Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

An ARM Simulator Application for Teaching Assembly Language

Finn Voorhees

April 2021

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
BAI (Computer Engineering)

Declaration

I hereby declare that this Final Year Project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Assembly language is used to teach computing concepts to students at a low level. This project aimed to create a simulator application analogous to an interactive textbook that can be used to engage and teach ARM assembly language to students. Taking into account current simulators and the existing research on motivation and engagement techniques, the application was designed to teach fundamental concepts using a visual sandbox that written programs interact with and control. The application was implemented as a self-contained iPad application, requiring little effort to download and run for free from the Apple App Store. This app enables students to learn fundamental computing concepts from included lessons and allows course instructors to develop custom lessons for use in an assembly language course.

Acknowledgements

Thanks to the supervisor of this project, Prof. Glenn Strong, who provided guidance and encouragement throughout the year.

Contents

List of Figures

List of Tables

1 Introduction

Assembly languages are low-level programming languages designed to interact directly with computer hardware. They are commonly taught in college-level computer science courses as a way to introduce students to processor and memory structure and how a computer works on a low level. Assembly languages are rarely used for day-to-day programming, but they are still necessary for tasks like OS development and embedded devices with limited price and power constraints.

ARMv7 specifically has a reduced instruction set and simple design, meaning it is one of the easiest introductions to assembly languages. Typical introductory ARM programs consist of register arithmetic, memory loading and storing, branches, loops, and boolean algebra. While these instructions are simple, it can be challenging for students to debug programs. If classrooms have access to physical ARM development boards, students can run programs that can control a few LEDs or switches, however it is not possible to understand what is happening at any given step in a program. Students can instead use ARM simulators, which allow stepping through instructions and viewing register and memory values, however this comes with the trade-off of not having physical feedback and interaction with the programs. The ACM Journal of Educational Resources in Computing (1) published two issues focused on general computer architecture simulators for education, and identified that simulators can highlight general concepts regardless of proprietary details, are available without needing laboratories or special equipment, are available for a variety of machines, and can facilitate both active and asynchronous learning.

The goal of this project is to create an ARM simulator that can be easily set up and installed and provides self-contained educational content and visual feedback while keeping students engaged.

This project proposes a tool that can be used to teach students ARM assembly language using an iPad application. The app contains lessons of increasing difficulty designed to walk students through basic ARM syntax and basic computing concepts. These lessons contain instructions, hints, and automatic grading designed to be possible to use without an instructor, however instructors can also create their own

lessons for students to complete. In order to provide more tangible feedback often given by physical microprocessor boards, this application contains a graphical sandbox that students interact with using assembly language, providing a way to make lessons interactive and visualize program execution.

This report details the process of identifying areas of improvement in tools for learning assembly language, designing an iPad application based on these areas, and implementing the application. Further to this, limitations in the final application are identified and possible future work is discussed.

This report is structured as below:

- *Chapter 2*: This chapter introduces assembly language and the use of ARM in an educational context. A set of concepts that should be focused on in an educational tool is identified, and the current state-of-the-art for educational simulators is evaluated. This chapter also presents research on what design choices and techniques can be used to improve learning in students.
- *Chapter 3*: This chapter outlines the design of the iPad application and what decisions were made to best incorporate the findings described in Chapter 2.
- *Chapter 4*: This chapter describes the implementation details of the iPad application and the challenges faced during implementation.
- *Chapter 5*: This chapter summarizes the results of the design process and how well the original goal was achieved. Future work and limitations are discussed and conclusions are drawn about the project.

2 Background

2.1 Assembly Language

Assembly languages are programming languages that provide a thin layer directly above writing machine code and can be used to program specific computer processors. Most higher-level languages such as C can be used to write programs that will compile to assembly language, and as such are more commonly used, however assembly languages can be used to teach low-level computing concepts and help students understand how a computer works. Assembly languages consist of a set of instructions that closely map to a processor's machine code instructions, meaning they are specific to a processor's architecture. Many modern processors have similar structures and instructions, however, so studying an assembly language can be useful for teaching general low-level concepts. Learning an assembly language can provide an understanding of how a computers works, can help to write efficient and fast code in higher-level languages, and can demonstrate concepts not possible in a higher-level language (2).

2.2 ARM Usage in Education

The ARM assembly language is a language designed for development on ARM processors. According to the Arm Group, "Our energy-efficient processor designs and software platforms have enabled advanced computing in more than 180 billion chips and our technologies securely power products from the sensor to the smartphone and the supercomputer" (3). ARM processors are in widespread use throughout the world and provide a useful learning environment due to the reduced instruction set.

While processors such as Intel are *Complex Instruction Set Computing* processors, ARM is a *Reduced Instruction Set Computing* (RISC) processor meaning it has a greatly reduced number of instructions and uses a simplified memory access model. Clements (4) discusses transitioning to using ARM to teach computer architecture because "ARM covers the requirements of existing curricula, is easy to learn, and has

an elegant and sophisticated architecture. Moreover, it is widely found in real systems". The elegant and sophisticated architecture allows assembly language instructions to be introduced one at a time, giving students an incremental understanding of each one. Clements also found that students are more likely to be motivated to learn assembly language if they are studying a processor that is in their mobile phones and other devices they commonly use.

While the ARMv8 and recently announced ARMv9 architectures are in use throughout the electronics industry, the ARMv7 architecture can provide a simpler learning environment. The ARMv7 architecture is limited to 32-bit instructions and does not contain more complicated features such as cryptography and machine learning that are beyond the scope of an educational introduction to computer architecture (5). ARMv7 contains instructions for operating on 32-bit registers and loading/storing to memory that provide an easy-to-understand way to teach computing concepts described below.

2.3 Computing Curriculum

The motivation for this project is to help students learn assembly concepts as well as fundamental computer programming concepts. Assembly language is a recommended topic of study under different guidelines and is commonly taught in computer architecture college courses. Assembly language can be used to teach fundamental computing concepts such as how a computer handles memory, binary arithmetic, number systems, and interrupt processing.

