C-Programming Project:
ARM11 Emulator and Assembler
Final Report

Louis Aldous, Tao Wang,
Mohammed Abdus Samad Hannan, Armaan Sabharwal

June 18, 2021

1 Introduction

For this project, we were given the task of implementing an ARM11 emulator and assembler given a subset of the ARM instruction set. The emulator would execute a binary instruction loaded from a binary file while the assembler would take an assembly source file and process the assembly code into a binary instruction which would then be executed by the emulator.

2 The Assembler

2.1 How we split up the work

As with the emulator, we believed the most efficient way to get everything done on time was to separate the task of implementing the assembler into different components which was then delegated to the various group members.

The work was delegated as follows:

- One group member was responsible for implementing the two-pass assembly system which included the main control flow in the assembly source file, the implementation of the symbol table ADT and the assembly function for Branch instructions. This group member also created the tokeniser which broke a line of assembly code into its label, opcode and operand(s) and was used for the assembly functions for every instruction type.

- Two members were assigned to manage the assembly of Data Processing instructions. One of these members also created the binary file writer. The other member was in charge of also implementing the Special instructions `andeq` and `lsl`. This was because the two special instructions could simply be implemented as an extension of the Data Processing instructions.

- The final member was given the task of implementing the assembly functions for both Multiply and Single Data Transfer instructions. This was due to the fact that Single Data Transfer was arguably the most difficult to implement and even included working with Data Processing thus it would take the most time and delegating other tasks to them would not be time efficient. However, they were also instrumental in handling the control flow of the program.

As a group, we decided to use the two-pass assembly system as opposed to a one-pass as we believed it to be the simplest way of performing the assembly.

2.2 Structure of the assembler

Each task had its own source file which the delegated member worked with. This was done in order to prevent as little merging conflicts as possible and also so that the code in the main assembly source file was kept as clean as possible. The only exception to this was the Special instructions which were implemented in the same source file as Data Processing Instructions. For each task or any major changes being made e.g. debugging, a new git branch would be created. Once the task was completed, thoroughly tested, and believed to be correct, the branch would be peer reviewed by other members of the group before being merged with the master branch. This made sure that the code in the master branch was always correct and functional.

In terms of dependency, this meant that the assembly source file is dependent on every source file made for the assembler either directly or indirectly, including the source file named `src/utility.c` which contained common utility functions and macros that were used across various source files for the assembler and the emulator e.g. a little endian converter. In order to implement this efficiently as opposed to compiling every source file for the assembler along with the main `src/assembly.c` file, the Makefile was used to prevent slowdown as the number of files for the assembler grew.

2.3 Implementation of the Assembler

Our main assembler function, outlined in `src/assemble.c`, when called on, takes in the first command-line argument, the name of the assembly file, and calculates the maximum number of labels from that assembly file i.e. the number of instructions. It then creates an instance of our hash table ADT of that size before passing the table and the assembly file through the two-pass assembly system, resulting in a number of binary instructions being written to the binary file specified by the second command-line argument.

2.3.1 The assembling functions

The two-pass assembly system was defined in the files `src/firstpass.c` and `src/secondpass.c`. In the first pass, we map each label to a memory address using our hash table data structure. In the second pass, a hash table is created to associate every instruction with their instruction type (1 for Branch, 2 for Single Data Transfer, 3 for Multiply, 4 for Special, 5 for Data Processing). A hash table is also made for mapping Data Processing instructions to their opcodes and Branch instructions to their condition codes. The relevant assembling function is invoked after looking up the mnemonic in the type table. The `secondPass` function terminates after using the binary file writer to write the 32-bit instruction returned by the assembling function to the binary file passed in as `assemble`'s second command-line argument.

In the file `src/assdataprocessing.c`, both main assembling functions take in a pointer to the Data Processing hash table and the instruction string and return the correctly assembled instruction as a 32-bit unsigned integer. `assembleSpecialInstruction` requires the data processing hash table due to the fact that `lsl` is translated to a `mov` with a shifted register operand, and `andeq` is an `and` instruction but with `eq` as the condition code. `assembleDataProcessing` tokenises the string instruction, then looks up the opcode in the hash table, and inspects both the number of operands and the operands themselves to determine the operand's type (immediate or shifted register).

In the file `src/assemblebranch.c`, a pointer to the Branch hash table, a pointer to the labels hash table, the string instruction and the value of the current program counter are all passed into the `assembleBranch` function. The string is tokenised, the opcode is looked up in the hash table, and the operand is inspected to determine whether it is a label or not. If so, the label is looked up in the labels table and the `addr` variable is given its value otherwise `addr` is given the value of the operand. Using this, the value of the program counter and the 8-byte offset due to the pipeline of the emulator, the offset is calculated, and the 32-bit instruction is assembled and returned.

