ARM11 Final Report

Devesh Erda, Eugene Lin, Raihaan Rahman, Prabhjot Grewal

June 14, 2019

1 ARM Assembler (Part II)

After completing the emulator we found that implementing the assembler was a simpler task, given that we had gotten to grips with some of the more theoretical concepts of the ARM platform, such as the way that the memory is read and written in little endian or how the different instructions were processed and constructed.

Given this newly-gained knowledge, we were able to reuse certain elements from the emulator in our assembler's implementation. This was not in the format of reusing exact code or files, but in how instructions were formatted and the logical processes that were common in both tasks. For example, the way the shifter is encoded in the emulator helped us to plan out and decode the different shifting cases when dealing with certain load instructions or data-processing instructions. Structurally, we knew how many and the sorts of information that was encoded into every instruction; for certain instructions such as branch, there were only a few pieces of information to decode from the assembly code, and a few flags to set, conversely the single data transfer instructions had a lot of intricacies to deal with.

1.1 Overview

We decided that it was much simpler and concise to use a 2-pass system to convert each of the instructions to their binary equivalents. The first pass reads in the file, building up a symbol table of labels so that any branch instructions could be given the necessary address to jump to. Furthermore in this first pass, we also keep a count of the number of instructions and how many load instructions that require memory storage. This is done so that we can dynamically allocate the correct amount of memory for storing the assembled binary.

In the second pass through the assembly source code, we decode each instruction depending on the instruction type, and then convert it into its binary representation. Then this binary representation is written to a file, taking into account any data that is appended to the end of the source code of the program.

Referring to code organisation, we separated the files into the main file (`assemble.c/h`), a set of helpful utility functions that were commonly used in different parts of the assembler's implementation (`assemble_utils.c/h`), a file with all of the parser functions (`parser.c/h`). Also, we included a header file containing struct definitions (`assemble_struct_defs.h`) and a set of constants that are used throughout the assembler, relating to the formatting of ARM's instruction set (`arm_assembler_constants.h`).

1.2 File Reading

When the file is read in, the first pass iterates through each line in the file. It checks for it being an instruction, a label or even an empty line, and acting appropriately: incrementing the instruction counter and possibly the number of extra data stored, adding a new entry to the symbol table according to the label name, or ignoring the empty line respectively. The symbol table is built as a linked list data structure holding the label name and relative address of the instruction the label points to.

If the current line is an instruction, the assembly code is also stored into a new array line-by-line, making sure to add a terminating character onto the end of the string. This array is dynamically allocated on the heap utilising the instruction count and data count from before. This array is then what is passed onto the next pass of the assembler to parse and then convert into binary.

1.3 Parsing Instructions

Iterating through each line of the instructions read in from the file, we first decode what instruction type the current instruction is. We then use a mapping from each instruction type (such as `add`, `bne`, `ldr`, etc.) to a parsing function that is able to handle the different cases and structures of the instruction.

Calling this parsing function invokes a series of string tokenisations and comparisons to decide which case the current instruction falls within, and so what each piece of information in the instruction represents. Additionally, it also identifies which flags and conditions need to be set if any.

The different cases were clearly specified and we are given the assumption that any assembly code that is fed into the assembly is well-formed and falls into one of the given formats. For example, the load instructions have 8 different cases due to the ability to perform loads using different addressing modes (immediate, direct, indirect), as well as pre and post-indexing addressing for the load instruction.

Within each of the different cases for each instruction, various flags must be set, registers utilised or shifting types specified. This is all encoded into binary using the same format given by the emulators specification, and is handled within the parsing function. So theoretically, the output from our assembler could be passed in as an input to our emulator or any ARM processor (with the same specification) and be run as a program on the machine-code level.

An additional caveat with one very specific case in the load instruction was that data had to be written to a memory location just after the end of the current programs source code. Then the instruction was able to reference this data by memory reference using an offset. So within the parsing function for load, that case also handles the writing to that memory location after the rest of the assembly code is written as binary.

1.4 Writing Binary to File

Following the conversion of all assembly instructions to binary, the binary representations are written to a file in the correct little endian format, which again matches the specification of the emulator. At the end of the programs source code, any additional data for load instructions have been appended in the same formatting, so that data is also written to the file.

