump.bin`) will be located at `build/platform/litex/vexriscv/firmware`.

4.5 Renode

Renode is an open-source development framework used to emulate physical hardware systems [29].

Renode provides detailed documentation and a set of useful tutorials which can be found at the main site.

4.5.1 Why Renode?

In the case of Malta and RPi3 boards, Mimiker relies on the QEMU system emulator [38]. Why would we make an exception for LiteX `RISC-V` boards?

Versatility

Whereas QEMU supports only a handful of boards that are compatible with some commercial products (e.g. HiFive Unleashed), Renode allows us to create our customized platforms using some basic building blocks.

Automatic platform generation

LiteX contains a utility that can generate a Renode platform description given a json file with LiteX CSR map.

4.5.2 Acquiring Renode

Renode can be installed from packages, extracted from portable release, or built from source.

Although simple and comfortable, there is a problem with the first two options. Renode is under active development and new issues and bugs are fixed daily. However, at the time of writing this thesis, the latest official release has been around for over nine months.

I encourage the reader to either build Renode from source or download the latest build from Antmicro's server [30].

4.5.3 Scripts

Renode scripts have a `.resc` file extension and are used to automate system emulation.

We will explore resc scripts by examining the script used for running the Mimiker `RISC-V` port in Renode. The script can be found in the Mimiker `RISC-V` hardware repository at `patches/renode/scripts/single-node`.

First, the script creates an emulated guest called machine.

Afterward, we load the target platform description in the context of created machine.

The `reset` macro will load the kernel image, DTB, OpenSBI firmware image, and initial ramdisk at specified locations in the main memory. After the loading is done, the control will be handed to OpenSBI. In terms of traditional boot flow, the `reset` macro plays the role of the bootloader.

In the end, the `reset` macro is executed and OpenSBI takes control.

```
1  $name="litex-vexriscv-mimiker"
2
3  using sysbus
4
5  @mach create $name
6  @machine LoadPlatformDescription
7      @platforms/cpus/litex_vexriscv_mimiker.repl
8
9  ### Launch script uses socket terminal integration for UARTs.
10
11 $opensbi=@fw_jump.bin
12 $kernel=@sys/mimiker.img
13 $dtb=@sys/dts/litex-riscv.dtb
14 $initrd=@initrd.cpio
15
16 macro reset
17 """
18     sysbus LoadBinary $kernel  0x40000000
19     sysbus LoadBinary $dtb     0x40ef0000
20     sysbus LoadBinary $opensbi 0x40f00000
21     sysbus LoadBinary $initrd  0x42000000
22
23     cpu PC 0x40f00000
24 """
25
26 runMacro $reset
```

Listing 61: LiteX VexRiscv Renode script (`litex_vexriscv_mimiker.resc`)

4.5.4 Platform descriptions

A Renode platform description (repl) defines devices composing the platform along with connections between the devices.

Although LiteX can generate an appropriate repl automatically, the generating script is obsolete, and thereby the resulting description is not adjusted to the latest Renode build. For this reason, we supply our repl (based on the generated one). The platform description can be found in the Mimiker `RISC-V` hardware repository at `patches/renode/platforms/cpus`.

MSWI and MTIMER interrupts are connected to HLIC interrupts 3 and 7, respectively.

PLIC has two contexts. One linked with the machine mode of the hart and one linked with supervisor mode.

The UART interrupt is connected to the first PLIC interrupt.

```
1  cpu: CPU.VexRiscv @ sysbus
2      cpuType: "rv32ima"
3      privilegeArchitecture: PrivilegeArchitecture.Priv1_10
4      builtInIrqController: false
5      timeProvider: clint
6
7  rom: Memory.MappedMemory @ sysbus 0x0
8      size: 0x000010000
9
10 sram: Memory.MappedMemory @ sysbus 0x10000000
11     size: 0x00002000
12
13 ram: Memory.MappedMemory @ sysbus 0x40000000
14     size: 0x10000000
15
16 soc_controller: Miscellaneous.LiteX_SoC_Controller @ sysbus 0xf0000000
17
18 uart0: UART.LiteX_UART @ sysbus 0xf0001000
19     IRQ -> plic@1
20
21 clint: IRQControllers.CoreLevelInterruptor @ sysbus 0xf0010000
22     frequency: 100000000
23     numberOfTargets: 1
24     [0, 1] -> cpu@[3, 7]
25
26 plic: IRQControllers.PlatformLevelInterruptController @ sysbus 0xf0c00000
27     numberOfSources: 31
28     numberOfContexts: 2
29     prioritiesEnabled: false
30     [0,1] -> cpu@[11, 9]
```

