EzASM

Introduction

EzASM is an assembly-inspired language designed specifically for teaching computer organization. While it bridges the gap between C and actual assembly languages, EzASM incorporates features typical of modern high-level languages. These include terminal functions, function invocation using labels, both stack and heap memory allocation, and pointer dereferencing. Instead of aiming for speed or efficiency, the primary goal of EzASM is to offer a beginner-friendly introduction to assembly languages, steering clear of the complexities of x86.

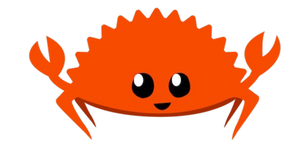
Developed with Java 17, EzASM provides an integrated environment complete with a dedicated text editor, syntax highlighting, an emulator, register view, and a memory viewer—all wrapped up in a user-friendly GUI. Additionally, there's a headless mode that allows for typical interpreted program execution.

Objectives

This semester, our ambition is to transition EzASM from Java to Rust. Java, with its design emphasizing readability, has some inherent challenges—especially when it comes to memory-intensive tasks like string handling and array operations. The language's high memory consumption, combined with the mandatory synchronization in its stream-based I/O, presents clear inefficiencies. Additionally, the recurrence of breaking updates in Java mandates regular code amendments, amplifying maintenance efforts.

On the other hand, Rust stands out with a series of compelling features. It ensures memory safety without relying on a garbage collector, leading to more consistent and efficient execution. Its 'immutable by default' philosophy encourages a more deliberate approach to data manipulation, enhancing both code clarity and reliability. Furthermore, Rust's ownership system elegantly sidesteps data races, making concurrent programming both safer and more intuitive. Also, moving away from the software bloat often associated with the JVM, Rust compiles directly to machine code, offering minimal runtime overhead.

Given Rust's blend of safety, performance, and backward compatibility, it emerges as an optimal choice for advancing EzASM's development.



Structure

EzASM was originally made using Java. We started porting it to Rust this semester to improve on the performance of the application, using React along with Rust for the UI. As we plan to fully move to the Rust version of EzASM, REzASM, we've been working on porting Java code to Rust while still maintaining the Java version, finding any bugs currently in it.

REzASM has 2 different cores: the main core (rezasm-core) and the web core (rezasm-web-core). rezasm-web-core contains all the functionality needed for the frontend GUI to interact with rezasm-core, while rezasm-core is where all the main functionality for REzASM lies. It contains code for simulating assembly, parsing user input, implementing instructions, and more. Specifically, rezasm-core contains the following:

- Utilities: Utilities such as errors and data representation that is used everywhere
  - Errors include divide by zero, reading out of bounds, negative memory addresses, and more
  - Data is represented as raw data, a vector of integers or floats
  - Word size of data is determined by the WordSize enum
- Simulation: Contains code about how registers and memory are represented. Everything gets stored in a simulator during runtime
  - During runtime, everything about the user's program gets stored in a simulator. The simulator contains everything about register data, memory, and more
  - Registers ($S0..9, $T0..9, $PC, etc.) are all defined by the instruction registry
- Parser: How user input gets parsed
  - The input is parsed by getting every line and going through every string in the line, determining what the string could possibly be (instruction, label, register, dereference, etc.)
- Instructions: How instructions are implemented and inputs and outputs are defined
  - Arithmetic instructions: add, sub, mul, div, ...
  - Float instructions: addf, subf, mulf, divf, ...
  - Comparison instructions: seq, sne, slt, sle, ...
  - Branch instructions: beq, bne, blt, ble, ...
  - Memory instructions: push, pop, load, store, ...