The `src/assemblemultiply.c` file contains the `assembleMultiply` function which is arguably the simplest type of instruction to implement as the only token needed to inspect after tokenising is the mnemonic as only the A bit differs depending on whether the instruction is `mul` or `mla`.

We also have a file `assembleSDT.c` which provides the assembling function for single data transfer instructions. `assembleSDT` takes in the string instruction, the last memory address, the program counter, a pointer to the Data Processing hash table, and an array containing the previous memory addresses. `assembleLDR` or `assembleSTR` is called depending on the mnemonic. `assembleLDR` requires a pointer to the Data Processing table as in some cases, it can be treated as a regular `mov` instruction thus it can be passed onto the `assembleDataProcessing` function in `src/assdataprocessing.c`.

2.3.2 The tools we needed to make

The hash table data structure is our symbol table ADT we implemented. This file outlines the structure of the hash table and contains the functions to create and add items to it as well as the main purpose used in our project which is a function to look up the value matched by a unique key.

In `src/tokeniser.c`, we have a function in which we pass in a string and the number of tokens we expect there to be and it returns an array of strings filled with those tokens in the order received. The delimiters used to separate the tokens are a space `' '`, a colon `':'`, a comma `','` and a newline character.

The binary file writer, defined in `src/binfilewriter.c` contains `fileWrite` which takes in a 32-bit unsigned integer instruction, and the name of the file to be written to. This function writes the binary instruction to that file.

Finally, we have a file `src/utility.c` which was also used in the emulator. It contains a variety of common utility functions and macros which are used within different files in our assembler and emulator, and it allows us to include the header file `src/utility.h` in the file(s) which we want to use any of these functions without repeating code or using magic numbers.

3 The Extension

3.1 Description of Extension

For our extension, we decided to implement a virtual emulation of the physical GPIO (general purpose input output) pins that would be found on a Raspberry Pi Model B (revision 2). Not only did this allow us to use our emulator and assembler to pass the automated tests which involved using the GPIO pins, it would also allow us to see the effect of certain instructions which use the GPIO pins without physically needing to have the device on which it would run on. In addition to this, we used ASCII art within our code in order to print out to the terminal a schematic

representation of the GPIO pins found on a Raspberry Pi which also included indicator signals for each of the individual pins and allowed us to display the status of each of these to the user of our program in an intuitive way, without requiring the design of a completely new graphical user interface which was not possible given our strict time constraints.

3.2 Design and discussion of Extension implementation

One of the issues we had in implementing the extension was how to implement it in such a way that it would not break the tests that have already passed. This was a concern as our extension would use stdout as an output stream so therefore it would break the emulator tests. This was fixed by creating a new directory exclusively for the extension which would keep it as its own standalone emulator and therefore would not break any of the tests. The ability to assemble GPIO instructions was also added to our assembler.

Our first task in implementing our extension was getting the GPIO tests to pass correctly, as this was not done in the first emulator section. This was not too hard, and only included the introduction of some print statements and also required a method of emulating the actual GPIO hardware that would be on the Raspberry Pi itself. For this we mapped the physical GPIO addresses to virtual ones in our emulator, which allowed us to emulate the GPIO pins themselves so we could then turn them on and off. After this it was just the case of adding some print statements in order to pass the necessary tests in the test suite. The next step in our extension was to display the status of the GPIO pins in a meaningful way to the user. This was done by using the mapped physical addresses to create a colour-coded interface in the terminal. We used the character 'o' to represent a pin, and if it was green in the interface then it means it was activated/turned on. Our interface was a grid of these pins with their respective pin number next to them (which was also colour-coded). Our testing of the GPIO built upon the tests already present in the test suite. This was done to fully check that all pins had the capability to be activated and properly emulated. The testing also included assembling GPIO instructions, which was successful. We believe this method of testing to be effective as the GPIO tests in the supplied test suite tested all types of instruction as well as different layouts of assembly file.

Unfortunately, due to time restrictions, we could not implement our plan to emulate hardware connected to the GPIO pins of the Raspberry Pi such as a LED bulb or a temperature sensor. However, as a group we are satisfied with the extension in its final state as it has achieved the main aim of what we planned for it to do. It correctly emulates the activation of GPIO pins, passes all emulator tests, and is capable of assembling GPIO instructions into binaries.

4 Group reflection of programming in a group

In order to designate the work between different members of our group, we initially asked if anybody had a preference to do a certain part of the project, to allow us to work to our strengths where possible. If not, the work for both the emulator was randomly designated to ensure that an equal amount of work was being assigned to each member of our group. Due to the nature of this project, it enabled us to carry out pair programming which we found was a very effective and quick way of completing the tasks as it allowed us to frequently review each others code to ensure functionality but also suggest any alternative, more efficient methods or stylistic changes that could potentially be made. Similarly, for the assembler we decided to use the same pairs of people in any instances

where pair programming was necessary as we had previously developed an understanding of each others style of programming which allowed for a much smoother workflow in the second half of the project. All in all, however, we made sure every member could understand every part of the code and there were times where the entire group would work together to solve specifically challenging problems or find and correct any difficult bugs.

