Porting mCertiKOS to RISC-V
A Senior Project in Computer Science

Adam Cimpeanu
CPSC 490: Special Projects
Advisor: Professor Zhong Shao
Spring 2016

The rise of embedded systems, SoCs, and FPGAs has resulted in growing interest in Reduced Instruction Set Computing (RISC) and more experimental Instruction Set Architectures (ISAs). mCertiKOS, a certified lightweight operating system kernel built by the FLINT group at Yale has rewritten their kernel using several ISAs as a way of both testing the limits of their hardware and specializing towards embedded systems applications. It has been modified to run on ARM chips and the Landshark UGV.

RISC-V is a new RISC ISA that generated much attention recently because it is the first open source ISA with a mature software stack, codebase, and active development team. Currently, the Indian government and IIT Madras are investing money and resources into developing their own 64-bit RISC-V chips. Active development is taking place on a working RISC-V Linux port. The instruction set is highly flexible in its design, allowing for 32-bit, 64-bit, and eventually 128-bit implementations, with four privilege execution levels and a large number of user-level and privilege level-specific registers.

In this project, a portion of the mCertiKOS kernel was ported to the RISC-V ISA and successfully run on a tethered ISA simulator running on top of a Linux virtual machine. Specifically, a bare metal kernel was built to run directly over the Berkeley Boot Loader (bbl), a bootloader provided by the RISC-V software stack. The stack was set up, allowing for development of C code on the RISC-V platform. Formatted print and debugging utilities were added. mCertiKOS’s MATInit, MAT Intro, and MATOp layers were added, providing the port with a physical memory management layer. This project serves as the keystone for future efforts to fully port mCertiKOS to RISC-V. When verified, mCertiKOS will be the first verified kernel running on this ISA.

1. Introduction

1.1 CertiKOS and mCertiKOS

The FLINT group at Yale, which focuses on almost all aspects of building certified systems software, is currently conducting research on an in-house, lightweight operating system kernel called mCertiKOS. Its parent project, CertiKOS (Certified Kit Operating System) was originally designed as a platform for secure cloud computing [1]. Since the completion of this larger verified kernel, the group has streamlined its kernel and verified a smaller version both to decrease development time upon successive iterations and to allow it to run on smaller hardware platforms including embedded systems. Projects like Embassies at Microsoft Research have demonstrated that a number of interesting research questions arise when working with minimal execution models [2].

The condensed version of CertiKOS is called mCertiKOS and currently exists as a set of verified kernels derived from the same
base uniprocessor version. Included in this set are the mCertiKOS-hyp kernel, which has been extended to run as a fully-functional hypervisor, mCertiKOS-rz, which supports “ring 0” processes, and mCertiKOS-emb, which has been stripped of virtualization, virtual memory, and user-space interrupt handling for the sole purpose of being used in embedded systems [3].

In addition to these functional variants to mCertiKOS, attempts have been made by the group to rewrite the kernel using other instruction set architectures (ISAs). The kernel was originally written using a combination of C and x86 assembly, but working versions have been written in AMD and ARM. Extending mCertiKOS to other instruction sets has allowed the kernel to run on other hardware platforms—the lab has experimented with Landshark UGV and the American-Built Car platforms—and the work is also a rich source of research, as different instruction sets make different assumptions about the underlying hardware and each version must be verified separately.

1.2 RISK-V

In 2010, researchers at the University of California Berkeley began the project of building an open-source ISA. Their design was based on the concept of Reduced Instruction Set Computing (RISC) in which a small number of optimized instructions are used instead of a larger, more complex set. Often, what gets sacrificed in specialization and versatility within an ISA is made up for by dramatic increases in the speed and efficiency of a microprocessor running that set. While other RISC instruction sets exist—most notably MIPS from Stanford—and other attempts have been made at building open-source instruction sets, what sets RISC-V apart is that it is both open-source and it was designed to keep up with the demands of more serious computing environments, such those seen in cloud and mobile platforms.

With its increasing hardware and software support, RISC-V has begun to fulfill some of the promises of RISC-based designs. The RISC-V Rocket core microprocessor is twice as energy efficient as the Cortex-A5, the most similar ARM implementation [4]. The potential promise of the platform has inspired the R&D division of the Indian government to invest $45 million to design a 64-bit processor based on RISC-V [5]. Many of the potential chipsets to come out of similar work at IIT Madras will be targeted towards IoT applications.