1.5 Problems we Faced

We initially struggled on how to efficiently and accurately allocate enough memory for the assembly binary instructions due to the massive variation in inputs containing labels, blank lines, and then possibly needing to add additional data at the end of the program source code. We solved this by deciding to keep a count of the instructions during the first pass of the assembly code, as well as immediately identifying any instructions that would require extra data to be stored and counting them too.

2 General Purpose I/O (Part III)

2.1 Adapting the Emulator

To test the kernel binary, we had to modify the emulator to check for accesses to the GPIO pins. This was done in the function which simulates the data transfer instructions. During a load instruction, the emulator checks to see if the instruction loads specific values, in particular `0x20200000`, `0x20200004`, and `0x20200008` (which check the pins 0-9, 10-19, and 20-29 respectively). If it is not one of these, the emulator then checks to see if the value is valid, if not it then prints an error message. For the store instructions, the same checks occur as for the load instruction. It then checks `0x2020001C`, and `0x20200028` to check if the pins need to be turned on, or off. Once again, if the value used is not valid, it prints an error message.

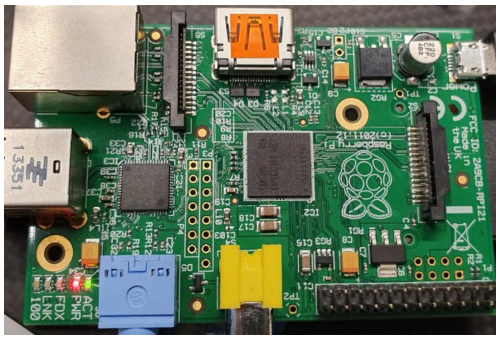
2.2 Blinking GPIO 16

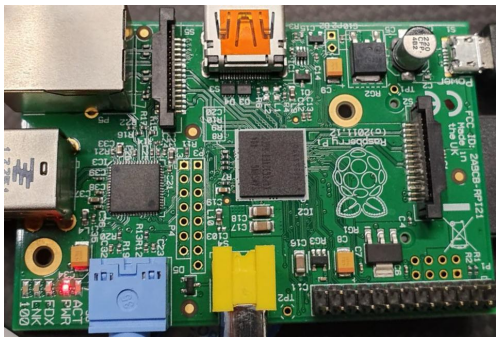
To allow our assembly to be more legible, we stored the addresses in registers 0-2. We first read the data from memory location `0x20200004`, since were dealing with pin 16, and then `AND` it with a bitmask to clear the pin, and then `OR` it with another bitmask to set it as an output pin. From here, we can start an infinite loop, which does the actual blinking. We store the value for the current state in register 8, and a mask for it in register 4. Every iteration, we set $\texttt{r8} \mapsto \texttt{r8} \oplus \texttt{r4}$, which allows us to toggle a single bit. This is then written to memory after the pin is cleared. Our delay is done by counting down from a high number (`0x00FFFFFF`), thus taking up cycles.

2.3 Problems we Faced

We initially struggled on how to efficiently and accurately allocate enough memory for the assembly binary instructions due to the massive variation in inputs containing labels, blank lines, and then possibly needing to add additional data at the end of the program source code. We solved this by deciding to keep a count of the instructions during the first pass of the assembly code, as well as immediately identifying any instructions that would require extra data to be stored and counting them too.

2.4 Blinking Lights Images





3 Extension (Part IV)

3.1 Our Idea

We created an LED matrix of addressable bi-colour LEDs which are configurable to show whatever simple graphical images or user interfaces we could want to display on the 7×7 grid. Utilising the matrix as a display, we have created a version of the popular game *Connect Four*. Then to expand on this and explore some more complex concepts, we implemented a "Computer" player to play against that uses Monte Carlo Tree Search for its decision process. The program controlling the LED matrix, running the game, and performing the heuristic search algorithm all run on a Raspberry Pi 3B and were written in C.

We thought the game *Connect Four* would be a simple way to utilise the LED matrix and the two colour functionality of the LEDs themselves, as well as being a fun and interactive way to utilise the Raspberry Pi and its GPIO capabilities. Furthermore, it gave us the opportunity to delve into experimenting with different data structures and the challenge of trying to implement certain algorithms in a language style many of us were just starting to become familiar with. For example, one algorithm we had to try to design and implement was how to check for a winning move without having to scan every single placed piece and its neighbours for an uninterrupted line of 4 pieces of the same colour.