Listing 62: `RISC-V` virtual memory mapping (`pmap.c`)

4.5.5 Integrating Renode with Mimiker

Mimiker has a dedicated `launch` script used for running the operating system in an emulated environment.

Before my changes were introduced, the QEMU emulator was assumed. In my solution, each target board has to explicitly specify the emulator of choice (QEMU or Renode).

Besides expanding the general logic to have regard to both emulators, I had to implement an object representing the Renod emulator, that is, the `RENODE` class.

For each provided UART device, we create a socket ❶ that exposes the virtual UART to the host system.

If the user has specified the debug option, we have to start a GDB server ❷ which can be connected to using the `target remote` GDB command.

Finally, we overwrite the kernel command line property of the DTB generated by LiteX script. We achieve this goal as follows:

1. We create a device tree overlay with command line taken from the user ❸.
2. We compile the DTS into DTB ❹.
3. We combine the DTB taken from the hardware repository with the created overlay to obtain a final device tree blob ❺.

```
1  class RENODE(Launchable):
2      def __init__(self):
3          super().__init__('renode', 'renode')
4
5          self.options = getopts('renode.options')
6
7          script = getvar('renode.script', failok=False)
8          self.options += [f'-e include @scripts/single-node/{script}']
9
10       ❶for i, uart in enumerate(getvar('renode.uarts')):
11             port = uart['port']
12             dev = f'uart{i}'
13             self.options += ['-e emulation CreateServerSocketTerminal ' +
14                              f'{port} "{dev}" False']
15             self.options += [f'-e connector Connect sysbus.{dev} {dev}']
16
17       ❷if getvar('config.debug'):
18             gdbport = getvar('config.gdbport', failok=False)
19             self.options += [f'-e machine StartGdbServer {gdbport}']
20
21         with open('bootargs.dts', 'w') as bootargs_dts_file:
22           ❸bootargs_dts_file.write(
```

```
23                  BOOTARGS_DTS_TEMPLATE.format(
24                      ' '.join(getvar('config.args')))
25              )
26          ❹subprocess.check_call(
27              'dtc -O dtb -o bootargs.dtbo bootargs.dts', shell=True
28          )
29          dtb = os.path.join('sys/dts/', '{}.dtb'.format(getvar('board')))
30          ❺subprocess.check_call(
31              'fdtoverlay -i {} -o {} bootargs.dtbo'.format(dtb, dtb),
32                  shell=True
33          )
34          os.remove('bootargs.dts')
35          os.remove('bootargs.dtbo')
36
37          self.options += ['-e start']
```

Listing 63: Renode launch class (`launch`)

4.5.6 Running Mimiker on Renode

To run the port in Renode, the user should perform the following steps:

1. Copy the resc script and the repl to the corresponding paths in the Renode installation directory.

2. Place the OpenSBI firmware binary in the main directory of the Mimiker repository.

3. Copy the generated DTB to `sys/dts` and rename it to `litex-riscv.dtb`.

4. Run the Mimiker launch script.

The launch script command-line options can be listed by executing
`./launch --help`

A typical invocation specifies the target board (the `--board` option) along with a single kernel argument defining the init program (`init=<absolute path>`).

The init program constitutes the first user-space process and in the majority of cases is set to the Korn shell (ksh) port prepared for Mimiker.