1.3 Porting to RISC-V

The most promising application for secure kernels is in the field of embedded systems. The FLINT group took a major step towards running on this platform when they rewrote their kernel using ARM and AMD. Rewriting the kernel in a RISC instruction set would seem to be the next logical step, as these reduced sets are ideal for embedded systems, and writing it in RISC-V would open many exciting new doors for the group. If verified, mCertiKOS on RISC-V would be the first verified kernel on this instruction set. Because of the continuing work on RISC-V at UC Berkeley, its BSD Open Source License, and the increased attention it is getting both from research labs in India and major companies like Google, HP, and Oracle, there is a very good chance that this instruction set will take a more central role in the research community in the years to come [6]. This makes it even more of a priority for the FLINT group to investigate ways of integrating RISC-V with their work.
2. Methods

2.1 Port Target

Porting the entire mCertiKOS kernel is outside the scope of a semester project. The targeted goal of this project was to port just a portion of a polished version of mCertiKOS. In the operating systems course taught at Yale in the fall, CPSC 422, a polished version of mCertiKOS is progressively built over the course of six labs. Building a working version of lab1, which includes everything up to physical memory management, was the goal of this project.

2.2 Computing Environment

All development took place on a 13-inch mid 2012 Macbook Pro running a dual-core 2.9 GHz Intel Core i7 processor and 8GB of 1600 MHz DDR3 RAM. On this workstation, VirtualBox (version 4.3.26 r98988) is running 64-bit Ubuntu 14.04 in a virtual environment that has 32 GB of storage space and 4 GB of RAM. The Ubuntu virtual machine was regularly updated throughout the semester using the native software updater. The InnoTek Systemberatung GmbH VirtualBox Graphics Adapter and Guest Service 3rd party drivers were used to allow resolution resizing of the Ubuntu desktop.

Version control for this project was done using git. Currently, the codebase for the the RISC-V port of mCertiKOS is stored in a private repository located at https://github.com/adamcimpeanu/cs490.

2.3 RISC-V Toolchain

The major tools for development on the RISC-V platform are packaged in the riscv-tools repository of the instruction set’s GitHub page. Instructions for installing this repository are freely available online [7].

The toolchain is organized across a number of subdirectories. riscv-gnu-toolchain contains a full suite of compiling utilities for building RISC-V binaries and linking them both against Newlib and GlibC binaries. All these utilities have the prefix riscv64-unknown-elf-.

riscv-isa-sim holds the source code for Spike, the main ISA simulator, and the project’s ‘golden standard’ for instruction behavior. For this project, Spike was preferred to QEMU, which also has a RISC-V port, both because it is lighter and easier to use for early kernel development, and because it continues to serve as the ISA ‘golden standard’. Opcodes for the simulator are stored in the riscv-opcodes subdirectory.

Also extremely important to the execution of the simulator is the RISC-V front-end server, located in riscv-fesvr. The RISC-V simulation operates using what the group at Berkeley calls the host-target interface (HTIF). fesvr is integral to this interface, as it is responsible for providing a layer that the simulator can communicate with in order to interact with the host machine, which in this case is the Ubuntu virtual machine.

The riscv-tests subdirectory contains assembly tests and benchmarks, but is actually useful because it contains a large number of examples of real RISC-V assembly, build logs, and linker scripts that proved to be very helpful when constructing the first, bare metal kernel.

For this project, the greatest resource was the riscv-pk directory. ‘pk’ stands for proxy kernel. The RISC-V proxy kernel is a program execution environment that can
host statically linked ELF binaries that have been linked with the RISC-V Newlib port, so the proxy kernel supports a more limited range of I/O features and reroutes most system calls through the fesvr. While the proxy kernel is useful for initially testing out the toolchain via a ‘hello world’ style program, the greater significance of the riscv-pk subdirectory is that the proxy kernel and the bootloader used in this project, the Berkeley Boot Loader (bbl), overlap almost entirely in their codebase. The source files in this subdirectory are extremely important for acquiring an understanding of all the machine mode bootstrapping that occurs before running any kernel on the RISC-V platform.

2.4 Other Ports

As a first attempt to demonstrate the legitimacy of the RISC-V ISA, the group at Berkeley released a port of the Linux kernel that could run on both Spike and the RISC-V port of QEMU. This project used the most recent version of the port, running on Linux 4.1.17. Details relevant to the RISC-V port are in the arch/riscv directory. Many of the source files in this directory are idiosyncratic and difficult to interpret because they must fit within the context of the rest of the Linux kernel. Therefore, it was most helpful to study the commonalities between these files and the files present in the riscv-pk directory, as there were many.

