C Project Final Checkpoint Report

Ifechukwude Ndozi, Amogh Shetty, Rohan Desai, Anshul Dani

June 20, 2025

1 Implementation and Structure of Our Assembler

1.1 Assembler Implementation Approach

As a group, we decided to implement the two-pass assembly method for our assembler, recommended by the specification.

- **First Pass:** The first pass through the assembly code file goes through each line, removes any leading and trailing whitespace from the line, and checks if it is a label. All labels end with a ':', so we only check for this character. If the line is a label, an entry is added to the symbol table, which associates the label with the current memory address. This is so that any label references in the second pass can be replaced with memory addresses, as defined during the first pass.

- **Second Pass:** The second pass through the code goes through each line, again removes whitespace, and deals with .int directives and instructions. We collectively decided to deal with .int directives directly, as all of these start with a '.', so they can be checked for directly in a similar fashion to labels in the first pass. The respective integer is stored directly at the current memory address. For instructions, we tokenise the line by splitting it up, using commas and spaces as delimiters. We then parse the instruction by inspecting the first token, which is the instruction mnemonic, and then calling the correct parse function on it by checking our parse function instruction table, which relates functions to mnemonics, as this was an approach recommended to us in the specification. The instruction returned is then stored at the current memory address. Once the assembly code has been passed through twice, the binary instructions created are written to a binary file line by line.

1.2 Data Structures and Helper Functions

- **Assembler State** (`ARM_STATE`) As in the emulator, we chose to represent the current state of the ARMv8 assembler as a custom struct. It consists of an array of 32 bit unsigned integers to represent all the binary instructions, the number of instructions in the assembly code, the current address of the program and the symbol table, containing all the label references. We chose to store the binary instructions created in an array during the second pass of the assembly code as it made it easier to write them all to the output binary file at the end.

- **Symbol Entry** (`SymbolEntry_t`): We designed a struct to store a single symbol table entry. The struct contains a field called `name`, which is a string of length `MAX_LABEL_LENGTH` (defined as 128 characters), and a member called `address`, which is a 32-bit unsigned integer storing the address of the label.

- **Symbol Table** (`SymbolTable_t`): Our initial implementation idea was to use a struct with a field called `entries`, an array of 256 symbol entries. However, based on discussions and insight from our C programming lectures, we decided to use a dynamic ADT with a resizing array. This approach is more memory-efficient, as most programs will not have anywhere close to 256 labels. In our final design, the `entries` field is a pointer to an array of symbol entries, and we include two additional integer fields: `no_entries`, which stores the current number of entries, and `cap`, which stores the current capacity of the array. These two fields allow for the `entries` array to be resized whenever needed. We also used a typedef for `SymbolTable_t` so it became an alias for *`SymbolTable_t`.

- **Functions** Alongside these data structures, we defined five functions for the symbol table and one static helper function which allowed us to make use of the symbol table by being able to add entries and retrieve addresses of labels already in the table.

1.3 Structure of `assemble.c`

As with the emulator, we neatly divided the assembler into its natural pipeline, the first pass and the second pass. The second pass involves the parsing the instruction, which we subdivided into 3 different instruction types: data processing, single data transfer and branch.

Again like the emulator, we split the file structure to match our implementation of the assembler and how we delegated functions to write between the group members. We also create a Makefile to include all dependencies we need to automatically compile our project.

- `assemble.c`: Contains the main function, which makes the symbol table, runs both the first pass and second pass of the assembler, and calls the binary file writer, which writes all the parsed binary instructions to the specified output file.
- `pass_one.c`: Runs the first pass of the assembler, creating a symbol table which links labels with memory addresses.
- `pass_two.c`: Runs the second pass of the assembler, dealing with .int directives, tokenising the instruction and parsing it by calling the relevant parse function.
- `parse_helpers.c`: Contains helper functions that are repeatedly used in all of the parse functions, such as getting register values, parsing immediate and signed immediate values, and parsing directives.

1.4 Parsing Assembly Instructions

The main bulk of work for this section was converting an assembly instruction into a 32 bit instruction. The way we decided to approach this as a group was to have different functions for each instruction type, which would break the instruction into tokens which could then be processed and encoded into the corresponding binary instruction.

To make the running of pass-two more efficient we decided to use function pointers which enables the parsing function used to be decided at run time. We did this by creating a `*parse_func` which would be a pointer to a parsing function and a `instruction_entry_t` struct which maps a mnemonic to its corresponding parsing function. This replaced what would have been a massive switch statement in our pass-two function, which gave us direct lookup from mnemonics to parsing functions and made the code much cleaner and faster. This approach worked really well for our team because each of us could work on individual instruction parsing functions independently, and when someone needed to add a new instruction, they just wrote their function and added it to the lookup table without touching anyone else's code.

