The RISC Takers:
Final Report

Sean Keever
swkeever@uw.edu

Yokesh Jayakumar
karthj@uw.edu

John McMahon
mcmahjoh@uw.edu

Abstract

The goal of our project is to create a means of letting a user not only use an OS, but to allow a user to see what is going on inside the OS and CPU. To this end, our project aims to maximize availability to users by providing a browser-based interface that lets the user visit a webpage and visualize an OS from the moment he or she visits the page. We hope that this project can grow to be an invaluable teaching tool for instructors, engineers, or anyone.

1 Introduction

There are multiple implementations of operating systems made available via a web browser. Instead of re-implementing that functionality, we chose to find a project that has the low-level emulation from C to JavaScript already handled. This way, we can focus on solving a new problem: providing an interface to visualize the OS and CPU.

We looked at multiple different OS implementations and decided on riscv-angel, which seemed to offer the functionality we needed without too much underlying complexity. This way, we could focus more attention to creating the visualization layers without having to worry too much about the low-level emulation details.

2 Initial Design

To start, we stripped the riscv-angel project to only show the Terminal. We got rid of all the elements and libraries that were not going to be used in our project. Before we started on our implementation, we prototyped the UI in Figma, shown in Figure 1.

Figure 1: Initial prototype of design

Figma allowed us to come up with a design before we implement anything in code. We aimed for a dashboard-like interface, showing the user information about the CPU and OS in real-time. The dashboard interface would allow the user to have all the information at his or her fingertips without having to constantly type commands to get the same information.

3 Beginning Development

Setting up a development environment

One of the challenges for the team was facing ambiguity in this project. It was difficult to interpret exactly what needed to be done.

To facilitate this problem and make development easier, we started approaching the project with an Agile workflow. We are using GitHub's Kanban board functionality to track issues and milestones. We hold weekly stand-ups to find out

- what we did over the past week, and
- what we will do in the next week.

We use the GitHub issues that we created to assign tasks for each member of a group. In doing this, we were able to be more productive by having concrete milestones in place in order to complete the project.

To ensure robustness and quality of our project, we have also developed some tooling for continuous integration. To ensure consistency in the codebase, we use ESLint, a tool that we have configured to apply the rules defined in AirBnB's style guide. We use `lint-staged` to run our linters before any commit. This way, we know if stylistic errors need to be addressed before pushing code to the repository.

We also set up GitHub Actions, which are essentially hooks that execute when code is pushed to the repository. These hooks will run the style checker and our unit tests before code is pushed to remote. The hook will reject any commits if any of these actions fail. The GitHub Actions are certainly overkill for our case, but we wanted to have the infrastructure in place in order to configure our CI easily.

Choosing technologies

We want our interface to be interactive, so we chose React.js as our library of choice, which offers a means of rapidly developing interactive client-side applications. React's rendering mechanism allows for changes in state to immediately update on the browser, which provides a user-friendly webpage.

This doesn't come without its challenges, though. The existing project uses vanilla JavaScript in order to run the emulator. The legacy code accomplishes this by spawning a worker thread that handles all the emulation tasks. The main thread interacts with this worker thread via message passing. We will go into more detail about how this message passing functions.

Working with the existing riscv-angel codebase was challenging. We resolved the issues by having rendering each dashboard component separately. This way, the dashboard components can coexist with the existing riscv-angel project code easily. This allowed members to develop features in parallel, without being blocked by another person's pending change.

4 Establishing a Design System

None of the members on our team are trained designers. We had to be resourceful to learn some of the patterns of good UI design. To make this easier, we took a lot of inspiration from professionals by using the Material Design system used and maintained at Google. We imported a Material Design kit into Figma, which includes many Material Design assets such as buttons, cards, etc. Doing so allowed us to wireframe a UI prototype that we feel was closer to our vision for the app's design. Figure **??** shows the first steps of prototyping our UI using Material Design and Figma.