The *ACM Curriculum Guidelines* (6) describe areas of study for undergraduate degree programs in computing. These guidelines are in widespread use in colleges and are in agreement with the *Computing Programmes of Study* in the National Curriculum in England (7). The guidelines recommend spending 60 out of 420 core hours on the area of Computer Architecture and Organization, and 50 out of 420 core hours on the area of Digital Design. These two areas consist of 22 knowledge units total, 9 of which are relevant and can be taught using an assembly language tool:

- Instruction set architecture
- Measuring performance
- Computer arithmetic
- Processor organisation
- Memory system organisation and architectures

- Input/Output interfacing and communication
- Peripheral subsystems
- Number systems and data encoding
- Boolean algebra applications

The *ARM Assembly Language: Fundamentals and Techniques* (8) textbook describes a set of core ARM fundamentals, the basics of which are listed below.

- Instruction sets
- Loads, Stores, and Addressing
- Constants and Literal Pools
- Integer Logic and Arithmetic
- Data-Processing Instructions
- Branches and Loops
- Subroutines and Stacks
- Interrupts
- Memory-Mapped Peripherals

The intersection of these sets of concepts can be summarised to find a set of learning objectives useful for creating an educational tool for teaching assembly and computing concepts as follows.

2.3.1 Instruction Set Architecture

An introduction to the ARM instruction set is necessary for learning ARM assembly language. The concept of registers must be introduced as the ARM RISC only allows arithmetic and data processing instructions to operate on registers. The instruction `MOV r0, r1`, for example, operates on the `r0` and `r1` registers and copies the value stored in `r1` to `r0`. The only way to interact with memory is to use the load and store instructions that load or store a value in memory to or from a register.

2.3.2 Arithmetic and Data Processing

The concept of arithmetic and data processing is important in the context of assembly language, as a large and most commonly used subset of the instruction set deals with register arithmetic and data processing. This concept can also act as an introduction to number systems, as integer literals can be initialised as decimal, hexadecimal, or

binary numbers. This general objective encompasses constants and literals, number systems and data encoding, integer logic and arithmetic, and ARM data processing instructions from the ACM curriculum guidelines and ARM fundamentals.

2.3.3 Boolean Logic and Arithmetic

Boolean logic and arithmetic closely relate to the arithmetic and data processing objective but focus more on boolean algebra. Boolean algebra can serve as a foundation for both *Signals and Systems* courses and low-level computer processing, and consists of instructions such as `AND`, `ORR`, and `EOR`.

2.3.4 Memory Management

The concept of memory management is best taught using assembly language as memory is directly loaded and stored by assembly language instructions and must be entirely managed by the programmer. Teaching memory management concepts allows students to understand what higher-level programming languages are abstracting away from the programmer and give a look into how memory is stored in a computer (2).

2.3.5 Branches and Loops

The concept of branches and loops can be used to teach students about conditional execution and the performance impacts of looping code. Conditional logic is a fundamental computing concept and can be taught using the ability of ARMv7 to conditionally execute any instruction based on condition flag states.

2.3.6 Subroutines

An understanding of subroutines is important due to their direct relationship to higher-level function calls. Functions are used in almost all higher-level languages to enable code reuse, and assembly language subroutines help introduce this concept. The subroutines learning objective can also be used to ensure students know how to save register states before a subroutine call and restore them after. It is also important to understand how parameters are passed to and returned from a subroutine call.

2.3.7 Peripherals

The concept of peripherals is important as it is how programs interact with external devices such as keyboards and displays. Two main peripheral access methods can be

used and introduced to students. Memory-mapped peripherals require changing values at a specific memory address to affect a peripheral in some way, while interrupt-based methods can be used to pause program execution and perform an action, which is commonly used to interact with an operating system.

2.4 Development Board vs. Simulator

Given the low level and hardware specific nature of assembly language, it can be difficult to set up an environment to write, test, debug, and run programs. The two primary ways to learn ARM assembly language programs are using an ARM development board or an ARM simulator.

A common way for students to execute ARM assembly language programs is to load and run them on a development board such as the ARM7TDMI. These development boards often come with an IDE or software for uploading assembly language programs to the board, however the process of setting up the IDE and figuring out how to connect it to a board can be a challenging task for a novice programmer (9). Programs such as Keil μVision provide a commercial IDE for writing and debugging programs and uploading them to development boards. The process of configuring the IDE for a specific type of board is different for each type of board, and finding a guide for creating a project in μVision to write basic ARM assembly programs can be difficult for both students and instructors (10).

Development boards cannot be used to inspect register or memory values without an external debugger, and will oftentimes run programs at a high speed making it difficult to distinguish and understand individual instructions. Physical development boards offer a similar experience to a real-world application of programming a microprocessor using assembly, however they are not as appealing for beginner programmers.

Clements (4) found that "It's virtually impossible to teach the architecture of a machine effectively without a working example that students can play with – preferably at home on their PCs". One way students can learn at home on their PCs is by using an ARM simulator. Simulators emulate a generic ARM processor and enable students to load an ARM program and step through each instruction, inspecting the memory and register values after each instruction. This allows students to understand where their program is branching to, which instructions are being conditionally executed, and the overall execution flow of the program. Simulators also allow each student to work on their own emulated processor, freeing up teachers to help students who are having difficulties (4).

ARM simulators can also provide simulated peripherals without needing additional costly components. Simulators can provide an all-in-one software package that does not rely on a specific development board or hardware setup. Students prefer using software that simulates microprocessors with simulated peripherals over working with development boards, and as such have been more successful in learning assembly language (11).

2.5 Related Work

2.5.1 Educational Technologies

Educational technologies have become an increasingly prevalent trend and can be used to assist in learning. Ardito et al. (12) identified the benefit of e-learning tools in scenarios such as distanced learning, where interacting with a teacher can be more difficult. They also discuss the requirements of a usable e-learning tool and find that tools must be interactive, provide useful feedback, have specific goals, and motivate or communicate a sense of challenge. A graphical user interface can also improve the time, correctness, and efficiency of students when learning a new concept (13).

