C Project Final Report

Dorian Turner, Ashwin Biju, Michael Song, Timofey Kolesnichenko

Abstract: This report outlines our group's division of tasks and coordination strategy for developing a C-based ARMv8 assembler, an assembly program to flash an LED via a rPi, and our extension. It highlights our teamwork dynamics, describes the emulator's structure and potential reusability for the assembler, and identifies anticipated challenges with planned mitigation approaches.

1 Part II: Assembler

Learning from the lack of proper planning for the emulator, we started off with a plan on how to tackle the assembler. We agreed that prioritising the coordination of the group is most important, so we first designed an intermediate representation. This allows the parser and the assembler stages to be decoupled, and makes the process easy to debug and legible. Additionally, all the assembler functions would have the same signature, allowing for an elegant function-pointer based approach in the final assembly pipeline.

1.1 Architecture of the assembler

We aimed to implement the assembler through a main loop that calls abstracted functions to perform the full sequence of steps to assemble a program, reading in an assembly file, forming a symbol table, then parsing, tokenising and encoding the instructions line by line, writing them out to a binary file.

The parsing stage involves converting the ARMv8 assembly instructions into structured intermediate representations that are to be encoded into machine code. The process begins with a symbol table population pass, where the assembler scans the input file to identify labels using regular expressions and maps them to the corresponding instruction addresses.

The parsing stage involves converting the assembly instructions into intermediate representations that can be encoded into machine code. The process begins with a symbol table population pass, where the assembler scans the input file to identify labels using regular expressions and maps them to the corresponding instruction addresses. The majority of the tokenising and parsing are handled within the `parseLine` function which tokenises each instruction line into at most 6 tokens to accomadate with optional shift modifiers and address formats. There are a series of comparisions that then take place to match the opcode against the known set of assembly mnemonics, which once identified can move on to parse both immediate and register operands. An `Instr` union is populated and returned by this function which will be encoded later in the loop.

1.2 Interesting Implementation Details

We opted to go for a 2-pass architecture as this simplifies the logic. Research of existing assembles revealed that most are 2-pass, or N-pass to be precise (to allow for optimisations) with very few 1-pass. This reinforced our decision.

The symbol table took the form of a hashmap from strings to integers (being memory addresses). We debated whether binary search trees or hashmaps are better for our use case. We agreed that hashmaps are the more performant solution and are the data structure of choice for real assemblers. Despite our test cases being very simple with very few labels, we wanted our assembler to be suitable for real use cases.

For the parser, we decided that it made sense to perform parsing and tokenising within the same function and use a bottom-up approach to create helper functions that progressed from removing whitespace from a line, to parsing a register, to parsing a line, to parsing a file.

When deciding on how to assemble instructions, looking at what we did within the emulator to decode instructions gave us inspiration to do the reverse. We would identify the parts relevant to each instruction, taken from the intermediate representation, and then create 32 bit integers with the binary values for relevant parts like opc, opr, sf, etc. Once we did this, we would return the result of all of these elements bit-wise or'd together. This proved to help a lot with readability and debugging as every element of the assembled structure was separate, so it was straightforward to fix bugs.

1.3 Test suite and debugging

We decided that the best approach for our team to debug each other's code was for us to all meetup in person.

We used a systematic approach to find and fix errors. We would identify a failing test case, identify what elements were failing with gdb, and then perform a sort of binary search where we went through the main call chain, and verified the behaviour we wanted, halving our search space each time.

This is where our focus on clarity of code and compartmentalization of code became useful. As once the error was identified, it was typically a small change to get the intended behaviour.

2 Part III: LED

Our solution to Part III is fully parametric, allowing arbitrary pin selection and timing via `.int` directives. This necessitates runtime computation of the correct memory addresses and appropriate bitmasks for register manipulation.

To set the selected pin as output, we first calculate the correct memory address for the `GPFSEL` register, adding 4 to the memory address of the first bank of the `GPFSEL` register, and subtracting 10 from the selected pin, until we reach the right set of addresses (pin < 10). Next, we left-shift the bitstring `001` to the appropriate position within that register, for the selected pin. Because dynamically-sized shifts are unsupported, we had to use a loop to achieve this.

