FINAL REPORT

Sanchit, Morkus, Luqman, Alex

Assembler Structure

After a constructive checkpoint meeting with our team mentor, we decided to restructure our project files. The underlying issue was there were too many lines of code and different functions present within one file which took away from the readability of the project as a whole. This led us to split our project into multiple folders: `emulator`, `assembler`, `common`. The `emulator` and `assembler` folders contained their respective files that were unique to their operation. This included specific constants files for each part as it became much clearer to an external reader where the definitions of the constants could be found. Our `emulate_utils.c` file was also split to show which helper functions were needed for each execution function. We split these Emulator files into `fetch`, `decode`, and `execute` files to clarify the purpose of the code further. This was because some instructions, such as Data Processing Instructions, required many helper functions but were all included in one file which became confusing. The `common` folder contained the constants and specific error handling and checking functions which could be applied in both parts of the project.

Essentially, our project evolved to a library-based structure instead of the mostly confusing linear arrangement, where we had all our files in one folder. As a result, we based our Assembler structure on the revised project structure and principle. We logically split our files to ensure readability was maintained and to clarify which functions were grouped. For example, our hash table ADT for the symbol table was included in its own files, where the header file defined the ADT and the .c file defined the functions needed for it to operate. We also created separate program files which were dedicated to handling the output of the machine code our functions produced. Breaking down our project in this way was incredibly helpful to us in maintaining the large codebase. *Diagrams showing our project structure have been included in the accompanying presentation slides.*

Assembler Implementation

Our Assembler implementation relied on three main aspects which worked together to produce the intended result. These were: populating the symbol table from the first pass and calling the designated assembly functions; generating the corresponding binary; writing the binary to the output file. We decided to use the two-pass assembly method, which allowed us to split the tasks for each person in the group more fairly. The file I/O was handled by *Morkus* as he worked on this type of task during the Emulator phase of the project. The functions required to generate the corresponding binary (i.e. the second pass) were assigned to the group members working on the same instructions in the Emulator *(Alex – DPI, Luqman – Multiply & Branch, Sanchit – SDTI).*

Our primary goal in implementing the assembly functions for the second pass was to produce each part of the instruction one at a time. For example, for a Data Processing instruction, there are RN and RD registers. These registers would have their own assignments and hence would make it obvious as to how their corresponding binary was produced. We did not want to combine multiple parts of an instruction into one variable as that would cause confusion between our own members and any external programmer. Breaking down the instruction into its constituent parts provided absolute clarity on the source and process in generating the binary. This technique meant that the final result returned by the assembly functions were multiple bitwise ORs of each part coming together in the exact order they appeared on the instruction type

summary.

To further explain our implementation choice, consider the format for a multiply instruction. It was structured as follows: [**Cond** | **000000** | **A** | **S** | **Rd** | **Rn** | **Rs** | **1001** | **Rm**]. As a result, you would notice the return value of our multiply function was:

```
return ALWAYS | accumulate | rd | rn | rs | MULT_HARDCODE | rm
```

where each variable was placed in the corresponding position (and S = 0 did not need to be included). We felt that this was a useful technique to keep it clear what was being returned and how the binary was being generated (by finding and combining the constituent parts).

Another implementation choice we made was to use a lookup table for the supported opcodes and the condition suffixes. This was needed as C does not provide any standard functions to convert from a String type to an Enum and using a long chain of *'strcmp'* calls to distinguish between opcodes was not a good style of programming. Converting to an Enum also brought us the benefit of having the condition code's binary representation as the value of the Enum. Hence, it did not require us to do further calculations or conversions.

While we were programming, we noticed we removed the first character from a string and applied an operation to it quite often. This was because constants and registers were preceded by the characters `'#'`, `'='` or `'r'`, and these were not desired to produce the correct binary output. Our first implementation choice simply used the `++` operator to increment the string pointer by one. However, this operator was also used in other cases such as incrementing a number or looping through a string, and the meaning behind what we were trying to do became unclear. This led us to produce several macro functions, including `REMOVE_FIRST_CHAR`, which allowed an external programmer to understand what was happening in the program. The removal of the first character and conversion to an integer was also a process we carried out often, and hence we produced another macro, `REM_INT`, which combined the two functions.