Also extremely helpful for this project were a number of difficult to find but publicly available RISC-V ports of other kernels. The most significant in terms of scope is FreeBSD. The port, spearheaded by Cambridge University research engineer Ruslan Bukin, is unique in that it is one of the only implementations that does not use bbl as a bootloader [8]. Instead, Bukin implemented all the low-level machine mode instructions himself and included them as part of the FreeBSD kernel.

The seL4 32-bit microkernel also has a RISC-V port, which is a useful reference because of its smaller codebase [9]. The Genode RISC-V port project is also a great resource because they are one of the only existing ports that has a detailed description of their porting process and some of its challenges [10]. Not used in the project but potentially useful in the future is the RISC-V port of Berkeley’s own Akaros operating system [11].

3. RISC-V Platform

3.1 Privilege Levels

In the RISC-V terminology, a running hardware thread is called a hart, and at any given moment, the hart is running using one of four privilege levels, as indicated by the mstatus control and status register (CSR). In increasing order of privilege, the hart can run in user, supervisor, hypervisor, and machine mode.

Code in machine mode is the most trusted, and therefore has the most access to hardware-level information. Supervisor and user mode take the place of kernel and user mode on more traditional hardware and exist to support the dual mode of execution for modern operating systems. Hypervisor mode support is currently a work in progress for the team at Berkeley, and is designed to support virtual machine monitors.

RISC-V was designed to be extremely flexible, and as part of this philosophy, a given hardware implementation does not need to implement all four modes. Only machine mode is required for execution
because it is the only one that has access to machine-dependent information.

The ECALL instruction requests that the hart be taken to a higher privilege level, whereas the ERET instruction moves execution down a privilege level. The last thing the Berkeley Bootloader does before exiting is call ERET so that the payload is executed in supervisor mode.

### 3.2 Registers

At all privilege levels, there are 32 integer registers labeled \( x_0 \) to \( x_{31} \), and one additional program counter register called \( pc \). Many of these registers have different abstract binary interface (ABI) names. \( x_0 \) is called zero because it is a constant 0 register. \( x_1 \) is \( ra \) and typically holds the return address for a function. \( x_2 \) is the frame pointer, \( fp, x_{14} \) is \( sp \), the stack pointer. \( x_{15} \) is the thread pointer (\( tp \)). \( x_{18}-25 \) are written as \( a_0-a_7 \) and hold function arguments. \( x_{26}-30 \) are commonly written as \( t_0-t_4 \), and are the five registers that hold temporary values.

There are also 32 floating point registers, \( f_0-f_{31} \) that are used when RISC-V is run with floating point compatibility.

In addition to these user-level registers, which can be accessed at all privilege levels, RISC-V has built in the compatibility for up to 4,096 control and status registers (CSRs). Currently, the top four bits of the 12-bit encoding space for these registers is used to delineate read and write privileges for the register based on privilege level. There are different sets of CSRs for each privilege level, with higher privilege levels being able to access the CSRs of lower privilege levels. These registers hold information about cycle counts and timers for the user level. Supervisor, hypervisor, and machine-mode CSRs also include information needed to handle traps when moving up or down a privilege level. There are a larger number of CSRs for machine mode execution because it is through these registers that machine mode instructions can access information about the CPU, hart ID, and physical memory.

### 3.3 Software Stack

Figure 1 is borrowed from a draft of the RISC-V Privileged ISA Specification, v1.7 [12]. It illustrates the different ways in which the instruction sets from different privilege levels can be assembled to create real-world software. Furthest to the left is an example of a simple Application Execution Environment (AEE), which runs an application that can run on top of the AEE and interact with it using an Application Binary Interface (ABI). This ABI would include the user-level ISA as well as a set of special ABI calls that would let the application interact directly with the AEE.

![Software Stack Diagram](image)

Figure 1: Different possible RISC-V software stacks using different privilege levels.
The middle diagram of Figure 1 illustrates what a Supervisor Execution Environment (SEE) would look like. This is the most important diagram as the SEE represents the foundational abstraction of the RISC-V mCertiKOS port. In this case, host applications interact with the operating system via the ABI instead of the execution environment directly. In turn, it is the operating system that interacts with the execution environment, this time through something that is called the Supervisor Binary Interface (SBI).

