Aalto University
School of Electrical
Engineering

Booting Operating Systems on RISC-V Processors

Exploring RISC-V ISA Capabilities for Operating Systems Support

---

Lenni Toikkanen

Bachelor's thesis

2025

Aalto University
School of Electrical
Engineering

Author Lenni Toikkanen

Title Booting Operating Systems on RISC-V Processors — Exploring RISC-V ISA Capabilities for Operating Systems Support

Degree programme Bachelor's Programme in Electrical Engineering

Major Electronics and Electrical Engineering

Supervisor D.Sc., Senior university lecturer Markus Turunen

Advisors M.Sc. (Tech.) Verneri Hirvonen

Date 11.05.2025 **Number of pages** 25 + 2 **Language** English

Abstract

RISC-V is an openly standardized instruction set architecture (ISA) that has recently gained popularity as a free alternative to commercialized ISAs. ISA defines how software controls a processor. An important feature of a processor architecture is the ability to run an operating system (OS). OS is a sophisticated program that manages system resources and provides useful services for user programs. The boot and runtime requirements for an OS are heavily platform dependant.

This work explores the boot process and the general hardware requirements for each individual OS service, and outlines how RISC-V ISA answers these requirements. It was found that the RISC-V ISA extensions A, Zicsr, Zifencei, Zicntr and Zihpm, the privileged architecture's interrupt architecture, machine, supervisor and user privilege modes, and the Sv32-57 page-based virtual memory schemes all play a crucial role in providing functionality required by common OS services. It was also found that the supervisor binary interface (SBI) serves a crucial role in isolating supervisor-level OS execution from machine-mode firmware.

The work also explored the practical hardware requirements of Linux and Zephyr operating systems. For this purpose, minimal hardware projects capable of running either OS were investigated. The findings indicate that Zephyr only requires the Zicsr and Zifencei extensions, support for hardware timer interrupts as well as basic I/O, memory and system bus facilities. Linux requires the M, A, Zicsr and Zifencei ISA extensions, machine, supervisor and user privilege modes, support for software, timer and external interrupts, the SBI interface and a page based MMU on top of the implicit I/O, memory and system bus requirements. The findings may not be the absolute minimum requirements for Linux or Zephyr but should point to the right direction.

Keywords RISC-V, Operating Systems, Instruction Set Architecture, Linux, Zephyr

Aalto-yliopisto
Sähkötekniikan
korkeakoulu

Tekijä Lenni Toikkanen		
Työn nimi Käyttöjärjestelmien käynnistys RISC-V prosessoreilla — Selvitys RISC-V ISA:n ominaisuuksista käyttöjärjestelmien tueksi		
Koulutusohjelma Sähkötekniikan kandidaattiohjelma		
Pääaine Elektroniikka ja Sähkötekniikka		
Työn valvoja TkT, vanhempi yliopiston lehtori Markus Turunen		
Työn ohjaajat DI Verneri Hirvonen		
Päivämäärä 11.05.2025	**Sivumäärä** 25 + 2	**Kieli** englanti

Tiivistelmä

RISC-V on avoimesti standardoitu käskykanta-arkkitehtuuri (ISA), joka on viime aikoina saavuttanut suosiota ilmaisena vaihtoehtona kaupallisille ISA:ille. ISA määrittää, miten ohjelmisto ohjaa prosessoria. Minkä tahansa prosessoriarkkitehtuurin tärkeä ominaisuus on kyky ajaa käyttöjärjestelmää (OS). Käyttöjärjestelmä on ohjelma, joka hallitsee järjestelmäresursseja ja tarjoaa hyödyllisiä palveluita käyttäjäohjelmille. Käyttöjärjestelmän käynnistys- ja ajonaikaiset vaatimukset ovat varsin alustariippuvaisia.

Tässä työssä tutkittiin käynnistysprosessia ja kunkin yksittäisen käyttöjärjestelmäpalvelun yleisiä laitteistovaatimuksia, ja hahmoteltiin, kuinka RISC-V ISA vastaa näihin vaatimuksiin. Työssä havaittiin, että RISC-V ISA-laajennukset A, Zicsr, Zifencei, Zicntr ja Zihpm, privileged-arkkitehtuurin keskeytykset, machine-, supervisor- ja user-oikeustilat, sekä Sv32-57 sivupohjaiset virtuaalimuistijärjestelmät ovat kaikki ratkaisevassa roolissa yleisten käyttöjärjestelmäpalvelujen edellyttämien toimintojen tarjoamisessa. Havaittiin myös, että SBI-rajapinta on tärkeä supervisor-käyttöjärjestelmän ja machine-laiteohjelmiston eristämiseksi toisistaan.

Työssä tutkittiin myös Linux ja Zephyr käyttöjärjestelmien laitteistovaatimuksia tutkimalla minimaalisia laitteistoprojekteja. Tulokset osoittavat, että Zephyr vaatii vain Zicsr- ja Zifencei ISA-laajennukset, tuen laitteistoajastinkeskeytyksille, sekä perus-I/O-, muisti- ja järjestelmäväylätoiminnot. Linux vaatii M-, A-, Zicsr- ja Zifencei ISA-laajennukset, machine-, supervisor- ja user-oikeustilat, tuen ohjelmisto-, ajastin- ja ulkoisiin keskeytyksiin, SBI-rajapinnan ja sivupohjaisen MMU:n, I/O-, muisti- ja järjestelmäväylävaatimusten lisäksi. Tulokset eivät välttämättä ole Linuxin tai Zephyrin ehdottomat vähimmäisvaatimukset, mutta niiden pitäisi osoittaa oikeaan suuntaan.

Avainsanat RISC-V, käyttöjärjestelmät, käskykanta-arkkitehtuuri, Linux, Zephyr

Foreword

I would like to thank my supervisor Verneri Hirvonen for providing assistance and guidance in this thesis project. Their guidance has helped me greatly in narrowing down the scope of the thesis and in forming a clear understanding of the overall topic. Verneri also provided an excellent example project, SERV, which is explored in Section 5 of this work.

I would also like to thank the github user *aarneng* for their Aalto thesis Typst template in github. The template has proved very useful, simplifying the writing and formatting process of the work.

11.05.2025 Lenni Toikkanen

Contents

Definitions & Abbreviations

ABI	Application Binary Interface
AEE	Application Execution Environment
CSR	Control and Status Register
EEI	Execution Environment Interface
GPIO	General-Purpose Input/Output
HBI	Hypervisor Binary Interface
HEE	Hypervisor Execution Environment
Hart	Hardware Thread
I/O	Input/Output
ISA	Instruction Set Architecture
MMU	Memory Management Unit
OS	Operating System
PC	Program Counter
RFS	Root File System
RISC-V	Reduced Instruction Set Computer Five
ROM	Read Only Memory
RVWMO	RISC-V Weak Memory Ordering
SBI	Supervisor Binary Interface
SEE	Supervisor Execution Environment
TLB	Translation Lookaside Buffer

1 Introduction

Reduced Instruction Set Computer Five (RISC-V) is an openly standardized *Instruction Set Architecture* (ISA), initially developed by the University of California, Berkeley in 2011 [1]. RISC-V has recently gained popularity as a promising free alternative to commercialized ISAs like ARM and x86. Many RISC-V hardware implementations are currently in development. Aalto University started their own open-source RISC-V microprocessor project *A-Core* in 2021 [2]. The project has evolved from a simple RISC-V core to a more customizable processor with signal-processing and energy-efficiency capabilities.

An especially important feature of a microprocessor is the ability to boot and run operating systems. *Operating system* (OS) is a sophisticated program that manages system resources and provides useful services for user programs. Typically, an OS provides functionality for program development & execution, access to I/O devices, controlled access to files, system access, error detection and accounting [3]. OS functionality can, however, differ significantly between different kinds of operating systems. As an example, real-time operating systems, targeting embedded devices, have very different features compared to 'regular' operating systems targeting servers and desktop use.

Booting an OS requires setup on both hardware and software level. Physical hardware has to be turned on and initialized by system firmware, which forms the software runtime and booting environment for the OS. The OS kernel has to be moved from secondary memory to main memory and placed into the execution stack of the processor. Finally, the control of the system is passed to the OS. Almost all steps in the OS boot process are implementation specific. To avoid redefining the whole process for each implementation, the ISA and target OS documentation provide specifications and interface abstractions of relevant areas.

This study explores RISC-V ISA features required to boot and run modern operating systems. Since the OS closely controls hardware of the system, the problem has two parts: What hardware facilities does the ISA provide for supporting operating systems? What systems software is involved in the boot process?

In this study, The RISC-V ISA's OS support is examined both by reviewing OS and RISC-V literacy, and by analyzing existing projects enabling operating systems on RISC-V. Existing projects serve as real-world examples of how RISC-V features are utilized to support operating systems. Linux and Zephyr project documentation are also used to outline their respective system requirements. Documentations of Linux and Zephyr OS kernels are credible sources, as they are maintained by the developers themselves who are experts in their field.

Linux and Zephyr RTOS are chosen as the operating systems of interest in this study because of their practical interest to the *A-Core* project and their relevance in the industry. Other relevant operating systems are not explored as they would require separate inspection. ISAs other than RISC-V ISA are also not explored as the study aims to address requirements for only the RISC-V platform and the results cannot be generalized across other platforms.