A similar strategy was used to calculate the bitmasks and offsets for the `GPSET` and `GPCLR` registers. If the pin number is ≥ 32, the registers' memory addresses were incremented by 4 and the effective pin number decremented by 32. The bitmask was set by left-shifting `1` by the remaining pin number. Once the masks are generated, the `GPFSEL` bitmask is written to the calculated `GPFSEL` memory address and the selected pin is set to output.

The main loop follows a straightforward sequence. First, `0` is written to the clear register, and the generated mask to the set register, turning the LED on. A delay loop then executes, controlled by a pre-defined iteration count, using `and w0, w0, w0` as a no-operation to consume clock cycles. After the delay, the LED is turned off by writing `0` to the set register and the mask to the clear register, followed by another delay cycle. The loop then restarts indefinitely.

3 Part IV Extension: Cocktail Machine

3.1 Overview - Classy Unit Mixer

For our extension we chose to design and implement an automatic cocktail mixing machine, which provides the user with a selection of cocktails to choose from, after which the machine will dispense the required amounts of each relevant liquid into the user's cup. The machine utilizes peristaltic pumps to transport liquids from bottle to cup via food-safe silicone tubing. The RPi is responsible for controlling which pumps are activated based on the user's selected drink. An example of use of the Classy Unit Mixer could be that it serves drinks at the Union Bar instead of a barman.

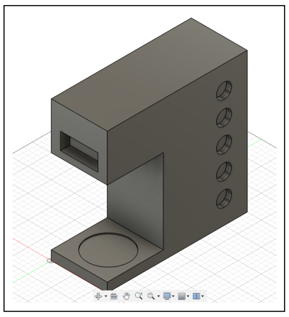
3.2 Hardware Design and Challenges

Being Computing students, designing, implementing and testing the hardware for the extension was challenging and unfamiliar, and required preparation and research. After several hours of research, we concluded that the chassis was to be made using CAD software and a laser cutter to cut out wooden plates with notches, which could be slotted together to form the full chassis. The circuit was designed as 5 peristaltic pumps wired in parallel, each connected to a MOSFET which control whether each pump is on. This was wired together using solder and placed inside the chassis.

We ran into several challenges when manufacturing this hardware. The first was when sourcing the tubing, we found that the material we chose at first was too stiff to use in the pumps. We overcame this by 3D printing several hosing connectors to connect our stiff tubing to newly sourced flexible food-safe tubing, meaning the pumps now worked and we could still use our original tubing material in our design.

Another challenge we were met with was getting to grips with using CAD/CAM software and machinery such as Autodesk Fusion and the laser cutter. None of our group were familiar with using such tools, so the better part of 2 days were spent focusing on getting familiar with using them. We then used the software to design and manufacture a chassis for the machine.

During initial testing of the electronics, we ran into an issue regarding the wiring of the MOSFETs connected to the pumps. Using a multimeter, we eventually discovered that the problem was a misunderstanding of the way MOSFETs were wired up compared to normal transistors. Once this was resoldered and fixed, we found there was a separate issue stopping the whole thing from working, which ended up being that the RPi and the machine itself did not have a common GND voltage. We fixed this by wiring together the GND wire for the machine to GND on the RPi. However, we were subsequently followed up by a new problem surrounding power distribution of the pumps. When multiple pumps ran in parallel, they would be pulling considerably less weight. After some investigation, we discovered to our disappointment that the 2A power supply we had sourced was not in fact doing what it promised, and was drawing less than 2A. This was fixed by switching it with a more reliable 5A power supply.


(a) Final CAD Design

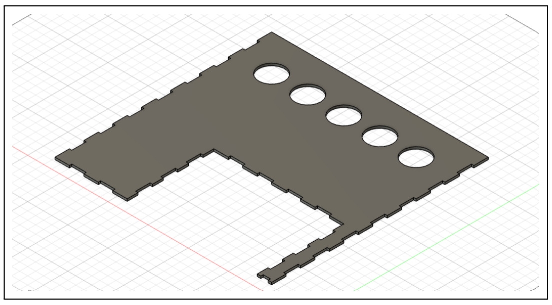

(b) Exploded Face with notches

Figure 1: CAD Design

3.3 Software

We heavily utilised the `pigpio` library for GPIO interaction, as we did not feel that such low-level control was within the scope of the extension. Additionally, this library has been extensively tested, meaning few issues are likely to arise from GPIO interaction, limiting the bugs we could encounter. The software was designed as a finite state machine, as this would simplify the menu and dispensing logic.

