



Hochschule
Augsburg University of
Applied Sciences

Master Project Report

Linux for the *ParaNut* Processor

Nico Borgsmüller (Mat.-Nr.: 2110949)

Faculty: Computer Science
Field of study: Master Computer Science
Workgroup: Efficient Embedded Systems
Submitted on: 03.07.2022
Supervisor: Prof. Dr.-Ing. Gundolf Kiefer



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.

Contents

List of Figures	iii
Acronyms	iv
1 Introduction	1
1.1 Motivation	1
1.2 Purpose of this Work	2
1.3 Structure of this Work	2
2 Background Information	3
2.1 RISC-V	3
2.1.1 Overview	3
2.1.2 The Base ISA and Extensions	3
2.1.3 Privilege Modes	4
2.1.4 Control and Status Registers	5
2.1.5 Virtual Memory	6
2.2 The ParaNut	8
2.3 Linux	10
2.4 OpenSBI	13
2.5 RISC-V Debugging	15
3 Linux on the ParaNut	19
3.1 Preparing the Kernel	19
3.1.1 Configuration Options	20
3.1.2 The Device Tree	21
3.2 Preparing OpenSBI	23
3.2.1 Disabling CoPUs	23
3.2.2 Atomic Instructions	23
3.2.3 The Serial Interface	25
3.2.4 Time Registers	25
3.2.5 Configuration and Compilation	25
3.3 Adaptations of the ParaNut Hardware	27
3.3.1 Virtual Memory Management	27
3.3.2 Debugging	29
3.3.3 CSR Refactoring	30
3.3.4 Running on the FPGA	31
4 Testing Linux on the ParaNut	33
4.1 Running on the Simulator and Hardware	33
4.2 Observed Output	34
4.3 Debugging Techniques	37

5 Conclusion	38
5.1 Summary	38
5.2 Known Issues and Next Steps	38
References	40

List of Figures

1 RISC-V privilege levels and interfaces (from [20, p. 2]) 5

2 Overview on the interfaces between software in different privilege modes
(from [17]) 13

3 RISC-V Debug Overview (Simplified version of figure 2.1 in [15, p. 6]) . . . 15

4 State diagram of the EXU states important for flushing 29

Acronyms

ABI	Application Binary Interface.
BBL	Berkeley Boot Loader.
BSS	Block Starting Symbol.
CePU	Central Processing Unit.
CoPU	Co-Processing Unit.
CPU	Central Processing Unit.
CSR	Control and Status Register.
DM	Debug Module.
DTC	Device Tree Compiler.
DTM	Debug Transport Module.
ELF	Executable and Linkable Format.
EXU	Execution Unit.
FDT	Flattened Device Tree.
FPU	Float Processing Unit.
GDB	GNU Debugger.
GNU	GNU's Not Unix.
GOT	Global Offset Table.
HTIF	Host Target Interface.
IFU	Instruction Fetch Unit.
ISA	Instruction Set Architecture.
JTAG	Joint Test Action Group.
LSU	Load/Store Unit.
MMU	Memory Management Unit.
OpenOCD	Open On-Chip Debugger.
OS	Operating System.
PCI	Peripheral Component Interconnect.
PIC	Position Independent Code.
PMP	Physical Memory Protection.
QEMU	Quick Emulator.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
SBI	Supervisor Binary Interface.
SEE	Supervisor Execution Environment.
SIMD	Single Instruction Multiple Data.
TCL	Tool Command Language.
UART	Universal Asynchronous Receiver Transmitter.
UEFI	Unified Extensible Firmware Interface.
USB	Universal Serial Bus.

1 Introduction

1.1 Motivation

Previous decades have shown a continuous increase in processing power requirements. While until now the actual processing power per chip area has increased drastically according to Amdahl's Law, this increase is slowing down due to physical limitations. To fight this, modern processors contain an increasing number of processor cores, thus multiplying their processing power by multiplying their resource requirements like area and energy. Alternatives to this include the use of SIMD extensions present in many modern CPUs. These Single Instruction Multiple Data units can execute the same instruction on multiple different data locations. They require much less space and energy than a whole processor core but are of course not as flexible. [8]

The goal of the *ParaNut* processor developed at the Augsburg University of Applied Sciences is to combine both approaches into a single parallelization concept. It has a configurable number of *Co-Processing Units* (CoPUs), where each core can combine both approaches of parallelization introduced above. They either offer full thread support in the fashion of simultaneous multithreading or they can run in linked mode, which means they execute the same instruction as the main core, but on different data. Additionally to most traditional SIMD extensions, the *ParaNut* does not need a specific instruction set for the linked mode, but can run any non-branching instruction of the standard instruction set. To save resources, every core except the main core can be configured to only support linked mode or both modes. When both modes are present, this offers a high level of flexibility to the user, as the *ParaNut* allows switching between the different parallelization strategies at run time. [12]

When writing applications for the *ParaNut*, they have to be specifically written for this bare metal target using a *ParaNut* support library. Existing applications and libraries must possibly be adapted to run on the target. This includes general adaptations that are required for every board as for example changes to the linker script, but also the possibility to exploit the special parallelism on the *ParaNut*. As most software applications are only available for operating systems, they would require much larger effort to be adapted to the *ParaNut*. To simplify this process and reduce the adaptation process to the new parallelization concept, an operating system should be used on the processor.

1.2 Purpose of this Work

As one of the most used operating system kernels, Linux was chosen to be ported to the *ParaNut*. This operating system takes care of abstracting device internals like drivers, it allows pseudo-parallel execution of tasks through scheduling and provides a simplified programming interface for user space programs. To be able to run on specific piece of hardware, Linux has a few requirements on the hardware as well as the software environment, which will be explained in further chapters of this work.

The purpose of this project work was to be able to run a Linux operating system on the *ParaNut* processor. As the goal of porting a full-fledged operating system would have been too complex, it was reduced to the following aspects:

- Identifying necessary features of the hardware for running the Linux kernel.
- Correctly configuring the Linux kernel so it can possibly run on the *ParaNut*.
- Testing the newly implemented Memory Management Unit of the *ParaNut* through the Linux kernel.
- Getting the Linux kernel to start and run as far as possible.

1.3 Structure of this Work

This work is structured as follows: The following chapters first provide background information on the instruction set architecture as well as on the required components to run Linux on the *ParaNut* processor. Afterwards, it is explained how the different components need to be modified and configured to reach this goal. In the next chapter, different debugging strategies are presented. Finally, a conclusion is drawn and missing steps for full Linux support are discussed.

2 Background Information

2.1 RISC-V

2.1.1 Overview

As the RISC-V homepage states, "RISC-V is a free and open ISA" [1]. This means on one hand that everyone can contribute to the different standards of this instruction set architecture. On the other hand, everyone can also use the standard to build their own processor core, commercial or not. The project was started in 2010 at the UC Berkeley and is coordinated since 2015 through The RISC-V Foundation with now over 300 member organizations [9]. It is called RISC-V as it is the fifth Reduced Instruction Set Computer (RISC) architecture which was developed at the UC Berkeley [19, p. 1].

The ISA was designed modularly to allow for different levels of complexity when building a RISC-V core. There are different address widths (32bit, 64bit or 128bit), different extensions, like multiplication or floating point numbers or the addition of privilege modes as required for the execution of more complex operating systems like Linux. Further, there exists a separate debug as well as a trace specification that can optionally be implemented. The following paragraphs will now present the most important aspects of the RISC-V ISA, which are required for the understanding of the following chapters. [19]

2.1.2 The Base ISA and Extensions

The main parts of the specification are separated into two parts. The first considers all unprivileged features and is thus called the "Unprivileged Specification" [19], while the second is called the "Privileged Specification" [20], which takes care of all the different features that require privilege levels in the processor. The unprivileged specification starts with some basic information like instruction encoding or an overview of the available registers. It also introduces the term "hart". A hart is defined as any part of hardware that can fetch and execute instructions. This definition both includes full cores as well as some kind of threads. Further, the different base instruction sets, depending on the address width, are explained. Most importantly, these are RV32I, RV64I and RV128I. These base instruction sets can be extended with a large number of extensions, of which the most important are as follows:

- **M-extension:** Integer multiplication. Provides multiplication and division operators.
- **A-extension:** Atomic instructions. Provides a set of instructions for atomic memory management.

- **F-extension:** Floating point support. Provides a set of floating point registers as well as arithmetic instructions to modify them.
- **C-extension:** Compressed instructions. Extends the normal 32-bit instruction set with some shorter 16-bit instructions that represent common operations which require fewer parameters.

The unprivileged specification document also contains the "Assembly Programmer's Handbook" which defines some pseudo-instructions for simpler programming. This includes for example the `li` (load immediate) instruction, which is translated to some other sequence of instructions during assembling depending on the exact immediate value. [19, pp. 137ff]

2.1.3 Privilege Modes

The next document is the privileged specification, which is the most important part for this project [20]. It starts with defining different privilege levels for the processor's execution:

- **Machine mode** (M-mode): This is the default mode where everything is allowed.
- **Supervisor mode** (S-mode): This mode is typically used for operating systems, which need a bit fewer permissions. This mode can for example manage virtual memory, which will be explained in more detail below.
- **User mode** (U-mode): Being the mode with the least privileges, it is normally used to run applications under an operating system. It generally has no rights to modify any settings related to the processor's execution.