The research goals of this study are: 1. To outline the what hardware facilities RISC-V provides to support operating systems. 2. To find practical hardware and software requirements for Linux and Zephyr, and to address the minimal set of RISC-V hardware features required to boot and run them. These findings should help the developers of the *A-Core* project to address the requirements of booting Linux and Zephyr, and highlight the key sections of the RISC-V ISA for exploring other operating systems on the *A-Core* processor.

In Section 2, the general OS boot process is outlined and each stage analyzed in detail. Section 3 explores the general functions of an OS and what requirements they pose to the ISA. In Section 4, the RISC-V implementation of these requirements are outlined and explored in detail. Section 5 and Section 6 review existing hardware projects that are able to run Zephyr and Linux operating systems. Finally, Section 7 concludes the findings and answers the research questions.

2 The Boot Process

The OS boot process is a complex combination of software components and hardware interaction. Almost all phases in the boot flow are implementation specific, but the responsibilities of each software component have become more or less standard. The key software components in the boot process are outlined and their most important functions are explained below. Figure 1 shows a typical boot process of an operating system.

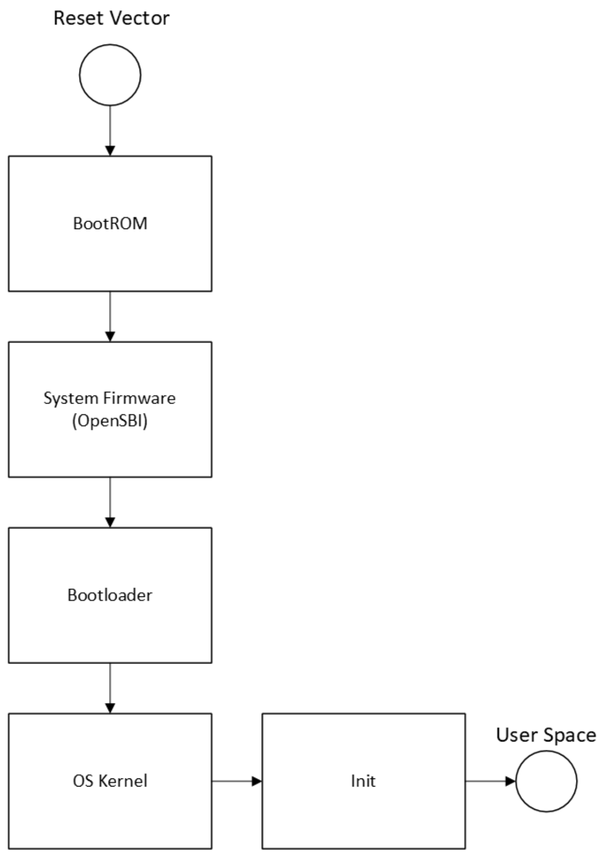

Figure 1: Boot process of an operating system on RISC-V hardware.

2.1 Reset Vector & Boot ROM

The boot process of a system starts from the processor's reset vector. Reset vector is an address that marks the initial memory location where the processor executes its first instruction. The reset vector is implementation specific, but usually it points to the Boot ROM (read only memory) a piece of non-volatile memory implemented separately on the hardware. The Boot ROM contains system firmware - instructions required to complete a basic hardware initialization of the system and load the bootloader into main memory for execution [4].

2.2 Firmware

Firmware is low-level software that controls hardware devices. It is commonly embedded into non-volatile memory directly on hardware. There are two types of firmware: device firmware and system firmware. Device firmware is associated with a hardware device in the system. It is responsible for initializing and controlling the device, and providing an abstraction layer between the OS software and the physical hardware. Device firmware is often programmed directly on the device's non-volatile memory.

System firmware provides boot and runtime services to the OS. It is responsible for initializing the minimum hardware devices and passing execution to the bootloader. Historically system firmware that handles these tasks on x86 architecture was called *basic input/output system* (BIOS). BIOS is a firmware system originally programmed by the company IBM for their first *personal computer* (PC) [5]. The original BIOS firmware was reverse-engineered by several companies and evolved over time into a kind-of-a standard. Today, system manufacturers have ran into issues with BIOS, which was historically limited to 16-bit operation and address space, leading into the development of *unified extensible firmware interface* (UEFI) standard.

UEFI has the same responsibilities as BIOS but handles them in a more standardized and modern way. Operation is 32-bit, 64-bit or 128-bit [6, p. 17], compared to the 16-bit, providing practically infinite possible address space. UEFI is also more modular and is written platform and architecture independently, making it possible to run it on architectures other than x86, such as RISC-V and ARM. Apart from UEFI, there also exist other system firmware specifications for modern architectures.

One such system firmware specification is the RISC-V architecture's *supervisor binary interface* (SBI). SBI defines a software interface between RISC-V SEE and the OS, or between a RISC-V hypervisor and the OS, depending on the implementation stack. The RISC-V Supervisor Binary Interface Specification [7] marks that the SBI allows supervisor-mode software to be portable across all RISC-V implementations. The OS kernel only has to implement support for the SBI interface to support the entire RISC-V platform.

The SBI Specification [7] also states that: "The SBI specification doesn't specify any method for hardware discovery. The supervisor software must rely on the other industry standard hardware discovery methods (i.e. Device Tree or ACPI)" Therefore, it is also required for an OS to provide a separate method for hardware discovery for the RISC-V platform.

2.3 Hardware discovery - Device Trees & ACPI

To allow portability across devices, the OS software has to be platform-independent. This is achieved by providing a hardware discovery interface between hardware devices and the OS. There are two prominent hardware discovery interfaces: *device-trees* and *ACPI tables.*

Devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property - value pairs that describe the characteristics of the device being represented [8]. During the system boot, the "boot program" (firmware/bootloader/hypervisor) loads a devicetree binary into memory space that's accessible to the "client program" (OS/bootloader/hypervisor) and passes a pointer to that memory [8], providing the OS with the required device information.

ACPI is a complex hardware discovery and configuration scheme. It is intended to be used with BIOS/UEFI and provides an interface between the system firmware and the OS for discovering and configuring hardware devices. Both the OS and the system firmware must be ACPI-compliant. In ACPI, hardware information is stored in ACPI Tables which describe the interfaces to the hardware [9]. As ACPI is a complex scheme and requires UEFI/BIOS implemented on the platform, this review will mainly focus on *devicetree* based hardware discovery.

2.4 Bootloader

The bootloader is a small program that calls the OS into main memory after the system is turned on. It is responsible for the initial boot process of the system, and for loading the kernel into main memory [3, p. 607]. A typical bootloader sequence consists of two stages.

The first-stage bootloader is loaded by firmware from secondary memory. It initializes the memory controller and a few peripheral devices and loads the second-stage bootloader into main memory [3, p. 607]. The second-stage bootloader then loads the OS kernel and root file system from secondary memory into main memory [3, p. 607].

2.5 Operating System Kernel & Root File System

Kernel is a piece of OS software that contains the core functionality of the OS. The kernel usually includes a number of separate modules, including: Memory management; Process/Thread management; Inter process communication, timers; Device drivers; File systems; Networking; Power management [3, p. 607].

Most OS kernels require a *root file system* (RFS), which is a basic file system containing files, commands and shared libraries required by the kernel [10]. Some operating systems use a temporary RFS to limit the size of the OS kernel and make it more portable [10]. Temporary root file systems are minimal file systems that get mounted before the final RFS. They provide minimal necessary files, commands and shared libraries for booting the OS kernel. After the OS kernel is booted, the final RFS is mounted to move forward with the boot process.

During the final phases of an OS boot process, the kernel calls a software component commonly referred to as the init system [11]. The init system's responsibility is to initialize the user space of the system. In practice, the init system starts and manages OS services like the window manager, file systems and network manager.

3 Operating System Facilities

Operating systems provide many facilities to the end user. The major facilities are:

- Process Management
- Memory Management
- File System Support
- Device Drivers
- Networking. [10, pp. 16-17]

In this section, the major OS facilities are outlined along with their hardware requirements.

3.1 Process Management

In operating systems, *process* is a unit of activity characterized with a sequential thread of execution, state and a set of associated system resources [3]. On a more technical level, a process consists of program code, associated data and a data structure called the *process control block* which holds the complete state of the process. As described in [3, p. 132], the process control block typically includes the following elements:

- **Identifier:** A unique identifier associated with a process, to distinguish it from all other processes.
- **State:** If a process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** Include pointers to the program code and data associated with a process, plus any memory blocks shared with other processes.
- **Context data:** Data that are present in registers in the processor while the process is executing.
- **I/O status information:** Outstanding I/O requests, I/O devices assigned to this process, a list of files in use by the process, and so on
- **Accounting information:** The amount of processor time and clock time used, time limits, account numbers, and so on.

The significance of the process control block is that it contains sufficient information so it is possible to interrupt a running process and later resume execution as if the interruption had not occurred [3, p. 132]. The process control block enables the OS to support multiple processes and their multiprocessing [3, p. 132].

Typically, operating systems provide the following process management functionality [3, p. 158].

- Process creation and termination
- Process scheduling and dispatching
- Process switching
- Process synchronization and inter-process communication
- Process control block management.