Extension Description

After creating an Assembler and Emulator we were proud of, we decided to incorporate our previous parts into the extension in a way that we had not seen before. Our idea for the extension is a text-based adventure game which explores the *South Kensington* Campus of *Imperial College London*. As you explore the map, you will find essential items to progress in your journey. Key passes are needed to gain access to some rooms where treasures or bosses await the player. The game consists of shops such as *Fusion* where you could spend your hard-earned cash to obtain items that can help you. Food items allow you to regenerate some lost health. Technological items, including keyboards and monitors, are needed to battle the boss with programming.

The twist we added to the text-based adventure is the ability to have coding battles with *'bosses'* which would be members of the DoC. These coding battles take place using the *Arm-11 assembly language* that we have been working on for the past few weeks. Users can write their code to solve the given problem, which then gets assembled using our Assembler and executed using our Emulator. If the outputs of their program match the expected outputs, they gain rewards such as cash and special items. Otherwise, they take damage to their health and may lose some of their items.

This exciting twist to a popular genre of games provides a new way of testing out your ARM knowledge in the comfort of your own home. Students would enjoy exploring the campus while being introduced to surprises along the way. Our adventure game has multiple uses which work well together. The simplest use case for the extension is as a way of relaxing and destressing for students by just playing and exploring. However, we decided to go beyond that with our Emulator and Assembler integration. Our extension has

the dual purpose of being a more creative and extensive test suite than the original tests. With the ability for users to input their own programs, the Assembler and Emulator can get tested to a higher degree than before. This means that students can now test their programs while enjoying a great adventure in a remake of the campus. We hope that this is the most fun testing facility for the Assembler and Emulator that the DoC has ever seen.

Extension Implementation and Challenges

We decided to challenge ourselves to purely use the C programming language to complete the extension. This is because we wanted to use the extension as an opportunity to fine-tune our skills for C and learn to solve problems which may be more difficult in C than other programming languages. As a text adventure game, our extension relied heavily on user input, and this came with a range of unique problems for the C language. We programmed the extension features with no reliance on other libraries, except for the C standard library, which improved the portability of our code. For example, we refrained from using the *Conio.h* header file (which would have allowed us to add colours to our textual UI) as its inclusion in the standard library depended on the platform it was used on.

We followed the general design principles used in our assembler and emulator implementations by ensuring our code was logically split into multiple files and folders. We included folders such as `boss` and `player` for the characters in the game.These files included the definitions for commands the user could type to explore the map along with the initialisers for each character's state. The code which provided the bridge between our game, the Assembler and the Emulator was placed in its own folder called `emulateARM`. Finally, we had folders `game_utils` and `print_utils` consisting of utility files for the game itself, such as creating the map, connecting rooms, and the prints needed to produce the textual UI.

The main function for the game used a while loop to persistently query the player's commands and check the type of command that was inputted. A switch case then allowed the correct process to be determined, carried out and outputted to the screen. Each function also had its own error checks, e.g. if an item could not be *'bought'* or if the user tried to *'drop'* an item they did not have. Upon the completion of a process, the user gained back control to type another command and this cycle continued until the user decided to save and quit the game.

The primary challenge we came across was ensuring the user's input was correctly parsed and have an appropriate corresponding process or output. This also meant that we had to account for any strange inputs the user may type including blank inputs, random spacing in commands, or commands that did not exist. Using a combination of *GDB* and *Valgrind*, we identified the points of the program which were most vulnerable to an erroneous input which could cause a segmentation fault and unexpected exit. This led us to provide many iterations and upgrades to our parsing and error handling, allowing us to reach the point where the user could no longer break the program with their inputs. As it is a text adventure game, we provided some creative and funny outputs for when the user typed incomplete or non-existent commands.

Another challenge we came across was implementing our save and load functions. This required us to use our experience with binary files from the previous parts to produce our own serialisation and deserialisation functions which is an element that we are proud of. We solved this problem by generating our own test suite, inspired by the formats of the tests used in the lectures. Using the test cases, we pinpointed specific problems and quickly solved them to enable this functionality in the game. External knowledge from the *Graphs and Algorithms* course also came in useful when traversing through each room and populating the items. We viewed our rooms and connections as nodes and arcs, allowing us to traverse our game rooms with inspiration from some graph algorithms such as *depth-first search*.