The Berkeley Boot Loader will be discussed in greater detail in Section 4.1, but this executable actually does the work to set up the SBI before loading the kernel. This particular SBI was designed to be able to query the memory map, access a basic console device, send and clear IPIs, handle IRQ masking and unmasking, and perform remote TLB shootdowns.

Similar to the vDSO concept in Linux, the functions that make up the SBI are actually mapped to fixed locations in the supervisor’s address space. They act like system calls to the SEE without the overhead of most standard system call implementations. For this reason, most kernels will have an SBI header file, like kern/lib/sbi.h in the mCertiKOS port that simply consists of a set of virtual memory mappings instead of more typical function definitions. The actual source code of the SBI can be located in risc-pk/pk/sbi_entry.S.

The final diagram in Figure 1 illustrates the largest of the abstractions, the yet to be developed hypervisor execution environment. Following the pattern of the other environments, the SBI now interacts with the hypervisor, which needs its own hypervisor binary interface (HBI) to interact with the execution environment.

3.4 HTIF

As part of the RISC-V project at Berkeley, the research team is building systems for users to interact with both tethered and standalone systems, however, their focus is on building tethered systems, which are characterized by their dependency on a host system to both boot and handle most I/O [13]. Spike was written to simulate a tethered RISC-V system, which means that it, along with the proxy kernel and many of the other components installed through the RISC-V toolchain, take part in the Host-Target InterFace (HTIF) protocol.

HTIF was mentioned before when describing fesvr. The front-end server is the host that calls the bootloading code, runs it, and helps properly map the ELF binaries of both the bootloader and the kernel payload. It interacts with the host RISC-V machine simulation at the granularity of system calls.

Inspection of the source code in the riscv-pk subdirectory reveals that a number of simple but reasonably mature operating system features are built into the proxy kernel and bootloader code, including but not limited to mmap and brk functions, a basic file system, and system calls. This is because when the target system attempts to perform these tasks within its simulation, code running on top of the front-end server must be able to interpret what the target is trying to do and translate it into commands that the host can actually perform on its physical hardware. This is particularly true of I/O, which is handled almost entirely in this manner. Note that riscv-pk/pk/frontend.c, which contains the system call code that sends system call requests from the target to the host, contains calls to a function called tohost_sync(). These system calls are then picked up by the
host in riscv-fesvr/fesvr/syscall.cc, where Spike or proxy kernel system calls are translated into actual Linux system calls for the host Ubuntu system.

4. Implementation

4.1 BBL

The Berkeley Boot Loader does much more than a typical bootloader, and so understanding every line of its execution is a crucial prerequisite to building any kernel on the RISC-V platform. What follows is a summary of the important milestones of the bootloading sequence, as this leads to a better understanding of subsequent parts of Section 4.

Via the linker script in the proxy kernel sourcefiles directory, the entry point of the bootloader is at address 0x100 of the riscv-pk/pk/mentry.S source file. This initial code zeros out the registers, sets the stack pointer, and checks the hart ID number. For the bootloader, this is the first hart, so this function directs the code to the main initialization function, init_first_hart(). This function in turn calls a number of other initialization functions. The mstatus CSR is initialized, as well as floating point mode, depending on the configuration of bbl’s build. Hart local storage is zeroed out, and the very simple target file system is initialized with three file descriptors for standard input, output, and error.

At this point, a function called parse_device_tree() is called. This function queries the hardware to acquire information about available memory and CPUs. The size of memory and number of harts is saved in two extern variables located in riscv-pk/pk/pk.h.

Virtual memory parameters are then initialized by vm_init(). Among them are the first free physical address, and first free pages for the mmap and brk utilities.

The last function of init_first_hart() is boot_loader(), which subsequently loads the ELF binary of the kernel, calls supervisor_vm_init(), prints the RISC-V logo, and enters supervisor mode while jumping to the entry point of the kernel binary. Section 4.4 will discuss vm_init() and supervisor_vm_init() in much greater detail when describing the memory management layer.

4.2 Bare Metal Kernel

A bare metal kernel is the ‘hello world’ of kernel development. It consists of the bootloader, where the payload is as simple as possible, written in assembly and outputting to the console in the most basic manner. Relevant files for this phase of the port are located at the first pull request of the project repository.