According to the specification, different combinations of supported privilege modes are allowed. There can be only the M mode, the combination of M and U mode to run low privilege applications or the full stack of M, S and U mode, which is the typical setup for running an operating system like Linux. Additionally, a Hypervisor extension is also defined, which leads to slight modifications and extensions of the given privilege levels. Since it is not relevant for this work, it won't be explained in more detail. The specification also defines the basic principles on how the different modes interact with each other. For the case of all three modes being supported, figure 1 shows the different layers involved. The user application uses an Application Binary Interface (ABI) to interact with the Supervisor, which are for example the Linux system calls. The Supervisor then also has the ability to access the Machine level software, which can also be called Supervisor Execution Environment (SEE). For that it uses a standardized Supervisor Binary Interface

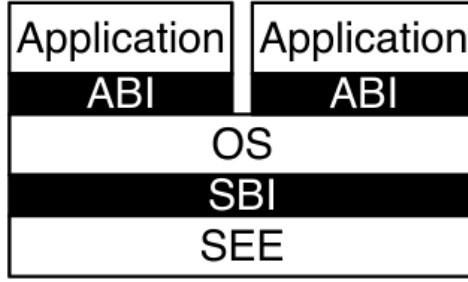


Figure 1: RISC-V privilege levels and interfaces (from [20, p. 2])

(SBI). SBI calls can be used for example to access machine level CSRs (Control and Status Registers) or to control the execution state of the available harts. More explanation on this follows in chapter 2.4 about OpenSBI.

2.1.4 Control and Status Registers

Additionally, the specification defines sets of Control and Status Registers (CSRs) for the different privilege levels. If the registers are only accessible by a specific privilege mode, their names are prefixed by the first letter of this mode (e.g. `mstatus` for M-mode). The most important registers for this work are the following:

- `mstatus`, `sstatus`: Contains a set of flags which control the current hart's state. Relevant fields will be explained when needed.
- `mtvec`, `stvec`: Trap vector address: Exceptions and interrupts jump to this address.
- `medeleg`, `mideleg`: Exception/Interrupt delegation: Which interrupts should be handled by S-Mode instead of the M-mode.
- `mepc`, `sepc`: Exception program counter: The PC value before an exception occurred.
- `mcause`, `scause`: Exception/Interrupt cause.
- `mtime`: Real-time counter. Note that this is not a CSR but provided as a memory-mapped register.

- `time` : Real-time counter CSR accessible by all modes. Should shadow the mtime register.
- `satp` : Supervisor address translation and protection. Important for the virtual memory support explained below.

Furthermore, the privileged specification adds some instructions related to the different privilege modes. The most important instructions are:

- `ecall` : The ecall instruction (environment call) calls the higher-privilege-level execution environment by producing an exception.
- `mret` , `sret` : Returns execution to the lower level privilege level by jumping to `mepc` or `sepc` respectively.
- `sfence.vma` : This instruction needs to be executed to tell the processor to refresh data structures that are located in memory, like for example page tables for virtual memory.

2.1.5 Virtual Memory

The next important aspect of the specification is the virtual memory management. This concept allows for separation of different sections in memory with different permissions, for example to separate different processes. These sections are called pages. The permissions include readability, writability or if the specific memory locations can be executed. It additionally allows to map arbitrary address spaces to different parts of the physical, contiguous memory. This allows that every process could possibly be located at the same virtual memory address while the actual data is spread over different regions of the physical memory. These features introduce the need of some kind of translation unit in hardware which needs to translate a virtual address into a physical address, which is called the MMU (Memory Management Unit). The RISC-V specification defines multiple types of virtual memory, but only a single variant is applicable for RV32 implementations, so only the Sv32 variant will be explained here:

The pages have a fixed size of 4 KB. Virtual addresses are made of a virtual page number (VPN), which identifies the page and a page offset to point to the memory location inside the page. The Supervisor level application now needs to provide a table (the page table) to define the different existing pages and their mapping to physical memory. The address where the table is located in memory must be written into the `satp` CSR register mentioned above. If it was changed, the `sfence.vma` instruction needs to be executed

so that the processor knows that the table has to be refreshed. An entry of the page table is identified by the first part of the virtual page number, which defines the offset from the page table address. If the valid flag of the entry is not set, it should not be considered and raises an exception. A valid entry is now either an actual entry or a pointer to a next-level page table. The next-level page table then uses the next part of the virtual page number to identify an entry. This concept allows to define different sizes of pages, the page is largest if the top-level page tables already contains an actual entry. An entry then consists of the higher bits of the physical address, which is combined with the page offset from the virtual address to form the final address. It further contains the R, W, X flags to define the actions that can be taken on the memory addresses of the specific page. All these actions and checks must be performed transparently by the hardware so that an access to a virtual address directly leads to the physical address being accessed, with no additional action necessary by the software. Note that virtual memory can only be active during U and S mode execution and is automatically turned off when switching to M mode. More information can be found in the specification [20] or Christian Meyer's work on the Memory Management Unit for the *ParaNut* [13].

2.2 The ParaNut

The *ParaNut* is a RISC-V compatible processor developed by the Efficient Embedded Systems group at the Augsburg University of Applied Sciences. It is FPGA-based, so the developed hardware can be programmed onto an FPGA device. The main motivation is its specialized parallelization concept as explained in the introduction. The *ParaNut* is implemented in the SystemC language, which is used by a High Level Synthesis tool to generate the necessary hardware. Additionally, this SystemC model can be used to run a cycle-accurate simulation of the processor without the need of an actual FPGA device. This simulator allows to generate different debug outputs that will be explained in chapter 4.3 *Debugging Techniques*.

The processor supports a subset of the RISC-V features, which will be presented in this paragraph. Firstly, the *ParaNut* implements the RV32I base instruction set with the Multiplication (M) and Atomic (A) extensions. While the M extension is fully implemented, the support for the A extension is only partial. The extension defines two types of instructions which can be considered redundant. The first consists of the two instructions `lr / sc` (load reserved/store conditional). With `lr`, a value is loaded from the given address. When the `sc` instruction on the same address is executed, it can only succeed when the memory at the given address was not changed since the `lr` instruction. If it was changed, the value is not written and the software implementation can jump back to the `lr` instruction and try again. This is enough to implement any kind of atomic instruction. Additionally, the specification provides the utility instructions `amoswap`, `amoadd` or `amoand` (and many more) to perform the three steps: Load, Operation, Store in one instruction. This is, for example, in the case of `amoadd`: Load a value from the given address, add a different value to it and finally store the modified value back to the given address. For simplicity reasons of the hardware, only the `lr / sc` instructions were implemented in the *ParaNut*.

Furthermore, all privilege modes are supported in the *ParaNut*. A certain set of CSRs is also implemented. This includes all of the CSRs introduced in the RISC-V chapter above. Further information on those registers can be found in the *ParaNut* manual [12]. All instructions necessary for the privileged modes to function that were introduced above are also implemented. The Sv32 virtual memory variant was implemented by Christian Meyer prior to this work [13], although some changes to this implementation were still necessary and are described later in this work.

Additionally to the standardized features, the *ParaNut* also provides some additional features. First and foremost, this is of course the support for the cores that are switchable between Linked and Threaded Mode. Since this feature was not relevant for the tasks of this work, it will not be explained in further detail. To be able to output data to

the console, the *ParaNut* provides a simple "tohost" interface, which allows the software implementation to print data over a serial interface both in the simulator and on the FPGA. This interface is a 8 bit symbol at a specific address where the application can write a character. This character is then read by the simulator or the firmware implementation on the FPGA and printed out to the console. The character is then cleared to signal the application that a new character can be written.

2.3 Linux

Linux was first released in 1991 by Linus Torvalds. It quickly developed from a small personal project to an international free and open source software with support for a large number of different processor architectures and hardware drivers. Linux is now the state-of-the-art operating system for many applications, including mobile phones [14], supercomputers [7] or embedded devices [6]. The term Linux generally only refers to the kernel without any user space software. To build a full operating system, an additional set of software is required. Often, GNU tools are used, which leads to the term GNU/Linux to refer to the full operating system. In this case, this work will only talk about the kernel itself, as the focus lies on the boot process before a user space process can even be created. Most information in this chapter was collected manually using the Linux kernel release 5.18.

The RISC-V port of the Linux kernel contains different adaptations for this instruction set architecture. Most importantly of course, the boot process must be adapted. Different parts of the kernel have to be set up. The boot process starts with some architecture specific assembly startup code in `arch/riscv/kernel/head.S`. This code establishes some expectations to the calling instance. The processor must be in S-mode and the memory has to be "identity mapped", which is achieved by not enabling paging. The `a0` register contains the current hart id and the `a1` register contains a pointer to a device tree binary located in memory. The device tree provides information on the hardware the current system is running on. Typical general-purpose computers, like for example PCs or servers, contain busses like PCI or USB. Those busses are enumerable, which means the system can find attached devices and their settings automatically. On embedded or smaller systems in general this is not always the case, so the developer needs to provide a topology of attached devices and their configuration. This information is provided to the Kernel by the standardized device tree format. Specifics on this topic will be provided below when introducing the necessary steps to get Linux running. [3]

To start the boot process, the entry point of the Linux kernel jumps to the `_start_kernel` symbol in `arch/riscv/kernel/head.S`. As the RISC-V harts can start in arbitrary order, Linux needs to decide which hart is used for booting. For this, the so-called "boot lottery" is used. The first hart to reach this point is selected as the boot hart, others are parked in a wait loop until a specific point in the boot procedure. The process continues with masking any interrupts and filling the BSS section with zeroes so that any variable that expects to be zero-initialized actually contains zeroes.

Already in the next step, virtual memory is set up. The kernel relocates its own execution from the initial address to `0xC0000000` by filling a page table with a linear mapping and enabling the MMU through the `SATP` register. It is activated by setting the trap

vector to the virtual address of the following instruction and executing an `sfence.vma` instruction. Once the paging was activated, the processor will produce a page fault exception as the current address space is not present in the page table. This leads to a jump to the previously set trap vector. Afterwards, the execution continues in this virtual memory space.

Afterwards, the trap vector is set again to a very simple exception handler that lets the processor run in a loop when an exception has occurred. Further, the C environment is initialized by constructing for example a stack frame and initializing the global and thread pointers. After these steps are completed, execution jumps to a generic `start_kernel` function in `init/main.c`, which is now architecture independent and written in C.