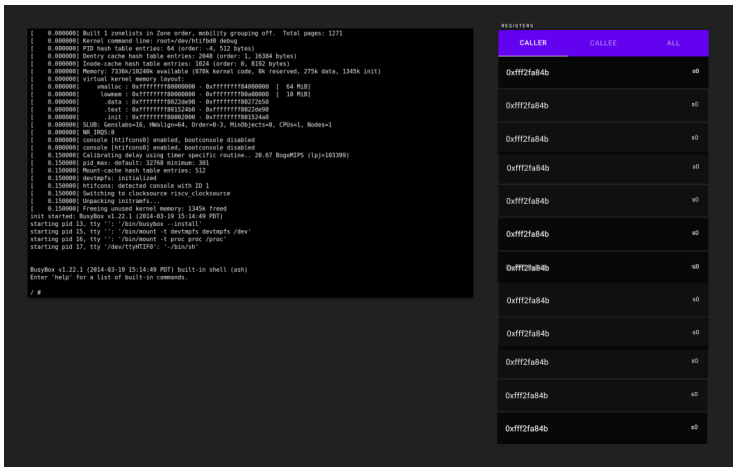

Figure 2: Initial prototype using Material Design

In addition to our team not being expert designers, we are also not experts in CSS, which is vital to making the application look good. We wanted our application to be as lightweight as possible, so we wanted to have our own custom CSS, as opposed to using a CSS framework like Bootstrap. To learn CSS and UI design patterns further, we read the book Refactoring UI, which provided many design tips from a developer's point of view.

We established a design system that is used throughout the app. We defined a system for colors, fonts, spacing/font size, and more. The system restricts properties of the UI to follow a discrete set of values. Although at first glance, it may seem that this restriction would limit our ability to create a design. In fact, the restriction provided by the design system helped to make the application feel polished and consistent.

5 Implementing a Working Prototype

A Modular Design

We describe our app as a modular, dashboard user interface that lets users use a basic OS and see properties about the OS and CPU in real time.

Our modular design means we have separate components for each visualization in the dashboard. For example, there is a separate component for the window showing the registers, a separate component for the view showing the ratio of instructions being executed, and so forth. We split our app into modular components for several reasons:

- easier to maintain and test
- allows features to rapidly developed

- offers more flexibility in UI structure

We will elaborate on each of these points.

First, because our dashboard interface uses modular UI components, whenever we work on a single component, we can be confident that breaking a single component won't break other parts of the app.

The modular design also allows us to split the work and develop features independently. This is great for us during these times of quarantine because of COVID-19. So, for example, while one team member is working on some feature *A*, another team member can be styling feature *B*. And since the components are modular, we can do so without worrying about conflicting with another person's work.

Finally, having the components act as modular entities allows us to change the layout of the UI very easily should we choose too. The components are simply independent building blocks that can be moved around as we please. Component indepedence is very useful for debugging purposes as well, as if one specific component isn't behaving as intended, then a developer can isolate the problem to that component alone. This flexibility will become very important when it comes time to finalizing our app.

Finishing our Prototype

There are two key files in the vanilla Javascript files that direct boot-up. Boot.js is run on start-up of the app and spawns a Webworker (webworker.js), which is a thread that will be running code given to it. This Webworker thread is in charge of actually computing instructions that will be passed in by boot.js. Once boot.js passes an insruction, webworker.js will load that instruction and execute it. It will then send the results of that instruction as a string to boot.js, which will then display the reuslts in the terminal.

This exchange between webworker.js and boot.js is an example of how we use message passing to pass data between what's shown on the terminal and what is being run in the cpu. Message passing will be the core method of how results from running instructions will be displayed on the webpage.

Information about the CPU state lives in the existing `riscv-angel` code, which is written in vanilla JavaScript. We bind a JavaScript object from the `riscv-angel` code to the globally accessible `window` object. To be specific, Webworker.js will send the state of the cpu to boot.js, which then binds that cpu state to the Window object, named as myCpu, which is globally accessible by the React code. We then create a custom React hook that safely retrieves this state to be used by React. To further optimize this message passing, webworker.js will only send a cpu state object to boot.js when the user is interacting with the terminal. If the terminal is idle, then no message will be passed. This is important to the speed and efficiency of our application, so that it is not slowed down by constant message passing when it doesn't have to happen. With the state of the CPU accessible in our React components, we could implement our dashboard.

In our app, we show the contents of the 32 user registers, the ratio of CPU instructions executed, and a time series graph showing memory utilization. These are expressed as React components that are stored in RegisterPanel.js, InstructionPanel.js, and MemoryPanel.js. On boot-up, the application runs index.js, which renders each of the above components. First we will look at the register panel.