In terms of communication, we utilised a group chat on Discord which allowed us to contact each other constantly via instant messaging whenever required, but also facilitated our calls on Monday, Wednesday and Friday mornings which allowed us to catch up with each other as a group all at once so that we were aware of the progress being made by other group members on their respective parts of the project and whether any help was needed or whether there was other work that could be done to further improve the program. In future scenarios, we believe that using a Discord server with separate channels could be more effective, as we could have a channel designated to each different section of the project, allowing for more direct communication as well as making it easier to refer back to previous conversations for reference. We believe this method would be an upgrade on the method of communication we used during this project and could have helped optimised our workflow even further.

5 Individual Reflections

5.1 Louis

The project has been a great learning experience for myself. I learn programming best by taking on big projects like this one to really get into the language I am trying to learn. C is completely different to any other language I have used before as it never felt like there were limits to what you could do with the code. It also gives control of the program completely to the programmer, everything that would be done automatically in other languages (e.g. memory allocation) must be done manually which is something I have found fun (although sometimes this has caused more harm than good). For me, the best way to describe the language would be rewarding but unforgiving. It has also been my first collaborative programming project and so I have also gained great experience with working in git (e.g. branches, semantic commit messages, merge requests) in addition to the general workflow for implementing an application collaboratively.

As this was my first project in C, many mistakes were made, and many lessons were learnt. At the start of the project, I had somehow convinced myself that a char only took one bit of memory rather than a byte which definitely caused some problems before I realised what a massive mistake that was. The biggest problem I came across was segmentation faults occurring in my code. Fortunately, GDB was a great tool to help me get rid of these errors and by the end of the project I was able to not only avoid segmentation faults but also fix them quickly if they did occur.

Overall, the project was very rewarding and a great learning experience. I am very happy that the project went so smoothly and that I got such a hardworking group.

5.2 Tao

I will admit that I was a little intimidated by the thought of having to participate in a group project to begin with, considering the fact that I've been studying remotely for the entire year so opportunities to socialise and make friends were few. However, working with people from my tutor group who I was familiar with quickly changed that. I feel that we set out on the right foot

by scheduling a call on the first day of the assignment to devise a strategy. This paid dividends as everyone was familiar with their tasks and it allowed for flexible working hours with little to no stress. Onto the more technical side of things, the process of learning C was easier thanks to the early release of lecture slides on Materials, in alignment with the conciseness of lectures and preparation for the Lexis examination. I think it's also worth giving a mention to the C Tools lectures, in particular Valgrind, which helped me swiftly identify where segmentation faults occurred (which felt inevitable). I am thankful for such an amazing group and look forward to future group projects.

5.3 Abdus

I thoroughly enjoyed working on this project. Not only was the project very interesting and fun to work on, I was with was an amazing, hardworking team that I was able to fit in well with and work efficiently with.
In terms of the project itself, I initially did not have high hopes for myself and my abilities since this was my first time working on a programming project as well as my first time learning C therefore the idea of learning C as the project went on was a concern. However, this did not seem to be the case and I seemed to learn C as I did my project work quite well. Also, having group members who were learning C for the first time as I was and having a group chat where we all helped each other was very effective in turning my weaknesses into strengths. Therefore, I would have to say the great communication we had as a group helped a lot with our progress as learners of the C language and also the progress of the project.
I think a problem I faced was having the C Programming Exam right in the middle of the project as it meant a significant portion of time had to be spent honing my skills to a good enough standard to be able to do well in the C Test. However, if I were to do this again, I would make effort to learn the language at a good enough level before the project starts. That way, not only would I have to worry too much about learning everything in time for an exam midway through the project, I would also be able to think of more possible extensions we could implement with a wider and deeper knowledge of the language and its capabilities.

5.4 Armaan

Personally, I believe that I fit very well into the group, as we all were able to choose our preference on certain parts of the projects which allowed me to work on the parts that I found most interesting, as well as allowing a continuity of work between both the emulator and the assembler, as by working on the data processing instructions in the emulator it gave me a better understanding of how it worked, which I believe positively impacted my work in the assembler.
I also think that something which helped quite a lot during our group work was our switch to using semantic commit messages on Gitlab, which made it easier to understand the changes that had been made to the code by other members of the group. This is certainly something that I would like to adopt when working on future group projects, but also when working individually. The frequent communication with other group members is something that I found very helpful as it ensured that all of us had the correct understanding of our tasks, as well as being able to help each other with their respective parts.