Software Overview

Year: 2025 Semester: Spring Team: 15 Project: αCassiopeiae

Creation Date: ­1/27/2025 Last Modified: January 27, 2025

Author: Caleb Shinkle Email: cshinkle@purdue.edu

Assignment Evaluation: See Rubric on Brightspace Assignment

1. Software Overview

1.1 Emulator for Intel 8080 CPU

The Altair 8800 originally used an Intel 8080 CPU as its processor, however for our project we are not using an actual Intel 8080 for our design, so we will be emulating it in software using Rust on a Raspberry Pi microprocessor (RP2350). The software will emulate the entire 8080 instruction set, which means that the software will mimic the original 8080 instructions and drive I/O pins on the RP2350, so that the instructions can communicate with the bus, access memory, and run programs. Instructions can take various numbers of clock cycles to complete, so the emulator will need to complete one instruction before fetching another from memory.

Additionally, since the clock for our system comes from the front panel, our emulator will need to listen to the clock from the front panel instead of just using the clock from the RP2350 that it will be running on. The clock should be able to switch between 2 MHz, single-step, and off, so the emulator needs to handle that so that the behavior of the Altair 8800 is mimicked as closely as possible. See Appendix 1.1 for a full flow chart of the emulator’s desired behavior.

With everything put together, the emulation software should listen for the clock pulse, execute instructions, and fetch instructions/data from the bus if necessary. The microcontroller will use the emulator as shown in Appendix 2.1: once a program starts it will run the emulator, execute the program, and return to an idle state.

* 1. Programmable I/O (PIO) for Raspberry Pi 2350

We will need to write programs to control the PIO blocks on the RP2350 for our design. The state machines responsible for controlling the PIO pins use a form of assembly with nine instructions, so we will have to write some assembly programs to control our I/O. By programming the PIO blocks on our RP2350, we will be able to “handle data coming in or out of the microcontroller, and offload some of the processing requirement for implementing communications protocols” [3]. They also provide descriptions of all the instructions that are available for use, for example “OUT shifts 1–32 bits from the output shift register to somewhere” [3].

We plan on using this PIO for bus control, which will allow the various components of our design to communicate with each other, including the CPU emulator, RAM, and front panel I/O (?).

* 1. Front Panel Software

We will also need to write some Rust code to handle input and output to and from the front panel. This will work with our emulator to make sure the emulator is listening to the clock signal provided from the front panel.

1.4 RAM Software

We will also be writing some Rust to control the RAM for our system. This software will make sure that the RAM will work with the 8080 bus.

2.0 Description of Algorithms

Instruction decoding will need to be handled by the emulator. When an instruction is fetched from memory, the emulator will need to break the instruction down into its components using lookup tables for opcodes, registers, etc.. Reference [1] shows the opcodes for every instruction in the 8080 instruction set, which our emulator will need to decode. For instance, if we provide the opcode 0x4, the emulator should execute the INR B instruction, which increments the value in the B register. Once instructions are decoded, the emulator will mimic the execution of the instructions and take the required actions on the RP2350’s I/O pins.

Additionally, as a stretch functionality if we have time, we plan on implementing UART, so we will need to follow the UART protocol. UART packets have a specific format and timing that must be followed, there are more details about that in Section 3.0.

3.0 Description of Data Structures

Other than the lookup tables for instruction decoding, we will not be using any meaningful data structures for our design. As a stretch functionality, we want to have two systems communicating over UART, in which case we would need to use the UART packet format to send data between the computers. A UART packet has an initial start bit, 5-8 bits of packet data, an optional parity bit, and a stop bit. If we get to it, we will likely use software to manage UART communication between two machines, Reference [2] provides a very thorough explanation of the UART protocol as well as the format of the data packets.

4.0 Sources Cited:

[1]

“Intel 8080 Opcodes,” *RetroComputerInstructionManual*, 2025. <https://grantmestrength.github.io/RetroComputerInstructionManual/intel8080.html> (accessed Jan. 27, 2025).