2.5.2 ARM Simulators

ARM simulators have been developed for both academic use and industry-oriented assembly development. In the following sections the effectiveness of each simulator in a teaching environment was evaluated and the unique features and peripheral support of each were investigated.

QEMU

QEMU is a command line tool that can be used to emulate and visualise a variety of different architectures including ARM processors. QEMU can emulate ARM assembly language programs, however it does not have support for debugging and can be difficult for beginners to set up. Malhotra et al. (14) found that using QEMU in a computer architecture course resulted in over 90% of students having issues. It also lacks peripherals and valuable feedback on running programs. QEMU does not contain any tools designed for goal-based learning or educational technologies.

Keil μVision

Keil μVision is an advanced IDE with features to write, compile, and debug embedded programs. It allows simulation of ARM programs and has support for stepping through code, but it is oriented towards advanced users and provides

features far beyond the scope of an educational tool. The software does not provide any simulated peripherals or ways to teach learning objectives and can be difficult to use to create and debug basic ARM programs (10). Keil μVision is only available to run on Windows computers and is focused more on targeting specific microcontroller boards than general ARM assembly.

ArmSIM#

ArmSim# is a desktop application available for Windows and Linux computers developed by members of the Department of Computer Science at the University of Victoria. ARMSim# was developed specifically for use in teaching and research and offers standard features, such as stepping through code, as well as features useful to students, such as simulated peripherals. Plugins can be developed to create peripherals that simulate real-world hardware, giving students visual feedback from their executed code (9). In a learning environment, ARMSim# relies on instructors to assign coding challenges and assignments for students to complete, and there is no automatic grading system or instruction.

VisUAL

VisUAL is a tool developed with a focus on making ARM Assembly language easier to learn. It is an emulator that supports a subset of the ARM instruction set and allows visualisation of aspects such as registers, memory, stack, and branches. It was designed for use as a teaching tool in the Introduction to Computer Architecture course taught at the Department of Electrical and Electronic Engineering of Imperial College London. While this tool does only support a subset of the ARM instructions, it includes most of the instructions typically taught in an ARM assembly language course. VisUAL offers many advanced visualisation tools but does not have any simulated peripherals and relies on course instructors to assign tasks to students.

2.5.3 Summary

There are a number of different ARM simulators available with a variety of features, however each one has been designed for a specific purpose and lacks some features necessary for teaching assembly. Industry-oriented simulators prioritise closely emulating real-world hardware and contain many complicated features unsuitable for an introductory course. Because of this, it can be difficult to set up a basic project. Education-oriented simulators more closely match the goals of this project, however they have been designed to be used in conjunction with a college course and rely on instructors providing and grading assignments for students. No simulators contain

educational content that teach students about various computing concepts, and most do not support any simulated peripherals and rely on debugging by inspecting register and memory values.

2.6 Engagement and Learning Success

Motivation and engagement are important to consider when designing an educational tool. While traditional college courses assign lessons to students and have instructors grade them, a self-contained simulator can contain multiple integrated lessons and an automatic grading method for correcting them. Engaging students with the content while still providing educational value can be done through gamification, visual feedback, and instant grading.

Gamification is the use of game theory and concepts in an environment that would typically not contain them. Gamification has long been used as a tool for assisting in teaching computer programming, with programs like *Robot Odyssey*, *Core War*, *Human Resource Machine*, and many more offering a video game based around simple programming or digital logic. Gamification can be a useful tool to motivate and engage students while learning and has been shown to have a positive effect on education (15). Lee and Ko (16) have shown how the presentation of game elements can affect students learning. They found that using vertebrate elements in an educational game, such as cats, resulted in students being more likely to continue playing an educational game on more difficult levels. They also found that purposeful goals play a significant role in the engagement of students in self-guided games.

Visual feedback is an important part of an ARM simulator and can provide students with a more in-depth look at what is going on in their programs. Studies done by Lee and Ko (17) found that the method in which feedback is portrayed has a significant impact on the success and motivation of students when programming. Visual feedback in a simulator application can consist of a simple view into the register and memory values of the simulated processor or can include more complicated visual displays. Simulated peripherals can mimic physical hardware such as LED's and buttons and can provide motivation to learn beyond simply writing device drivers (18). Visual feedback can also consist of gamified elements such as a visual sandbox with game characters that update based on written programs.

Automatic and instantaneous grading not only allows students to immediately understand if their program works, but has also been found to increase engagement and cause students to interact with an e-learning tool for a longer period of time (19).

An automatic grading system allows students to test their programs without the need of a dedicated instructor and frees up instructors to help students who may need more assistance. According to Swigger and Wallace (20), "programming courses that require the student to write programs and then run these programs may convey information at a level that will allow prolonged retention of the material". Instant grading can consist of predefined lessons for students to complete and a method for a simulator to automatically check the validity of the programs that students write. Shan (21) has suggested adopting a new approach to teaching assembly language that not only requires students to study fundamental techniques but also requires them to apply these techniques to practice projects. Simulators can also display helpful error messages when programs fail to assemble, preventing students from becoming discouraged when there are issues in their programs.

2.7 Summary

This chapter has given an understanding of assembly language and its uses in education and has explained the role of ARM assembly language in teaching fundamental computing concepts. A set of learning objectives for the project have been listed based on ACM guidelines and ARM fundamentals, and an explanation of the importance of each of these learning objectives has been provided.

An important distinction has been made between teaching ARM assembly using a physical development board and a simulated processor, and research has been conducted showing the benefits of a simulator over a physical board.

An analysis of related work was conducted on both general educational technologies and specific ARM simulators developed in the past, and it was determined that there is a gap in the research conducted to date for a self-contained educational simulator that contains specific educational content and grading mechanisms. Expanding on this, the impact of gamification, visual feedback, and grading on the motivation and engagement of students were investigated to determine ways an educational tool can keep students engaged.

This research has provided a good foundation for a more specific project goal and application concept outlined in the following chapter.