OS Process management is typically implemented by utilizing two common hardware constructs: *control and status registers* (CSRs) and *interrupts*. CSRs are processor registers that are employed to control the operation of the processor. Typically implemented CSRs include [3, p. 153]:
- **Program counter**: Contains the address of the next instruction fetch.
- **Condition codes**: Result of the most recent operations.
- **Status information**: Interrupt enabled/disabled flags, execution mode.

Interrupts are methods by which other modules such as I/O or memory, can interrupt the normal sequencing of the processor [3, p. 36]. Interrupts may be caused by both software or hardware events. Table 1 lists the most common types of interrupts in a computer system.

Table 1: Classes of interrupts. [3, p. 36]

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

Interrupts are defined by the instruction set architecture of the hardware together with the microarchitecture of the computer system. The ISA defines the types of interrupts the processor can handle while the microarchitecture dictates which of them are generated. Interrupts can be generated at runtime by different hardware devices in the system. Interrupts are handled by the operating system's interrupt handler which "does some basic housekeeping" and branches to an OS routine that concerns the particular type of interrupt that has occurred [3, p. 160].

3.2 System Calls & ABI

Operating systems provide user applications with a compatibility interface called *application binary interface* (ABI). The ABI defines a standard for binary portability, and a system call interface to the OS services and hardware resources in the system [3, p. 71]. System calls are means by which processes can request specific kernel services from the OS kernel. System calls can be roughly grouped into six categories: file system, process, scheduling, inter-process communication, socket (networking), and miscellaneous [3, p. 116].

System calls can translate to *supervisor calls* that cause explicit interrupts within the OS [3, p. 160]. These interrupts are handled by the OS's interrupt handler. Supervisor calls are typically defined as instructions in the ISA. Calling the instruction from a user program triggers an interrupt that requests an operation based on passed parameters or register data values [3, p. 160].

3.3 Threads, Synchronization & Atomicity

Process can be thought of as a program under execution. The process can have one or more paths of execution referred to as *threads.* The ability of an OS to support multiple threads within a single process is called multithreading. Each thread holds its own execution state, thread context (independent program counter), execution stack, static storage for local variables and shared access to the memory and resources of its parent process [3, p. 179].

Multithreading is a form of concurrency. As listed in [3, p. 224], other forms of concurrency in operating systems include:

- **Multiprogramming**: The management of multiple processes within a uniprocessor system.
- **Multiprocessing**: The management of multiple processes within a multiprocessor
- **Distributed processing**: The management of multiple processes executing on multiple, distributed computer systems.

Concurrency arises naturally in contexts where an OS manages multiple applications, applications with concurrent processes or threads, or when the OS itself is implemented as a set of processes or threads. Concurrency presents a host of design issues: processor time needs to be allocated between processes, processes and threads need to communicate, share and compete for resources as well as synchronize their activity with others [3, p. 224].

Synchronization refers to mechanisms and rules used to coordinate the execution of concurrent processes and threads, so that the integrity of shared data objects is ensured [10, p. 175]. The basic requirement for supporting concurrency is the ability to enforce mutual exclusion; the ability to exclude other processes or threads from accessing shared resources while one is granted access [3, p. 225]. One common synchronizing tool to enforce mutual exclusion is atomicity.

An atomic sequence of operations is always executed as a indivisible group that cannot be interrupted. Atomicity guarantees operational isolation from other concurrent processes and threads [3, p. 225]. Some operating systems, like Linux, require hardware implementations of some base set of atomic operations to operate [3, p. 316].

3.4 Memory Management

To support modular programming and flexible use of data, the OS has five principal storage management responsibilities [3, p. 87]:

1. **Process isolation**: The OS must prevent independent processes from interfering with each other's memory.
2. **Automatic allocation and management**: Programs should be dynamically and transparently allocated across the memory hierarchy as required.

3. **Support of modular programming**: Programmers should be able to define program modules, and to dynamically create, destroy and edit the size of them.
4. **Protection and access control**: The OS must allow portions of memory to be accessible and inaccessible on various ways depending on the access control scheme.
5. **Long-term storage**: Many application programs require means for storing information for extended periods of time.

Generally, operating systems meet these requirements with *virtual memory* and *file system* facilities. The file system implements a long-term storage with files. A file is a convenient unit of access control and protection within the OS [3, p. 87].

Virtual memory is a facility that allows programs to address memory without regard to the amount of physical memory available [3, pp. 88-89]. It enables multiprogramming by allowing multiple concurrent user jobs in main memory at the same time [3, pp. 88-89]. Virtual memory is usually managed by a hardware unit called the *memory management unit* (MMU). As shown in Figure 2, the MMU translates virtual memory addresses into physical memory addresses on main memory and disk addresses in secondary memory.

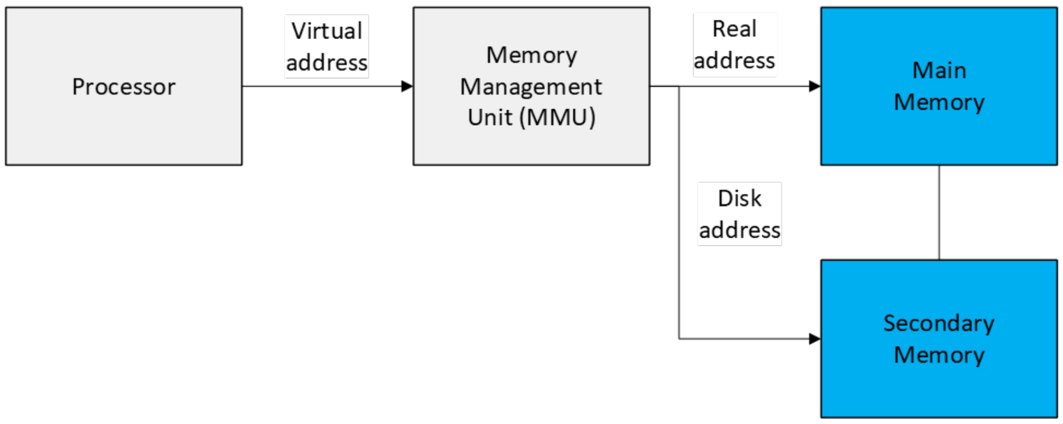

Figure 2: Virtual Memory Addressing. (Adapted from [3, p. 89])

There are two prominent virtual memory management methods: paging and segmentation. In paging, processes can be comprised of a number of fixed-sized blocks called pages. A program references a word by means of a virtual address that consists of a page number and an offset within the page [3, p. 88].

In segmentation, processes comprise of one or more dynamically sized blocks called segments. Similarly to paging, in segmentation a program references a word by means of virtual address that consists of a segment number and an offset within the segment. Segmentation results in a less fragmented memory and better efficiency. However, the dynamic size of segments adds complexity to the management of memory and addressing in the MMU. Because of this added complexity, most modern operating systems only support paging.

3.5 Privilege Rings

Operating systems provide different mechanisms to protect data and execution environments from faults. One such mechanism is to separate execution of software components to different privilege levels. Hierarchical privilege levels are called protection rings, which have become a standard practice in OS design, and commonly enforced by the hardware architecture.

Hardware architectures provide differing number of privilege levels of which most operating systems often only use two. Usually, an OS uses separate protection rings for kernel software and user applications [3, p. 388]. Separating the two protects the most important system processes from unwanted interference by other processes.

3.6 Real Time Systems

Embedded systems are computing systems that have very specific functions or sets of functions. Often, embedded systems are tightly coupled to their environment and require complex real-time constraints [3]. Systems that have complex real-time constraints are called real-time systems. Many such systems can be controlled with simple special purpose programs, but typically, more complex embedded systems include an OS for control and management.

As described in [3, pp. 604-605], real-time operating systems have unique characteristics and design requirements compared to general purpose operating systems:

- **Real-time operation**: The correctness of operation depends on the time it's delivered (hardware timing).
- **Reactive operation**: Software may execute in response to external events (external interrupts).
- **Configurability**: Large variation in the requirements for embedded OS functionality.
- **I/O device flexibility**: Device support is limited to the use case.
- **Streamlined protection mechanisms**: Systems are designed for limited, well-defined functionality.
- **Direct use of interrupts**: Embedded systems can be considered thoroughly tested.

Real-time operating systems have similar boot flow and software components as general-purpose systems. However, The firmware, bootloader and OS kernel are more specialized for the platform and task than on general-purpose systems. The hardware is often less-impressive and thus the software has to be smaller and better optimized than in general-purpose systems.

4 RISC-V Hardware Facilities

The RISC-V is a standardized modular ISA. It consists of alternative base integer ISAs, one of which have to be present in any implementation, and optional instruction-set extensions [12, pp. 12-13]. The base integer ISAs are carefully restricted to minimal set of instructions that provide a reasonable target for compilers, linkers, assemblers and operating systems (with additional privileged operations) [12, p. 13]. They provide a convenient ISA and software toolchain skeleton which can coupled with customized processor ISAs. The base integer ISAs are characterized by their corresponding width of integer registers and the size of their address spaces [12, p. 13].

The RISC-V ISA manual is separated into two volumes: *Volume I: Unprivileged Architecture* [12] and *Volume II: Privileged Architecture* [13]. [12] covers the design of the base unprivileged instructions, as well as optional unprivileged ISA extensions. Unprivileged instructions refer to instructions that are generally usable in all privilege modes in privileged architectures. [13] provides the design of the privileged architecture that covers all aspects of RISC-V systems beyond the unprivileged ISA, including privileged instructions as well as additional functionality required for running operating systems and attaching external devices.