‌

[2]

Charlet Priscilla, “‘Understanding UART: A Complete Guide to Universal Asynchronous Receiver-Transmitter’ – Pantech.AI,” *Pantech.ai*, 2019. <https://pantech.ai/understanding-uart-a-complete-guide-to-universal-asynchronous-receiver-transmitter> (accessed Jan. 27, 2025).

‌

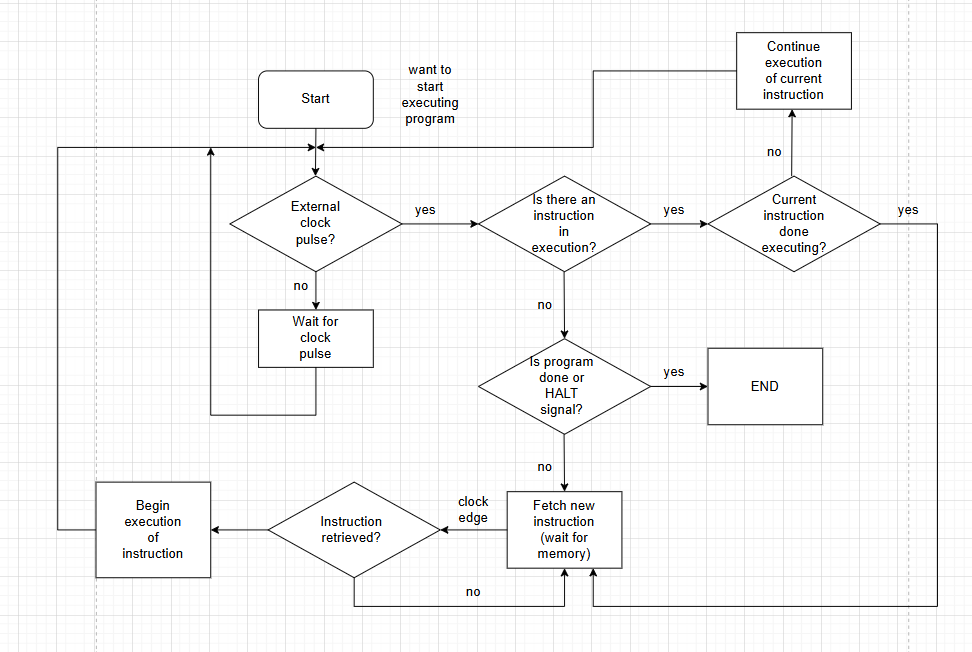
[3]

“What is Programmable I/O on Raspberry Pi Pico? — HackSpace magazine,” *HackSpace magazine*, Jan. 21, 2021. <https://hackspace.raspberrypi.com/articles/what-is-programmable-i-o-on-raspberry-pi-pico> (accessed Jan. 29, 2025).

‌

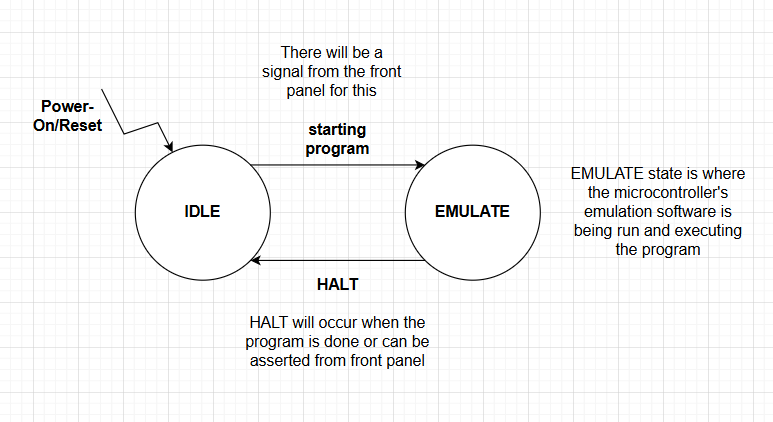
Appendix 1: Program Flowcharts

Appendix 1.1: Emulator for Intel 8080 CPU

**

Appendix 2: State Machine Diagrams

2.1 Basic Microcontroller Behavior FSM

**