3 Design

3.1 Overview

Based on the original project goal and the research conducted in Chapter 2, the objective of the project was expanded upon and refined. It was decided that a feasible solution to the problems discussed in Chapter 2 was to create a self-contained iPad application that included a development environment for writing ARM assembly language that can interact with a visual gamified sandbox. The app contains lessons of increasing complexity that serve to teach computing concepts. The lessons were designed based on the curriculum discussed in Chapter 2.3. A simulated ARM processor is used to run code written by students based on the findings in Chapter 2.4, and engagement techniques discussed in Chapter 2.6 are used to keep students involved with the content.

The iPad was chosen as the target device for this application due to the ease of installation, large screen size, and support for educational content. Developing an app for iPad allows the app to be uploaded to the Apple app store, enabling anyone to easily install the app to their device without the need for complicated installations or required dependencies. The screen size and split-view mode allow students to have an IDE and sandbox side by side and optionally open ARM documentation to the side for reference. The iPad also supports *Apple Classroom* (22), which allows instructors to guide students through educational content and monitor progress. Previous learning environments such as *Swift Playgrounds* (23) have also been developed for iPads and successfully used in educational settings. Developing an app for iPad also reduces the challenges discussed in Chapter 2.4 related to difficulties setting up an environment to run assembly programs. All necessary dependencies can be bundled with the application, meaning students can immediately open the app and start writing and running ARM programs.

The application was designed analogous to a textbook, where a list of lessons is provided on different ARM topics. Each lesson is similar to a chapter in a textbook and introduces a new concept that builds upon previously learned concepts. Instead

of simply describing a concept, however, the application requires students to write a simple ARM program demonstrating the concepts learned in the lesson. The lessons teach a new ARM instruction or concept and then give instructions on what the program should do. A visual sandbox was designed instead of relying on inspecting register values. While many other simulators allow students to step through programs to show register values being updated, this application shows a character moving around the scene and interacting with number blocks based on the written ARM program. Programs must take number blocks from the input, perform some action on them, and place them in the output. This gives visual feedback and an easier way to understand the execution flow of a program. If there are errors in a student's program, they will be immediately evident when the sandbox does not act the way they expect. The application also allows stepping through code so that students can understand every instruction in a program. This solves the issues described in Chapter 2.4 related to understanding the execution flow of an ARM program.

The interface of the application was designed to show an ARM development environment on the left side and a sandbox area that interacted with programs on the right. The left side shows instructions for lessons and allows students to enter code to complete the lesson. Programs written by students interact with the sandbox through various ARM instructions.

3.2 Lessons

Five lessons were designed based on the learning objectives outlined in Chapter 2.3. As previously mentioned, lessons were designed to incorporate more advanced concepts as they progressed and continually build upon previously taught concepts. Each lesson requires students to design a fully functional ARM program and verifies the program by checking the output of the sandbox against the input. This ensures students have error-free programs and understand where any programs are malfunctioning.

The goal of each lesson is centralised around a core theme of interacting with number blocks. Lessons require programs to control a character in the sandbox and instruct him to pick up numbers from the input, place them in memory-mapped slots, edit them using assembly language, and place them in the output. The human-like character was included in the design based on the impact of vertebrate objects discussed in Chapter 2.6. The use of numbers in the lessons was decided based on the ease of developing programs that interact with numbers. Numbers can be loaded from memory and operated on using various instructions, and as such operating on

numbers was a logical way to teach students assembly language.

3.2.1 Introduction Lesson

The introductory lesson presents a necessary tutorial on how ARM programs interact with the sandbox. This lesson does not teach ARM fundamentals, but it is important to introduce students to how their programs will interact with the sandbox to complete the following lessons. This lesson introduces the software interrupts necessary to interact with number blocks in the sandbox and the objective is to pick up number blocks from the input and place them in the output.

3.2.2 Arithmetic Lesson

The arithmetic lesson acts as an introduction to the data processing instructions subset and instructs students to take number blocks from the input, add `10` to the current value, and place them in the output. While the main focus of this lesson is to introduce arithmetic instructions and teach the learning objective described in Chapter 2.3.2, it also acts as an introduction to the load and store instructions, as it is necessary to load numbers from memory to interact with the sandbox. This is an important part of the Memory Management learning objective outlined in Chapter 2.3.4.

3.2.3 Boolean Logic Lesson

The boolean logic lesson is a simple expansion of the arithmetic lesson, as it also uses data processing instructions. This lesson can act as a way to learn the boolean algebra learning objective and consists of ANDing numbers from the input with the value `100`. This lesson teaches the learning objective described in Chapter 2.3.3.

3.2.4 Loops Lesson

The loops lesson is a further expansion of the arithmetic lesson and requires students to take numbers from the input and add `10` to them in a loop. The lesson is complete when this has been completed five times. This lesson requires the use of a basic loop using branch instructions, teaching commonly used ARM instructions.

3.2.5 Branches Lesson

The branches lesson introduces the concept of comparisons and conditional branches in ARM and requires adding a different number to the output depending on the value of the input. This gives an introduction to the CPSR register, condition flags,

and branching based on these values. This lesson, and the preceding loops lesson, acts as an evaluation of concepts described in Chapter 2.3.5.

3.2.6 Custom Lessons

In addition to the built-in lessons, lessons can be easily added by instructors to expand upon the included learning objectives. In a classroom setting, instructors can design further lessons based on the curriculum of the course, and students can create lessons for others to complete. Lessons consist of a title, image, instructions, and a function to verify the output of a program with the input. This can be used to construct a variety of lessons ranging from simple to complex assembly language programs. These custom lessons can be created directly from the app interface, meaning no recompilation of code is needed.

3.3 Instruction Set

When designing the application, it was necessary to decide which ARM instructions to support. Looking at the ARM instruction set (5) it was decided that the data processing, branch, multiply, data transfer, and software interrupt instructions should be implemented. This covers all of the ARM instructions used in introductory courses and excludes instructions irrelevant to an educational simulator. These instructions were necessary to cover the concepts and learning objectives proposed in Chapter 2.3. While the included lessons did not use more advanced instructions such as multiply and data swap, these instructions were included in the simulator so that more advanced lessons could be designed by instructors if needed. Software interrupt instructions were chosen to act as a way to bridge program execution with the visual sandbox. The sandbox can be thought of as a simple operating system, where different software interrupts perform different actions in the sandbox. This gives immediate visual feedback while executing a program. Loading and storing memory, for example, is immediately evident in the sandbox when the number inscribed on the number blocks updates to the new value from memory.