In the context of this project, a bare metal kernel is essentially bare metal assembly code that can be called by bbl. Inspiration for this step came from a post on StackOverflow that demonstrated how to run simple assembly directly on the Spike simulator [14]. The key is to make sure that when linking the binary, the .text section is set to begin at address 0x200. It is also important to make sure that the marked entrypoint in the linker script corresponds to the .globl start location in the assembly code.

Looking at an elfdump of the compiled Linux kernel image provided the next important clue, as it was linked such that the .text section began at address 0xffffffff80000000. This is the only
requirement for running an ELF binary on top of bbl. It falls in line with a convention in more modern operating systems to map kernel code in the top half of the virtual address space.

Once the design behind the SBI abstraction was better understood, figuring out how to print from the console was simply a matter of looking for assembly code in the right place. As mentioned before, the assembly definitions of all SBI functions is located in risc-pk/pk/sbi_entry.S. The printed character is placed in the a0 register, an mcall constant called 
`MCALL_CONSOLE_PUTCHAR` is placed in the a7 register (both of which are conventionally function argument registers), and the ECALL instruction is called, which makes a request for a machine mode trap to handle the mcall. This is sufficient to print a character to the console.

### 4.3 Stack and Formatted Print

The next most important step in the mCertiKOS port was write a proper entrypoint for the kernel such that the stack is properly set up, and the assembly can properly jump to the first C function. The changes involved in this phase are represented in pull request 2, 4, and 5 in the repository for this project.

The entrypoint function for the mCertiKOS port, located in kern/init/entry.S, is largely an adaptation from a file from the RISC-V toolchain test directory: riscv-tests/benchmarks/common/crt.S. In this file, integer registers are cleared, the global pointer is initialized to its linker script-defined value, and the stack is given 128KB of space. The kernel then jumps to the _init() function. An `early_printk()` function is called within, and this function calls the SBI function, `sbi_console_putchar()`. The SBI function is defined in a very simple assembly file that directly maps the function to its preexisting location in memory.

This phase of implementation also involved reorganizing the directory structure of the repo and compilation system to introduce multiple directories and a multi-directory build system. For consistency, the recursive Makefile structure used in the lab1 version of mCertiKOS was adopted for this project. Very little was changed, however, given that CompCert has not been ported to RISC-V, CompCert-related variables were removed. One consequence of this step was that the linker script was removed and replaced with linker flags.

The last step in this phase involved porting large parts of the mCertiKOS /lib directory, with its many string and formatted printing functions. Almost everything from lab1 remained intact. The `cputs()` function was modified so that it used `sbi_console_putchar()`, and few types in the string manipulation files needed to be changed because the source version of mCertiKOS is 32-bit, whereas the new port is 64-bit.

### 4.4 Physical Memory Management

This final phase involved porting the /pmm directory from lab1 to set up the mCertiKOS physical memory management layer. The work for this phase involves the changes represented in pull requests 6 and 7 in the GitHub repository for this project.

In preparation for the memory management layer, the `memcpy()` and `memset()` functions were adopted from the proxy kernel code to write `memcpy()`, `memset()`, and `memzero()` functions for the port. Some typedef’d types
had to be changed for 64-bit compatibility to avoid compilation errors.

Properly appreciating the design of this portion of the port requires going back to a portion of the bootloading code in more detail, more specifically, two functions in \texttt{riscv-pk/pk/vm.c}: \texttt{vm\_init()} and \texttt{supervisor\_vm\_init()}.

\texttt{vm\_init()} first sets two important parameters, \texttt{mem\_size}, and \texttt{current\_first\_free\_paddr}. These values are extremely important because they’re the only memory-related values that can be read in supervisor mode via the SBI function, \texttt{sbi\_query\_memory()}. The first is the size of available RAM to the SEE. The second is the physical address of the first available page, which is calculated by taking the address of the 	exttt{_end} section of bbl’s linker script, rounding it to a page size, which is a standard 4KiB, and adding one page for each executing hart. The first free page, for mmap and brk purposes is then calculated, which is not relevant to this project.

\texttt{supervisor\_vm\_init()} is a much longer function. It starts by making sure \texttt{mem\_size} is no larger than calculated range of possible virtual addresses, which is unlikely. The rest of the function is devoted to building two layers of page table entries. The page table entry pointer, \texttt{sbi\_pt} points to the primary level, and \texttt{middle\_pt} points to the secondary level and is used when creating the page table entries for the secondary level.