Implementation Testing

To test and improve our code for the Emulator and Assembler, we used a combination of our own test files and the standard test suite. The standard test suite allowed us to find which functions were providing incorrect outputs as some lines did not match the expected answers. However, in some cases, more accuracy was required to find where the fault lied. This led us to produce our own tests which we designed to work on specific functions. The combination of the tests we produced and the test suite along with *GDB* and *Valgrind* allowed us to effectively test our program for various inputs and ensure that it worked in all cases defined by the specification. One improvement we could have made was to include even more tests to thoroughly check the programs, as originally we were carrying out manual tests rather than automated tests. Although the manual tests were useful at first, they did not transfer well when the code scaled, which is a great lesson our team learned and hence switched to automated testing.

Our extension had two types of testing: an automated test suite and user-perspective testing. Functions which produced comparable outputs, such as loading in a save file, were tested using automated tests in the style of the lectures. These outputs were checked against hardcoded expected outputs and worked well in identifying which parts of the program were not working. As our game relied on user input and text output, our second form of testing was to play the game and note down any unexpected results when carrying out a command. This was incredibly useful as it led us to improve and fix many functions that were not working as expected, and hence we believe it was an effective technique. As discussed already, our extension allowed us to test further the Emulator and Assembler programs which was a useful bonus.

Group Reflection

Overall, the group functioned well in coping with the Assembler and extension implementations. The group decided to use the *Agile Scrum methodology* as our previous style of working was similar to this. Hence, clearly highlighted deadlines and broad inspections into each other's current tasks led to the project being completed not only smoothly, but with greater quality. The task delegation within the group for Assembler was well managed and non-time consuming. As discussed, we delegated tasks for the Assembler based on the tasks each person carried for the Emulator. This allowed group members to contribute their uniquely learnt skills to their respective tasks and ensure consistency across our implementation.

Communication was a strong point within the group. As expected, not all members fully understood the more complicated elements of the task (like the *ADT* used to traverse our *symbol table*). However, every member made an effort to provide explanations until we reached the point where everyone was happy with the task. This meant the group was well informed of the problem at hand, improving the time we spent on writing quality code. Daily meetings on *Discord* kept the team correctly addressed of their progress and problems with the code, allowing everyone to stay on track to meet their individual sprint deadlines.

After the delegation of tasks, we had a problem with the group leaving the completion of their tasks too close to the deadline. This was mainly raised due to other situations, such as preparation for the C final test, given higher priority than the project. The problem caused a landslide of further issues, such as other members in the group not being able to implement their tasks, because they required parts of other tasks. However, after a small intervention on *Discord*, we unanimously agreed to extend all internal deadlines to after the C test. This led to better focus and greater dedication to finish.

For the extension, our cohesive planning strategies enabled the team to make decisions regarding the idea of the extension swiftly. *Sanchit* took the initiative to create a *Google doc* in which we mapped our ideas into. With each group member researching and reading/asking questions on *Piazza*, we were able to make a weighable list of pros and cons which aided our final extension idea. The tasks for the extension were split equally: *Sanchit* worked on the UI and generating the rooms for the map; *Luqman* worked on the player

input and commands to play the game; *Alex* worked on creating the boss characters and battle functions; *Morkus* worked on the save/load feature and integrating our extension with the Assembler and Emulator. Our planning and breakdowns played a major role in succeeding with our implementations and is one aspect of the project we will keep the same for the future.

As we had only a limited time to complete the extension, several improvements could be made, such as new features. We had put measures in place to prevent a user from breaking the program when typing in the standard commands to play the game. However, we could not implement an assembly syntax checker, within the time we had, which would parse the user's assembly code input for the battles and ensure it had no syntax errors defined by the ARM manual. This led us to make the assumption that the ARM assembly user input was syntactically correct, which was justifiable as this assumption was made in the Assembler and Emulator. If we had more time, this would be a feature we would have tried to include.