In order to make the software more maintainable and expandable for the future, it was chosen to make the state logic and effects of the logic as decoupled as possible. Button input is handled through the use of `pigpio` callbacks on button state changes. To avoid issues with async functions, these buttons would only change the current state. The loop inside the `main` function would then handle the state changes according to the FSM diagram in Figure 2a. All state transitions are detected in the main FSM processing loop and the string corresponding to the new state is written to the LCD.

If the new state is Dispensing, then the previous state must have been the selected drink. This calls an async function, for the purposes of allowing the main loop to continue processing the FSM changes. This async function spawns a thread for each active pump, corresponding to the ingredients of the selected drink. This approach was favoured as we elected to make the system dispense the drinks in parallel, instead of sequentially, in favour of speed. Each thread is responsible for turning its pump on for a specified duration. These threads are all joined to ensure that the whole dispensing process only terminates once all pumps are off.

Due to electrical noise and the sensitivity of GPIO input, special debounce logic was necessary to filter out button presses too close in proximity, limiting double-input. [TODO: talk aboit this later as it is not fully fixed]

Clever preprocessor tricks were used in order to make the modification of the drinks simple. A single `#define` block defines all the states, their strings and the ingredient ratios. The enums and arrays are generated from this using macros. The pouring logic is similarly parametric, with all pouring times and volumes defined as constants.

All envisaged errors are cleanly handled, with extra logic detecting possible glass overflow, and handling all initialisation and `malloc` errors.

After the software was complete, the executable was installed as a `systemd` service and managed by `systemctl`. This was done to allow the machine to run automatically on startup, and to automatically restart the program in the event of a crash, as you would expect of a machine.

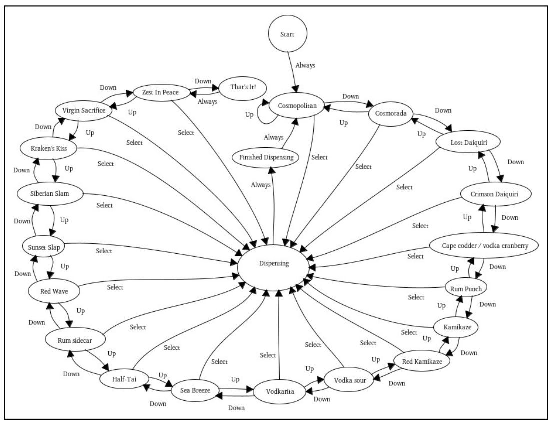
3.4 Testing and Debugging

As the software and hardware were developed in parallel, it was necessary to test the software before the hardware was complete.

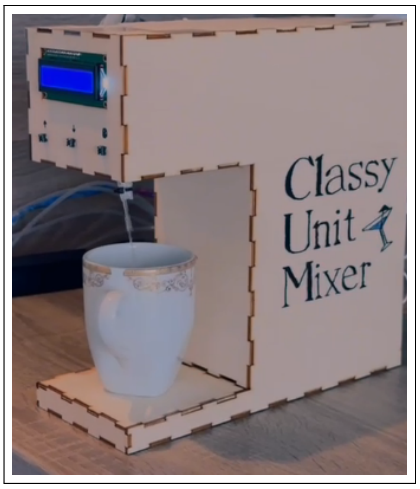
The first stage of software testing was local, on the Linux machine used for development. It was not possible to install `pigpio` on this machine, so it was necessary to create a dummy header file to emulate GPIO functionality. This was implemented through `stdio` output.

A breadboard was utilised for testing the outputs and correctness, with buttons for input (as they would actually be in the final product) and LEDs for showing pump outputs. The LCD was also connected to see the printed text.

Testing our machine involved calculating flow rate of liquid given by the pump, so that we could calibrate our program with actual values corresponding to the volume of liquid dispensed by the pump.