4.1 Hardware Threads & Traps

OS threads can be implemented either in hardware, software or both. *Hardware threads* (harts) have a special role in RISC-V. From the perspective of software running in a execution environment, they are resources that autonomously fetch and execute instructions within an execution environment. Each hart contains a full set of architectural registers and executes their program independently of other harts. The forward progress and control of a hart are governed by the execution environment [12, pp. 11-12].

In RISC-V *exception* refers to an unusual condition at run time associated with an instruction within a hart. *Interrupt* refers to external asynchronous event that happens unexpectedly. The term *trap* refers to the transfer of control to a trap handler caused by either an exception or an interrupt [12, p. 18]. Table 2 lists the possible types and effects of a RISC-V trap from the perspective of software running in an execution environment.

Table 2: Runtime effects of a trap. [12, p. 19]

Trap type	**Effect**
Contained trap	The trap is handled by software running inside the execution environment.
Requested trap	The trap is an explicit call to the execution environment requesting an actn on behalf of software inside the execution environment.
Invisible trap	The trap is handled transparently by the execution environment and execution resumes normally after the trap is handled.

Trap type	Effect
Fatal trap	The trap represents a fatal failure and causes the execution environment to terminate execution.

4.2 Interrupt Architecture

RISC-V interrupt architecture is defined in [13] as a bunch of instructions and registers. Most interrupt-related registers are duplicated for machine and supervisor privilege levels to allow privilege-specific interrupt behavior. Table 3 lists the central components of the RISC-V interrupt architecture.

Table 3: Central pieces of RISC-V ISA's interrupt architecture.

Component	Purpose
`MIE`/`SIE` fields in `mstatus`/`sstatus` register	Indicates globally if interrupts are enabled on a hart. [13, pp. 25-26, 90-91].
`mie`/`sie` register	Indicates which types of interrupts are enabled on a hart [13, pp. 36-39, 93-94].
`mip`/`sip` register	Indicates currently pending interrupts [13, pp. 36-39, 93-94].
`mtvec`/`stvec` register	Holds trap vector configuration: BASE address and vector MODE [13, pp. 34, 92-93].
`medeleg` & `mideleg` registers	Indicate which exceptions/interrupts should be processed directly by a lower privilege level. [13, pp. 35-36]
`mscratch`/`sscratch` register	Can be used to hold a pointer to a M/S-mode hart-local context space. [13, pp. 41, 95].
`mepc`/`sepc` register	Holds the virtual address of the trapped instruction. [13, pp. 41-42, 95-96].
`mcause`/`scause` register	Holds the cause of the last exception / interrupt. [13, pp. 42-44, 96-97].
`mtval`/`stval` register	Holds exception-specific information. [13, pp. 45-46, 97-98].
`mtime` & `mtimecmp` registers	Spawn timer interrupts when `mtime` contains value greater than `mtimecmp`. [13, pp. 49-50].
`ECALL` instruction	Used to generate exception to a more-privileged execution environment [13, p. 50].
`EBREAK` instruction	Used by debuggers to generate breakpoint exceptions [13, p. 50].
`MRET`/`SRET` instruction	Used to return from a trap handler [13, p. 51].
`WFI` instruction	Used to wait for an interrupt in stall mode [13, pp. 51-52].

By default, RISC-V traps at any privilege level are handled in machine mode [13, p. 35]. When an interrupt is encountered:

1. The hardware checks if interrupts are enabled from the global MIE bit in `mstatus` and the right bits in the `mie` register, and marks interrupt pending in `mip` register.
2. The hardware captures the interrupted `pc` into `mepc` register and interrupt cause to the `mcause` register.
3. If redirection is required: The `mepc` and `mcause` values are moved to the right privilege-level's `epc` and `cause` registers, and `mepc` is set to target trap vector in `mtvec`
4. `MRET` is executed, either returning from the trap or moving to the next handler.

- Optionally: the hardware can check `medeleg` and `mideleg` registers and select the right privilege-mode interrupt handler before going to the M-mode handler.

In RISC-V implementations, interrupt architecture is commonly managed through two non-standard hardware modules [14]: the Core Local Interruptor (CLINT) and Platform-Level Interrupt Controller (PLIC). CLINT is used to generate software and timer interrupts into `mtime` and `mtimecmp` CSRs and `mip`'s MSIP field. PLIC is used to route external peripheral interrupts to harts and privilege levels. RISC-V interrupt architecture can be implemented in any way as long as the implementation follows the privileged specification. PLIC and CLINT are default implementations.

4.3 Privilege Levels

RISC-V currently defines three privilege levels used to provide protection between different components of the software stack. The privilege level is encoded as a mode in one or more CSRs (control and status registers). Operations not permitted by the current privilege level will cause an exception to be raised. The exception will normally cause a trap in the underlying execution environment [13, p. 9].

The privileged specification states: "The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively" [13, pp. 9-10].

Table 4: Supported combination of privilege modes. [13, p. 10]

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M,U	Secure embedded systems
3	M,S,U	Systems running Unix-like operating systems

Table 4 shows the supported combinations of privilege modes. All RISC-V hardware implementations must support the M-mode, which provides direct access to the whole machine. Many implementations support at least U-mode to protect the system from application code. S-mode can be added to provide more isolation between M and U-modes [13, p. 10].

In a conventional system supporting at least two privilege levels, a hart typically executes in U-mode until a trap forces a switch to a trap handler that runs in a higher-privileged mode. After the trap-handler has executed in higher privilege-mode, the execution will resume in U-mode [13, p. 10].

4.4 Software Execution Environments

The behavior of a RISC-V program depends on the execution environment it runs on. According to the unprivileged specification: "A RISC-V execution environment interface (EEI) defines the initial state of the program, the number and type of harts in the environment including the privilege modes supported by the harts, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on each hart (i.e., the ISA is one component of the EEI), and the handling of any interrupts or exceptions raised during execution including environment calls [12, p. 11]." Examples of EEIs include *application binary interfaces* (ABIs), *supervisor binary interfaces* (SBIs) and *hypervisor binary interfaces* (HBIs). The RISC-V execution environments can be implemented purely in hardware or software, or as a combination of both [12, p. 11].

Figure 3 shows some of the supported software stacks in RISC-V architecture. In a simple embedded system, the *application execution environment* (AEE) is not isolated from the application; the system supports the execution of only a single application. The application is programmed to run with a particular ABI to hide details of the AEE from the software [13, p. 8].

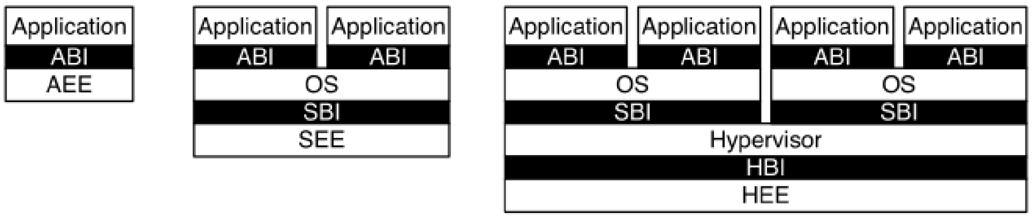

Figure 3: Different implementation stacks supporting various forms of privileged execution. [13, p. 8], CC BY 4.0 © 2024 RISC-V International.

In a more sophisticated Unix-like configuration, with at least 2 privilege levels present, the AEE is implemented by the OS running on a more privileged mode. The OS is programmed to run in a particular SBI which hides details of the *supervisor execution environment* (SEE). [13, pp. 8-9].

In the right-most virtual machine monitor configuration, multiple operating systems are supported by a single hypervisor. Each OS implementing the AEE communicates via SBI with the hypervisor which forms the SEE. The hypervisor is programmed to run in a particular HBI that hides details of the *hypervisor execution environment*

(HEE) [13, p. 9]. Virtual machine monitor configurations are used in high-end server and software virtualization applications.

4.5 ISA Extensions & Profiles

RISC-V comprises of the base integer ISA and optional ISA extensions. There are currently around 50 ISA extensions ratified by the RISC-V Foundation with lots under active development. Appendix -A lists the unprivileged RISC-V ISA extension, currently ratified by the RISC-V foundation, and their identifiers. Appendix -B lists the currently ratified privileged RISC-V ISA extensions. RISC-V ISA string symbol G is reserved as a shorthand for extensions IMAFD_Zicsr_Zifencei.

To allow software to rely on the existence of a certain set of ISA features on application processors, RISC-V provides standardized ISA extension profiles for particular generation of RISC-V implementations. RISC-V ISA extension profiles align vendors targeting binary software markets, allowing better software compatibility between different implementations as well as greater competitiveness against other architectures. RISC-V profiles consist of selections of mandatory, optional and transitory ISA extensions [15].

4.6 A - Atomic Instructions

As listed in Appendix -A, the standard ISA extension “A” provides atomic instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. “A” provides two forms of atomic instruction: load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency ordering semantics [12].

Atomic instructions behave as a sequence of other instructions that appear to be indivisible; no other process can see an intermediate state or interrupt the operation. Atomicity provides isolation from concurrent processes, which is crucial for multi-programming operating systems and applications [3].