The instructions implemented by the assembler and emulator were taken from the 14 major instruction categories described in the *ARM7TDMI-S Technical Reference Manual* (5). Eight of these categories were determined to be necessary for the application based on their usage in common ARM programs and effectiveness in teaching the learning objectives outlined in Chapter 2. These instruction categories are described below:

3.3.1 Branch and Exchange (BX)

The *branch and exchange* instruction performs a conditional branch by replacing the contents of the program counter with the value stored in a register. It is commonly used after a subroutine call to return to the previous program execution location. This instruction can be used in combination with the branch with link instruction to teach students about subroutines in assembly language. Supporting the branch and exchange instruction is required to teach the learning objective described in Chapter 2.3.6.

3.3.2 Branch and Branch with Link (B, BL)

The *branch* and *branch with link* instructions perform a conditional branch by adding a specified offset to the program counter. This allows moving program execution to a different point in a program. The branch with link instruction will store the old program counter into the link register before branching. This allows the link register to be used with a branch and exchange instruction to return to the previous program counter after a subroutine call. The branch and branch with link instructions can be used to teach students about conditional branching, loops, and subroutine.

3.3.3 Data Processing

The data processing instruction category contains 16 different instructions, each focused on performing an arithmetic or logical operation on one or two operands. The first operand is always a register, and the second operand can be either a register or an immediate value. Unlike the branch instructions, which control the sequence of execution on the processor, the data processing instructions operate on data to produce a new value. These instructions can be used to teach computer arithmetic and boolean algebra to students.

3.3.4 Multiply and Multiply-Accumulate (MUL, MLA)

The *multiply* and *multiply-accumulate* instructions are also arithmetic instructions that will perform multiplication on the given operands. They can be used to teach more advanced computer arithmetic.

3.3.5 Multiply and Multiply-Accumulate Long (MULL, MLAL)

The *multiply long* and *multiply-accumulate long* instructions perform multiplication on two 32-bit operands but produce a 64-bit result. These instructions can be used to teach students about number size and overflow in computers.

3.3.6 Single Data Transfer (LDR, STR)

The *single data transfer* instructions can be used to load and store a single byte or word into memory. The instruction will load data from an address specified by the value of a register into another register. This instruction will fetch or store data from the MMU. The load and store instructions can be used to teach students about low-level memory management and the stack in a computer.

3.3.7 Single Data Swap (SWP)

The *single data swap* instruction swaps the value of a register and the value in memory at a specified location. It does this by performing a memory read and write where the processor cannot be interrupted between these two operations. This is commonly used to implement software semaphores, which are beyond the scope of an introductory assembly language course but can be implemented to enable more advanced lessons in the future.

3.3.8 Software Interrupt (SWI)

The *software interrupt* instruction is passed a comment field, which has no effect on the processor, but can be used to specify which software interrupt to call. The software interrupt instruction is commonly used to invoke a system call, enter supervisor mode, or communicate with a kernel. These are not relevant to a simple processor emulator, however software interrupt instructions can be used as a means of communicating with an external sandbox easily.

3.3.9 Unimplemented Categories

Coprocessor instructions were not included as the simulator did not implement any coprocessors and did not include features like a cache where coprocessor instructions would be useful. These instructions are also beyond the scope of introductory ARM courses. Status register transfer instructions are used to change processor mode and update PSR's, which are not included in the simulation and did not need to be implemented. The 16-bit Thumb subset of ARM was not implemented, as code size was not a concern for the simulation. Lessons were designed to result in small and simple programs that would teach individual concepts.

3.4 Editor

The editor component of the application is responsible for providing an environment for students to write and run ARM assembly language programs. It consists of an ARM IDE with basic features for writing assembly language programs. Above the IDE is text displaying instructions and hints for the currently selected lesson. It also contains buttons to step through code and reset the simulator.

The ARM IDE was designed to be an easy way for students to write ARM programs without needing any additional setup. The IDE displays a text input where programs can be written and does not contain any distracting project management features or code completion. These features may distract from the main objective of providing a straightforward way for students to write and test ARM programs.

In order to teach and explain each lesson to students, instructional content and hints must be displayed in the interface. The editor interface design was inspired by *Swift Playgrounds* and contains the lesson title followed by instructions and hints for the lesson above the editor. These instructions were designed to give necessary information to complete the current lesson and instruct students on how to complete a task.

The step and reset buttons give students control over the simulator and allow stepping through each instruction. The step button executes one instruction, and the reset button will reset the emulator state, memory, and sandbox state.

In order to provide students with meaningful feedback when their programs fail to assemble, error messages are shown as alerts. Error alerts contain a message describing what the assembler has identified is wrong with the written code and where it was unable to assemble. Meaningful error messages aid in the debugging process and prevent students from becoming discouraged and giving up.

3.5 Sandbox

To help alleviate difficulties students face when trying to debug and understand the execution of their programs, the sandbox area allows programs to directly manipulate a sandbox visually. Instead of relying on students to look at register values or figure out what is happening on a development board, the application contains a 2D scene for each lesson, containing a character and number blocks to manipulate with assembly language.

The overall design of each lesson required students to pick up numbers from the input, perform an action on them, and place them in the output. This was influenced

by the game *Human Resource Machine* which requires players to perform actions on numbers using block-based coding. This provides a unique gamified approach to programming while still maintaining the core concept of numerical operations. This gamified approach and the use of vertebrate characters were incorporated in the sandbox based on the findings discussed in Chapter 2.6.