The `start_kernel` function performs a total number of around 90 setup function calls in a specific order. Most of them won't be explained in detail, but some general information can be found in [10], [18] or the in-tree Linux documentation. After some very basic initialization functions, there is again an architecture-specific `setup_arch` function. It invokes 16 additional setup functions, which perform a few architecture-dependent initialization steps. The device tree is parsed for further usage and a more complex page table for IO, allocatable space and more is set up. Further, the Supervisor Binary Interface is detected and configured. This interface to the Supervisor Execution Environment will be explained further in the following chapter. Next, early logging is initialized to provide the user with log output as early as possible. If available, the serial output functionality of the SBI (Supervisor Binary Interface) is utilized. At this point, the other parts of the system are started. They each get a very basic thread and stack context and are then available to be assigned with tasks. Lastly, the `setup_arch` function sets up the memory management sub system of the Linux kernel.

The architecture-independent code now continues with parsing the kernel command line as well as setting up interrupts and the interrupt controller. As the RISC-V interrupt controller is abstracted as a driver and identified by the device tree, the code doesn't have to be architecture-specific. With the same principle, the timer is initialized. This driver is implemented by reading the RISC-V `time` CSR. Afterwards, many more initialization functions follow, including file system setup, although they are not that relevant to the goals of this project work. The setup process finishes by performing the first context switch to the "init" process.

In general, only a very small part of this boot process is architecture-specific to RISC-V, most things are simply abstracted by the kernel. Further architecture-specific adaptations in the Linux Kernel include the context switching itself, the handling of atomics and synchronization primitives as well as the memory handling and interactions with the MMU. Even though these topics are still relevant for porting Linux in general, they won't be

explained in detail here as they did not need to be touched in the process of getting Linux to run on the *ParaNut*. The context switch implementation is most likely going to be adapted when trying to take advantage of the special parallelization concept of the *ParaNut* in a future project.

2.4 OpenSBI

OpenSBI [16] is an implementation of the standardized Supervisor Binary Interface (SBI) [17]. In this case, it acts as the SBI but also as a bootloader that loads the Linux Kernel. OpenSBI could alternatively be used in combination with a different bootloader like "Das U-Boot" instead [5]. In this case, it would only provide the environment call interfaces and some RISC-V specific library functions for the actual bootloader. Different alternatives for OpenSBI also exist. The earliest official implementation of the SBI standard was the Berkeley Boot Loader, or BBL for short. It was kept quite simple and was replaced by OpenSBI. One alternative that claims to be feature-equivalent to OpenSBI is RustSBI, fully written in the Rust programming language. In this work, the OpenSBI version 1.0 will be used, which was released in December 2021.

The following paragraphs will now present the RISC-V Supervisor Binary Interface specification itself. The SBI acts as the interface between the Supervisor Execution Environment (SEE), which is some kind of firmware or bootloader and is generally more specific to the platform and an operating system running in S-mode, which can be kept more general-purpose. It enables the supervisor (OS) to perform privileged operations through the `ecall` instruction. This call is the equivalent of system calls, which are between U-mode and S-mode. This setup is illustrated in figure 2. The SBI offers a variety of different interfaces in the form of extensions, of which most are optional. The base extension allows to retrieve information on the SBI itself, on the machine as well as on the list of available extensions. Important interfaces include printing and reading characters to or from the console, setting a timer, sending inter-process interrupts to other harts, executing fence instructions on other harts or managing the hart state by starting, stopping or suspending specific harts. It further provides the possibility of a system reset for performing a reboot or shutdown as well as managing the performance monitoring registers (e.g. `(m)cycle` for cycle counting) by changing relevant M-mode CSRs.

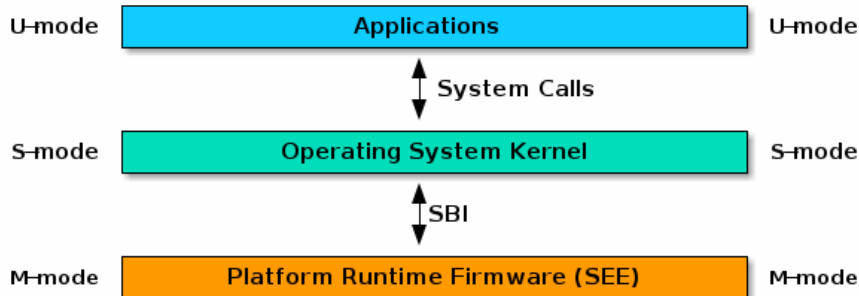


Figure 2: Overview on the interfaces between software in different privilege modes (from [17])

In addition to these interfaces, OpenSBI performs some more tasks. It can emulate unsupported instructions when they are detected through an "Illegal Instruction Exception" as well as handle unsupported CSRs by capturing reads and writes to them.

During the startup process, OpenSBI performs a number of setup steps. Execution starts in `firmware/fw_base.S` at the `_start` assembly label. It first decides on a boot hart to continue and afterwards initializes the global offset table when position independent code (PIC) is enabled. This table contains pointers to functions and other symbols and allows to run at an arbitrary address that is not known beforehand. The startup process further resets all registers and fills its BSS section with zeroes just like the Linux kernel. Afterwards, a temporary trap handler and well as a stack is set up. Next, the platform specific code gets the chance for basic initialization by calling the `fw_platform_init` function. This function will, in the general case, parse the device tree and try to detect the actual platform. Lastly in this assembly script, the `sbi_init` C function is called. In the case of the boot hart, this function calls a set of initialization functions for the different features and extensions. It defines memory domain regions and detects which feature are supported by the processor. These features include physical memory protection (PMP), different performance counters and numerous CSRs that might or might not be implemented. It further initializes the console output, which is supported by a set of different drivers, as well as the interrupt controller as defined in the device tree. It also sets up the PMP if available. Subsequently, OpenSBI cleans up the device tree by only setting certain harts as active that have all necessary features and disabling all OpenSBI-specific features. Finally, the privilege mode is switched to the configured target mode, the registers `a0` and `a1` are filled with the hart id and a pointer to the device tree respectively and the execution jumps to the entry point of the provided payload, the Linux kernel in this case.

Different platforms have different prerequisites and hardware configurations. For this reason, OpenSBI can be compiled for a variety of RISC-V targets. These can either be configured at compile time, or the "generic" platform target can be used. In the latter case, OpenSBI detects the system configuration based on the provided flattened device tree. Different drivers exist for different tasks, for example the serial console, the timer, the interrupt controller, etc. Additionally, the "generic" platform can still be extended with specific callback functions for specific detected platforms. This detection takes place through the `compatible` field of the device tree.

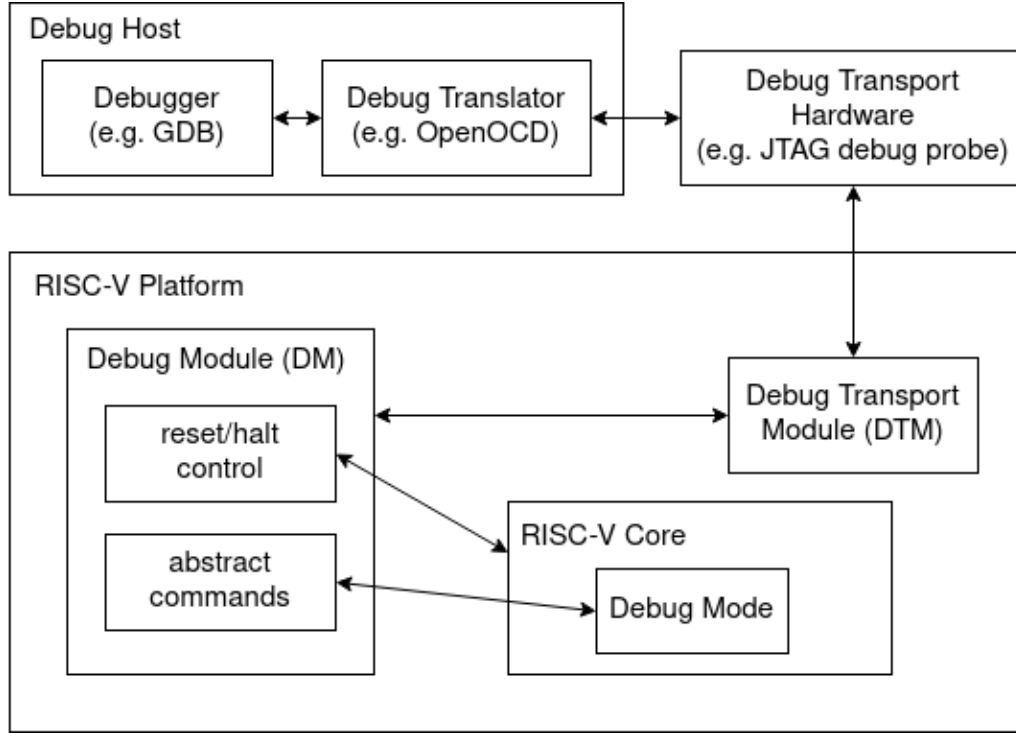


Figure 3: RISC-V Debug Overview (Simplified version of figure 2.1 in [15, p. 6])

2.5 RISC-V Debugging

Running freshly-build software for the first time often reveals unforeseen bugs and other problems. This is even more the case when porting it onto a new platform. Even though the Linux kernel was tested and run successfully on other RISC-V systems, there are still numerous ways it can fail on a new target system. On the one hand, configuration needs to be adapted and can have a large impact on behaviour. On the other hand, hardware bugs or inconsistencies on the *ParaNut* side can cause unexpected behaviour. Especially issues from the second category can often not be detected by the Linux kernel, so the log output might not be useful in this case. That’s why debugging utilities play an important role when porting new software to this platform.