A command line used to run ksh on the `RISC-V` port in the emulated environment may look like this
`./launch --board litex-riscv init=/bin/ksh`

4.6 Verilator

Verilator is an open-source tool used for generating a cycle-accurate C++ model of the hardware given a hardware description written in Verilog [31].

4.6.1 Why do we want a cycle-accurate simulator?

Such a simulator is an exact reflection of the target hardware, thereby, allowing us to robustly test the software without the need for any interaction with real hardware.

4.6.2 Litex VexRiscv simulator

The Mimiker `RISC-V` hardware repository provides a script for running a Verilator generated simulator. The created model simulates the board for which the basic build (using `litex/make.py`) has been performed, thereby, a build must be performed first.

The script file is `litex/sim.py`. As usual, the `--help` option can be utilized to view all command line parameters.

Before the simulator can be started, the user must provide the following files and place them in the `litex/images` directory:

- `Image` – the raw binary image of the built kernel (`mimiker.img`),
- `rootfs.cpio` – ramdisk image (`initrd.cpio`),
- `opensbi.bin` – the raw binary image of the built OpenSBI firmware (`fw_jump.bin`).

The most straightforward way for running the simulator is by executing
`./sim.py`

4.7 Running Mimiker RISC-V on FPGA

To run Mimiker on the LiteX VexRiscv hardware platform the following steps must be performed (the presented commends employ the Arty A7 board):

1. The bitstream needs to be built. This requires Xilinx Avado and is issued by executing
   `./make.py --board arty_a7 --build`
2. The built bitstream must be loaded to the FPGA board by issuing
   `./make.py --board arty_a7 --load`

3. As for the simulator, the target images must be placed in `litex/images`.

4. Finally, the `lxterm` tool provided with LiteX is used to boot LiteX on the target board (the user should determine the exact ttyUSB identifier of the connected board)
   ```
   lxterm --speed 1e6 /dev/ttyUSB? --images images/boot.json
   ```

For the time being, the user must build the bitstream on his or her own. However, we plan to provide a downloadable prebuilt bitstream for users not interested in SoC development.

Chapter 5

Summary

It is a challenging task to prepare a port of an operating system. It requires a fair knowledge of the target architecture and associated software and hardware environments, along with a thorough understanding of the most vital components of the target operating system.

Although it is a demanding process, there is certainly wisdom unutterable to be had from it. Operating system development is sublime art and it has changed my life.

5.1 Contributions

As the effort of the Mimiker `RISC-V` project is divided into software and hardware components, I will present my contributions applying the same split.

5.1.1 Mimiker RISC-V hardware

I have provided an implementation of the execution environment for the Mimiker `RISC-V` kernel along with a platform description and a script supplied to the Renode system emulator.

All remaining elements, including the Mimiker SoC building scripts and simulator, have been provided by Marek Materzok.

5.1.2 Mimiker RISC-V software

I have implemented the abstract CPU model employed by Mimiker, provided drivers for devices required by target architecture and target platform, and supplied implementation of machine-dependent parts of Mimiker's system libraries.

5.1.3 Results

As the result of my work, Mimiker has been adapted to a wide range of LiteX platforms incorporating a `RISC-V` softcore.

The presented port is fully operative in the emulated environment provided by Renode. All userspace programs are responsible and usable just as for other ports.

Although it is possible to run Mimiker on a cycle-accurate simulator employing a single host thread and obtain proper operation, issues arise as the number of threads grows. Similarly, problems occur when running on actual hardware.

5.1.4 Future work

Although a lot has been done, there is more to be done.

First of all, the vast majority of described contributions have not been merged yet. My main effort will concentrate on introducing the existing changes to the main branch.

All issues exposed by running on the simulator and real hardware must be fixed.

Some significant components have not been integrated with the `RISC-V` port yet. These are:

- kernel address space sanitizer (KASAN),
- CFI directives,
- CI automated tests,
- architecture customized userspace tests.