Automatic grading is another important component of the sandbox and can increase student's motivation and engagement as discussed in Chapter 2.6. Automatic grading was included in the sandbox design by displaying visual feedback when running programs. When a number block is placed in the output, the sandbox will check whether or not the output matches the expected output for the current lesson, and if it does it will display a checkmark. If the output does not match, the sandbox will display an "X" mark, signaling to students that they have not completed the lesson. When a lesson is successfully completed, the lesson will be closed and students may move on to the next lesson.

While the 2D sandbox offers valuable visual feedback, a view into register values during program execution can also be useful for testing and debugging. Because of this, a register view was included in the design of the sandbox. The register view displays a list of each simulated register and can be expanded to show more at once. The displayed register values will update as each instruction is executed, allowing students to inspect each value. This is especially useful for viewing the program counter register which shows what instruction is being executed. This facilitates debugging loops, branches, and subroutines where the program counter may jump around.

3.6 Editor Sandbox Interaction

The sandbox was designed to act as both a memory-mapped and a software interrupt-based peripheral. The number blocks and memory slot described in Chapter 3.2 are contained in the sandbox and the memory slot acts as a bridge between the editor and sandbox. When number blocks are placed in the memory slot, a program can interact with the number block value using memory load and store instructions in the editor. This enables seamless interactions between programs and the sandbox and simulates a memory-mapped peripheral, where changing a value at a certain memory location will have an effect on the peripheral.

Software interrupt triggered actions are the other method of interacting with the sandbox. Software interrupt instructions in ARM are typically used to call operating system routines on whatever operating system a program is running on. In this

application, the sandbox area is analogous to a basic operating system that provides specific software interrupt routines. Programs written in the editor can take advantage of software interrupt instructions to perform certain actions in the sandbox. These actions consist of moving the character around the sandbox and moving the number blocks. Instead of relying on students to move the character directly step by step, these software interrupts abstract the movement to simple actions. This allows students to quickly understand how to interact with the sandbox, and is introduced in the first lesson.

4 Implementation

The iPad application was implemented as four individual components that were developed separately. A simple ARM IDE is necessary for students to write and run programs, an assembler assembles written programs into machine code, an emulator runs the machine code with simulated registers and memory, and a sandbox gives visual feedback to running programs. Developing these sections as self-contained components allowed tests to be written for each one, verifying that each worked as intended before connecting them. It also enables further work to use these components in other environments, such as in a web app version of the application. Developing each component separately also allowed each to be written in a programming language that best suited the specific needs of the component, such as using Swift for the assembler due to its easy string manipulation. A diagram of each component of the application can be seen in Figure 4.1. The editor and sandbox are user-facing, whereas the assembler and emulator act behind the scenes to tie everything together.

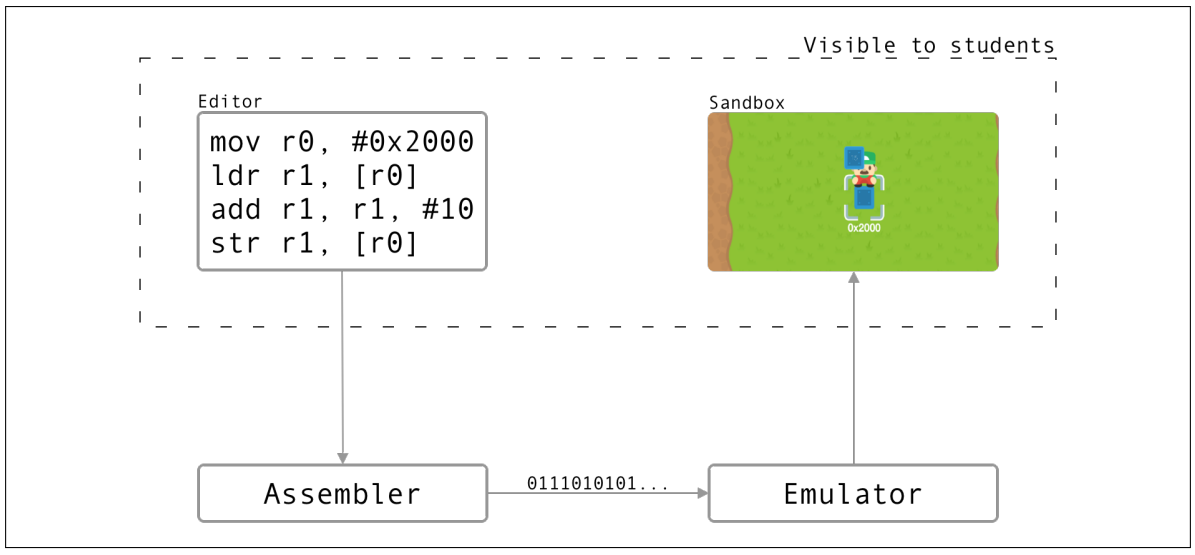

Figure 4.1: An overview of the four main components of the application.

The application interface was designed according to the Apple Human Interface

Guidelines. The Human Interface Guidelines "offer a high-level and comprehensive view of key UI elements and associated APIs, and best practices to help you implement features into your app" (24). Following these guidelines when creating the application ensured that any users who are familiar with iPad software already will have a basic understanding of the interface, and will be able to navigate it easily.

The implementation of the application consisted of designing an interface, developing the four components necessary for writing, assembling, and emulating ARM programs, and linking them all together into an iPad application.

4.1 Technology Decisions

After determining the application would run on an iPad, *Swift* was chosen as the development language for user-facing components of the application. iPad applications can be developed primarily in Objective-C or Swift, but Swift is a newer programming language designed specifically for iOS and iPadOS development. Lower-level components such as the emulator were programmed using C because of its speed and portability. In order to interact with the C code from the higher level Swift code, wrapper classes were created which call the C code directly. Swift automatically translates C code into Swift objects, but these wrapper classes allowed for a simpler, more native, and type-safe way of calling functions.