The RISC-V debug specification [15] standardizes the hardware parts of a debugger implementation and gives advice on how to implement the software parts that interact with it. Figure 3 shows an overview of the different modules involved in a debugging progress. The software part runs on the debug host in the form of a debugger and a debug translator. These parts need to communicate with the hardware through some kind of transport hardware like a JTAG debug probe. The hardware side of this interface is the Debug Transport Module (DTM) which passes the commands to a Debug Module (DM). This is the most relevant, as well as implementation-specific item of the debugging architecture as it takes care of controlling and modifying the actual processor core.

The RISC-V debug specification defines an additional privilege mode (the Debug Mode) when the hart is halted for external debugging. During this time, the debugger can use two different ways to access the processor, but only one needs to be implemented by the hardware. The first one is "Abstract Command Based". In this case, the execution of the processor is halted and registers or memory locations are accessed through specific commands sent by the debugger. The alternative is "Execution Based", where the hart jumps to a predefined memory location provided by the Debug Module when entering Debug Mode. The hart then repeatedly polls for updates from the DM. This can either be a command to resume execution or to jump to a program buffer. The program buffer can be filled with arbitrary instructions from the debugger or the debug module and thus allows to access all memory and registers from the view of the hart. Debug Mode can be entered either by executing an `ebreak` instruction, which was placed at a certain address by the debugger in the form of a breakpoint. Alternatively, the debugger can activate single stepping where the Debug Mode is entered after every instruction that was executed. To leave Debug Mode, the hart should execute the `dret` instruction, although this is inserted by the Debug Module and not the debugger directly.

The debug specification adds additional CSRs that must be implemented:

- `dcsr` : Flags to control entering and the execution of the Debug Mode.
- `dpc` : The program counter before entering debug mode.

The Debug Module is required to implement a defined set of operations, but can also be extended with some optional features. The debugger must be able to halt and resume a hart, get information on the implementation as well as on the hart status, read and write general purpose registers and be able to reset the hart for debugging from the first instruction. The Debug Module has to provide at least one of the following three methods to access the hart's memory: The first is using the functionality to execute arbitrary instructions from the program buffer as explained above, the second is to access memory through the abstract commands and the third is direct access to the system bus. Further features are defined, but not relevant for this work.

There are three abstract commands defined, although only the first one is mandatory to implement:

- **Access Register:** Allows to read and write registers, most importantly, the program buffer if supported. The command also contains a flag to start execution of the program buffer.
- **Quick Access:** Halt the hart, execute a program buffer and then resume the hart.

- **Access Memory:** Allows to directly access memory.

The Debug Module provides over 30 registers to the debugger over the Debug Transport Module. Those include locations for the abstract commands and their data, the program buffer as well as information on the DM and hart states. Additional extensions like authentication or the system bus access can also be controlled through these registers.

The Debug Transport Module (DTM) controls the debug interface of the system. It can use USB or JTAG for example. The DTM then uses the Debug Module Interface to command the Debug Module. The specification supports any number DTMs as well as DMs to work together. For using JTAG, the specification further defines the exact set of JTAG registers as well as the pinout of recommended connectors, although more detail is not relevant for this work.

The next part of the debugging chain is the Debug Translator. It works closely together with the Debugger and translates the commands to the underlying protocol (e.g. JTAG). It also manages most of the platform-specific features like waiting for commands to finish, catching errors or accessing memory through the supported method. The translator first needs to probe for available functionality. This happens partly through executing abstract commands and waiting for an error response or through writing into registers and reading back if the modifications persisted. The general available commands include accessing registers, reading memory and writing memory. Those actions should be enough for a working debugging setup.

Lastly, the debugger provides some kind of user interface to allow for simpler interaction. It can for example provide means to display the currently executed instructions or the registers and it can provide actions for the user to place breakpoints and step through the program.

In the case of the *ParaNut*, in accordance with figure 3 from above, the following components for debugging are implemented or used:

- The Debug Module supports only the first Access Register abstract command. To provide further access, the "Execution Based" approach using the program buffer is implemented.
- The DTM supports a JTAG-based interface. For the simulation, the `remote_bitbang` driver is used on top to be able to send the JTAG data over a network socket interface.
- As a Debug Translator, OpenOCD is used. OpenOCD is available for numerous

platforms and architectures and provides multiple interfaces like a TCL scripting interface and a GDB server.

- For debugging, the well-known GDB client can be employed. It connects to OpenOCD and supports a large amount of commands for debugging. To provide further information on the program, it can load the ELF file of the executed binary to load its debugging information.

3 Linux on the ParaNut

The different components that were introduced in the previous chapters must now be built, integrated and potentially adapted to be able to run the Linux Kernel on the *ParaNut* RISC-V processor. The Linux boot process on the *ParaNut* works as follows:

1. *ParaNut* loads the binary and starts executing OpenSBI.
2. OpenSBI sets up the hardware, checks for available features, prepares registers and switches to S-mode.
3. It jumps to Linux, which further prepares the hardware, sets up available memory and loads required device drivers.
4. Once Linux is fully prepared it can pass execution to a U-mode `init` process that eventually forks all future processes that should run on the operating system.

During the OpenSBI and Linux boot stages, the debugger combination GDB + OpenOCD can be used to find issues in soft- and hardware.

3.1 Preparing the Kernel

This chapter will describe requirements for a working Linux Kernel binary to run on the *ParaNut*. Taking the basic knowledge of the Linux boot process into account, the following is a selection of features in addition to the base instruction set which must be available on a RISC-V processor core to be able to boot Linux:

- Exceptions and interrupts must be supported. An interrupt controller must be available.
- Virtual memory must be available to be able to separate the different user processes.
- A timer device has to be present. It is used for timekeeping and regular interrupts.
- Atomic instructions must be supported for providing synchronization primitives.

The first three bullet points are natively supported by the *ParaNut* processor, even though there are some more steps necessary in OpenSBI to expose the timer as expected by the kernel (see the chapter below). The fourth step is only partially supported as explained

in chapter 2.2 *The ParaNut*. The `amo*` instructions thus need to be emulated by the machine level software (see OpenSBI chapter below).

3.1.1 Configuration Options

The following steps are necessary to prepare the Linux kernel for the *ParaNut*. First, the kernel has to be downloaded from <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git>. The latest version that was tested has the tag `v5.18.9`. Additionally, a RISC-V Linux compiler toolchain has to be installed. A pre-built version can be downloaded from [2]. For all of the following `make` commands, the environment variable `CROSS_COMPILE` has to point to `<toolchain folder>/bin/riscv32-linux-`. Once the toolchain is installed, a minimal configuration can be created by running `make ARCH=riscv tinyconfig` in the Linux folder. The following configuration options then need to be adjusted (using `make ARCH=riscv menuconfig`):

Option	Meaning	State
<code>MMU</code>	Enable MMU	Enable
<code>ARCH_RV32I</code>	Use RV32I ISA	Select
<code>RISCV_ISA_C</code>	Emit compressed instructions	Disable
<code>FPU</code>	FPU support	Disable
<code>RISCV_SBI_V01</code>	Legacy SBI support	Enable
<code>EFI</code>	UEFI support	Disable
<code>TTY</code>	Enable TTY	Enable
<code>HVC_RISCV_SBI</code>	SBI Console Support	Enable
<code>SERIAL_EARLYCON_RISCV_SBI</code>	Early console using SBI	Enable
<code>PRINTK</code>	printk support	Enable
<code>PRINTK_TIME</code>	Display time in log messages	Enable

This enables the RV32I ISA and the MMU, disables the C extension and FPU which are unsupported by the *ParaNut*. The legacy SBI, TTY and the two SBI console support options are required to be able to print console messages over the SBI interface. The early console is used to log messages at a time where drivers are normally not yet loaded. The two `printk` options are used to configure the logging in general. An optional option is `DEBUG_INFO_DWARF_TOOLCHAIN_DEFAULT`, which allows to use the resulting ELF file in the debugger by providing additional information. After setting the different options, the kernel can be built using `make ARCH=riscv CROSS_COMPILE=...`. The resulting pure binary image can be found in `arch/riscv/boot/Image`, while the ELF file with debugging data is in the main folder as `vmlinux`.

3.1.2 The Device Tree

To provide the kernel with a device tree, the device tree compiler at `scripts/dtc/dtc` can be used. As explained above, the device tree is necessary to inform the kernel and the bootloader of the different parts of the hardware it runs on. As the name suggests, it is a tree-like structure. Most leaf nodes contain a `compatible` field, which is used by the kernel to find the correct driver. The device tree is located at `sw/linux/paranut.dts`. The following paragraphs will present its content step by step:

```
1 #address-cells = <1>;
2 #size-cells = <1>;
```

At first, the two given variables are set to a default value of 1. This means that any register definition of the device tree which doesn't have those settings set, contains a single address and a single size. The `compatible` string was set as "hsa-ees,paranut" to identify our device.

```
1 cpus {
2     timebase-frequency = <25000>;
3     #address-cells = <1>;
4     #size-cells = <0>;
5     cpu@0 {
6         compatible = "hsa-ees,paranut", "riscv";
7         device_type = "cpu";
8         reg = <0>;
9         riscv,isa = "rv32im";
10        mmu-type = "riscv,sv32";
11        cpu0_intc: interrupt-controller {
12            #interrupt-cells = <1>;
13            compatible = "riscv,cpu-intc";
14            interrupt-controller;
15        };
16    };
17 };
```

The CPU configuration first defines the frequency of the implemented timer, 25kHz in this case. The address and size fields always need to be in the configuration for CPUs as the `reg` field contains only the CPU id with no size. The CPU is again identified by "hsa-ees,paranut", but also as "riscv" compatible. Furthermore, the implemented ISA (rv32im), the MMU variant (sv32) as well the interrupt controller are defined.

```

1 memory@10000000 {
2     device_type = "memory";
3     reg = <0x10000000 0x10000000>;
4 };

```

The memory node contains the address and size of the configured memory, which in this case is 256 MB of size.

```

1 chosen {
2     stdout-path = "/serial0";
3     bootargs = "console=hvc0 earlycon=sbi";
4 };
5 serial@0 {
6     compatible = "hsa-ees,pn-tohost";
7 };

```

The `chosen` node configures the standard output device, where `serial0` is used. `serial0` is defined at the bottom as the *ParaNut*-specific output mechanism `hsa-ees,pn-tohost`, which is the "tohost" interface already explained above. Note that this is only used by the bootloader and not the Linux kernel. The address of the output symbol resides in bootloader memory and the kernel should not access this section. Additionally, the Linux boot arguments are provided here to instruct the kernel to use the SBI driver for log output. This means that OpenSBI is called through an environment call to print out text.

```

1 timer@20000000 {
2     compatible = "riscv,aclint-mtimer";
3     #address-cells = <2>;
4     #size-cells = <2>;
5     reg = <0x20000000 0x8 0x20000008 0x8>;
6     interrupts-extended = <&cpu_intc 7>;
7 };

```

This section defines the timer, which is located at `0x20000000`. Two addresses and two sizes of the memory mapped registers are given here. The first part of the `reg` field points to the two register halves `mtime` and `mtimeh` which together form the full 64bit (8 byte) time value. The second part references the `mtimecmp` and `mtimecmph` register halves used to define a time at which the mtimer module should fire an interrupt. Lastly, the interrupt source is defined. The timer interrupt has the ID 7 and is handled by the interrupt controller that was defined in the CPU section.