4.7 Zicsr - Control & Status Registers

Control and Status Registers (CSRs) are a variety of processor registers that control the operation of a processor [3]. Typically the most essential CSRs on a system are the program counter (PC), instruction register (IR), memory address (MAR) and buffer (MBR) registers, as well as input/output address (I/O AR) and buffer (I/O BR) registers [3]. Out of these, RISC-V ISA only implements the PC register. Instead of a dedicated IR register, RISC-V instructions use instruction fetch pipelines. RISC-V architecture follows a load-store architecture where only load and store instructions access memory and arithmetic instructions only operate on CPU registers [12, p. 31]. Memory and I/O accesses are both similar load/store operations to different addresses that are temporarily stored on general purpose registers. Figure 4 shows the encoding of 32-bit base ISA’s load and store instructions.

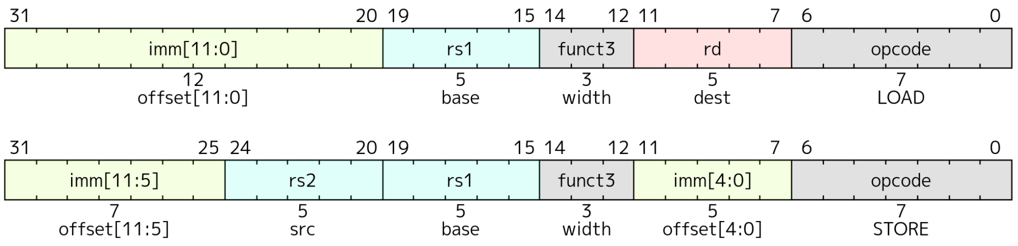

Figure 4: Encodings of the RV32I `LOAD` and `STORE` instructions. [12, p. 32], CC BY 4.0 © 2024 RISC-V International.

RISC-V defines a separate address space of 4096 CSRs associated with each hart. The Zicsr ISA extension defines instructions for reading and writing CSRs on this address space. [12, p. 46] Figure 5 shows the encoding for all RISC-V CSR instructions. All Zicsr instructions are encoded with the `SYSTEM` `opcode` field. The `func3` field encodes the CSR instruction in question.

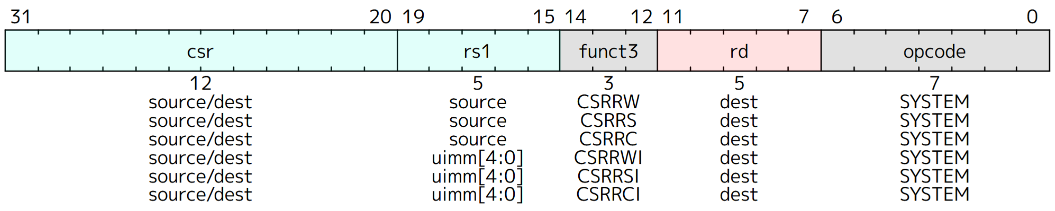

Figure 5: Encodings of the instructions defined in the Zicsr extension. [12, p. 46], CC BY 4.0 © 2024 RISC-V International.

4.8 ZiFencei - Synchronization

RISC-V does not guarantee that stores to execution memory will be made visible to instruction fetches on a hart until the hart executes an explicit synchronization instruction [12, p. 44]. RISC-V defines a relaxed memory ordering scheme: *RISC-V Weak Memory Ordering* (RVWMO) which doesn't enforce strong memory ordering between harts [12, pp. 84-93]. Explicit synchronization is required due to the possibility of out-of-order execution.

The Zifencei ISA extension includes the `FENCE.I` instruction that is used to explicitly synchronize between writes to instruction memory and instruction fetches on a hart [12, p. 44].

4.9 Zicntr & Zihpm - Counters & Timers

In modern general purpose systems and real-time applications (briefly outlined in Section 3.6) it is often necessary for a system to contain real-time timers and performance counters to allow real-time operation. RISC-V ISA provides a set of up to thirty-two 64-bit counters and timers that are accessible via unprivileged read-only CSR registers. These counters are divided between the “Zicntr” and “Zihpm” extensions [12, p. 50].

Zicntr standard extension for base counters and timers defines three counters: `cycle`, `instret` and `time` which each have dedicated functions: cycle count, instructions retired and real-time clock respectively. Pseudo-instructions for accessing the counters are mapped as `csrrs` calls to counter addresses [12, p. 50].

Zihpm extension for hardware performance counters up to 29 additional unprivileged counters `hpmcounter3-hpmcounter31`. The specification states: "The implemented number and width of these additional counters, and the set of events they count, is platform-specific. Accessing an unimplemented or ill-configured counter may cause an illegal instruction exception or may return a constant value [12, p. 52]." The execution environment should allow means to determine the number and width of the additional implemented counters [12, p. 52].

4.10 Sv32, Sv39, Sv48 & Sv57 - Virtual Memory

To support operating systems' typical page-based memory management, discussed in Section 3.4, the RISC-V privileged supervisor-level ISA defines one 32-bit and three 64-bit page-based virtual-memory addressing schemes: Sv32, Sv39, Sv48 and Sv57 respectively. In each Sv (supervisor virtual addressing) scheme, the virtual addresses are translated into physical addresses by traversing a radix-tree page table [13, pp. 104-114]. The Sv32, Sv39, Sv48 and Sv57 schemes define the radix-tree page tables to be 2, 3, 4 and 5 levels deep respectively [13, pp. 104-114]. The supervisor address translation and protection register `satp` is used to hold the selected Sv mode and to hold a pointer to the root node in the radix-tree page table [13, pp. 104-114].

The RISC-V supervisor-level ISA defines the default page size for each Sv scheme to be 4 KiB. Additionally each Sv scheme supports using non-root level page-table nodes other than the last level as leaf nodes, providing support for 4 MiB *megapages*, 1 GiB *gigapages*, 512 GiB *terapages* and 256 TiB *petapages* depending on the number of levels implemented. Megapages and gigapages are especially useful, as bigger pages are commonly utilized by operating systems to reduce *translation lookaside buffer*'s (TLB's) page entries by mapping special structures, such as kernel data or frame buffers into larger pages. TLB is a memory cache that is used to store recently and frequently used virtual memory translations [3, p. 379].

5 Zephyr

Zephyr is a real-time operating system (RTOS) developed under the Linux Foundation. It is based on small-footprint kernel designed to be used in resource constrained embedded systems. The Zephyr kernel supports many hardware architectures including RISC-V [16, p. 1].

Zephyr OS follows a 3-layer structure where the core OS consists of the small-footprint kernel, and selected OS and application services. The structure allows for greater resource flexibility as the OS can be configured with or without support for different low-level services. Figure 6 shows the internal structure of Zephyr.

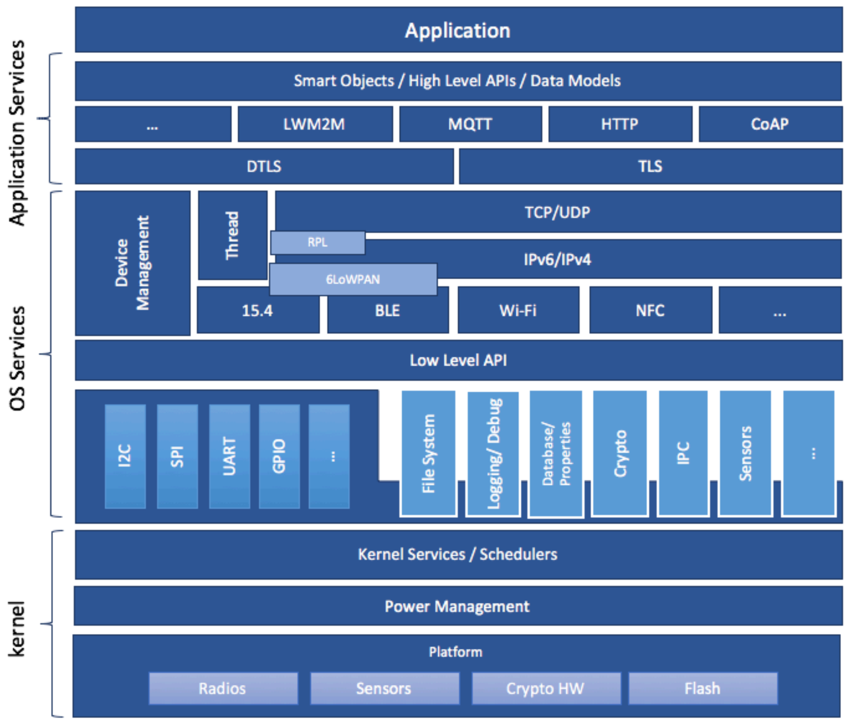

Figure 6: Zephyr system architecture from [16, p. 1633],
Apache License 2.0 © 2023 Zephyr® Project, a Linux Foundation Project.

Zephyr relies on two global build-time configuration frameworks for selecting which OS features to include: Kconfig and devicetree. Kconfig defines the overall OS configuration while devicetree provides the hardware abstraction for software. Hardware description schemes other than devicetree are not supported by Zephyr.