Open source frameworks have been developed for ARM assembling and emulating, however it was decided to program these from scratch in order to be license-compliant and provide all necessary features. A framework like Unicorn (25) can be used to emulate an ARM processor, however it is licensed under GPLv2. Distributing apps on the Apple App Store that contain GPLv2 licensed code would violate the license, which specifies that you may not impose additional restrictions when you redistribute the GPL licensed code of others. App store distribution would require imposing additional restrictions. Other more openly licensed frameworks can be used, however in order to include all the necessary features and simplify use with other modules, it was decided to program each component without the use of third-party frameworks or libraries. Developing the emulator, for example, allowed the use of callback functions to hook into program emulation at any point. After attempting to write a program that parsed ARM assembly language programs, the open source software Tree-sitter was added, which was used to generate a parser for the assembler. Developing an ARM parser in plain C would have taken a large amount of time due to the number of edge cases that must be handled.

Swift Package Manager was used to compile all of the component modules into the

final application. Swift Package Manager allowed the creation of separate packages for each component, and methods and classes could be specified as public to be used in other packages. This allowed code only relevant to one component to be hidden away in the package to avoid calling it from elsewhere. For components written in Swift, a simple Swift Package was created that contained all the source files. For components written in C, a Swift Package was created that contained all the C source files, and another Swift Package was created that contained wrapper classes and functions around the C code and contained the C-based Swift Package as a dependency. The final application contained all of the component Swift Packages as dependencies.

4.2 Editor

The editor component is responsible for providing an easy-to-use ARM integrated development environment where students can develop short ARM programs. The editor is visible to students and was written in Swift. It was built using UIKit, the framework responsible for all basic UI components on iPadOS, and uses a text view for the primary input of programs. The editor has text above the text view which displays the instructions for the current lesson, and a navigation bar containing the lesson title, reset, and step buttons as seen in Figure 4.2.

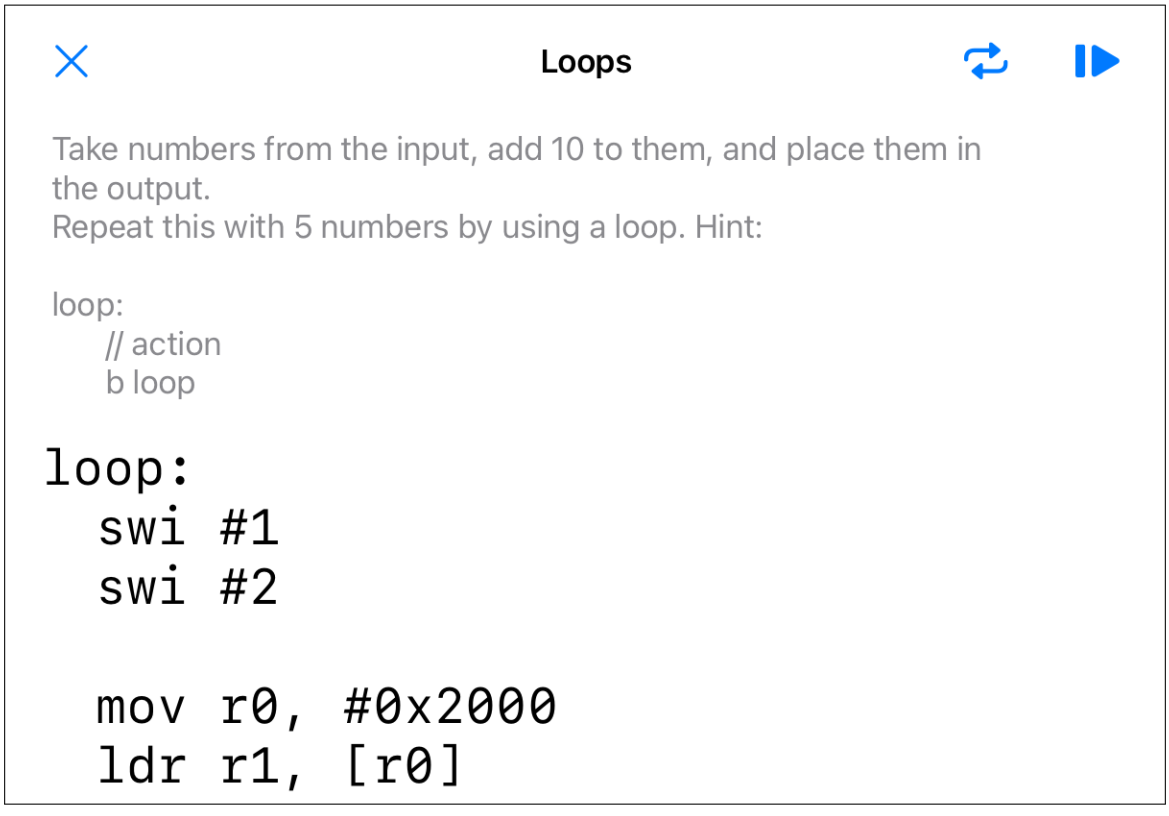

Figure 4.2: A screenshot of the assembly language program editor.

The reset button will reset the memory and sandbox to their default states, and the step button allows students to step through each instruction in their program. When the step button is first tapped after modifying a program, the program is automatically assembled and loaded into the emulator before the first instruction is stepped through. This gives students instant feedback when writing programs, and avoids the complicated task of assembling a program in a specific format and uploading it to a physical development board. The instant assembling and emulating of programs allows students to quickly make changes to their ARM programs and see the outcome in the sandbox while avoiding losing motivation due to long assembly and upload times. The editor also contains alerts that show when a lesson is complete and when the assembler finds a problem with a program that a student has written. Descriptive error messages are shown to students to aid in debugging.

4.3 Assembler

The assembler consists of a two-part pipeline that transforms a written ARM assembly program into machine code. The first step generates a syntax tree from the program which enables the second step to parse the syntax tree and create the machine code for each instruction.

4.3.1 tree-sitter-ARMv7

The first half of the assembler pipeline takes an ARM program as the input. This program is input as a string directly from the editor. A module was designed that converted the program to an abstract syntax tree. After attempting to create a parser from scratch in C, the complexity of handling each case resulted in the decision to turn to an open source framework. Tree-sitter (26) is described as a parser generator tool and an incremental parsing library that can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited. Tree-sitter generates parsers in pure C, meaning they can be used from a variety of programming languages. Parsers have been created for a large number of programming languages, but ARM is not one of these. In order to create a parser for ARMv7 assembly language, a custom Tree-sitter parser had to be created for the language.