3.2 Preparing OpenSBI

As identified by the previous sections, some kind of bootloader as well as an implementation of the SBI interface is required to successfully run Linux. The following features are required by OpenSBI to reach this goal:

- Support for the RV32IMA ISA.
- Availability of S-mode on at least one hart.
- The trap vector CSR `mtvec` must support direct mode, which means that all exceptions jump to the same address.

The first requirements is only partially supported as the atomic extension is not fully implemented. The necessary steps to fix this are described below. The S-mode is available and `mtvec` supports direct mode.

3.2.1 Disabling CoPUs

In OpenSBI, a much larger number of modifications is necessary than in the Linux kernel. This is also caused by the fact that it was tried to keep changes away from the Linux kernel and abstract them into the more platform-specific bootloader. The first necessary change is caused by the CoPUs of the *ParaNut*. All cores of the *ParaNut* start their execution on system reset. Although normal core execution would be no problem for the boot loader as it would select a single core for booting automatically, in the case of the *ParaNut*, only the main core has all CSRs implemented. This would lead to numerous problems during the start process. That's why the CoPUs are disabled in the first lines of the entry point function, which is located in `firmware/fw_base.S`:

```
1 csrr t0, 0xcd4 // Read current hart id
2 _wait_copu:
3 bnez t0, _wait_copu
4
5 csrwr 0x8c1, 0x1 // Only enable the CePU
```

3.2.2 Atomic Instructions

As noted above, another problem was the missing support for the `amo*` instructions in the *ParaNut*. All occurrences of those had to be replaced with the functionally equivalent `lr` / `sc` instructions. Luckily, OpenSBI only contained a few spots that needed action

to be taken. The following is an example of an `amoadd` instruction that was replaced (with changed register names):

```
1 amoadd.w a1, a2, (a3)
```

This instruction loads a word from the memory address contained in `a3`, stores this value into `a1` adds the content of `a2` to it and writes the new value back to the memory address from `a3`. The replacement code looks as follows (annotated with comments):

```
1 0:
2  lr.w a1, (a3)      // Load word into a1
3  add t0, a1, a2      // Store sum in t0
4  sc.w t0, t0, (a3)   // Store word back into memory
5  bnez t0, 0b         // Retry if not successful
```

As there are multiple instructions involved, the atomicity of the instructions might be violated. The `sc` instruction returns a result for this reason. The `bnez` instruction jumps back to the start as often as needed until the store is successful. To replace other `amo*` instructions, the `add` instruction needs to be exchanged. Luckily, most of the atomic instructions were contained in the `riscv_atomic.c` file in `lib/sbi/`. All helper functions in this file were extended with a `lr / sc` variant. Additionally, a single atomic instruction was located in the startup code (at `firmware/fw_base.S`) of OpenSBI and had to be replaced similarly. Another occurrence was found and replaced in `lib/sbi/riscv_locks.c`.

As explained above, the Linux kernel also requires support for those atomic instructions. This problem is solved by emulating them in OpenSBI. If an `amo*` instruction would be executed, the *ParaNut* throws an "Illegal Instruction Exception", which is captured by the OpenSBI trap handler. OpenSBI already had support for emulating missing instructions, so adding emulation for the atomic instructions was as simple as adding a new function and putting it into an array. The new function can be found in `lib/sbi/sbi_illegal_insn.c`. In this function, the expected operation and the input registers are read. Then the value is loaded from memory. Afterwards, the operation is executed before the value is stored back into memory. Even though this sounds quite simple, a major problem arised once the virtual memory was enabled. The addresses of the instruction and the memory location are virtual while the trap handler runs in M-mode on physical addresses. To solve this, RISC-V provides the `MPRV` flag in the `mstatus` register. If this flag is set, all load and store memory accesses should be performed as if the current mode is equal to before the exception. This old mode is stored in the `MPP` (previous privilege) field of `mstatus`. Instruction fetches are still to be performed without virtual memory active. Details on the implementation are provided in the *ParaNut*

chapter below. Once this flag was supported, all memory accesses were performed in three steps: Enable `MPRV`, access memory, disable `MPRV`. This allows the Linux kernel to use all atomic instruction as if they were natively supported.

3.2.3 The Serial Interface

Next is the "tohost" interface. As explained it is used on the *ParaNut* to print to the console. To support this, a serial driver was implemented into OpenSBI. This driver is matched when a serial device with the `compatible` string of "hsa-ees,pn-tohost" was found. It implements the `putc` and `getc` functions, which are implemented by writing to and reading from the symbols `pn_tohost` and `pn_fromhost`. Previously, those symbols were just called "tohost" and "fromhost", although this led to a collision with the more complicated HTIF (Host Target Interface) protocol, which is implemented by for example the SPIKE RISC-V simulator or QEMU [11]. The solution of renaming the symbols was deemed the simplest, although implementing the HTIF pseudo-standard would also be a possibility.

3.2.4 Time Registers

The last aspect that needs to be managed by OpenSBI for the Linux kernel to function properly is the `time` CSR. This CSR is available to all privilege modes and can be read using the `rdtime` pseudo-instruction. The time is always 64bit long and thus needs to be implemented with two registers in 32-bit RISC-V implementations. Those two are called `time` and `timeh`. These registers should shadow the memory-mapped `mtime` and `mtimeh` registers. As per the implementation hints in the specification, two implementation variants are possible [20, p. 36]. Either the processor should translate an access to a load of the memory-mapped registers, or the M-mode software, which is OpenSBI in this case, should emulate the CSR read. When Linux tries to access the unimplemented register, the *ParaNut* throws an "Illegal Instruction Exception". This exception can be caught just like the atomic instructions and emulated by OpenSBI. Luckily, OpenSBI already supports the emulation of the `time` registers by accessing `mtime`, so this variant was chosen. The only configuration that was necessary was to define the timer module in the device tree as explained above.

3.2.5 Configuration and Compilation

After OpenSBI now supports all necessary features to run itself and start the Linux kernel, it must be configured and compiled correctly. For a running operating system, an image consisting of multiple components must be created. These are the Bootloader, the Kernel,

the device tree as well as a root file system where the user space executables and other files are located. Since the project is limited on the basic kernel booting, a file system is not necessary at this point. The OpenSBI build infrastructure directly offers different possibilities to pass execution to next booting stage (e.g. the kernel):

- **Dynamic:** A possible previous boot stage would give information to OpenSBI on the next stage to boot.
- **Jump:** OpenSBI is configured with a fixed address where the execution jumps to after the bootloader has completed its steps. The image has to be included with a different mechanism beforehand.
- **Payload:** The image is included in the OpenSBI image. It is also possible to bundle the device tree so that the result is a single combined binary to be executed by the *ParaNut*.

All settings can be configured by using `Makefile` parameters. The following settings were applied for this project:

- `FW_DYNAMIC=n` `FW_JUMP=n` `FW_PAYLOAD=y` : Enable the payload variant.
- `FW_TEXT_ADDR` : Address of the OpenSBI image at run time (0x10000000 in case of the *ParaNut*)
- `FW_TEXT_START=0x10000000` : Start of the text section
- `FW_PIC=n` : Position independent code would require some kind of loader to initialize the GOT for example. The GOT is the global offset table, which contains information on where symbols are actually located. Since in the *ParaNut* case, OpenSBI is the first loader, the addresses must be known at compile time.
- `FW_FDT_PATH` : Path to the flattened device tree, which was compiled by the device tree compiler as explained above
- `FW_PAYLOAD_PATH` : Path to the Linux image, which is located in the Linux directory at `arch/riscv/boot/Image`

Additionally to these firmware settings, a few more settings regarding the platform are necessary:

- `PLATFORM=generic` : Choose the generic platform variant (see the OpenSBI

background chapter for explanation)

- `PLATFORM_RISCV_ABI=ilp32` `PLATFORM_RISCV_ISA=rv32ima`
`PLATFORM_RISCV_XLEN=32` : Used for setting the correct compiler options

Finally, the `CROSS_COMPILE` and `ARCH=riscv` variables have to be passed for compilation just like for compiling the Linux kernel. After running `make`, the output can be found in `build/platform/generic/firmware/fw_payload.elf`.

3.3 Adaptations of the ParaNut Hardware

In parallel to the efforts taken to configure Linux and OpenSBI correctly, numerous changes were also necessary in the *ParaNut* hardware. A finished implementation of virtual memory as described in [13] is used as a basis here. There were four major categories which needed tweaking. First, the Virtual Memory Management had some issues that needed to be fixed and a new feature had to be implemented. Secondly, the Debugger needed to be adapted to the new privilege modes and virtual memory. Additionally, some modifications to the flashing process were necessary to run Linux on the FPGA with the *ParaNut*. As a bit more unrelated topic, the CSR module was refactored to provide a cleaner interface to the EXU.

