ARMv8 AArch64 – Assembler and Emulator

Final Report

Team 54

Date: 20 June 2025

Group Members:
Richard Baca
Zayan Baig
Prasanna Sivakumar
Jaivir Sohal

1 Assembler (Part II) - Design and Implementation

1.1 High-Level Design: A Two-Pass Approach

To handle forward references to labels, we opted for the classic two-pass assembly process, simplifying the logic for branch and literal load instructions.

1.1.1 Pass 1: Symbol Resolution

In the first pass, the assembler reads the input `.s` file line-by-line to build a symbol table. We maintained a program counter, starting at address zero, which was incremented by four for each instruction or `.int` directive encountered. When a token ending in a colon (`:`) was found, we stripped the colon and added the resulting label along with the address to our symbol table.

Our data structure of choice for the symbol table was a simple linked list, where each node contained the label, its corresponding address, and a pointer to the next node. We opted for this data structure due to its simplicity, memory and insertion efficiency, and ease of implementation.

1.1.2 Pass 2: Code Generation

The second pass rereads the `.s` file, this time with a complete symbol table. For each line, we invoke the tokenizer to break it into its components. A central if-else structure then inspects the mnemonic (the first token) to call the appropriate assembly function (e.g., `assemble_dp`, `assemble_ldr`, `assemble_b`). The resulting 32-bit instruction word is then written to the output binary file using `fwrite` in little-endian format.

1.2 Key Implementation Modules

1.2.1 Tokenizer

Our initial implementation treated chunks like `[Xn, #imm]` as single tokens. This proved impractical as it made parsing the varied data transfer addressing modes too complex. We refactored the tokenizer to treat special characters (`[`, `]`, `,`, `#`, `!`) as individual tokens. This change created a more predictable and granular token stream, which significantly simplified the logic in our data transfer and data processing instruction assemblers.

For the implementation, we used a simple resizing array for memory efficiency. We first skip any whitespace, then save the start of the token, and finally continue until we hit a whitespace or special character. Then we extract the token based on the start and current pointer and add it to the array.

1.2.2 Data Processing (DP) Instructions

To handle the large number of DP mnemonics, we implemented a lookup table that maps each mnemonic to one of ten DP mnemonic "types" (e.g., `ARITH`, `LOGICAL`, `MOVX`, etc.), based on the categorization found in Table 2 of the spec. For each type, we wrote a dedicated assembly function that handles the specific encoding logic. This modular approach allowed us to easily handle mnemonic aliases, for example, the instruction `mul rd, rn, rm` is assembled by passing the appropriate tokens to the assembly function for instructions of type `madd rd, rn, rm, rzr`.

We also implemented a number of helper utility functions such as `set_bits` to make the encoding logic cleaner and more maintainable and `get_register_number` to avoid duplicate code when converting register names to their corresponding numbers.

We encountered a challenge of many tests failing which we traced back to the handling of tokens by the DP assembler. The initial implementation assumed the first, less granular version of the tokenizer. After refactoring the DP assembler to work with the correct tokenizer, we fixed around 570 DP-related test cases.

1.2.3 Data Transfer (LDR/STR) Instructions

The data transfer instructions were the most complex to implement due to their varied and flexible addressing modes. Our improved tokenizer was essential here. By examining the pattern of tokens - such as the presence of an exclamation mark for pre-indexing, or the number of tokens between the square brackets - we could reliably distinguish between the different addressing formats and call the correct assembly logic. For example, to identify the unsigned immediate offset addressing mode, we looking for the presence of a hashtag '#' within a pair of brackets. We could then confirm this was not a pre-indexed address due to the lack of an exclamation mark at the end of the address.

1.2.4 Branch Instructions

The core logic for assembling branch instructions is the calculation of the branch offset from a label. The offset is calculated using the formula: `offset = (target_address - current_instruction_address) / 4`. We ensured our implementation correctly handled this as a signed, 4-byte-aligned, word-based offset. The target address was retrieved from the symbol table built in the first pass. The C code correctly handles sign extension for the `simm19` (B.cond) and `simm26` (B) fields implicitly. When the word offset is calculated, if it's negative, the resulting `int32_t` value naturally holds its two's complement form. Subsequent bitwise ANDing with the field-specific mask (e.g., `0x03FFFFFF` for `simm26`) correctly truncates this to the required bit width while preserving the sign for negative offsets.

2 Raspberry Pi (Part III)

2.1 Assembly Program Design (led_blink.s)

Our assembly program, `led_blink.s` used to blink an LED on the Raspberry Pi works as follows:

- Load the base address of the GPIO peripheral registers into a register.
- Configure the target GPIO pin (e.g., GPIO 21) as an output by writing the bit pattern `001` to the appropriate bits in the GPFSEL2 register.
- Enter an infinite loop labelled `main_loop:`.
- Turn the LED on by writing to the GPSET0 register.
- Implement a delay using a nested loop that decrements a register a large number of times.
- Turn the LED off by writing to the GPCLR0 register.

- Implement another delay.
- Use an unconditional branch (`b main_loop`) to repeat the cycle indefinitely.

2.2 Challenges

Our emulator and assembler both passed all tests, but we faced challenges when deploying the assembly program to the Raspberry Pi. The main issue was that connecting the SD card containing the assembled `kernel8.img` file did not flash the LED as expected. At first, we thought the issue was with the hardware, but after connecting the LED to the steady 5V pin, we confirmed that the LED and the GPIO pin were functioning correctly. In the end, we discovered that we were connecting the SD card to the Raspberry Pi incorrectly; we were plugging in the USB card reader into a USB port instead of the SD card itself into the SD card slot. After correcting this, the LED started blinking as expected.