After implementing the simple *Connect Four* game using standard input and output relatively quickly, we began working on the two more advanced sections of the extension: building and controlling the LED matrix, and the Monte Carlo tree search algorithm.

The bi-colour LED matrix uses a shared cathode for both the red, and yellow, LEDs, with individual anodes for each colour. Our prototype board is a 7×7 matrix, which requires 14 pins for anodes (since each LED requires 2 anodes), and 7 pins acting as cathodes. It uses the standard design for an LED matrix, which turns on each row (hence setting the cathode to a low value ($\approx$ 0v)), and setting the anodes to high ($\approx$ 3.3v) if theyre enabled.

The heuristic search algorithm was implemented to expand the functionality beyond just a simple 2-player connect 4 game. The algorithm is able to pick what it deems to be the most optimal move by searching the current game space in a tree structure and using random simulations to build up the tree and each moves' relative "value". Monte Carlo Tree Search is a greedy algorithm, but with tunable parameters to optimise for the trade-off between exploration and exploitation.

An additional piece of functionality we added to our extension was the ability to show scrolling text, of which is used extensively in digital signage. We utilised it at the end of the game to show the winner of the game, but it has been built to be able to be expanded to other use cases and theoretically LEDs matrices of larger sizes.

3.2 Connect Four Implementation

After displaying an initial welcome screen, it takes in an input to choose which "game mode" to run. This is specifying whether player 1 and 2 are human players (therefore inputting their choices via standard input), or should be a "Computer" (which runs the Monte Carlo Tree Search algorithm for its move-making decisions). The core game functionality runs in a simple game loop:

1. Display current game board
2. Request an input from current player (whether human, or computer)
3. Place the piece on the board
4. Check if it was a winning move, or if the board is full
5. If it was not a winning move, and the board isn't full, switch current turn to be the other player, and continue from step 1

6. Otherwise, display the end state, and display scrolling text corresponding to the state

The game board is represented as a 2-D array of enums of size 7 by 6. The enum used, `piece`, has three possible states: `RED`, `YELLOW`, and `CLEAR`. These are also used to represent which player is currently playing and which player has won (`CLEAR` as the winner means that no winner has been decided yet, however if the board is full, and `CLEAR` is still the winner then a draw state is reached).

3.3 LED Matrix Implementation

For ease of explanation, its better to consider each LED as two individual LEDs (one red, one yellow), with a shared cathode. The cathodes of each row are joined together, and connected to the same GPIO pin. The anodes of each column are joined together, and connected to the same pin. Our build was done on a copper stripboard, as that allowed us to skip having to manually connect the anodes - but it forced us to have to break connections for the cathodes to prevent short circuits. We used female PCB headers to allow easy insertion, and removal of jumper cables. No resistors were needed, since we were pulsing the LEDs rapidly.

We used Gert van Loo & Dom's method (available at `https://elinux.org/RPi_GPIO_Code_Samples`) to directly access the GPIO pins on the software side - thus allowing for faster operations. In our initialisation code, we first turn off each row by setting the cathode to `HIGH`, and all the anodes to `LOW`. We start our scan from the top, and turn on the row by setting the corresponding cathode to `LOW`, thus allowing current to flow through the diode. The anodes are then enabled from left to right, if they need to be turned on. There is then a delay to allow for the lights to be brighter, and then each column is turned off in the order in which they were turned on. Finally, the row is then turned off. As this is done very quickly, it appears to be a solid image, with very little flickering.

3.4 Monte Carlo Tree Search

The Monte Carlo tree search algorithm uses a tree data structure to make its decisions. It is implemented using two structs - one for the whole tree, one for each node in the tree. The tree struct holds a pointer to the root node and the current game board. Each node in the tree has a pointer to an array of child nodes (can be NULL if the node represents a terminal state in the game or children have yet to be "discovered"), a pointer back to its parent node, the move in the game that the node represents (e.g. column 5 by `RED`), the number of visits the node has, and the number of wins that node has been on the path of.