3.3.1 Virtual Memory Management

The first set of issues happened on enabling virtual memory. This happens in the Linux kernel by setting the `satp` CSR, running `sfence.vma` as well as setting the trap vector to virtual address of the next instruction. When execution continues, the following address is still physical and thus a "Page Fault Exception" is thrown. This exception jumps to the trap vector where the execution continues with virtual addresses. The following issues were discovered and fixed in cooperation with the project described in [13] while testing the Linux kernel boot process:

- **Issue:** The *ParaNut* hangs after running the `sfence.vma` instruction, the Memory Management Unit (MMU) does not respond at all.
Cause: There were some issues with building physical addresses which led to a loop in certain cases.
- **Issue:** Jumping from OpenSBI to Linux prevented the processor from continuing its execution.
Cause: When enabling S-mode but not yet virtual memory, the default values of

the access control bits, which normally come from the page tables, were considered.

Further issues were discovered during the execution with virtual memory enabled:

- **Issue:** A store word instruction does not always work.
Cause: The `bitselect` signal, which defines the size of the written data, was not set correctly when the MMU was active.
- **Issue:** The load reserved/store conditional instructions never returned success.
Cause: On one hand, the `lr` reservation was not set in all cases, on the other hand `sc` didn't write its result correctly even though the transaction was successful.
- **Issue:** Exceptions after the initial page fault were no longer handled correctly.
Cause: *ParaNut* had an "inside exception" signal, which was never reset as the "trap handler" did never return. This prevented nested exceptions from working correctly.

As described in the OpenSBI chapter above, `MPRV` support had to be implemented. When this bit is set, loads and stores should behave as if the old privilege mode from the `MPP` field is active, while instruction fetches still depend on the current mode. The `MPP` field is set to the previous privilege mode for example when entering an exception handler. To implement this into the *ParaNut*, the paging mode was separated into two signals instead of one. The Load Store Unit (LSU) and the Instruction Fetch Unit (IFU) got each their own signals. The LSU signal is implemented as follows:

```
1 sc_uint<2> effective_mode =  
2     csr_mstatus_MPRV.read() ? csr_mstatus_MPP.read() : priv_mode;  
3 exu_lsu_paging_mode = csr_satp_mode.read () && effective_mode != Machine;
```

The effective mode now either is the current privilege mode or the contents of `MPP`. Paging is enabled if `satp` is configured and the effective mode is not the M-mode. The IFU then simply uses the current privilege mode instead of the effective mode:

```
1 exu_ifu_paging_mode = csr_satp_mode.read () && priv_mode != Machine;
```

This newly implemented features proved as a great stress test for the memory management as the mode is constantly switched. Instruction fetches directly access physical addresses while loads and stores require the translation of virtual addresses. Some minor bugs were discovered like that, including an issue where signals were assigned too late to handle this.

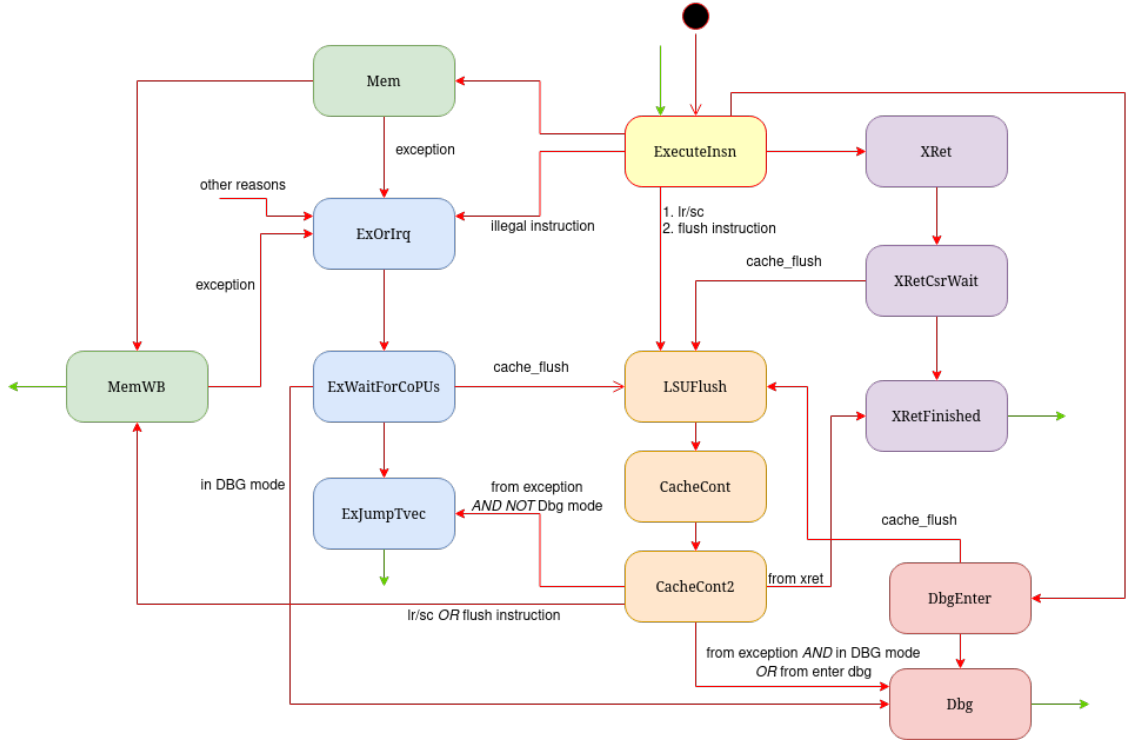


Figure 4: State diagram of the EXU states important for flushing

3.3.2 Debugging

To still allow debugging, while a lower privilege mode or virtual memory is active, some modifications were necessary. When entering debug mode, the processor should automatically switch to M-mode and store its old privilege in the `prv` field of the `dcsr` register. When leaving debug mode, the `dret` instruction is executed, switching the privilege mode back to the stored value. As this switching of the privilege mode can also change the state of virtual memory, flushing of IFU, LSU and the cache may be required on entering and leaving debug mode. To improve performance, the EXU first probes the CSR module, if a flush is actually necessary. This is decided by checking if the paging mode actually changed. If necessary, the EXU performs the required flushes and sets an acknowledge bit for the CSR module. Only if that is set, the CSR module updates all the different CSRs to reflect the new state. As such paging mode changes can also happen when entering or leaving an exception or interrupt handler, this logic was generalized. The CSR module is called on entering an exception handler or debug mode as well as on the different `ret` instructions (`mret`, `sret`, `dret`). It returns, if a flush must be performed and only updates the different exception or debug CSRs once the acknowledge signal is high. To handle these different situations properly, the EXU had to be adapted.

Figure 4 shows the EXU states involved in the different flushing variants. All instructions start the `ExecuteInsn` state. When entering debug mode, it switches to `DbgEnter`.

This state requests entering the debug mode from the CSR module and either switches to `LSUFlush` if a full flush is necessary (i.e. the paging mode has changed) or directly to `Dbg` if it's not. When an exception or interrupt occurs from any source, the `ExOrIrq` state is entered. It waits for the CoPUs to halt in the `ExWaitForCoPUs` state and then also requests from the CSR module if a flush is necessary. If yes, it switches to `LSUFlush`, if no and the debug mode or single stepping is currently active, the `Dbg` state is entered (and the debugger is invoked). Else, the `ExJumpTvec` state is entered and passes execution to the current trap handler. When leaving debug mode or returning from an exception, the `XRet` state is entered, which again probes the CSR module for the need of a cache flush and moves on to `XRetCsrWait`, which either goes to `LSUFlush` or to `XretFinished` depending on the response. All these three possibilities to enter the flushing state depend on the CSR response. The final state of these chains then always sets the acknowledge bit to actually perform the respective actions in the CSR module. Two further reasons to enter the `LSUFlush` state are the load reserved/store conditional as well as the *ParaNut* specific cache flush instructions. Depending on their exact form, those don't necessarily require a full cache flush, but they can give parameters to the respective modules. Once the `LSUFlush` state is active, it clears both the IFU and LSU caches and then continues to `CacheCont`. This is a waiting state for one clock cycle so the recent changes can fully propagate. `CacheCont2` then finally commands a potentially full cache flush. It waits for it to finish and then passes execution back to the correct state depending on where it previously came from. Although this change was initiated by the intention of debugging, it led to cleaner and more deterministic situation for paging mode changes and flushing states.

There were also some smaller problems with debugging which could be fixed very quickly: The simulator showed sudden non-deterministic crashes when connecting OpenOCD to the simulator. This was caused by OpenOCD commanding a reset while the RAM was still working on an action. The *ParaNut* tried to read a response and failed, which ended the simulator. The solution was to ignore issues in that case and just print a warning, as the value would not be used anyway. Additionally, OpenOCD started to crash regularly at a certain point. An update to the newer 0.11 version solved this problem.

3.3.3 CSR Refactoring

Although not necessary for Linux to work, the CSR module of the *ParaNut* was refactored to provide a simpler and cleaner implementation. Previously, the CSR module was separate from the EXU module. Due to the EXU needing access to most of the CSR's signals (over 50), they had to be routed between EXU and CSR. When adding a new signal, it needed to be added to the CSR header and source, to the EXU header and source as well as to the connection parts of the `nut.cpp` and `nut.h`.