3 The Extension

3.1 Description and Goal

Our extension, the "Guide Glove," aims to assist visually impaired individuals in navigating complex or confined spaces where a traditional cane might be cumbersome or less effective. By integrating an ultrasonic sensor, a buzzer, and a vibration motor with a Raspberry Pi, the glove provides real-time, multi-modal feedback about the proximity of nearby objects. The ultrasonic sensor measures the distance to surfaces in front of the hand, and this data is translated into varying intensities of haptic (vibration) and auditory (buzzer tone/frequency) feedback, which intensify as an object gets closer, thereby enhancing spatial awareness and user safety.

3.2 Design and Implementation

The "Guide Glove" extension was realized as a C application on the Raspberry Pi, interfacing with an HC-SR04 ultrasonic sensor, a piezo buzzer, and a vibration motor, initially prototyped using a breadboard and necessary resistors. The program utilized the `pigpio` library for GPIO control.

Core logic involved continuously reading a distance value (in cm, from a file populated by the sensor process) and translating this into synchronized feedback. Auditory feedback featured a dynamically pitched buzzer and a rhythmic beeping where the silent interval decreased with proximity. Haptic feedback mirrored this rhythm with the vibration motor. A critical "danger zone" (5cm) triggered constant, high-intensity sound and vibration. If no object was in range, outputs were silenced.

3.3 Example of Use

The Guide Glove serves visually impaired users in two primary scenarios. Firstly, for general navigation in unfamiliar environments, such as a new restaurant, the user can orient the glove (e.g., sensor facing forward from the hand or near the head) to receive haptic and auditory warnings about approaching obstacles like walls or low-hanging objects, preventing collisions. Secondly, for close-proximity tasks, the user can employ a sweeping motion with the glove to locate specific items

within a confined space, for instance, finding a mug in a cupboard without accidentally knocking over other objects, guided by the intensifying feedback.

3.4 Challenges and Learning

The primary challenge was iterative hardware refinement. Initial breadboard connections proved unreliable, prompting a move to soldered components on perfboard. This transition introduced new issues like wire breakages and misconnections, which were time-consuming to debug. Our key learning was the necessity of rigorous, incremental testing after each hardware modification to quickly isolate faults, which significantly improved our debugging efficiency and the final prototype's robustness.

4 Raspberry Pi and Extension Testing Strategy

For our C-based "Guide Glove" extension, testing was performed directly on the Pi. After compiling with `gcc -lpigpio -lrt -lpthread`, we ran the executable and systematically presented objects at varying distances to the ultrasonic sensor. We verified: (1) accurate distance reporting to the terminal, (2) corresponding activation of the vibration motor, and (3) correct dynamic pitch and rhythm changes from the buzzer, including the distinct "danger zone" feedback. Iterative testing after hardware modifications (soldering, pin connections) was crucial to ensure component reliability.

5 Group Reflections

We used Slack as our main communication platform which provided useful productivity tools such as file sharing, task management, and communication channels. Our workload split allowed us to work independently and without much overlap after implementing the core files, which sped up the workflow immensely. At the start of the project we didn't have frequent in person meetings, however when we started using hardware we naturally had to meet up more to work on the project. We found that meeting more frequently reduced the number of conflicts and code duplications and improved overall workflow - so in the future we will try to meet as often as possible.

At the start we all agreed on the git conventions we would use for naming commits and branches etc. this proved to be very important in keeping our workflow clean and organised and knowing what each branch and commit does - preventing confusion between teammates.

6 Individual Reflections

6.1 Reflection from Richard

I feel like I made solid contribution to our project, especially in fixing most of the failing test cases and implementing the instruction decoding and data processing assembly. I also spent a fair bit of time reviewing merge requests, which proved tricky when the branches were large but I tried to stay as thorough as possible. Overall, I think my strengths lied in problem-solving and consistency. If I were to do anything differently, it would be to allocate more time to complex code reviews to

avoid potential oversights. I've learned a lot from this project and will definitely carry that forward into future group work.

6.2 Reflection from Zayan

My main contributions to the group were through my sections of the code. For both our emulator (Part I) and assembler (Part II), I implemented the execution and parsing of the data processing instructions. I also wrote the high level code for our assembler to link everyone's work individual work together so that our assembler could actually run. It was noted that my biggest strength was being thorough and getting "it" right first time, meaning that my work needed minimal changes during code reviews and testing. One thing that I have learned is that I need to ensure that I keep closer contact with team members when writing code - to ensure that we don't accidentally overlap and write the same code. This happened once in Part I, and we learned from our mistake and it didn't occur again.

6.3 Reflection from Prasanna

My primary contributions included implementing the DP executor and register shifting for the emulator (Part 1), the branching logic for the assembler (Part 2), and contributing to the hardware implementation and testing for our Part 4 extension. I consistently aimed to produce well-commented, functional code, particularly for the assembler's branching. Initially, I recognized a weakness in my proactivity regarding group communication, testing, and peer reviews. However, I consciously worked to improve this, becoming more engaged during the extension phase. This project reinforced the value of rigorous Git practices and highlighted the importance of consistent team communication to prevent conflicts - a key area I will focus on more in future group work.

6.4 Reflection from Jai

As team leader, my main contributions were planning the project's structure and developing the core software architecture. I built the foundational components for both the assembler and emulator, including the tokenizer and the symbol table that the team depended on. For our extension, I then wrote the C control system for the sensor glove, designing the non-linear algorithm for its dynamic pitch and rhythm feedback. I believe my strength was in creating a stable architectural foundation that allowed the team to work in parallel effectively. The greatest challenge was the shift from deterministic software emulation to debugging the physical hardware, which taught me the value of methodical, incremental testing to isolate problems.