# Final Report: ARM11 Assembler and Extension

Seiya Aoki, Jimin Park, Alex Liu, Isabel Li

14th June 2019

## 1 Short introduction to the project: Assembler and Extension

Our task was to implement an ARM assembler for translating an ARM assembly source file into a binary file, that could for example be executed by our emulator. We decided to make an extension consisting of implementing a 4-bit binary counter using a Raspberry Pi and LEDs.

## 2 Implementation of Assembler

We decided to structure our assembler using two passes. An Assembler struct was used to carry information that is used across multiple instruction. Further information is given below.

### 2.1 File Loading

We first loaded the file into a buffer of maximum instruction size 511 characters, and maximum instruction count 32768. The instruction count is 32768 for valid inputs because the maximum possible instruction count is 16384 assuming 4 byte instructions and a 64KB memory machine. If every instruction had a corresponding address, this will double the instruction count, thus giving us 32768.

### 2.2 First Pass

Secondly, we iterated through the instructions, and associated any labels with a memory address in a hash table. We also split each instruction into a split string structure as described below. An Assembler struct was declared here, which stores the label symbol table.

- We therefore needed to write our own hash table data structure. We made a structure for the items in the hash table, which would contain key/value pointers of type void* so that we could insert pairings of any type into the hash table. However, this meant that we needed a wrapper function that cast the keys and values as required when inserting and searching. We created such a wrapper for a string key and int value pair. We initialised hash tables at run-time relating mnemonics and suffixes to their corresponding enums, so that we could easily convert the strings from the assembly source file to enums that would set the correct flags in the instruction struct. The same combination was used for the symbol table that related labels with memory addresses.

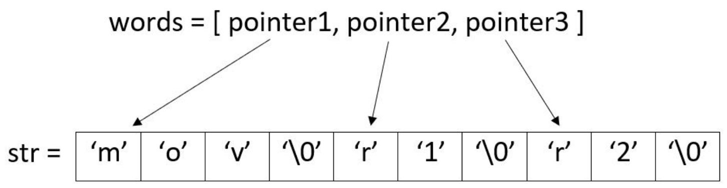

Figure 1: SplitString

- We then tokenised each instruction, splitting by spaces and commas (whilst preserving square bracketed terms). We decided to create a SplitString struct to contain the tokenised instruction, as it conserved memory. (refer to Figure 1). The struct stored the number of words in the instruction, an array of pointers to the beginning of each word in the instruction, and a variable of type char* storing the words and their sentinel characters. This was so that freeing the allocated space to the array was simpler, and made our code more efficient with less redundancy.

## 2.3 Second Pass

Before starting the second pass, we must first declare and set the index where constants would need to be stored in assembling certain load instructions. Following this, we performed the second pass.

- Our struct converter would take the SplitString, and convert it into our pre-defined instruction structure from our emulator. The Assembler struct is also passed in. We used an array of function pointers to call the appropriate struct converter based on the instruction type, deciphered using our mnemonic to enum hash table.
- Since some of the parameters that were passed into our struct converters, such as binary output, current index and constant index, were independent of the current instruction, we implemented an Assembler struct to simplify the parameters. Assembler also contains all the hash tables we need, so that the hash tables wouldn't need to be created every time we searched through a hash table, therefore making our code more efficient.
- Each struct converter would set the correct flags depending on the type of instruction given. Setting the operand for `to_struct_data_processing` and setting the offset for `to_struct_single_data_transfer` sometimes had corresponding logic, and therefore we conflated some of the code using helper functions, such that there was no duplication. We utilised the hash tables we'd already created for replacing label references with their corresponding address, and using the mnemonic to enum hash tables to set the correct flags for the generated instruction structs.
- Finally, our binary converter would generate its corresponding binary encoding from each structure using a series of shift and bit-masking operations, and write it to the given output file.

- We ran valgrind to check for memory leaks, and initially there were quite a lot of memory leaks since we didn't consider freeing mechanisms for our structs. As a result, we created freeing functions for SplitString and hash table and made sure to free any space that we had allocated but no longer needed, so that our memory didn't become bloated.

We reused the structures we created, and the bit-masking and shift operations from our emulator in our design for assembler.

## 3 Extension: 4-bit Binary Counter using a Raspberry Pi and LEDs

### 3.1 Specification

Our extension is to implement a 4-bit binary counter with parallel LED outputs using a Raspberry Pi. Each LED represents a digit in the binary number system, and the binary 1 and 0 are represented by turning LEDs on and off. The four LEDs are connected to Raspberry Pi through GPIO pins with resistors in series.
You can see an example of our extension in action here. `https://youtu.be/ckxNMscShpg`

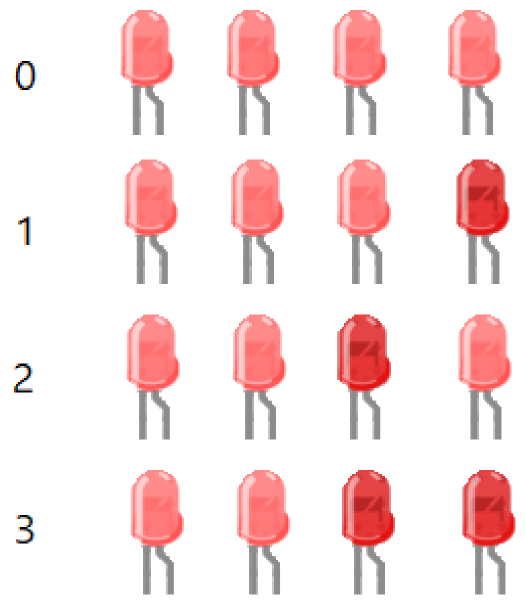

Figure 2: LED Binary counter

### 3.2 Implementation

For this project, we used a Raspberry Pi kit, a breadboard, 4 LEDs and 4 resistors. We connected 4 selected GPIO pins (pin 14, 15, 18, 23 for the 1st, 2nd, 3rd and 4th binary bit respectively) to the corresponding LEDs via their limiting resistors.