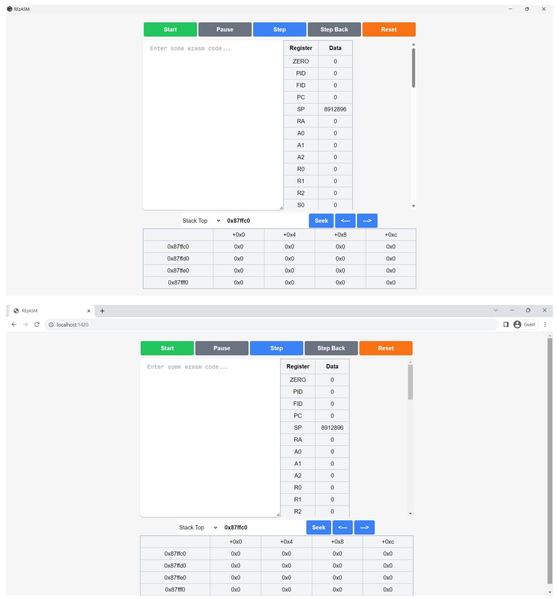
REzASM's GUI is made with React. The GUI's components are:

- Code Area: Where the user can input assembly code to be run
- Memory View: View all memory addresses and data stored within
- Registry View: View all registers and data stored within
- Toolbar: Where the user can start, pause, step through, and reset their program

REzASM has 3 different ways of being run: tauri, wasm, and CLI. Tauri is used to run the desktop application version of REzASM, wasm, or web assembly, is used to run REzASM on a website, and CLI is used to run REzASM from the command line. Each version of REzASM uses different parts of the rezasm-core and rezasm-web-core. For example, the tauri and wasm versions of REzASM will use the rezasm-web-core while the CLI version won't.

Progress

So far this semester, we've worked on error handling, instruction handling, GUI components, and importing and fixing instructions. We've implemented a lot of the basic instructions used in assembly, debugging any issues found along the way. We've also created GUI components (memory view, registry view) to help the user view the data being stored in memory and registers.




```
lazy_static! {
    pub static ref ADD: Instruction =
        instruction!(add, |simulator: Simulator,
                           output: InputOutputTarget,
                           input1: InputTarget,
                           input2: InputTarget| {
            let value1 = input1.get(&simulator)?.int_value();
            let value2 = input2.get(&simulator)?.int_value();
            let k = value1 + value2;
            return output.set(simulator, RawData::from_int(k, simulator.get_word_size()));
        });
    pub static ref SUB: Instruction =
        instruction!(sub, |simulator: Simulator,
                           output: InputOutputTarget,
                           input1: InputTarget,
                           input2: InputTarget| {
            let value1 = input1.get(&simulator)?.int_value();
            let value2 = input2.get(&simulator)?.int_value();
            let k = value1 - value2;
            return output.set(simulator, RawData::from_int(k, simulator.get_word_size()));
        });
```

```
#[macro_export]
macro_rules! instruction {
    ($name:ident, |$simulator_name:ident: Simulator, $($names:ident: $types:ty),*| $func:tt) =>
    ({
        let mut v: Vec<std::any::TypeId> = Vec::new();
        $(v.push(std::any::TypeId::of::<&mut $types>());)*
        fn $name($simulator_name: &mut crate::simulation::simulator::Simulator, types: &Vec<std::
            let mut _counter: usize = 0;
            $(
                #[allow(unused_mut)]
                let mut $names: $types = match arguments[_counter].clone().try_into() {
                    Ok(value) => value,
                    Err(error) => return Err(error.into()),
                };
                _counter = _counter + 1;
            )*
            $func
        }
        let mut instruction_name = std::stringify!($name);
        instruction_name = instruction_name.trim_start_matches('_');
        Instruction::new( instruction_name.to_string(), v, $name )
    });
}
pub use instruction;
```

Conclusion

Transitioning from Java to Rust presented challenges due to our reliance on object-oriented features like polymorphism and reflection, which aren't inherent in Rust. While this necessitated rebuilding numerous features, the end result was a more streamlined and optimized codebase.

References

https://github.com/ezasm-org/ezasm

https://github.com/ezasm-org/rezasm

Acknowledgements

Prof. Kuzmin, Trevor Brunette, Wyatt Ross, Nathan Paul, Sebastien Brand, Michael Ni, Hui Zhang, Siyan Zuhayer