The algorithm itself is a general search algorithm that works for any decision space where each move can be described as a state and action and simulations can be run to playout a state until a terminal state is reached. There are 4 stages:

1. Selection Phase

   - Starting at the root node, recursively choose the most promising child node until a node with uninitialised (but possible) child nodes or a terminal node is reached. The algorithm evaluates how promising each child node is by using the Upper Confidence Bound 1 formula (UCB1) applied to trees. It is a formula that evaluates each node with a trade-off between choosing child nodes it knows already perform well (exploitation) and choosing child nodes it does not have relatively significant amounts of information about (exploration).

     $\text{UCB1}(\text{node}_i) = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln(N_i)}{n_i}}$

     w_i - number of wins of the node

     n_i - number of visits to the node

     N_i - number of visits to the node's parent

     c - tunable exploration constant (i.e. the higher its value the more bias the selection process will have to less-explored nodes); after researching and testing, we discovered that the theoretical optimal value of c is $\sqrt{2}$

2. Expansion Phase

   - If the leaf node is not in a terminal state, create a child node from the leaf by randomly choosing a move.

3. Simulation Phase

   - From this new node (or the terminal node) simulate the rest of the game until a terminal state is reached.

4. Backpropagation Phase

   - Taking the result from the simulation, backpropagate the result back up the tree until it reaches the root node, incrementing the number of visits and incrementing the number of wins if the simulations winner matches the player for that particular node.

This cycle is run several times, according to `COMPUTATIONAL_BUDGET` - a constant predefined in `constants.h`. Then once the cycles end, the algorithm must choose the most optimal move by picking the best child node of the root node. As Monte Carlo tree search is a greedy algorithm, picking the final decision move is done in a greedy manner by choosing the child node with the highest number of visits. This move is then returned back to the rest of the game loop.

3.5 Scrolling Text

We decided to use a 5×5 pixel font (available at `https://www.dafont.com/5x5-pixel.font`). Our method was to reserve the first 25 bits of a `uint32_t` to store whether a pixel was white (`0`), or black (`1`). Then we have 4 unused bits, which are set to `0`, and the final 3 bits are reserved for storing the width of the character (and can be easily read with a bitmask). The first 5 bits are used to store the first row of the character, and the next 5 for the second row, and so on.

3.6 Problems we Faced

Initially we had problems with deciding on a feasible, yet intriguing idea for an extension. We thought about utilising the Raspberry Pi and a camera input to perform object tracking in real time, possibly connected to servos in order to keep the subject in the centre of the frame. Unfortunately this idea would have likely required OpenCV which uses C++, which was not in the scope of this project.

After deciding on the project format, we encountered some limitations with our design of the LED matrix. Even with the vastly improved Raspberry Pi 3B, we were limited by the number of GPIO pins to control each of the LEDs. So we decided on a design for a 7×7 grid of LEDs so that we still had some spare GPIO pins for another possible extension of using physical buttons for human input. Unfortunately, we did not know how to do inputs with the GPIO pins in C easily and then be able to read it into standard input. So in the given time frame of the project, we decided this was beyond the scope of our extension for now.

Once the LED matrix was built, we noticed an issue with the left-most LED being dimmer, since it was being turned off earlier. This was fixed by fully lighting up a row, and then turning it off from left to right. We added a delay between each row in order to allow for the LED to be lit up fully, and increase brightness.

With the Monte Carlo tree search algorithm, we faced issues in the design of our implementation due to C not being an Object Oriented Programming language, so we had to think about how to correctly implement the data structures and the functions without using objects and methods.

3.7 Testing the Implementation

For testing the basic game logic, we manually inputted the various game states and edges cases to see if the win-checking logic was sound. This included testing winning condition cases in each direction (vertical, diagonal left, diagonal right, horizontal), drawing states, states when a winning move is made that also fills the game board.

When a human player inputs a move, we made sure to sanitise the input sensibly and be defensive in accepting inputs, asking for the value to be inputted again if it was either an invalid move or erroneous input. Due to our initial plans of using physical inputs, these checks were not completely exhaustive, but reasonably comprehensive.