Zephyr on RISC-V can be configured either as a simple embedded system supporting only M-mode, or as a secure embedded system with support to M- and U-modes. Zephyr's U-mode support on RISC-V requires that the platform to support *physical memory protection* (PMP)(See [13, pp. 59-63]). [16, p. 1398] As Zephyr isn't a supervisor-level operating system, SBI or other runtime firmware interfaces are not required; Zephyr can run directly on hardware.

5.1 SERV - The SErial RISC-V CPU

One minimal RISC-V project capable of running Zephyr is the *SErial RISC-V CPU* (SERV) project by Olof Kindgren [17]. SERV claims to be the world's smallest RISC-V core, implemented entirely in bit-serial architecture [17]. In bit-serial architecture the internal datapath is only one bit wide and only one bit is processed per clock cycle. SERV is intended to be used for embedded systems, IoT devices, education and research [18].

SERV core features RISC-V RV32I_Zifencei ISAs with optional extensions for C, M, Zicsr and timer interrupt support. SERV is structured in a modular manner with pre-designed convenience wrappers to simplify platform integration. SERV project includes a small FPGA-focused reference platform called *Servant*, which is capable of running Zephyr OS 3.7 [18].

Servant builds upon the *Servile* SERV convenience wrapper that contains:

- a memory bus intended to be connected to a combined data and instruction memory
- an extension bus for peripheral controllers and accelerators
- and an interface for connecting to an SRAM for GPR and CSR registers.

Servant interfaces Servile with a timer (for timer interrupts), some memory and a 1-bit GPIO output pin. Servant SoC is easy to port to all kinds of FPGA boards [18]. Servant uses the following KConfig SoC configuration [17]:

1. "`select RISCV`": Enables RISC-V architecture support [16].
2. "`select ATOMIC_OPERATIONS_C`": Use atomic operations implemented in C [16].
3. "`select RISCV_ISA_EXT_ZICSR`": Enables support for Zicsr extension [16].

5.2 Requirements & Zephyr on A-Core

As mentioned in Section 5.1, Servant can be thought of as being close to the minimal requirements for Zephyr OS. Servant implements SERV with RV32I_Zicsr_Zifencei ISA configuration, which represent the minimal ISA requirements required to run Zephyr on a RISC-V platform. In addition to the ISA requirements, Zephyr requires data and instruction memory, hardware timer for timer interrupts, some basic I/O functionality, as well as memory and peripheral buses for data access. For meaningful embedded operation, the peripheral bus should be extensible with custom controllers and accelerators. In addition to the hardware requirements, Zephyr kernel requires a devicetree hardware description file loaded to the memory during kernel boot.

Aalto University's A-Core is a 7-stage pipelined processor, following the Harvard microarchitecture. It currently implements RV32IM_Zicsr ISAs with partial A support, program and random access memories, memory mapped IO and some other peripherals such as JTAG programming interface and DMA through AXI4-Lite peripheral bus [19]. As outlined above, Zephyr requires only RV32I_Zicsr_Zifencei ISAs, some memory and GPIO, and support for timer interrupts. Therefore, A-Core should be close to being capable of running Zephyr, only missing the Zifencei ISA extension and support for timer interrupts.

6 Linux

Linux is a family of UNIX-like operating system initially developed by Linus Torvalds in 1991. Since its initial release, Linux has matured into one of the most used operating systems on servers, mobile devices and desktop computers. Linux OS comprises of a bootloader, the Linux kernel, an init system and a set of system programs. Different bootloaders, init systems and sets of system programs: daemons, graphical servers and desktop environments, etc. exist, forming a plethora of Linux flavors referred to as distros. In this review, only the Linux kernel will be investigated as it's the most central piece of the Linux OS in operating system point-of-view.

Linux kernel follows a monolithic kernel approach where the whole kernel runs in higher-privilege kernel mode completely free from the lower-privileged user-mode application. The Linux kernel functionality is structured in modular subsystems as shown in Figure 7 [20].

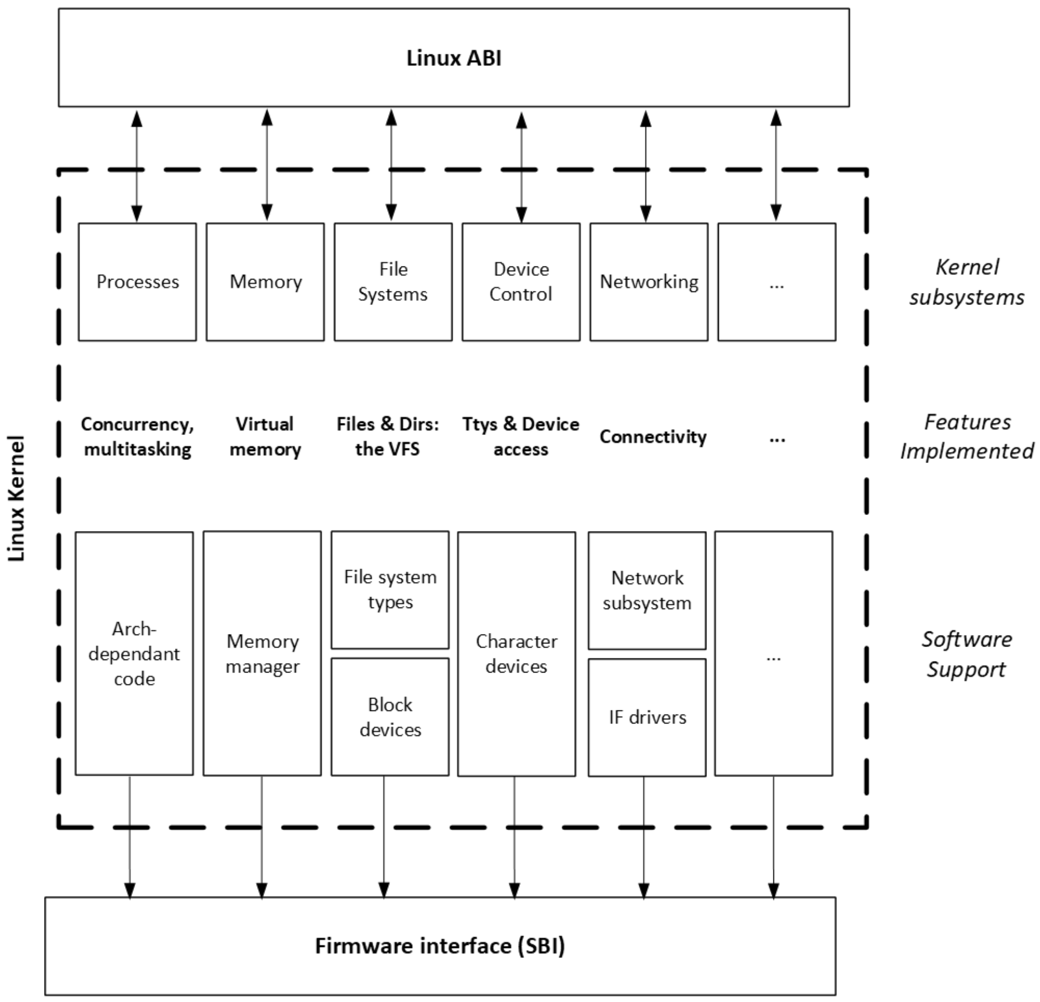

Figure 7: Linux system structure. (Adapted from [20, Fig. 1-1].)

Each subsystem implements some OS feature and provides the user-mode applications with software support through the Linux ABI. The subsystem's kernel-mode software interfaces with the hardware system's firmware interface [20].

Linux kernel supports both SBI and UEFI firmware interfaces on RISC-V. The firmware needs to provide the Linux kernel a hardware description file as either ACPI Tables or a devicetree [21]. Devicetree are either passed to the kernel by the previous stage using the `$a1` general purpose register, or using the EFI configuration table when booting with UEFI [21]. ACPI Tables can only be passed via the EFI configuration table [21].

6.1 Chips Alliance - Rocket Chip

One minimally-configurable RISC-V project that is capable of booting Linux is the Chips Alliance's *Rocket Chip Generator* project [22]. The Rocket Chip Generator is an open-source *system-on-chip* (SoC) description generator that generates synthesizable *register transfer language* (RTL) code [23]. Rocket Chip generator generates general-purpose RISC-V processor cores. The Rocket Chip generator can generate both in-order *Rocket* cores and out-of-order *BOOM* cores [23]. "Rocket Chip has been taped out (manufactured) eleven times, and yielded functional silicon prototypes capable of booting Linux [23]."

Rocket generator is a 5-stage in-order scalar core generator that can implement general purpose RISC-V cores, which support RV32G or RV64G ISAs and machine, supervisor and user privilege levels [23]. *Rocket* includes an MMU generator that supports Sv32, Sv39, Sv48 and Sv57 page-based virtual memory scheme configurations [22]. It also includes a non-blocking data cache and a variable size TLB [23].

Rocket Chip Generator couples the Rocket (or BOOM) core with BootROM, PLIC, CLINT and additional caches, TLBs, co-processors and system buses into a compatible and synthesizable SoC description [22]. It also generates BootROM firmware and compiles a devicetree hardware description for operating systems [22].

