Software Formalization

Year: 2025 Semester: Winter Team: 15 Project: αCassiopeiae 8800

Creation Date: ­2/29/2025 Last Modified: February 22, 2025

Author: Seth Deegan Email: sdeegan@purdue.edu

Assignment Evaluation: See Rubric on Brightspace Assignment

1.0 Utilization of Third Party Software

We intend to use the [rp-rs RP2350 HAL](https://github.com/rp-rs/rp-hal/tree/main/rp235x-hal) Rust crate to program our microcontroller. [1] It is a Rust-based HAL that fully supports configuring our RP2350 microcontroller. This library is released under the MIT license and is free to use without requiring attribution.

As a tool to first test our CPU and memory cards without requiring us to write the emulator ourselves first, we plan to use the [rs80 8080 Rust-based emulator](https://github.com/cbiffle/rs80). [2] We will modify the source code so that instructions and data are retrieved and executed in response to external machine cycles (clock and bus signals). When the functionality of the CPU has been verified to be correct, we will write our own emulator to be dropped in as a replacement. This software does not specify a license, however it is publicly available, and it will not be included in our final implementation, so it is permissible to use in the development stage of our project.

We will utilize the [probe-rs](https://probe.rs/) embedded toolkit to program and debug our microcontroller. [3] It comes with a VSCode extension which we will use to assist in debugging. This toolkit is released under the MIT license and is free to use without requiring attribution.

2.0 Description of Software Components

The packages involved in our project will include ”tram”, “cycle”, “recall”, “ubahn”, “interchange”, and “runner”. These are all cargo (Rust’s package manager) packages that are placed in a cargo workspace in our repository.

Tram will be a common interface between the software dedicated for each card and the 8080 style bus. It will be a series of function pointers and provide callbacks to run functionality specific to each card when bus signals are to be received and sent. It will be responsible for instantiating the microcontroller’s programmable IO (PIO) state machines that will handle reading and writing to our 8080 style bus without requiring a dedicated 8080 bus interface or bit banging. Data read or written will be sent from PIO over the FIFO lanes and to the CPU. An interrupt will be triggered for every bus interaction to read from the FIFO lanes. Read and write functions will be created to handle this.

Cycle, recall, and ubahn will be the code to be run on the CPU, memory, UART cards respectively. Interchange will be the code to run on the front panel. Each package will instantiate tram to communicate with the bus and connect internal functions to respond to bus signals.

Runner will be our implementation of an 8080 emulator. It will remain separate from the cycle package for abstraction and testing purposes. During development, we will use the rs80 package instead of runner. Runner will include a struct to define the CPU’s state (this will include the registers, pc, and flags) and a function to execute an instruction that will update such state. Additionally, functions for reading and writing to external memory will be created and used by the execute instruction function for memory read and write instructions.

An example of how our packages will interface with tram and expose functionality can be shown by the pseudocode below. Cycle will instantiate an instance of tram and runner and connect the two. Additional functions in cycle or other card-specific packages can be defined in the package to respond to signals from tram.



Interchange will need to process signals from our I2C interface coming from the IO expanders for the front panel lights and switches. It will simply convert these signals to bus signals.

3.0 Testing Plan

Software testing for pure software components can be done in software. This would apply to all of our packages in some way. For example, for a given series of bus signals, the output signals from the CPU simulator will be exactly the same. A software test harness like this is developed to perform hardware-out-of-the-loop testing for CPU, RAM, Front Panel, and UART simulator correctness.

Software testing for bus communication logic (tram), requires hardware-in-the-loop testing, because of the use of PIO assembler routines. These routines can first be tested hardware-in-the-loop on a single board, where the board “talks to itself” through its own pins, and can then be expanded into two (2) board, and then three (3) board setups.

Once these software components are independently validated, we perform end to end (E2E) testing of all components, which is the integration of all simulators (CPU, RAM, Front Panel, UART) with the communication routines. E2E testing can be staged, where an initial test with both CPU and RAM simulators can be loaded onto the same board, and communicate by “talking to itself”. Further E2E stages include additional simulators, and split the functionality across separate boards, so to test multi-board correctness.

Because all of our code will be done in Rust, we will utilize the cargo Rust testing framework to write tests. Tram is a critical part of our design and will be written and tested first before all other software and tests. The remaining tests will have equal importance as most depend on each other and the final testing of the E2E system is what matters.

4.0 Sources Cited:

# Bibliography

|  |  |
| --- | --- |
| [1] | rp-rs, "rp-hal," [Online]. Available: https://github.com/rp-rs/rp-hal/tree/main/rp235x-hal. [Accessed 22 February 2025]. |
| [2] | cbiffle, "rs80," [Online]. Available: https://github.com/cbiffle/rs80. [Accessed 22 February 2025]. |
| [3] | "probe-rs," [Online]. Available: https://probe.rs/. [Accessed 22 February 2025]. |

Appendix 1: Software Component Diagram

*A screenshot of a computer screen

AI-generated content may be incorrect.*