In order to test the Monte Carlo tree search algorithm, a debugger was used extensively to check the state of each of the nodes at various game states, and at different points in the algorithm. We ran various game cases that could occur to check: the tree was correctly built; memory was allocated and freed as needed; nodes were initialised correctly; backpropagation of results updated the right nodes; the selection function for exploration vs. exploitation was correctly implemented; that the optimal move was chosen from what the algorithm had discovered.

Testing the LED matrix was done incrementally as the matrix was built. Then once fully built, several sub-modules and tests were created to check various aspects of the matrix's functionality. We checked if there was noticeable flickering of the LEDs using a slow motion camera, and that each LED could cycle through each colour correctly and update in sync with each other. In addition, sample patterns and simple graphics were created to test the effectiveness of the display and to see if the colours were distinct enough for our use case.

Lastly, we made sure to run Valgrind on the extension program, to ensure all memory was correctly allocated and then freed - preventing memory leaks.

4 Reflection

4.1 Group

Overall, we worked well together as a team. We found that we were able to trust each other to get our respective tasks done on time. This is because we split the tasks in a way that enabled each member to put forward their strengths, making sure we completed each part of the project on time. We communicated effectively through instant messaging as

well as clear commit messages in each of our respective branches; these together made sure that we had minimal merge conflicts in our Master branch of the git repository. Furthermore, branches were used to implement only a single aspect of the project, worked on by only one person at a time. If we were to work together again, we would like to use software such as Trello to manage tasks, as well as creating a Gantt chart to help us keep on track with our tasks.

4.2 Devesh

Having previously programmed in C/C++ I felt comfortable when dealing with any code requiring the use of pointers and dynamically allocated memory. This allowed me to help others in the group by explaining nuances (e.g. when dealing with strings), and showing them the correct syntax to use. However, this was also a weakness as sometimes I would assume everyone has been introduced to a concept that I am using. As well as this, I sometimes found that when I was giving ideas I would make jumps in my thinking without explaining how I got there. When working with a different group of people I would continue to explain things to others in the group in a way that makes it fairly straightforward to understand. Also, I would change the way in which I explain ideas to make it make cohesive for the others in the group.

4.3 Eugene

While I lacked understanding in parts of the programming language initially, working with more experienced C programmers in our group allowed me to grasp basic concepts fairly quickly. Since I have worked with this group before on other projects, we were able to bounce ideas off of each other, and help each other with any issues, or ask for help when needed. I felt my major weakness was a lack of familiarity with the language, and my tendency to fall into using conventions from other programming paradigms. On the other hand, I was able to help the group with some hardware knowledge, and designing the circuitry we used in our extension project, as well as assembling the prototype board where we displayed the output from the program. Overall, I would not hesitate to work with this group again.

4.4 Raihaan

Having collaborated with the other members of the group before hand, I knew some of the strengths and weaknesses of each member. As the group leader, this was very useful as it meant that the tasks I allocated were achievable within the deadlines we set ourselves. At first, we thoroughly designed how we would implement each part of the project, making sure that we accounted for all cases in every type of instruction. I then split up the tasks, giving everyone an opportunity to learn C and develop their programming skills during the project. From the tasks I allocated to myself, I learnt how to deal with pointers, a concept that was at first difficult for me to get used to. However I am still not 100% comfortable with certain concepts in C, such as memory allocations, as these tasks were allocated to other members of the group. My strengths and weaknesses lined up with how I expected them to be. Overall we worked well together and I definitely would enjoy working with this group again.

4.5 Prabhjot

Before working on this project, I was unfamiliar with C as a programming language and reflecting back I am now very comfortable writing clean, efficient and readable code. In the project I expressed my strengths to be with algorithmic design and theoretical aspects of the control flow of the programs. So I believe I was able to contribute significantly to the planning and organisation of the project, as well as implementing some of the more complex aspects of the project such as the Monte Carlo tree search algorithm. However, I believe I need to improve on my organisation as I began to get lost in the rapid progress of my group - despite this I was able to catch up by spending time combing through the code, commit messages and comments left by my group. Additionally, although I began the project with a slightly weaker understanding of the low level programming aspects of the project, working on the emulator and parsing cases of the assembler improved my understanding to the point where I am now much more comfortable dealing with hardware. I thoroughly enjoyed working with my group due to us all being able to communicate openly and effectively, making sure to help one another with our individual strengths.