2 Our Implementation of Part III

2.1 Hardware Interaction

A new challenge we encountered in Part 3 was controlling the GPIO pins on the Raspberry Pi in order to continuously blink our LED. This was because instead of using specialised instructions, the CPU communicates with the GPIO controller by reading from and writing to specific physical memory addresses. The GPIO controller's registers are mapped to a memory block beginning at the base address `0x3F200000.` Our implementation consistently uses this address as a base to access the necessary registers. As specified in the spec, all interactions with these registers must be 32-bits wide, hence we have only used W-registers in our assembly code.

2.2 GPIO Pin Configuration

We chose GPIO pin 17 for connecting our LED circuit. The first step was to set the function of the GPIO pin as output, handled by the `GPFSEL` registers. The relevant register for pin 17 was `GPFSEL1` located at address `0x3F200004.` As a method of defensive programming, in order to avoid the configuration of other pins managed by the same register (pins 10-16 and 18-19), we employed a strategy that ensured only the bits for pin 17 were altered. First, the current value of the `GPFSEL1` register is read to memory. We then clear the bits corresponding to pin 17 (bits 21-23) by creating a bitmask and applying a bitwise AND. A new bitmask with the value 001 (representing "output" mode) is applied using a bitwise OR, setting pin 17's function to output. The final, modified value is then written back to the `GPFSEL1` register.

2.3 Blinking Loop

The program then enters an endless loop, where the LED is turned on, waits for some time, LED is turned off, etc. This is marked by the blink-loop label and concludes with an unconditional branch that returns it to the start.

- **Turning the LED on:** To turn the LED on, a high signal needs to be sent to the GPIO pin. To achieve this, we write a bitmask to the GPIO Output Set register, `GPSET0`, which is located at address `0x3F20001C`. Our code creates a mask where only the 17th bit is set and writes it to `GPSET0`.
- **Delay:** A delay is required to allow us to visualise the blinking of the LED. We decided to implement this by loading a large constant into a register and decrementing it to zero, which pauses the program execution for a short period of time.
- **Turning the LED off:** In a similar fashion to turning the LED on, to turn the LED off, a low signal needs to be sent to the GPIO pin. To achieve this, we perform a similar bitmask operation on the GPIO Output Clear register, `GPCLR0`, located at address `0x3F200028`. A second equivalent delay follows this to keep the LED off and make the blinking visible before the main loop repeats.

3 Our Extension

3.1 Brief Overview

As a group passionate about creating technology with social impact, we were inspired to develop a memory-training game aimed at supporting cognitive health, particularly in the context of dementia, a condition that affects millions of people worldwide. Therefore, we chose to build an LED memory game on a Raspberry Pi designed to stimulate short-term memory and cognitive sequencing, both of which are known to deteriorate in the early stages of dementia. We wanted the final product to be interactive, intuitive, and accessible, while also giving us hands-on experience working with hardware systems.

The game operates by flashing a randomly generated sequence of LEDs, which the user must then replicate using a set of buttons positioned in front of each light. If the sequence is entered correctly, a new LED is added to the pattern and the updated sequence is displayed again. This process continues until the user makes a mistake, at which point the both of the red LEDs flash to indicate failure, and the system resets to an idle state awaiting a new game.

3.2 Design Details

We began our project by designing a high-level structure for the game logic. The initial plan involved creating a central game logic file responsible for managing the overall flow and calling relevant helper functions to keep the code modular and readable. In parallel, we started experimenting with the hardware, developing a basic test script that allowed us to operate three LEDs using individual buttons to turn them on and off. This helped us familiarise ourselves with input/output handling on the board.

3.2.1 First Prototype

Our first prototype implemented the core game functionality with three LEDs. The game displayed a growing sequence of lights, which the user had to replicate using corresponding buttons (as described in the overview). At this stage, we encountered some key design questions. One of the main ones was how to clearly signal failure. Initially, we used a red LED flashing three times, but this wasn't visually distinct or intuitive. We adjusted the flashing frequency to be significantly faster, making it more obvious that it signalled an error.

We also began considering ideas to improve user experience, such as implementing a separate controller with the buttons to make the game more portable and ergonomic. However, more critically, we focused on making the design scalable, allowing us to increase the number of LEDs to enhance the cognitive challenge. This meant structuring our code without hardcoded values, using lists and variables to dynamically control LED behaviour based on the number of LEDs in play.

3.2.2 Second Prototype

In our second prototype, we experimented with the external controller concept. However, the additional wiring obstructed the LEDs and reduced visibility, and the physical separation between buttons and LEDs introduced

the unnecessary task of mentally mapping buttons to lights, which detracted from the core memory challenge. Simplicity was key, so we reverted to our original layout but scaled it up to six LEDs, spread across two breadboards.