6.2 789 KB Linux Without MMU on RISC-V

Even more minimal configuration for Linux is possible by configuring Linux without MMU support. Uros Popovic's blog guide *789 KB Linux Without MMU on RISC-V* [24] explains how Linux can be stripped out of its virtual memory management and run on QEMU virt RV32GC emulator without MMU support. Linux without MMU support was initially separate project called uClinux but was mainlined to Linux via configuration options [24].

According to [24], removing virtual memory poses challenges in the Linux environment:

- ELF executable file format explicitly assumes virtual address space and doesn't work. Luckily, bFLT binaries can be used instead.
- Pointers point to physical memory directly which is dangerous.
- The `fork` system call relies on virtual memory, `vfork` should be used instead.
- Compiling correct bFLT binaries is tricky.
- Most system calls and even `printk` (for printing) are not supported.
- Separate kernel and user modes aren't fully supported. (No U-mode support)

To create the right configuration, Linux is first configured to the minimal `tinyconfig` configuration. After that, the configuration is edited as follows [24]:

1. `CONFIG_NONPORTABLE: y`. Enables non-portable builds.
2. `CONFIG_MMU: n`. Disables MMU and virtual memory.
3. `PHYS_RAM_BASE_FIXED: y`. Sets kernel address as fixed.
4. `CONFIG_PHYS_RAM_BASE: 0x80000000`. Sets the kernel address to 0x80000000.
5. `CONFIG_BLK_DEV_INITRD: y`. Adds `initramfs` support.
6. `CONFIG_BINFMT_FLAT: y`. Adds bFLT support.

Using this configuration, Popovic was able to boot Linux and print a message to a UART. In [24], Popovic also configured another MMU-less Linux image with added support to TTY and UART drivers as well as other useful functionality, which was also able to boot without the MMU. The final kernel was still lacking basic Linux kernel functionality, but proved that Linux can be configured to run on RISC-V without an MMU.

6.3 OpenSBI

On RISC-V, Linux is a supervisor-level OS and requires an interface between platform-specific firmware running in M-mode and OS-code running in S-mode. As mentioned in Section 6, Linux supports both SBI and UEFI firmware interfaces for this purpose. SBI is the typical interface on RISC-V systems.

One notable SBI implementation is the OpenSBI reference implementation by RISC-V International. The main component in OpenSBI is the platform-independent static library `libsbi.a` that implements the SBI interface [25]. A firmware or bootloader implementation can link against the library to ensure compatibility with the SBI [25]. OpenSBI supports SBI specification v0.2 and the *hart state management* (HSM) SBI extension (defined in [7, pp. 26-31]). The HSM extension allows S-mode software to boot all harts in a defined order, enabling commonly required OS features [25].

OpenSBI's platform requirements are listed in `platform_requirements.md` documentation file [25]. According to the file, the base requirements for OpenSBI are as follows:

1. At least RV32IMA_Zicsr or RV64IMA_Zicsr ISAs on all harts.
2. At least one hart with S-mode support.
3. `MTVEC` CSR on all harts must support direct mode.
4. Optional `PMP` CSRs for protecting M-mode firmware and secured memory regions.
5. `TIME` CSR implemented or 64-bit hardware MMIO counter emulating it.
6. Hardware support for injecting M-mode software interrupts on multi-hart platform.

6.4 Requirements

Rocket Chip serves as a good reference implementation for the hardware and architectural features needed to run a full Linux operating system on a RISC-V platform. Rocket Chip implements RV32/64G ISAs, short for RV32/64IMAFD_Zicsr_Zifencei. Rocket Chip also implements the RISC-V M-,S- and U-modes, support for software, timer and external interrupts as well as RISC-

V Sv32-57 page based MMU with TLB. Implicitly, Rocket Chip includes sufficient memory, peripheral buses and functionality required to run Linux.

The RISC-V single-precision floating point and double-precision floating point extensions F and D are required for Linux [24] and can be software emulated. However, not having floating point support may impose a large bottleneck for user-space programs. Further investigation is required to confirm if these extensions are necessary for Linux in practice.

As noted in Section 6.3, Linux is a supervisor-level OS and requires a firmware interface between M- and S-modes. The reference firmware implementation OpenSBI requires at least RV32/64IMA_Zicsr, `MTVEC`, `TIME` and optional `PMP` CSRs, hardware timer and S-mode support.

Linux can be stripped out of MMU and U-mode dependencies and configured to run on a wildly reduced feature set [24]. However, this results in the OS lacking basic functionality outlined in Section 6.2. In addition to the hardware requirements, Linux kernel requires a devicetree hardware description file loaded to the memory during kernel boot.

6.5 Linux on A-Core

As mentioned in Section 5.2, A-Core implements RV32IM_Zicsr ISAs with limited A extension support and no MMU. Minimal Linux requires at least RV32IMA_Zicsr_Zifencei, M and S-modes and support for software, timer and external interrupts on top of the implicit memory, I/O and bus requirements. However, less barebones Linux would additionally require support for page-based MMU, U-mode and RISC-V floating point extensions F and D.

For the most minimal Linux, A-Core is missing at least the A and Zifencei extensions, S-mode support and software, timer and external interrupt support. For normal Linux functionality, page-based MMU, U-mode and floating point should also be supported. SBI should also be implemented by the platform's M-mode execution environment, and devicetree hardware specification of the system should be passed to the kernel at kernel boot time using the `$a1` general purpose register.

7 Summary & Conclusions

This study explored RISC-V ISA features required to boot and run modern operating systems. The purpose of the study was to answer two research questions. 1. What hardware facilities does the RISC-V ISA provide for supporting operating systems. 2. What are the minimal set of RISC-V ISA features required to boot and run Linux and Zephyr operating systems.

In Section 2, it was found that the OS boot process is a chain of hardware and software layers and interfaces. The boot process begins from the processor's reset vector, pointing to a data address in the boot ROM memory. The boot ROM contains platform specific system firmware that is used to load the next stage bootloader and provide runtime services to the above layers. The bootloader is used to load the Operating system and pass the required hardware discovery binary to the Kernel. The OS Kernel is responsible for the core functionality of the OS, such as memory management, process management and file systems. To end the boot process, the kernel starts a software component called the init process, which initializes the user space of the operating system.

Section 3 explored common OS facilities and their typical hardware requirements. It was outlined that operating systems commonly provide: process management using CSRs and interrupts, system call and ABI interface via supervisor calls, multiprogramming and concurrency support through atomic operations and synchronization, memory management with - page based - virtual memory, privilege rings using ISA privilege levels, and real-time operation using hardware counters and timers.

In Section 4, the RISC-V implementation of these requirements was outlined. RISC-V privileged specification [13] defines the interrupt architecture of the RISC-V platform as a collection of instructions and memory mapped CSRs as listed in Table 3. RISC-V currently defines three privilege levels: machine, supervisor and user. The privilege level of a hart is encoded as a field in one or more CSRs. RISC-V defines an address space of 4096 CSRs for each hart. These CSRs are accessed by the CSR read/write operations defined in the Zicsr ISA extension. Atomic operations on RISC-V are defined in the A ISA extension. Synchronization in RISC-V is handled by the FENCE instruction, defined in Zifencei ISA extension. RISC-V also supports one 32-bit and three 64-bit page-based virtual-memory addressing schemes Sv32-57. RISC-V supports hardware timers and counters as well as timer interrupts using the `mtime` and `mtimecmp` registers defined in the privileged specification [13]. In addition to the OS requirements, RISC-V ISA also supports the SBI system firmware interface. SBI is a RISC-V firmware interface between platform-specific firmware running in M-mode and OS-code running in S-mode.

In Section 5, the practical RISC-V requirements for Zephyr OS were explored through the SERV project. SERV claims to be the world's smallest RISC-V core [17], implemented entirely in bit-serial architecture. Servant is a convenience wrapper that extends the SERV core with a hardware timer, memory and a 1-bit GPIO pin. Servant is capable of running Zephyr version 3.7. The hardware requirements for Zephyr, based on the SERV project, are RV32I_Zicsr_Zifencei ISAs, data and instruction memory, hardware timer for timer interrupts, some basic functionality and memory and peripheral buses for data access.

In Section 6, the practical RISC-V requirements for Linux were explored through the Rocket Chip and 789 KB Linux Without MMU projects. Rocket Chip Generator is an open-source RTL generator that generates general-purpose RISC-V processor cores. Based on the Rocket Chip project, the minimal hardware requirements for Linux are RV32/64IMA_Zicsr_Zifencei ISAs, machine and supervisor privilege modes, page-based MMU for virtual memory, and support for software, timer and external interrupts on top of the basic memory, and bus requirements. Linux can be stripped out of the page-based MMU requirement by wildly reducing the kernel functionality via configuration. In Section 6, it was also noted, that on RISC-V Linux is a supervisor-level OS and requires the RISC-V SBI firmware interface between machine and supervisor privilege modes. The most common firmware interface for this purpose is the OpenSBI reference implementation by RISC-V international. OpenSBI requires: implementation of at least RV32IMA_Zicsr ISAs, at least one hart with supervisor mode, support for external machine-mode interrupts, `MTVEC` in direct mode, and a TIME CSR or 64-bit MMIO timer.

As described in Section 5.2, Aalto University's A-Core processor currently implements the RV32IM_Zicsr ISAs with limited A extension support and no MMU. The research results suggest that A-Core is close to being capable of running Zephyr, only missing the Zifencei ISA extension and support for timer interrupts. The results further suggest that to run Linux, A-Core would require at least A and Zifencei extensions, supervisor mode, software, timer and external interrupt support, and for normal operation, a page-based MMU and user mode.