The counter starts from 0 (all LEDs off) and counts up to 15 (all LEDs on) incrementing every second. This is equivalent to counting from 0000 to 1111 in binary. When the counter reaches 15, it overflows and takes the next value as 0. This repeats until the program terminates.

In the code, we first set pins to their output modes. Then in a loop, it processes and increments the counter every second. The function `process_number` first prints out the counter in decimal and binary on the terminal, then extracts each bit from the counter by applying corresponding bit masking, and sets each GPIO pin mapped to that bit.

We saved the code in `binary_counter.c`, so we can compile and run this program on a Raspberry Pi to start the counter.

### 3.3 Challenges encountered

- We did not initially have access to a USB keyboard, so we were forced to write commands in a text file on a pen drive on a different computer, and paste them with a mouse on the Pi later.
- We naively did not realise that the the index of pins were not the same as the GPIO number in the wiringPi library. This caused our output to be different than expected. We fixed this by checking which wiringPi pin index corresponds to which GPIO pin, and correcting our previous oversight.

### 3.4 Testing

We tested our extension by compiling, executing and observing if our binary counter was correct and working. Initially, our output was incorrect as we had the aforementioned problem with the discrepancy between the wiringPi pin number and GPIO pin number. Once we fixed this, our output was working as intended.

### 3.5 Example uses

- Teaching purposes, for example our counter can be used to educate children on how to count in binary and how the binary system works.
- As a 15 second timer, as currently it counts in one second intervals.

### 3.6 Future Work

In the future, we will potentially implement:

- A way to set the time it takes for the counter to increment.
- Add a functionality to set the beginning and end number.
- Add greater functionality to include further LEDs to allow greater numbers it can count to.
- Replace our LEDs with ones which can have more than one colour, so it could count in ternary or binary for example.

## 4 Addendum

### 4.1 Emulator

We also changed some minor features of Emulator. Firstly, we changed the file structure by organising the files into a folder. Secondly we optimised the decode function in our emulator so that the overall code was shorter, and reduced the amount of function calls. This was so that it could be run on slower machines.

### 4.2 Utils

We created a separate utils folder in the `src` directory to contain utilities used in our emulator and assembler. Some were only used specifically in assembler.

## 5 Working in a group

We split the work of writing the assembler quite naturally following on from writing the emulator. For example, members would work on the same instruction types in both parts. We set out a list of functions or structures that needed to be written or fixed, and worked through them together such that every member had work to do. We convened at least three times a week to work on the project, and if we weren't together we were in contact via Discord or Messenger. In this way, if any of us were stuck, it was easy to ask for help from the other members. Our ability to keep communicating with each other about our progress and assisting one another is something that we would keep the same.

Whilst working through the project, we realised that parts of our design were inefficient, and therefore had to make changes to the rest of our program. So if we were to do things differently, we think we should've spent more time doing preliminary discussions about our implementation to avoid these problems.

Overall it was enjoyable to program in a group. We feel we learnt valuable skills, like the importance of frequent merging on Git and commenting our code such that other members can understand the code we'd individually written.

## 6 Individual reflections

### 6.1 Seiya

I found that working together in this group was both very productive and enjoyable. We were all quick to lend a hand when help was needed, and our strategy to split the workload and help the others when finished worked out well for us. I believe that we were able to maximize each of our potentials during this project due to this. One of my main weaknesses that I came across in this project is my lack of ability to plan the overall structure of the project at early stages. Due to this, we had to carry out refactoring and restructuring, as many parts were inefficient or redundant. Despite this, I believe we were able to eventually create a strong project structure. When working with a different group of people, I hope to be able to carry on being able to set and meet realistic and effective deadlines throughout the project, as well as my ability to debug the code to meet the specification.

### 6.2 Jimin

It was an exciting experience to work on the project. I think we worked efficiently as a group since we tried to plan before writing code, balance the workload evenly, and helped each other. One of my weaknesses was inconsistency in my code. My code style and variable naming was inconsistent, so I had to refactor and restructure my code. In the future, I would try to be more consistent and add meaningful comments as well as using GitLab branches, and write meaningful commit messages, so that other team members can understand what I'm doing more easily. I would also try to be as cooperative and collaborative as I did for this project when working with a different group of people.

### 6.3 Alex

I found working on the C project with this group an enjoyable yet also challenging new learning experience. One of my weaknesses was forgetting to comment my code after I had written a block so that others would have to ask me what the function did and how it was done, instead of being able to understand it using comments. In future, I would definitely be more mindful of this, to help make progress flow more smoothly. I think that I fit well into the group and allowed myself to be helped by others as well as offering help to others. As for working with a different group of people I would like to maintain the high level of communication I had with this group.

### 6.4 Isabel

I had a lot of fun working on the project together. One of my weaknesses was that I had a lot of bugs in my code, and inefficient function structures, which I then had to spend time trying to optimise and get rid of my bugs. Therefore in ensuing projects, I think I need to be more conscious of writing correct code, and logically work through what the most efficient way to write my code is, such as using ternary operations, or combining or reducing the number of loops. It was invaluable to be able to ask my peers for assistance whether it was in person or online when working on my section of the project. In the future, if I were working with another group of people, I would try to be just as collaborative and communicate well with my other team members, as we have here. I hope that I'll work efficiently and effectively, so that I won't have to spend too much unnecessary time fixing my own code.