To improve on this architecture, the CSR methods are now technically part of the EXU module. To still provide a cleaner development experience, they are placed in a separate file called `exu_csr.inc.cpp` which is included at the bottom of the `exu.cpp` file. All CSR signals that were previously in `csr.h` and most of them duplicated in `exu.h` are now moved to `exu_csr.inc.h` which is included in the EXU header. The `CSRMethod` of the EXU, which previously took care of outputting values to the CSR module is no longer necessary. All EXU methods can now directly access the same variables as the CSR methods. Additionally, all sensitivity lists were checked and missing signals were filled in. As a side-effect, an issue was also found in the performance counter support, where the wrong value was filled into a CSR due to a copy-paste error. Some registers were readable, but not writable even though they should have been and some registers were not initialized correctly. Those issues were also fixed.

The resulting CSR file is still separated into a few different methods. The `PerfcountMethod`, if enabled by the config, takes care of increasing the different implemented performance counters, like for example the cycle count `mcycle`. `HandleMethod`, `ReadMethod` and `WriteMethod` form together the core of the CSRs. On read, the combinatorial read method directly outputs the current value of the selected CSR. On write, the value is either written directly or the read value is modified according to the given bitmask and then written afterwards. The `csrrs` and `csrrc` instructions allow to set or clear certain bits by providing this mentioned bitmask. The bitmask handling itself as well as a privilege check is performed by the `HandleMethod` before setting the respective signals to command a write. The `WriteMethod`, which is clock-synchronous, then modifies the selected CSR as commanded. Both the read and write methods first differentiate if they are part of the CePU or a CoPU. Different sets of registers are available for each of them. They then each have a large switch-case statement to select between the different registers. Those registers can either hold a constant value, are represented by a single 32-bit register or are constructed through a combination of constant bits and signals of smaller width. The write method has an additional part for handling exceptions and privilege mode changes as explained in the Debugging chapter above, which updates the necessary registers as requested by the EXU. Finally, the `OutputMethod` takes care of setting different `m3` signals used to interact with the *ParaNut* CoPUs.

3.3.4 Running on the FPGA

Most tests and development efforts were performed using the *ParaNut* simulator. To further test the boot process and to be able to demonstrate it in a reasonable time, it was further tested on the FPGA hardware. Most hardware parts are covered by Christian Meyer's project work [13], although one aspect is going to be covered in the following

paragraphs. The output ELF binary of Linux has a size of around 5 MB. When producing a binary blob that can be flashed, the size increases to around 25 MB due to empty space between the different sections. Combining Linux with OpenSBI doesn't change the size much as it is quite small. An inconvenience that was discovered when trying to upload the binary to the board over UART is that the upload of a binary of this size took around half an hour, which is quite long when repeated testing is required. Different solutions were possible:

- Increasing the UART Baud rate: This solution was not really possible as the firmware couldn't keep up with receiving data at a higher rate.
- Putting the binary on an SD card: Although this a possible solution and a great addition for the future, it was not chosen due to the complexity of understanding this interface in the scope of this project work.
- Compressing the binary: As this can speed up the upload process with manageable effort, it was chosen.

As the binary contains large amounts of empty space and also other repetitive content, including equal instructions and more it is compressed before uploading. The firmware then takes care of decompressing it on the board. Through the compression with `gzip`, the binary was shrunk to around 700 kB which now happens automatically through the Makefile if necessary. When flashing with the `pn-flash` tool, if a `gzip` file is detected, a new `compressed` flag is set before the data is transferred. The firmware then uses the small `uzlib` library for decompression. It can handle stream decompression to receive the data byte-by-byte and also output the decompressed data byte-by-byte. Once a decompressed byte is produced it is directly send to the Xilinx memory region, where the program should be stored. This means that almost no additional memory is used, except for a smaller decompression dictionary. A problem that occurred after this features was finished, was the decompression stopping at random points. As the binary contains a lot of zeroes at fixed points of the data, the decompression had to produce a lot of output while only consuming little input. This led to overflowing of the receive buffer of the UART interface as it is only 64 byte in size. The solution was to implement some kind of speed control. The `pn-flash` tool now only sends packets with 64 bytes and expects an acknowledge response to continue. With this mechanism and the compression itself, the upload speed could be improved from 30 minutes to around 3 minutes for this 25 MB Linux + OpenSBI image.

4 Testing Linux on the ParaNut

4.1 Running on the Simulator and Hardware

A pre-configured setup for compiling and configuring Linux, OpenSBI as well as the device tree can be found in `sw/linux`. The included Makefile automatically clones the Linux Kernel version 5.18.9, OpenSBI version 1.0 and an additional compiler toolchain for Linux. It then applies the prepared Linux configuration and patches OpenSBI with the necessary changes that were introduced in previous chapters. To be able to easily adapt to the current *ParaNut* config, some values like the memory address and size as well as the timer address and frequency are computed based on the *ParaNut*'s `config.mk` file and inserted into the device tree template. Once the device tree compiler of Linux is ready, the *ParaNut* device tree is compiled. Finally, Linux and OpenSBI are both compiled and integrated to form a resulting binary. The Makefile infrastructure also offers targets for compressing the binary, running it in the simulator or flashing it to the FPGA board. In addition to the production of the binary, the *ParaNut* needs to be configured correctly. The following configuration options are relevant for the execution:

- `CFG_NUT_RESET_ADDR` has to be equal to the compilation variables of OpenSBI (e.g. `0x10000000`).
- `CFG_NUT_MEM_SIZE` has to be equal to the memory size given in the device tree (in this case 256MB).
- `CFG_NUT_MTIMER_ADDR` has to be equal to the timer memory address given in the device tree.
- `CFG_NUT_MTIMER_TIMEBASE_US` has to be configured correctly in relation to the given timer frequency in the device tree: $\text{Timer frequency} = \text{Clock frequency} / \text{Timebase}$.
- `CFG_NUT_CPU_CORES_LD` can be set to zero to remove all CoPUs and speed up the simulation.
- `CFG_EXU_M_EXTENSION` and `CFG_EXU_A_EXTENSION` have to be set.
- `CFG_PRIV_LEVELS` must be set to three.
- `CFG_MMU_TLB_ENABLE` can be enabled to speed up the execution.

4.2 Observed Output

When running Linux in the *ParaNut* simulator or on the FPGA, the following can be observed: First, OpenSBI starts with it's boot logo, which confirms the functioning "tohost" interface:

```
OpenSBI v1.0-1-g401c59a

  ----
 /  _  \          /  _  \  _  _  _  \
|  |  |  |  _  _  _  _  _  _  |  (  _  |  |  )  |  |  | | | | |
|  |  |  |  '  \  /  _  \  '  \  \  _  \  |  _  <  |  |
|  |  |  |  |  )  |  _  /  |  |  |  _  )  |  |  )  |  |  |
 \  _  /  |  _  /  \  _  |  |  |  |  _  /  |  _  /  _  _  |
   |  |
   |  |
```

Afterwards, it begins to detect features, setup drivers and prepare for jumping to the Linux kernel. It prints a summary before doing so:

```
Platform Name           : Generic
Platform Features       : medeleg
Platform IPI Device     : ---
Platform Timer Device   : aclint-mtimer @ 25000Hz
Platform Console Device : paranut
[...]
Firmware Base           : 0x10000000
Firmware Size           : 232 KB
Runtime SBI Version     : 0.3
Domain0 Name            : root
Domain0 Boot HART       : 0
Domain0 HARTs            : 0*
Domain0 Region00        : 0x20000000-0x2000000f (I)
Domain0 Region01        : 0x10000000-0x1003ffff ()
Domain0 Region02        : 0x00000000-0xffffffff (R,W,X)
Domain0 Next Address    : 0x10400000
Domain0 Next Arg1       : 0x12200000
Domain0 Next Mode       : S-mode
Domain0 SysReset        : yes
Boot HART ID            : 0
Boot HART Domain        : root
Boot HART ISA           : rv32imasux
Boot HART Features      : none
```

```

Boot HART PMP Count      : 0
[...]
Boot HART MIDELEG        : 0x00000222
Boot HART MEDELEG        : 0x0000b109

```

One can see numerous detected properties here: The timer as well as the output device were identified. Different memory regions ("Domain0 RegionXX") were separated. The first are memory-mapped registers of the timer and the second is the region of OpenSBI. The "Next" settings contain the Linux entry point as well as a pointer to the device tree. The section ends with some information on the boot hart.

After the execution jumped to the Linux kernel, it runs the first initialization functions before logging can appear. The Linux output looks as follows:

```

[ 0.000000] Linux version 5.12.0-rc8+ (eeslabor@eesvm)
(riscv64-linux-gcc.br_real (Buildroot 2020.08-14-ge5a2a90) 10.2.0,
GNU ld (GNU Binutils) 2.34) #1 Fri Jun 24 17:54:50 CEST 2022
[ 0.000000] OF: fdt: Ignoring memory range 0x10000000 - 0x10400000
[ 0.000000] Machine model: hsa-ees,paranut
[ 0.000000] earlycon: sbi0 at I/O port 0x0 (options '')
[ 0.000000] printk: bootconsole [sbi0] enabled

```

These first initialization steps show that Linux is starting and the SBI serial driver was enabled.

```

[ 0.000000] Zone ranges:
[ 0.000000]   Normal   [mem 0x0000000010400000-0x000000001fffffffff]
[ 0.000000] Movable zone start for each node
[ 0.000000] Early memory node ranges
[ 0.000000]   node    0: [mem 0x0000000010400000-0x000000001fffffffff]
[ 0.000000] Initmem setup node 0 [mem 0x0000000010400000-
0x000000001fffffffff]

```

An early memory setup phase takes place. Execution continues with probing the SBI interface:

```

[ 0.000000] SBI specification v0.3 detected
[ 0.000000] SBI implementation ID=0x1 Version=0x10000
[ 0.000000] SBI v0.2 TIME extension detected
[ 0.000000] SBI v0.2 IPI extension detected
[ 0.000000] SBI v0.2 RFENCE extension detected