(a) FSM Diagram



(b) Final Product

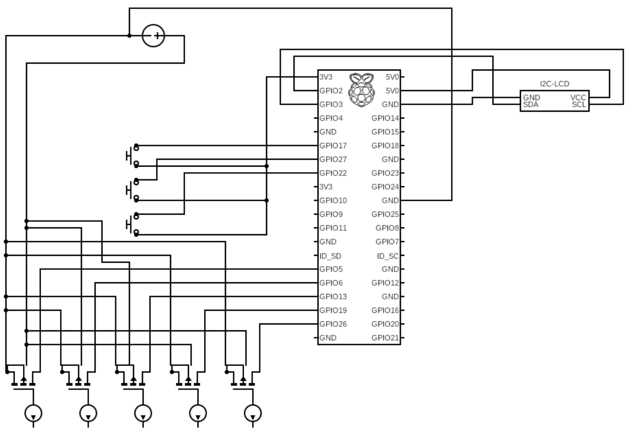


Figure 3: Wiring Diagram

4 Final Reflections

4.1 Group Reflections

Upon reflecting as a group, we believe that this project was a very valuable experience. We learnt key skills that will help us within the industry and believe we approached the task well. At the start, we decomposed the larger tasks into smaller ones and delegated depending on each of our strengths which proved effective due to our rapid advancement of the tasks. For instance, in the extension, some members focused on the software whereas some focused on the hardware. Our communication was efficient as we held regular meetings and communicated online. However, whilst this was efficient the majority of the time, there were times where not everyone in the group was clear on the overall structure of the project, which led to some confusion and duplication of code. To avoid this next time, we will aim to spend more time on debreifing the group and ensuring all members are clear on their respective tasks and how they all fit into the bigger picture of the project. Ultimately, the teamwork, communication, and commitment were strong and we hope to echo this in future projects.

4.2 Individual Reflections

4.2.1 Ashwin Biju:

My main strengths are decomposition of the complex high-level problem to smaller more manageable chunks to then delegate to the team. I took lead in verifying outputs, improving code quality and keeping the team on track to meeting deadlines. However one area that I recognise I have to improve on is time management as some tasks took longer than estimated which led to minor delays in the development. In the future, I plan to improve my time estimation for tasks and to communicate any such delays with my team. Overall, I found working in a team very insightful, and would like to bring the same level of organisation and commitment as I have done on this project.

4.2.2 Dorian Turner:

My main strengths were in writing code that was well laid out and, consequently, straightforward to debug. I also felt quite strong when it came to debugging the code of others. In particular, during assembler, I solved over 400 failing test cases due to minor errors in my group members' code, as well as fixing memory leaks with valgrind. Based off the feedback from peer assessment, I have identified a weakness in my git commit messages, which made it harder for my teammates to track progress and any changes I made. To fix this, I have watched the DocSoc lectures on git messages and have since thought more carefully about my commits to ensure that they are atomic and that the message accurately describes what has changed within the code. Overall, I have enjoyed working as part of a team and feel that I have learned a lot about working in a way that allows my teammates to collaborate with me on projects of a large scope. In the future, I will aim to bring the same level of code clarity and communication as I have learned to use here.

4.2.3 Tim Kolesnichenko:

My main strengths are high-level designing of the system, and solving technical issues in code. This led me to take initiative during the opening and closing stages of the project, planning the architecture and taking a key role in debugging of the code. Based off the feedback from my team, I recognise a key weakness in my communication, causing team member's to be unaware of the plans for the system. As a result, I aim to involve the team more in future design discussions, distributing knowledge and fostering better discussion around the design. This would lead to more robust solutions and identification of potential pitfalls. Overall, I found working as a team highly rewarding, and in future projects would aim to bring the same commitment and technical involvement as I have done in this one.

4.2.4 Michael Song:

My main strengths were writing clean, good quality code which joined nicely with the code written by the rest of the group, and debugging this code efficiently and effectively. I believe this is due to the quality of communication between me and my group, as I was able to easily understand the structure of their code. However, I recognize from feedback from my peers that I should be more proactive within the group regarding anticipating what is needed to be done. In future, I will aim to take more initiative in planning ahead, which would result in improvement in the group's efficiency and more ease in code-writing in general. Overall, I have found working on this project in a group to be highly rewarding and has taught me a considerable amount about the importance of communication within a team, hassles of using git with a large number of people in the same repository, learning and understanding other people's code style and subsequently becoming more familiar with programming this kind of system as a whole.