1 Introduction

This report provides an overview of the Assemble and Raspberry Pi project for Group 2. The first part of the project aimed to design and implement an Assembler which would translate assembly code written using AArch64 instruction set to binary code. The second part aimed to write assembly code to make the light on the Raspberry Pi 3B blink infinitely, using part one to translate to binary code. The report covers the project's structure, implementation, challenges faced, testing approaches, group, and individual reflections.

2 Assembler Structure and Task Division

2.1 Structure

Assemble (Ifaz)
This includes the main body of the program, consisting of 4 phases, connects and uses all the modules and their functionalities.

Structure (Ifaz)
This includes struct and type definitions, along with functions to create, operate and free them.

dataProcess, aliases, dataProcesImm and Reg (Mahdi)
This deals with data process instructions.

Branching (Suweda), dataTransfer (Areeb)
These deal with their respective instructions.

2.2 Reflection

We split the project and worked on separate files from the start after reflecting upon emulate, where we had lots of merge issues and additional bugs when we tried to break our file into modules and connect them. Whenever a module was dependent on a functionality of another module, we would pass each other the function signature which would allow all of us to work on our modules without thinking about how those functions were to be implemented. One issue we faced with modularising was with global variables. At the beginning we were getting lots of null pointers when we tried using certain global variables but those were fixed by properly declaring global variables across the modules.

Util (Everyone) This contains helper and debugging functions needed throughout the code.

3 Design and Implementation

In terms of design, we decided to have an internal representation of the instructions by using nodes and linked list. The advantage of using linked list was that we would only allocate memory that was needed. The nodes consisted of the memory address, the type of instructions (a string for name of instruction, eg "ldr") and their arguments. This made it much easier to deal with the

instructions. We also implemented a symbol table, which consisted of an array of symbol entries containing the label and its memory address. This allowed us to go through the source code in just one pass. Finally, we used a function pointer table containing an array of entries that holds the function pointer and the type of instruction it works on. This removed needing to implement a large if and else section for finding which function to use for which instruction. We have mainly utilized malloc to allocate memory for our structures as we wanted to consider cases where we had to translate a large file. For the symbol and function pointer table, we defined the size of the array at the top of the file, which can be changed if needed. Our program has 4 main phases, each dealing with a specific part of the whole task.

3.1 Phase 1: Initialisation

Initialises and creates the function pointer table from two arrays, one containing all the function pointers and another one with types in the corresponding index. Also initialise counters and sets counter to 0 for the size of symbol table array, memory address, current number of arguments and malloc an array of strings to extract and hold arguments from each instruction. We also open the source file for reading, create a buffer to hold each line and initialise the list and initialise a node to hold current node and another one to hold previous node.

3.2 Phase 2 (Parsing and forming symbol table and list)

In a while loop which keeps going until the end of file is reached, we use string tokenizer to break down each line into the type and arguments. We first check if the first token is a empty or just a newline, in that case we continue. We then check if the string is a label, if so we add it to the symbol table along with its memory address and increase the count. If the token is none of the above, then it must be an instruction so we use another while loop to go through each of the tokens and store them in the array of strings. When we have reached a null token, we initialise our node, connect it to the previous node (unless this is the first node) and reset the argument counter to 0 and increment the memory address by 4. By the end of phase 2, we should have the linked list with all the instructions represented through the nodes and the symbol table, which is passed through a function so that branch and dataTransfer can access them when they have to replace a label with its memory address.

3.3 Phase 3 (Creating and writing the word)

We open the file for writing and loop through each of the nodes. We use the function table to get the corresponding function pointer for the type of the node and pass the node into the function to create the word. This is where the functions from the different modules are used. Instructions with labels use the symbol table to replace the labels with their actual memory address. We write the word onto our file. Once the loop has finished, we close the file.

3.4 Phase 4 (Freeing)

Since we used a lot of memory allocation, we free up all the structures : linked list, function pointer table, symbol table and the memory allocated array of string.