The identified requirements may not represent the absolute minimum requirements for Linux and Zephyr OS, as they are mostly derived from practical implementations instead of experimenting with the minimal requirements of operating systems. However, the findings should point to the right direction and perhaps be useful in developing operating system support for Aalto University's A-Core processor.

Bibliography

[1] RISC-V International, “About RISC-V International.” Accessed: Apr. 29, 2025. [Online]. Available: https://riscv.org/about/

[2] Aalto University Department of Electronics and Nanoengineering, “A-Core · GitLab.” Accessed: May 06, 2025. [Online]. Available: https://gitlab.com/a-core

[3] W. Stallings, *Operating systems: internals and design principles*, Ninth edition, global edition. Harlow, England: Pearson, 2018.

[4] N. Bin *et al.*, “Research and design of Bootrom supporting secure boot mode,” in *2020 International Symposium on Computer Engineering and Intelligent Communications (ISCEIC)*, Aug. 2020, pp. 5–8. doi: 10.1109/ISCEIC51027.2020.00009.

[5] L. Dailey Paulson, “New technology beefs up BIOS,” *Computer*, vol. 37, no. 5, p. 22–, May 2004, doi: 10.1109/MC.2004.1297231.

[6] UEFI Specification Working Group (USWG) and Tianocore Community Members, “Unified Extensible Firmware Interface (UEFI) Specification,” *Release 2.11*, Nov. 2024, Accessed: Apr. 30, 2025. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI_Spec_Final_2.11.pdf

[7] A. Chang, A. Stone, A. Jones, A. Patel, and RISC-V Foundation, “RISC-V Supervisor Binary Interface Specification,” Jan. 2024, [Online]. Available: https://drive.google.com/file/d/1U2kwjqxXgDONXk_-ZDTYzvsV-F_8ylEH

[8] Devicetree.org, “Devicetree Specification Release v0.4,” p. 61, Jun. 2023, Accessed: Feb. 28, 2025. [Online]. Available: https://www.devicetree.org/specifications

[9] ACPI Specification Working Group (ASWG), UEFI Specification Working Group (USWG), and Tianocore Community Members, “Advanced Configuration and Power Interface (ACPI) Specification,” *Release 6.5 Errata A*, Nov. 2024, Accessed: Feb. 28, 2025. [Online]. Available: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.5a_Final.pdf

[10] K. C. Wang, *Design and Implementation of the MTX Operating System*. Cham: Springer International Publishing, 2015. doi: 10.1007/978-3-319-17575-1.

[11] W. Almesberger, “Booting linux: The history and the future,” in *Proceedings of the Ottawa Linux Symposium*, 2000.

[12] A. Waterman, K. Asanović, and RISC-V Foundation, “The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture,” Apr. 2024, Accessed: Jan. 24, 2025. [Online]. Available: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view

[13] A. Waterman, Y. Lee, R. Avižienis, D. Patterson, K. Asanović, and RISC-V Foundation, “The RISC-V Instruction Set Manual: Volume II: Privileged

Architecture," Apr. 2024, [Online]. Available: https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view

[14] Y. Ren and N. Tan, "Enhanced platform-based interrupt controller for RISC-V MCUs," *Microelectronics Journal*, vol. 159, p. 106628, May 2025, doi: 10.1016/j.mejo.2025.106628.

[15] RISC-V Foundation, "RVA23 Profiles," Oct. 2023, Accessed: Mar. 17, 2025. [Online]. Available: https://drive.google.com/file/d/12QKRm92cLcEk8-5J9NI91m0fAQOxqNAq/view

[16] The Zephyr Project Contributors, "Zephyr Project Documentation Release 4.1.99." Accessed: Apr. 10, 2025. [Online]. Available: https://docs.zephyrproject.org/latest/zephyr.pdf

[17] O. Kindgren, "olofk/serv." Accessed: Mar. 26, 2025. [Online]. Available: https://github.com/olofk/serv

[18] K. Olof, "SERV user manual — SERV documentation." Accessed: Apr. 15, 2025. [Online]. Available: https://serv.readthedocs.io/en/latest/index.html

[19] Aalto University Department of Electronics and Nanoengineering, "A-Core / A-Core_Chisel / ACoreChip · GitLab." Accessed: Apr. 15, 2025. [Online]. Available: https://gitlab.com/a-core/a-core_chisel/ACoreChip

[20] J. Corbet, A. Rubini, G. Kroah-Hartman, and A. Rubini, *Linux device drivers*, 3rd ed. Beijing ; Sebastopol, CA: O'Reilly, 2005.

[21] A. Ghiti, "RISC-V Kernel Boot Requirements and Constraints — The Linux Kernel documentation." Accessed: Apr. 12, 2025. [Online]. Available: https://www.kernel.org/doc/html/latest/arch/riscv/boot.html

[22] "chipsalliance/rocket-chip." Accessed: Apr. 10, 2025. [Online]. Available: https://github.com/chipsalliance/rocket-chip

[23] K. Asanović *et al.*, "The Rocket Chip Generator," Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[24] U. Popovic, "789 KB Linux Without MMU on RISC-V." Accessed: Feb. 04, 2025. [Online]. Available: https://popovicu.com/posts/789-kb-linux-without-mmu-riscv/

[25] "riscv-software-src/opensbi." Accessed: Apr. 14, 2025. [Online]. Available: https://github.com/riscv-software-src/opensbi

Appendix

-A Ratified Unprivileged RISC-V ISA Extensions

Table 5: Ratified Unprivileged RISC-V ISA Modules [12].

Identifier	**Extension**
Zifencei	Extension for Instruction-Fetch Fence
Zicsr	Extension for Control and Status Register (CSR) Instructions
Zicntr	Extension for Base Counters and Timers
Zihintntl	Extension for Non-Temporal Locality Hints
Zihintpause	Extension for Pause Hint
Zimop	Extension for May-Be-Operations
Zicond	Extension for Integer Conditional Operations
M	Standard Extension for Integer Multiplicatn and Divisn
Zmmul	Extension for the multiplicatn subset of the M extension
A	Standard Extension for Atomic Instructions
Zawrs	Extension for Wait-on-Reservatn-Set Instructions
Zacas	Extension for Atomic Compare-and-Swap (CAS) Instructions
RVWMO	RISC-V Memory Consistency Model
Ztso	Extension for Total Store Ordering
CMO	Extensions for Base Cache Management Operation ISA
F	Standard Extension for Single-Precisn Floating-Point
D	Standard Extension for Double-Precisn Floating-Point
Q	Standard Extension for Quad-Precisn Floating-Point
Zfh	Extension for Half-Precisn Floating-Point
Zfhmin	Extension for Minimal Half-Precisn Floating-Point
Zfa	Extension for Additnal Floating-Point Instructions
Zfinx	Extension for Single-Precisn Floating-Point in Integer Registers
Zdinx	Extension for Double-Precisn Floating-Point in Integer Registers
Zhinx	Extension for Half-Precisn Floating-Point in Integer Registers
Zhinxmin	Extension for Minimal Half-Precisn Floating-Point in Integer Registers
C	Standard Extension for Compressed Instructions
Zc*	Extensions for Code Size Reductn
B	Standard Extension for Bit Manipulatn
V	Standard Extension for Vector Operations
*Zbkb	Extension for Bit-manipulatn for Cryptography
*Zbkc	Extension for Carry-less multiplicatn for Cryptography
*Zbkx	Extension for Crossbar permutatns

Identifier	Extension
*Zk	Extension for Standard Scalar cryptography
*Zks	ShangMi Algorithm Suite
*Zvbb	Extension for Vector Basic Bit-manipulatn
*Zvbc	Extension for Vector Carryless Multiplicatn
*Zvkg	Vector GCM/GMAC
*Zvkned	NIST Suite: Vector AES Block Cipher
*Zvknhb	NIST Suite: Vector SHA-2 Secure Hash
*Zvksed	ShangMi Suite: SM4 Block Cipher
*Zvksh	ShangMi Suite: SM3 Secure Hash
*Zvkt	Extension for Vector Data-Independent Execution Latency

-B Ratified Privileged RISC-V ISA Extensions

Table 6: Ratified privileged RISC-V ISA Modules [13].

Identifier	Extension
Machine ISA	Machine-Level Support (required)
Smstateen	Extensions for State-enable
Smcsrind/Sscsrind	Extensions for indirect CSR Access
Smepmp	Extension for PMP Enhancements for memory access and execution prevention in Machine mode
*Smcntrpmf	Extension for Cycle and Instret Privilege Mode Filtering
Supervisor ISA	Supervisor-Level Support
Svade	Raise exceptions on accessing improper A/D bits
Smrnmi	Extension for Resumable Non-Maskable Interrupts
Svnapot	Extension for NAPOT Translation Contiguity
Svpbmt	Extension for Page-Based Memory Types
Svinval	Extension for Fine-Grained Address-Translation Cache Invalidation
Svadu	Extension for Hardware Updating of A/D Bits
Sstc	Extension for Supervisor-mode Timer Interrupts
Sscofpmf	Extension for Count Overflow and Mode-Based Filtering
H	Extension for Hypervisor Support