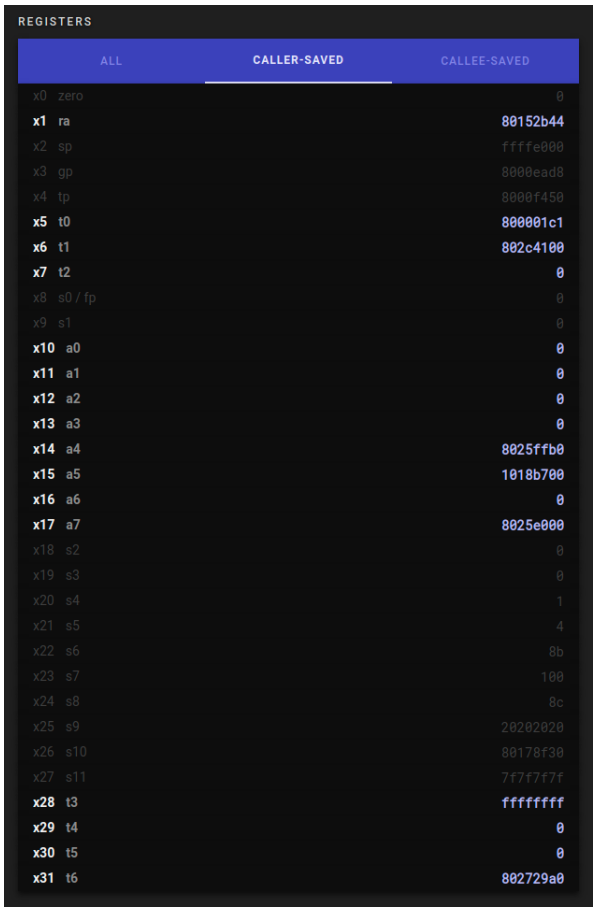

Figure 3: Register panel

As Figure 3 shows, the register panel shows the register name, ABI name, and the state of each register. The register value will be displayed in 64bit Unsigned Hexadecimal. We partition the registers into three categories: All, callee-saved, and caller-saved. The react hook use-cpu will query for the latest cpu state from the Window object, and set the local cpu state to be equal to it, on an interval. One this new cpu state is fetched into the React scope, the register panel will re-render, and display the latest state of the registers. The following components will follow the same pattern of updating.

The register panel is convenient for the user as all the information is contained in one location. The user is not forced to be in GDB mode and type commands to get the state of each

register. When the cpu state changes, the register panel updates, so the user does not have to re-type his GDB command. The register panel also allows the user to validate the proper contents of a register. One future improvement idea is to introduce a GDB mode, where the user can GDB through programs and have relevant information automatically be shown on the webpage. The user can use our register panel to see the contents of important registers, such as ra, sp, etc. Finally, the register panel is helpful to a user who may not be in GDB mode as well. If a user is running a program, the user can can keep track of the stack pointer and make sure it stays within a relative area, as this could signify that the program is running as intended, and didn't drastically change as a result of overflow.

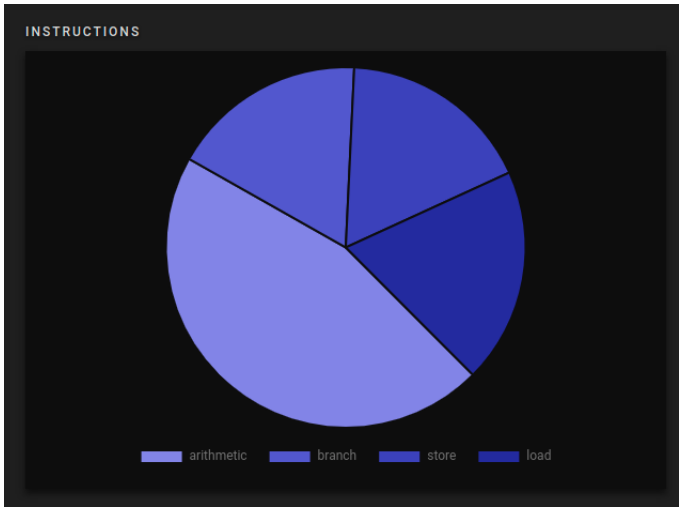

Figure 4: Instruction panel

Next is the instruction panel shown in Figure 4. The instruction panel shows a running average of four types of instructions: Arithmetic (ADDI, ANDI, etc), Control Transfer (JAL, JALR), Store (SW, SH, SB, etc), and Load. This lets the user see the ratios of each type of instruction and offers insight into what's happening beneath the surface. This information can be used by researchers or hardware engineers to optimize around this ratio. For example, if it looks like arithmetic instructions occur the most often, those developers can design hardware that prioritizes computation of those instructions over the other types. We actually found that the Memory Ordering instructions occured the least, which was always less than 1% of the total number of instructions being run. Since the ratio always stayed less than 1%, we decided to omit graphing this type of instruction usage. Memory Ordering instructions are commonly used to order device I/O and memory accesses as viewed by hardware threads, and the base linux vm is being run on one process, which is why this type of instruction is barely used.