```



```
[ 0.000000] riscv: ISA extensions im
[ 0.000000] riscv: ELF capabilities im
[ 0.000000] Built 1 zonelists, mobility grouping on. Total pages: 64008
[ 0.000000] Kernel command line: console=hvc0 earlycon=sbi
[ 0.000000] Dentry cache hash table entries: 32768 (order: 5,
131072 bytes, linear)
[ 0.000000] Inode-cache hash table entries: 16384 (order: 4,
65536 bytes, linear)
[ 0.000000] Sorting __ex_table...
[ 0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[ 0.000000] Memory: 234672K/258048K available (986K kernel code,
8638K rdata, 4096K rodata, 4116K init, 174K bss, 23376K reserved,
0K cma-reserved)
```

The different messages show initialization of caches, exception tables as well as the kernel memory allocator. In the following steps, interrupts as well as the timer device are initialized:

```
[ 0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[ 0.000000] riscv-intc: 32 local interrupts mapped
[ 0.000000] riscv_timer_init_dt: Registering clocksource cpuid [0]
hartid [0]
[ 0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff
max_cycles: 0x179dd7f66, max_idle_ns: 112843571735600 ns
[ 0.000120] sched_clock: 64 bits at 25kHz, resolution 40000ns,
wraps every 140737488340000ns
[ 0.005520] printk: console [hvc
```

As one can see, the last two lines have a timestamp at the beginning. This is a sign that the timer could be read correctly. As the "sched_clock" line shows, the scheduler also was already initialized and is upgraded with a clock device here. Sadly, execution stops in the middle of the "printk" line, which is probably happening during the call to the `console_init` function. The simulator shows accesses to invalid addresses at this point. A possible cause could be problems with the timer interrupt, as interrupts were enabled just before this function call. Further debugging at this point is left for the future.

To summarize, around 50 of the 90 different initialization functions in `start_kernel` in `main.c` were successfully executed, most architecture-specific steps are completed and log output is shown. Furthermore, the SBI interface, the kernel memory allocator,

interrupts as well as the timer driver are already initialized at this point. Future steps include support for forking processes, security mechanisms as well as general process management before forking the init process. To execute Linux until this point on the simulator it takes around two hours while the FPGA hardware takes about ten seconds. Overall, taking the frequency of 25 MHz into account, this gives a cycle count of around 250 million cycles that were executed successfully.

4.3 Debugging Techniques

A large number of very different issues occurred during the development of this project. A combination of multiple debugging techniques was required to identify and solve them successfully. The simplest measure is of course the printing of debug messages. On one hand they can be added to the hardware simulator, where state changes or the execution of unexpected paths can be logged to the console and enriched with further information. This proved useful for some cases of illegal instructions or illegal memory accesses. On the other hand, debug messages can also be implemented into the executed software, although the console mechanism must be already working. Linux output messages and kernel panics for example are immensely useful when trying to find out the location of a problem. A special case of these logging messages is the instruction trace which is implemented into the simulator. It prints out every instruction that was executed as well as the registers that changed during this time. It can be used to find out which branches are taken at a certain point of the program if the exact variable values are not known.

The next debugging technique is the use of an actual debugger like GDB. It can be connected with OpenOCD, which in turn connects to the *ParaNut* simulator as explained before. Additionally, it can be given the elf file that is executed to load debug symbols. Note that debugging symbols must be explicitly enabled in the Linux Kernel for example. When those are active, GDB can show the current line of code, correct function and parameter names and it allows to pretty-print structs and their fields with the simple `print` command. When debugging OpenSBI, the OpenSBI elf binary has to be provided. When debugging Linux, the `vmlinux` elf file must be used as it correctly contains the functions with their virtual addresses at run time. The most common actions taken in the debugger are setting a break point where the execution will halt with e.g. `break main`, continuing execution or single stepping through the program. With paging enabled, single stepping sometimes shows problems which are probably caused by the too-long wait time due to two cache flushes at leaving and entering the debug mode. For this reason, break points can be used more extensively. Addresses, variables or registers can also be printed out by GDB using the `print` or `x` commands (E.g. `print id`, `print $a0` or `x 0x10040000`).

The last, and most detailed method for finding issues in the hardware, is the generation of the signal trace file. All signal changes are logged by the simulator and can be viewed with for example GTKWave. As the simulator's execution slows down largely and the generated file grows quickly in size, it should only be used if other debugging techniques did not lead to a reasonable result. If the resulting file is larger than the available memory on the system, it must be compressed before opening using the `gtkwave -o <file>` command. In the viewer, specific addresses can be searched or status signals relevant to the observed problems should be inspected. For example `dbg` or `priv_mode` are useful for finding the exact position of an issue. When the location was found, further signals can be added to trace the execution step by step and find the problem.

Some of these techniques can also be combined. Execution can be manually interrupted through the debugger after a certain log output was seen on the console or the debugger can be used to identify addresses of possible problems, which can afterwards be searched for in the signal trace.

5 Conclusion

5.1 Summary

This work has introduced a setup and configuration for running the Linux Kernel on the *ParaNut* RISC-V processor. All necessary components were introduced and their procedures were shown. The Linux Kernel needs a bootloader for basic hardware setup, which is implemented by OpenSBI. Both require a device tree, which contains the structure of the hardware they are running on. The processor hardware has to support numerous features, which include privilege modes, atomic instructions as well as virtual memory. Needed modifications as well as configuration changes were done and explained. Different hardware issues were identified and solved and debugging was adapted to work with virtual memory. The Linux Kernel together with OpenSBI was run in the simulator as well as on an FPGA. The boot process has progressed to a stage where most processor-specific features are initialized and working, but there is still some more work to do.

5.2 Known Issues and Next Steps

A few issues are still to be solved, the following are a few aspects that were not yet finished in this project work. Firstly, the Linux boot process has to be debugged further to fix the current problem mentioned above as well as future problems that may arise in the following boot steps. Secondly, Linux has to be equipped with a file system to be able to switch to a simple init process. Later, Linux can be adapted to the *ParaNut* specifics

including the CoPUs in threaded or linked mode. To implement this, the other harts must be able to be started by the Kernel and the context switch has to be modified to save and restore additional states. More information on that can be found in [4]. As mentioned in the "Debugging Techniques" chapter, single stepping is causing some problems which are probably caused by the processor being too slow with responding. If possible, the debug mode entering and leaving must be accelerated. Another useful feature for debugging would be the possibility for the signal trace to start at a certain address. This would remove the large size and time requirements of this feature and allow for more targeted debugging.

References

- [1] *About RISC-V*. RISC-V International. URL: <https://riscv.org/about/> (visited on 07/03/2022).
- [2] *Cross-compilation toolchains for Linux - riscv32-ilp32d toolchains*. URL: https://toolchains.bootlin.com/releases_riscv32-ilp32d.html (visited on 06/27/2022).
- [3] Palmer Dabbelt. *All Aboard, Part 6: Booting a RISC-V Linux Kernel*. SiFive. Oct. 9, 2017. URL: <https://www.sifive.com/blog/all-aboard-part-6-booting-a-risc-v-linux-kernel> (visited on 06/28/2022).
- [4] Palmer Dabbelt. *All Aboard, Part 7: Entering and Exiting the Linux Kernel on RISC-V*. SiFive. Oct. 23, 2017. URL: <https://www.sifive.com/blog/all-aboard-part-7-entering-and-exiting-the-linux-kernel-on-risc-v> (visited on 07/03/2022).
- [5] *Das U-Boot - the Universal Boot Loader*. DENX Software Engineering GmbH. June 30, 2022. URL: <https://www.denx.de/wiki/U-Boot> (visited on 07/02/2022).
- [6] *Distribution of operating systems used for Internet-of-Things (IoT) devices, as of 2016*. Statista. Apr. 2016. URL: <https://www.statista.com/statistics/659581/worldwide-internet-of-things-survey-operating-systems/> (visited on 07/02/2022).
- [7] *Distribution of the 500 most powerful supercomputers worldwide from 2017 to 2021, by operating system*. Statista. Oct. 2021. URL: <https://www.statista.com/statistics/565080/distribution-of-leading-supercomputers-worldwide-by-operating-system-family/> (visited on 07/02/2022).
- [8] Mark D. Hill and Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* 41.7 (2008), pp. 33–38. DOI: 10.1109/MC.2008.209.
- [9] *History of RISC-V*. RISC-V International. URL: <https://riscv.org/about/history> (visited on 07/03/2022).
- [10] Alan Holt and Chi-Yu Huang. “Overview of GNU/Linux”. In: *Embedded Operating Systems*. Springer, 2018, pp. 11–40.
- [11] *How to tohost and fromhost work?* URL: <https://github.com/riscv-software-src/riscv-isa-sim/issues/364> (visited on 07/03/2022).
- [12] Gundolf Kiefer et al. *The ParaNut Processor. Architecture Description and Reference Manual*. Version v1.0.0-gd3eb3c6*. University of Applied Sciences. Nov. 22, 2022.
- [13] Christian H. Meyer. *A Memory Management Unit for the ParaNut*. July 2022.
- [14] *Mobile operating systems’ market share worldwide from January 2012 to January 2022*. Statista. 2022. URL: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/> (visited on 07/02/2022).

- [15] Tim Newsome and Megan Wachs, eds. *RISC-V External Debug Support*. Version 0.13.2. RISC-V International. March 2019.
- [16] *RISC-V Open Source Supervisor Binary Interface*. URL: <https://github.com/riscv-software-src/opensbi> (visited on 06/28/2022).
- [17] *RISC-V Supervisor Binary Interface Specification*. URL: <https://github.com/riscv-non-isa/riscv-sbi-doc/blob/master/riscv-sbi.adoc> (visited on 07/03/2022).
- [18] Chris Simmonds. *Mastering Embedded Linux Programming*. Packt Publishing Ltd, 2017.
- [19] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 20191213. RISC-V Foundation. December 2019.
- [20] Andrew Waterman, Krste Asanović, and John Hauser, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20211203. RISC-V International. December 2021.