We further refined the failure display by adding red LEDs on either side of the board that flashed simultaneously. This simultaneous behaviour was not used elsewhere in the game, making the failure state immediately recognisable.

Thanks to our scalable code, expanding to six LEDs was straightforward. Most logic extended naturally by modifying the LED list and updating the configuration variable for the number of active LEDs. One limitation we encountered was reduced access to hardware, which required us to make efficient use of the components we had.

3.2.3 Reflection

Overall, our iterative prototyping approach helped us steadily build complexity into the game without being overwhelmed by too many features at once. It also allowed us to identify usability issues early and ensure each version was functional, intuitive, and ready for further development.

3.3 Testing

Due to our iterative approach, we were able to test early and test often. We conducted hardware testing on our first prototype, running a simple script to test our ability to handle button inputs and LED outputs. We then moved onto system testing and user acceptance testing for the game as a whole. We ensured to test a wide range of valid, invalid, boundary, and erroneous test cases to catch any bugs and refine the user experience. This included rapid double clicks, long button presses, and pressing multiple buttons at a time to assess the program's defensive capabilities. We also outputted key information to the terminal to evaluate the correctness of our game logic. We fine-tuned constants such as the debounce delay, which prevents accidental double-clicks and ensures smooth user interactions, and the LED flash speed to create more engaging gameplay.

Our iterative testing approach enabled us to improve robustness and enhance gameplay through new features. For example, user feedback led us to implement a timeout feature in the second iteration, forcing quicker responses and creating more thrilling gameplay. Good team communication also enabled us to iterate quickly towards a strong final prototype. Our user-centered testing approach proved effective, but integrating automated unit and system tests earlier would have improved efficiency by simplifying later phases.

3.4 Challenges and Problems faced

- **Working with Raspberry Pi and GPIOD library:** As all of us were inexperienced with using a Raspberry Pi, setting it up to run our code took a long time, as we had to install its OS and enable a SSH connection to our personal devices in order to continue making changes to our code while still being able to run it on the Pi. This slowed down the development of our extension, and took away from valuable implementing and testing time.

- **Implementing the hardware:** Due to limited hardware experience, we encountered initial setup challenges. We addressed this by starting with simple validation scripts for one button and one LED, progressively scaling to three, then six components as our confidence grew. We felt limited by the hardware available so tested several different layouts, to optimise for accessibility and ergonomic user interaction within our available resources.

- **Handling button presses:** To prevent accidental double presses and missed inputs, we implemented careful button handling. Initially, we used a debounce delay that forced the program to sleep after registering a press, giving time for users to release the button. However, user feedback revealed issues with this approach, prompting us to switch to edge detection. By storing the previous button state, we only registered presses when the state changed from unpressed to pressed. This method proved far more robust, delivering a smoother gameplay experience.

3.5 Future Improvements

Looking ahead, our planned improvements focus on enhancing both the input and output aspects of the game to create a more engaging, reliable, and cognitively effective experience. On the input side, upgrading hardware

components will improve usability and durability, while on the output side, improvements in visual and sensory feedback aim to strengthen the user's cognitive engagement.

- **Hardware Improvements:** To improve the overall user experience, upgrading the hardware components will be essential. This includes using higher-quality LEDs and buttons that offer better tactile feedback and durability. Additionally, investing in better wiring, such as shorter male-to-male and longer male-to-female cables, will help reduce clutter and improve the device's portability and aesthetic appeal.
- **Expanding Game Modes:** Introducing different game modes could add variety and increase the cognitive challenge for users. For example, timed modes, reverse sequences, or pattern-matching variations could make the game more engaging and allow users to train various aspects of memory and attention. Also having levels of difficulty would make the game more accessible, whilst also providing a clearer way of measuring progression.
- **Display Board Upgrade:** One of the biggest improvements would be to add a dedicated score display board. Showing the user's progress and highest sequence reached would provide clear feedback and motivation, making the game more interactive and rewarding.
- **Haptic and Audio Feedback:** Integrating haptic feedback and distinct audio cues for each LED and button press could significantly strengthen the cognitive response. For instance, assigning unique sounds to each light and synchronising the button press with the corresponding sound would enhance multi-sensory engagement. This approach could improve memory retention whilst also making the game more immersive and enjoyable for users.

4 Reflection on Programming in a Group

4.1 Key moments and Lessons Learned

Early in the project, we faced a few challenges with miscommunication regarding task allocation. This occasionally led to overlapping work and wasted time. We addressed this directly by becoming more diligent with assigning tasks and upping the frequency and clarity of our team communication. Pair programming also naturally helped to mitigate these issues, as two sets of eyes were often on the same bit of code.