The provided port is meant for LiteX generated platforms incorporating a `RISC-V` 32-bit CPU equipped with CLINT and PLIC devices. It would be beneficial to drop the CLINT and PLIC assumption and add support for platforms lacking any of these devices. Moreover, we could lift the 32-bit assumption thereby enabling Mimiker on 64-bit `RISC-V` CPUs. Finally, by adding appropriate device drivers, we could run Mimiker on entirely new boards, for instance, HiFive Unleashed.

Bibliography

[1] VexRiscv repository, https://github.com/SpinalHDL/VexRiscv

[2] PicoRV32 repository, https://github.com/YosysHQ/picorv32

[3] Krste Asanović, David A. Patterson
Instruction Sets Should Be Free: The Case For RISC-V

[4] Mimiker web page, https://mimiker.ii.uni.wroc.pl

[5] Malta-R development platform,
MIPS® Malta™-R Development Platform User's Manual

[6] Paweł Jasiak
Port of Mimiker Operating System for AArch64 Architecture

[7] RISC-V ISA GitHub page, https://github.com/riscv

[8] RISC-V non-ISA GitHub page, https://github.com/riscv-non-isa

[9] Jason Lowe-Power, Christopher Nitta
The Davis In-Order (DINO) CPU

[10] Stephen A. Zekany, Jielun Tan, James A. Connolly, Ronald G. Dreslinski
RISC-V Reward: Building Out-of-Order Processors in a Computer Architecture Design Course with an Open-Source ISA

[11] Marek Materzok, *DigitalJS: a Visual Verilog Simulator for Teaching*

[12] *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Version 20191213*

[13] *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 20211203*

[14] RISC-V GCC options
https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html#RISC-V-Options

[15] FreeBSD manpages, *pmap(9)*
https://www.freebsd.org/cgi/man.cgi?query=pmap&apropos=0&sektion=0&manpath=FreeBSD+13.0-current&arch=default&format=html

[16] Linux kernel documentation, *RISC-V Hart-Level Interrupt Controller (HLIC)*
https://www.kernel.org/doc/Documentation/devicetree/bindings/interrupt-controller/riscv%2Ccpu-intc.txt

[17] *RISC-V Platform-Level Interrupt Controller Specification Version 1.0*

[18] *SiFive E31 Manual v19.08p0*

[19] *RISC-V Advanced Core Local Interruptor Specification Version 1.0*

[20] LiteX repository https://github.com/enjoy-digital/litex

[21] *RISC-V Supervisor Binary Interface Specification Version 1.0-rc2*

[22] OpenSBI repository
https://github.com/riscv-software-src/opensbi

[23] OpenSBI on LieX VexRiscv
https://github.com/litex-hub/opensbi/tree/0.8-linux-on-litex-vexriscv

[24] *RISC-V ABIs Specification Version 1.0-rc1*

[25] *Devicetree Specification Release v0.3-40-g7e1cc17*

[26] NetBSD source code – sys/arch/riscv/include/asm.h

[27] NetBSD manpages, *setjmp(3)*
https://man.netbsd.org/setjmp.3

[28] Mimiker RISCV hardware repository
https://github.com/tilk/mimiker_riscv_hardware

[29] Renode web page, https://renode.io/

[30] Latest Renode builds, https://dl.antmicro.com/projects/renode/builds/

[31] Verilator web page, https://www.veripool.org/verilator/

[32] NetBSD manpages, *callout(9)*
https://man.netbsd.org/callout.9

[33] Mimiker source code – include/sys/vm_physmem.h
https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/vm_physmem.h?r=093d1520

[34] Mimiker source code – `include/sys/klog.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/klog.h?r=6429a3a2`

[35] Mimiker source code – `include/sys/kenv.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/kenv.h?r=bf94105b`

[36] Mimiker source code – `include/sys/malloc.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/malloc.h?r=e9ef8e55`

[37] Mimiker source code – `include/sys/kmem.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/kmem.h?r=03df3238`

[38] QEMU web page, `https://www.qemu.org/`

[39] Shiva Chen, Hsiangkai Wang
Compiler Support For Linker Relaxation in RISC-V