One issue that we would like to improve for the future is to plan a logical and coherent code and file structure before beginning programming. This problem led us to spend large amounts of time cleaning and refactoring code in the earlier stages. Another group issue we noticed was making many mistakes during the last few days of the project as we felt under pressure. This led us to, accidentally, merge incorrect branches which had to be reverted. As a group, we feel we need to improve how well we work under pressure and to ensure that a calm, level-headed mind is kept throughout the project. We know that enhancing this trait will benefit us across any task we do, not just in programming.

Individual Reflection

Luqman

This was the first time that I worked on a large programming task in a group, and it was an incredible learning experience. My knowledge of *Git* and features such as branching and merging improved vastly. I learnt to provide meaningful commits for each file that was changed and ensure different changes had different commits, which was not something I did at first. This was useful as it made it much easier for us to find and fix errors in the code which may not have been present before. I was not acquainted with all members of the team at first, which was a worry for me as I did not know how we would work together. However, after a few days of working on the project, our group bonded well, and I was no longer worried.

One strength of mine I discovered during this project was consistently conveying the meaning of each function we wrote through comments and reports. I felt that one of my weaknesses was working with Makefiles, as there were some errors I relied on my teammates to solve because of my lack of understanding. In the future, I would try to take on as many different tasks as possible to gain a deeper understanding of all aspects of a project. I felt that I had made a strong commitment to the team and would maintain this if I worked with different people.

Sanchit

Having never worked on a group project of this scale before, I feel it went surprisingly well. My knowledge of *Latex* improved immensely and I am now able to, not only write reports but also use latex libraries to implement diagrams for file structures. This is a handy skill I know will be used in future projects throughout Imperial as it betters the representation of a project's structure. Having not used Git for team projects I expanded my current knowledge of Git features such as branching and committing. This was not an important part of previous projects, which I did alone, because I didn't have to worry too much about the messages I commit with. Now I understand how important it is to commit frequently and with accurate messages to avoid my team members being confused about the progress I've made.

On the other hand, a weakness I still carry is the ability to merge properly, which led me to rely on other team members to merge my branches for me. This slowed down the progress I made, and so I hope to resolve this issue in upcoming projects. The group project also helped highlight strengths that I didn't know I had; more specifically my strength in a program's UI and the front end of a program's design for the user. Overall, regardless of the complexity of the task, I feel I had a solid work ethic within the group project.

Alex

I was quite worried with how a programming group project would be carried out at the start. However, as we started planning our project, it became clear that the use of *Git* would be the bedrock of group programming. My *Git* knowledge tremendously improved during as we progressed through the project. I thought that my strengths would be in improving code hygiene. However, I realised that maintaining code hygiene to a good level in a group project was more challenging than previously thought due to everyone having different programming styles and habits. Therefore, I advised the group to maintain the same code conventions and use the same code formatter for consistency across our code. I felt that this boosted our productivity and reduced the time we spent on fixing inconsistent code and I would do try to keep a similar procedure for future projects.

Morkus

I came into this project having no prior experience coding in a team, but have learnt many valuable lessons along the way. This was the first time where I had used Git for a large project, but I quickly became comfortable with its many features and realised its potential in group projects through the powerful branching/merging system. I enjoyed the challenge of working with C and Makefiles as this was far more low-level than anything I had done before. I also did not use an IDE to hold my hand, which meant I was able to strengthen my knowledge on the fundamentals of memory management and what goes on behind the scenes when I run my code. This is knowledge that I'm sure will prove valuable in my future work. The biggest challenge I faced was perhaps the task of devising a load/save system in our extension. I initially thought this only needed to save the single struct representing the current state of the game. However, I realised I would also have to save the multiple structs, elsewhere in memory, that the state held references to, such as the player's inventory. In the end, I devised a text document called `save_file_format.txt` which highlighted a proposed structure of the data in the save file, which meant I then just had to read/write the data into the file following this simple specification.

In hindsight, the only thing I would have done differently is read through the whole of the project spec first and plan the file structure ahead of time as this would have saved a lot of headache and man-hours when it came to refactoring and polishing our code. Overall, It has been a pleasure working in this team as we collaborated well, letting each other shine in our strengths, and had full trust in each other's abilities, meaning that I am walking away from this experience feeling proud of what we have accomplished.