EmuGator Alpha Test Plan

Fiona Cahalan Nikhil Iyer Liam Rosenfeld

Rohan Simon Christopher Tressler

I. Front-end

A. Browser Compatibility

To ensure accessibility for all students, EmuGator will be tested on all common web browsers.

a) *Test Procedures:* On each web browser, test adding break points, assembling the example program, viewing the registers and data memory, and stepping through the program. The web browsers to test this on will be: Google Chrome, Safari, Microsoft Edge, and Firefox. Other common browsers include Opera and Brave. However these are both Chromium-based, like Google Chrome, so they will not be tested.

b) *Expected Behavior:* EmuGator is expected to behave exactly like it does on Google Chrome. When the user clicks “Assemble” a purple “Next Clock” button should appear. The instruction memory and data memory should be populated correctly, the registers should be viewable, and a blue outline should appear around the instruction that will be ran next, matching the program counter. As the user steps through the program by clicking “Next Clock”, the instruction memory, data memory, and register views should update accordingly. The blue outline should move with the program counter, and an orange outline should appear when performing a jump instruction.

B. Usability Testing

a) *Test Procedures:* We will ensure that EmuGator is useable by asking peers (who were not involved with the development of EmuGator) to perform actions within the user interface. We will ask them to upload a sample text program, assemble the program, step through the program to completion, check the register view, and toggle between instruction and data memory views.

b) *Expected Behavior:* It is expected that the interface will be intuitive for users, even those new to running assembly code. Thus, users are expected to be capable of performing all five basic tasks. We will log all users’ results and take special note of any tasks that could not be completed by standard users. We will use this to continually improve the front-end design.

II. Emulator

A. Instruction Execution

a) *Test Procedures:* Automated test cases for the 2-stage pipeline for every instruction the emulator can execute have already been created, and are run by executing `cargo test` in the project root, and is automatically run on each push using GitHub’s CI/CD.

b) *Expected Behavior:* After every clock step within the test cases, an assert statement is present to ensure that the emulator state gets updated accordingly. After a once over on the test cases, these asserts should continue to pass as it did before. No assertions should fail.

III. Assembler

A. Automated Functional Testing

a) *Test Procedures:* Currently, 39 unit tests have been created that assemble several different programs. These programs are inclusive of every currently supported instruction. They happen automatically using Rust’s built-in testing framework. The output of the assembler is three memory maps. Each output memory map is compared to a manually calculated theoretical value. For further randomized testing, we plan to implement some randomized instruction testing using a framework such as Proptest. These tests are also executed by `cargo test` in the project root and GitHub’s CI/CD.

b) *Expected Behavior:* The expected behavior is that no matter the instructions or memory allocation specified, the assembler produces three correct memory maps (instruction memory, source map, and data memory). This is automatically verified.

IV. Full System Testing

In addition to the isolated tests for each module, we will also create a test bank for all of the modules connected together. It will test providing source assembly and an optional breakpoint, and verify that all of the state is as expected when the breakpoint or end of program is hit. This architecture can additionally be reused for the auto-grader tool we are planning to create a secondary target for at some point.