We also had an instance where a merge request inadvertently removed some essential code. This incident highlighted the critical need for even more rigorous merge request reviews. We learned from this experience, taking the review process far more seriously and exercising greater caution to prevent similar occurrences in the future.

Finally, our initial commit messages were often a bit vague. When we looked back through the project history, these undescriptive messages made it challenging to understand the purpose or changes within early commits. We quickly recognised the importance of clear, concise commit messages and made a conscious effort to improve this habit moving forward, which greatly aided our understanding of the project's evolution.

4.2 What We Did Well

Pair programming really proved useful. Working in pairs allowed us to collaborate with each other for innovative solutions, catch bugs and issues early and gain a deeper understanding of the entire project through discussions. This collaborative approach significantly improved our overall code quality and style.

Our thorough merge request reviewing process was another major win. Coupled with working on well-named branches and frequently pushing our code, this ensured transparency and allowed team members to track progress effectively. This careful approach was vital in catching most bugs before merging, significantly reducing integration and compilation headaches down the line.

Crucially, we broke down every section into a clear plan right from the start. This gave each team member a proper understanding of their responsibilities and the duties for their specific components. Our modular approach helped us avoid code duplication; for instance, knowing a function was being developed by another team member meant we could simply plan to call it, understanding its inputs and expected outputs. This foresight saved us time and streamlined the merging process.

5 Individual Reflections

5.1 Ifechukwude

Reflecting on this group project over the last month, my communication skills shone, which I am happy about. This could be seen in the peer feedback, where my team members pointed out how my clear explanations of my design decisions and listening well while pair programming helped us to work better together as a team. I also discovered some new strengths and weaknesses. I found I was able to break down seemingly challenging problems into more manageable subproblems, and I developed good version control skills. Despite this at times my time management was not always the best as I would spend too much time hung up on small details of the code, which was often irrelevant. My main weakness was not always knowing the next steps to take due to the ambiguity of the specification, instead of just attempting the problem and iteratively modifying my solutions. If I worked with a different group, I would again maintain clear communication with my team as well as encourage pair programming wherever possible, and I would push myself to be more decisive and dive headfirst into problem-solving when facing uncertainty, as well as improve my time management.

5.2 Amogh

I have really enjoyed working on this project over the past 4 weeks, as it has really improved my teamwork and communication skills and I have gained a lot of experience in working with C and using git in a group programming project. I believe I was a great fit in this group, and as the project progressed, my role within the project changed quite a bit. There were times where I had to take initiative and share my thoughts with the other group members, whereas at other times I was the one listening and gathering information from other members who were more experienced with a specific task. This fluid dynamic allowed us to work well as a team and ultimately utilise all of our strengths and weaknesses effectively and make significant progress in a short amount of time. I also learned a valuable lesson about my own weaknesses. I realised I wasn't always reading the full project specification before starting, which sometimes caused issues down the line. I focused on improving this, and by the end, I was much better at understanding the entire problem before breaking it down.

5.3 Anshul

This project proved to be one of the most engaging experiences I've had in our course so far, primarily due to the loose guidelines, which forced me to think more critically about problem solving and my approach. Working in a team revealed important insights about my development process. Initially, one of my key weaknesses was always wanting to dive straight into coding with a rough idea but without extensive planning. Through collaborating with others and learning to properly decompose problems, I learnt the crucial importance of spending more time figuring out my approach before implementation to avoid committing significant effort to approaches that turn out to be flawed in their initial logic. This lesson became one of my strengths in tasks 2, 3, and the extension, where I learned to step back and think in-depth through the problem architecture first. One of my existing strengths was communication, with regards to making sure everyone's ideas were being heard within group discussions and that everyone had clarity on what to do, which proved valuable when structuring our approach and contributing to our collaborative development using Git. Overall, the project was a really great experience, and I had an amazing group alongside me.

5.4 Rohan

Working on this project with my group has been highly rewarding. Our team functioned well through frequent, clear communication that enabled us to divide our work, organise meetings, raise concerns, and more. When team members faced extra-curricular commitments, we successfully adapted through transparent communication, ensuring equal contribution from everyone. I discovered my versatility across multiple roles, pair programming, independent coding, system design, and debugging, which broadened my technical confidence and highlighted the importance of communication skills in a group. During the project, I identified some areas for personal growth, in particular I found myself being too pushy at times, and I am learning to develop better judgement about when to check in with teammates versus allowing independent work. I also gained valuable technical experience in C programming and Git version control, both of which I have thoroughly enjoyed. For future projects, I would continue prioritising clear communication to improve efficiency and team morale. However, I often found myself getting caught up with smaller details in the specification, trying to understanding everything before coding. Next time, I would encourage myself to dive into the code earlier, as hands-on experience often gave clearer insights.