3.5 Challenges

Since we utilized a lot of memory allocation, we initially had some memory leaks. However those were fixed by carefully breaking down structs and freeing each part.

4 Design and Implementation

Reflecting upon emulate, we decided to test most of our functions in isolation before merging and testing everything together. To help with this, we used functions such as PrintNode, PrintSymbolTable and PrintBits to look at the output of different functions based on given output. This made it much easier to debug once we joined all the modules together and ran the whole program. We primarily used Clion to debug as it allowed us to easily set breakpoints and look at the values at each step. Alongside this, we used valgrind from checking for memory leaks.

Since we divided our program into 4 phases, we tested each phase separately before moving to the next phase to find problems and debug more efficiently.

5 Raspberry Pi

5.1 Structure

We planned out the structure of our code before starting, but had to make edits along the way. We started with initialising registers with important memory addresses – e.g., of the buffer and mailbox registers. This made it easier to use later on, as we wouldn't be worried about accidentally using the wrong address in a later register. We then had a loop, which we called while, and first checked whether we were trying to turn on the LED or not, as our w7 register held the value of LED status. Next, we wanted to write a request, so first we had to make sure the F flag of the write status register was clear, and when it was we composed our request and then sent it. To free up the response queue, we immediately read the response, first checking if the E flag is clear. Once this is done, we delay using a loop which counts down, before restoring the buffer to the original – the only difference being changing the request to a turn-on request, if it was originally a turn-off, and vice versa. At the end, we made sure to follow the spec guideline and have the buffer at the end. We then used the amount of lines to find out the position of the buffer, and used nops at the start of the code in order to make sure the address was 16-byte aligned.

Here is a list of initialised registers and what they store:

- 0 – request buffer address
- 1 – address of mailbox read register
- 2 – write status register
- 3 – message to be sent
- 4 – E flag
- 5 – the response from the write register

- 7 – LED status - 0x0/0x1
- 10 – address of mailbox read status register
- 11 – address of mailbox write register
- 12 - address of mailbox write status register
- 20 – delay counter – decrements in delay loop
- 25 – 130 – the value at 0x14 offset from buffer start in the request, used for overwriting
- 30 – 0x1 for checking against LED status

5.2 Problems

We struggled with understanding everything in the spec, such as why we'd have to have a copy of the request buffer, when we can just overwrite it each time we get a response (which is at the end of our loop) instead. Also in the suggestions section of the spec was what to do when you didn't have 'mov'. We didn't really have a time where this was a big problem, instead using other functions like ldr more – this worried us a lot as we thought there may have been something wrong with our methods of implementation. Another big problem was that our assembly program had persisting issues until pretty late into the project, and so we couldn't really test our Pi as much as we would've wanted, or even ask about the problems we had in lab sessions/EdStem. This negatively affected our testing for this a lot, and more-so since we couldn't use QEMU to try get a log of mailbox interactions to see what was going on. If we had more time, I think we would have been able to focus on the Pi more and get it working, but due to time-constraints, we were not able to turn it on.

6 Group Reflection

Working as a group has provided valuable insights into the dynamics of collaboration and communication in software engineering. One key reflection for all us was understanding that our own solutions may not always be the best and many times when we discuss with each other, we would find better solutions or spot mistakes quicker.

A key strength of our group was that we maintained an open line of communication throughout the project, worked together in-person, shared ideas and discuss challenges. This allowed us to stay updated on each others progress and the project objectives.

Initially, we faced some challenges in finding the right balance with task division. However, through regular discussions and brainstorming sessions, we were able to identify each member's strengths and assign responsibilities accordingly. By establishing clear deadlines and milestones, we ensured that tasks were completed in a timely manner and progress was consistently tracked.