RISC-V has two schemes for virtual memory management, Sv32 for 32-bit machines and Sv39 for 64-bit machines. bbl implements the base component of Sv39 by setting up these page table entries. Figure 2 illustrates both the virtual addressing scheme and the structure of page table entries [10]. The 12 least significant bits give the offset of the address within the page. Bits 12 through 20, 21 through 29, and 30 though 38 are the first, second, and third level indexes of the page tables.

A page table entry has corresponding physical page numbers (PPN) for the virtual address’s virtual page numbers (VPN) as well as a host of bits for read/write permissions and a valid bit. These are the kinds of page table entries that the \texttt{supervisor\_vm\_init()} function is initializing in its two primary for loops. At the end of the function, the address to the base of the primary level of the page table is written to the \texttt{sptbr} CSR, which stands for supervisor page table base register.

The ported Linux RISC-V kernel reads from the \texttt{sptbr} CSR and builds off of the secondary level page tables when managing its own physical and virtual memory.
Querying addresses in supervisor mode indicates that both paging and virtual addresses are in use. However, these features do not restrict other kernels running on bbl from managing their own memory. This is because they are free to manipulate the paging structure and change the sptbr CSR. Moreover, very simple address translation exists when the supervisor payload is first loaded. If $\text{min\_vaddr}$ is the smallest virtual address of the supervisor payload ($0xffffffff80000000$ in this implementation) and $\text{first\_free\_paddr}$ is the first free physical address after the kernel address space, define $\text{bias} := \text{min\_vaddr} - \text{first\_free\_paddr}$. Then any virtual address $\text{vaddr} \geq \text{min\_vaddr}$ can easily be converted to a physical address $\text{paddr} = \text{vaddr} - \text{bias}$.

Because initialization of memory management is passive in this manner, much of the pmm layer from mCertiKOS can be directly borrowed from its original source code. The only major work at this level involved modifying the MATInit layer. As a preliminary step, an mboot source and header file was written to query and save memory information through the SBI. The procedure in the mem_info_init() function in many ways mimics the setup_bootmem() function in arch/riscv/kernel/setup.c of the Linux port. A memory block info struct is passed to the target SEE to query information about the size of available memory, and the smallest free physical address. Adding these two values together and bit shifting them to remove the 12 least significant bits calculates the last freely available page number. Reading the sptbr CSR, bit shifting it, and adding 1 gives the first freely available page. These page numbers of the first and last available pages are what are calculated and saved by kern/dev/mboot.c, and are later queried by the MATInit layer when assigning free and taken pages. In this manner, the pmm layer is easily ported to RISC-V, and passes slightly modified tests from lab1.

5. Conclusion

5.1 Next Steps

The motivation behind this project was to provide the initial impetus for the RISC-V port of mCertiKOS by creating enough of a foundational layer of code and knowledge to easily continue and hopefully accelerate the project. Many components still need to be built.

The first of which is a get_char module. While printing to the console is streamlined by the SBI, grabbing text from the keyboard requires more complex interactions using the HTIF protocol. The arch/riscv/htif directory of the Linux port provides what looks like a driver to help the kernel better interface with the more complex sbi_send_device_request() and sbi_receive_device_response() functions.

Then next obvious step would be to work through lab2 through lab6 of the CPSC 422 course, porting each respective layer. Important milestones include virtual memory support, user space support, and trapping and simple system calls between the user and supervisor levels. With context switches working, process management can be implemented so that the port becomes a multiprocessing kernel.

Then there are whole host of stretch goals including interrupts and preemptive multiprocessing, floating point support, and interprocess communication.

Also, the mCertiKOS port was written for a
64-bit platform only because it is easier to run such an environment using Spike and tools provided in the toolchain. However, before continuing the project, there should be an evaluation that conclusively decides on whether or not to make this kernel 32 or 64-bit. Luckily, very little of the current source code would need to be changed to make it a 32-bit kernel.

On a final node, it may be worth transitioning kernel development from Spike to QEMU as the kernel becomes more complex and more intricate debugging utilities are required.

5.2 Acknowledgments

I would like to thank Albert Ou and Sagar Karandikar at U.C. Berkeley for being available via email and providing key insights about the inner workings of RISC-V. I would like to thank Cambridge University research engineer Ruslan Bukin for telling me about his process in porting FreeBSD. Most of all, I would like to thank Professor Shao and Newman Wu, my advisor and mentor for this project for their guidance.

References