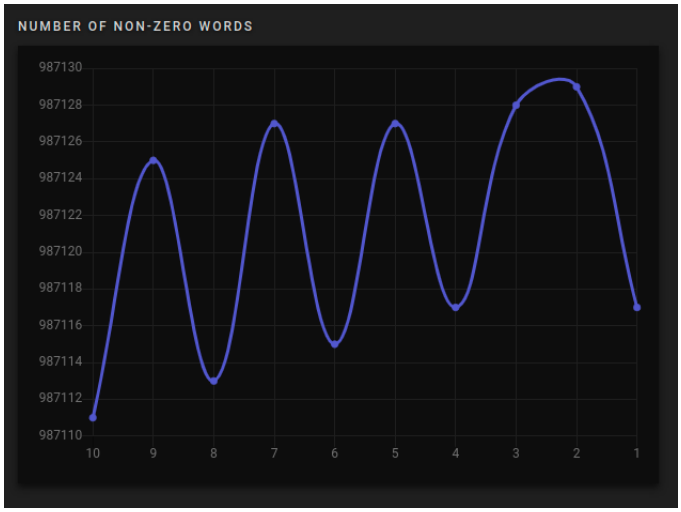

Figure 5: Memory panel

The memory panel in Figure 5 displays the number of nonzero words in the full memory region and shows the percentage of words that are not zero. The upper and lower bounds of the graph also dynamically change such that changes in nonzero word count are visible to the human eye. This is especially evident when the user creates, modifies, and deletes a file. This graph allows the user to see how much memory has been used, and how much memory is left in the operating system. This is a useful metric so a user will know if he or she has enough space left to download a file. As a user uses the terminal to create and edit a file, the upper trend of word usage that will stay constant in the graph is useful so the user can be aware on-the-fly of how much space is left. Finally, if the user wants to attempt a risky deletion process (clearing out many unused directories), the graph will show a steep decline, as many bytes of memory will clear up on deletion. If the user accidently deletes something without intending to, this graph's steep drop will alert the user when otherwise the user may not know such deletion ocurred.

We feel that this set of features offers a useful glance at what is going on under the hood of an operating system utilizing the RISC-V instruction set.

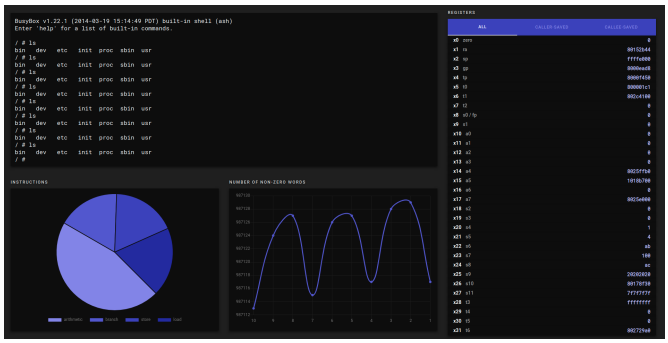

Figure 6: MVP

Deployment

After our prototype is complete, we prepare our application for deployment. To keep things as simple as possible, we host the app on the CSE department's servers at University of Washington. In this way, this project can easily be used as an example in future iterations of the CSE481a capstone course.

Next Steps

This project has a lot of room to grow. There are features still to be desired. Namely,

- A way to see inside the file system
- A way to get information about running processes
- An interactive GDB mode

One future goal is to bridge the gap between the vanilla Javascript files and our React code. Instead of message passing the state of the cpu every time there is an update, the ideal architecture would have the cpu code within the React scope, such as the cpu state changes, it causes the React components to re-render. Furthermore, if we are able to encompass a given OS binary into the React scope, a future goal is to allow any OS binary to be swapped with the base riscv linux vm. This will allow users more flexibility to see metrics on any operating system of their choosing.

We feel our project provides a modular foundation for these features to be developed in the future. In hopes to reinvigorate life back into the riscv-angel project, we submitted a pull request to merge our extensions into the base riscv-angel branch, from which this project is based on.

Conclusion

In this project, we've taken an existing project, riscv-angel, and extended it to display an interactive dashboard that lets users see the internals of an OS and CPU running on a virtual machine. We believe our project could be used by educators to easily show students properties of an OS in real time. We designed and developed this application with modularity in mind. Our React component-based architecture allows features to rapidly developed and added to the UI. We believe this application has the potential to reinvigorate interest in the original riscv-angel project. At the very least, people can use our app to learn and experiment with an operating system running on a RISC-V architecture with very little barrier to entry.

Acknowledgments

Thanks to all of the contributors of riscv-angel, in which this project is based on.

Availability

This project is open-source and is available at https://github.com/swkeever/riscv-angel-extended