One weakness we faced was our inexperience with coding compared to most CS students, which meant we had to spend significantly more time coding. However, we were able to mitigate that through planning effectively. Some issues we faced with setting deadlines included underestimating how much time some tasks would take, especially with debugging, or getting stuck on certain tasks due to lack of understanding. This slowed down the overall project progress as

we would be spending significant time just trying to understand what we are meant to do and how. Looking ahead, we believe that maintaining effective communication and collaboration will continue to be crucial in future group projects. We value the establishing of clear roles and responsibilities from the outset, whilst also allowing flexibility for individual contributions. Additionally, we plan to implement more frequent check-ins and progress updates on everyone throughout the project and plan accordingly.

7 Individual Reflection

7.1 Ifaz

Reflecting on my experience, I feel that I fitted well into our group. Initially I was hesitent on being the leader, but I discovered additional strengths that I hadn't anticipated, such as my ability to delegate tasks and ensure effective communication within the group. One area where I believe I excelled was keeping everything organized and not tunnel visioning.

One area that I recognized as a weakness was time management. There were instances where I struggled to set appropriate deadlines due to underestimating the time required for certain tasks, especially when it came to debugging. This experience has highlighted the importance of better planning and setting realistic expectations for myself and the group. In future group projects, I would prioritize improving my time management skills to ensure timely completion of tasks and minimize last-minute stress. I will also make sure to utilize in person meetings as they were

7.2 Suweda

During our group projects, I found that my strengths met my expectations: I am good at staying organized and proactive, managing my own tasks effectively and contributing to a team. Additionally, I actively participate in discussions and am always available to my teammates when needed, using my strengths to support collective efforts. However, along the way, I also faced challenges. Sometimes I have trouble fully understanding project specifications and finding the right balance between focusing on my own code and reviewing other people's work, which hindered my attempts to help them later down the line. This experience taught me the importance of taking the time to understand requirements and actively participating in teammates' tasks to collaborate more effectively.

An important lesson I learned was that project planning requires flexibility. We sometimes had a hard time setting realistic deadlines, either because we are too ambitious or not ambitious enough. Going forward, I hope to find a better balance by being responsive to the team's progress and setting achievable deadlines. Additionally, I have found pair programming to be a valuable technique that can provide diverse insights and efficient problem solving, and is definitely something I'd like to bring to the next project.

These experiences have shaped my approach to future team work, where I maintain my strengths, address weaknesses, and prioritize effective collaboration and adaptability when planning projects.

7.3 Areeb

Overall, my experience within the group was positive. I learnt how to effectively communicate my ideas in a concise manner, which improved the overall clarity of our discussions on the overall design implementation. I improved my adaptability in modifying code to fit changing project structure, showcasing flexibility and problem-solving skills. My strengths in programming, particularly in loading and storing functions to and from memory, were utilized well and contributed to the group's progress. However, I did encounter a weakness in debugging, that I overcame by seeking help from my peers. One area I intend to improve is my time management, as I underestimated the time required for each task. Moving forward, I better prioritize my responsibilities. One aspect that greatly contributed to our group's success was open communication through consistent meet-ups and collaborative problem-solving. I recognize the value of this approach and aim to maintain it in future group projects.

7.4 Mahdi

I motivated the team and lead by example. I worked with 2 of my friends so I was mostly comfortable working with the group, as we had worked together in the past. Suweda was a new addition, but I quickly became acquainted with her, and we did quite a bit of pair programming. I spearheaded the debugging process and made it easier by creating various printing programs. I worked very hard and stayed behind extra hours to ensure the emulator and assembler was complete and running as soon as possible. However, I received feedback saying that I needed to relax and in hindsight I could have been more productive at times by taking rest breaks. I also may have pressured my group because I expected the same level of input. All in all, I believe I applied myself to the fullest and contributed to a positive working environment.

8 Conclusion

The Assemble and Raspberry Pi project involved developing an Assembler and writing assembly code to control the Raspberry Pi's LED light. The project utilized modular design, linked lists, symbol tables, and function pointers for efficient code organization. Challenges such as memory leaks were addressed, and testing was conducted in phases with debugging tools. Effective communication and organization were key factors for success, although time management was identified as an area for improvement. Overall, the project showcased collaborative work, problem-solving skills, and valuable learning experiences for future projects.