Creating a Tree-sitter parser required downloading the template for a new language and defining a list of grammar rules using JavaScript. The Tree-sitter grammar DSL contains a number of functions for creating grammar rules, such as regex literals, numerical precedence, and sequences. Using these functions, 29 rules were created that enabled the parsing of ARM programs. These rules ranged from simple string

matches to complicated sequences and optional literals that resulted in a token. A data processing instruction, for example, consists of a sequence of a regex literal to match the mnemonic, an optional condition code, an optional set flag, whitespace, a register, a comma, a second register, another comma, and a second operand. The JavaScript code to define this rule can be seen in Figure 4.3. Symbols prefixed by a $ sign represent a reference to a symbol defined elsewhere in the grammar file.

```
data_processing_instruction: $ => choice(
   seq(
      /and|eor|sub|rsb|add|adc|sbc|rsc|orr|bic|mov|mvn|cmp|cmn|teq|tst/,
      optional(field('condition', $.condition)),
      optional(field('set_condition_flags', $.set_condition_flags)),
      ' ',
      field('Rd', $.register),
      ',',
      field('Rn', $.register),
      ',',
      $._data_processing_instruction_op2
   ),
)
```

Figure 4.3: An example Tree-sitter grammar rule definition for a data processing instruction.

Once the JavaScript grammar file was created, the `tree-sitter` command line tool was used to convert this grammar file into a parser written in C. This parser, along with some common Tree-sitter C files, was used to parse ARM code. A small Swift binding library was created to simplify the process of parsing a Swift string into a tree. An example of an S-expression created by the parser for an ARM instruction can be seen in Figure 4.4.

4.3.2 swift-tinyasm

Once a syntax tree for an ARM program was created, it was then necessary to convert this into machine code. The first step of this process required extracting any labels from the syntax tree and converting them to an absolute offset. ARM assembly machine code has no concept of labels and instead has relative and absolute offsets that are derived from labels by the assembler. Because ARMv7 instructions are all 32-bits, calculating the offset for each label was done by counting how many instructions came before it. The instructions were then converted into machine code. This involved setting specific bits of a 32-bit unsigned integer for each instruction type. A module was created, called swift-tinyasm, to perform this conversion. The module was written in Swift because it required complicated string parsing. Swift

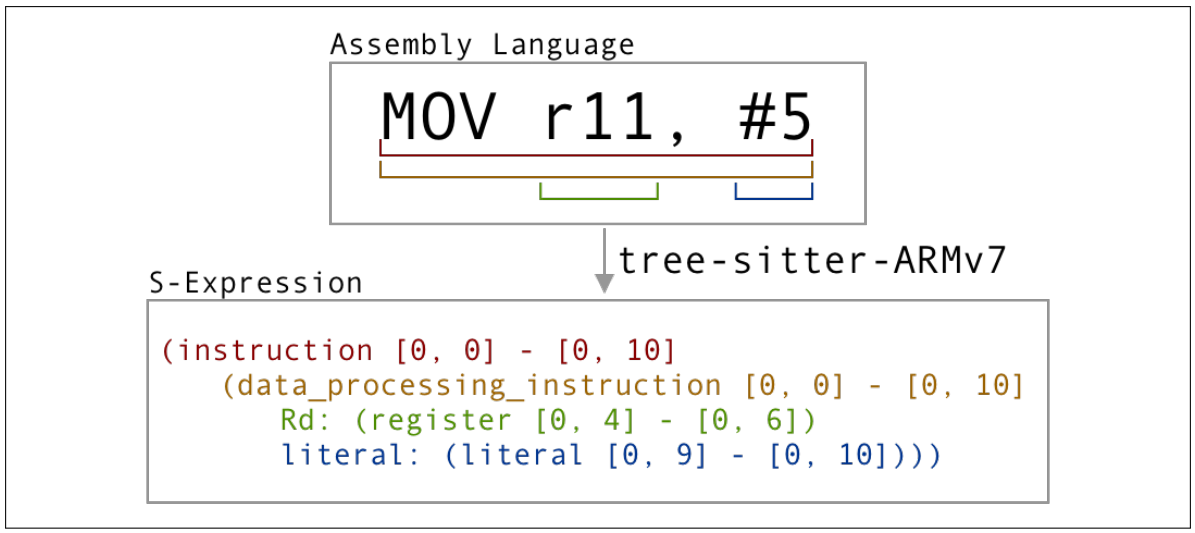

Figure 4.4: An example of an s-expression generated by tree-sitter-ARMv7 representing the components of an ARM instruction.

contains a fast and modern `string` implementation (27) which is ideal for interacting with the string ranges given by each node in the syntax tree.

Converting the syntax tree to an array of 32-bit integers was possible by iterating over each instruction node in the tree. An instruction node represented a single ARM instruction and each instruction occupied 32 bits. Each instruction category described in Chapter 3.3 was encoded using a separate function. The data processing category, for example, was encoded as follows.

Data Processing Instruction Encoding

The condition bits of the data processing instruction were encoded first. If the instruction node contained a condition child node, the condition bits were set based on hard-coded values for each condition. If the instruction node did not contain a condition, the condition bits were assigned a value of `AL`, or "always execute". Next, the "set condition codes" bit of the instruction was encoded. If the instruction node contained a `set_condition_flags` child node, the "set" bit was set high, otherwise it was low. The opcode bits were set based on hard-coded values for each mnemonic, which was extracted as the first three characters of the instruction. The register bits were set based on the register child nodes `Rd` and `Rn`. The register number was encoded in the bits of the registers. The `Operand 2` bits of the instruction were encoded either as a shifted register or a rotated immediate value based on what child nodes were present in the instruction. The instruction encoding layout for a data processing instruction is shown in Figure 4.5. A diagram of how a specific instruction correlates to each encoded bit can be seen in Figure 4.6.

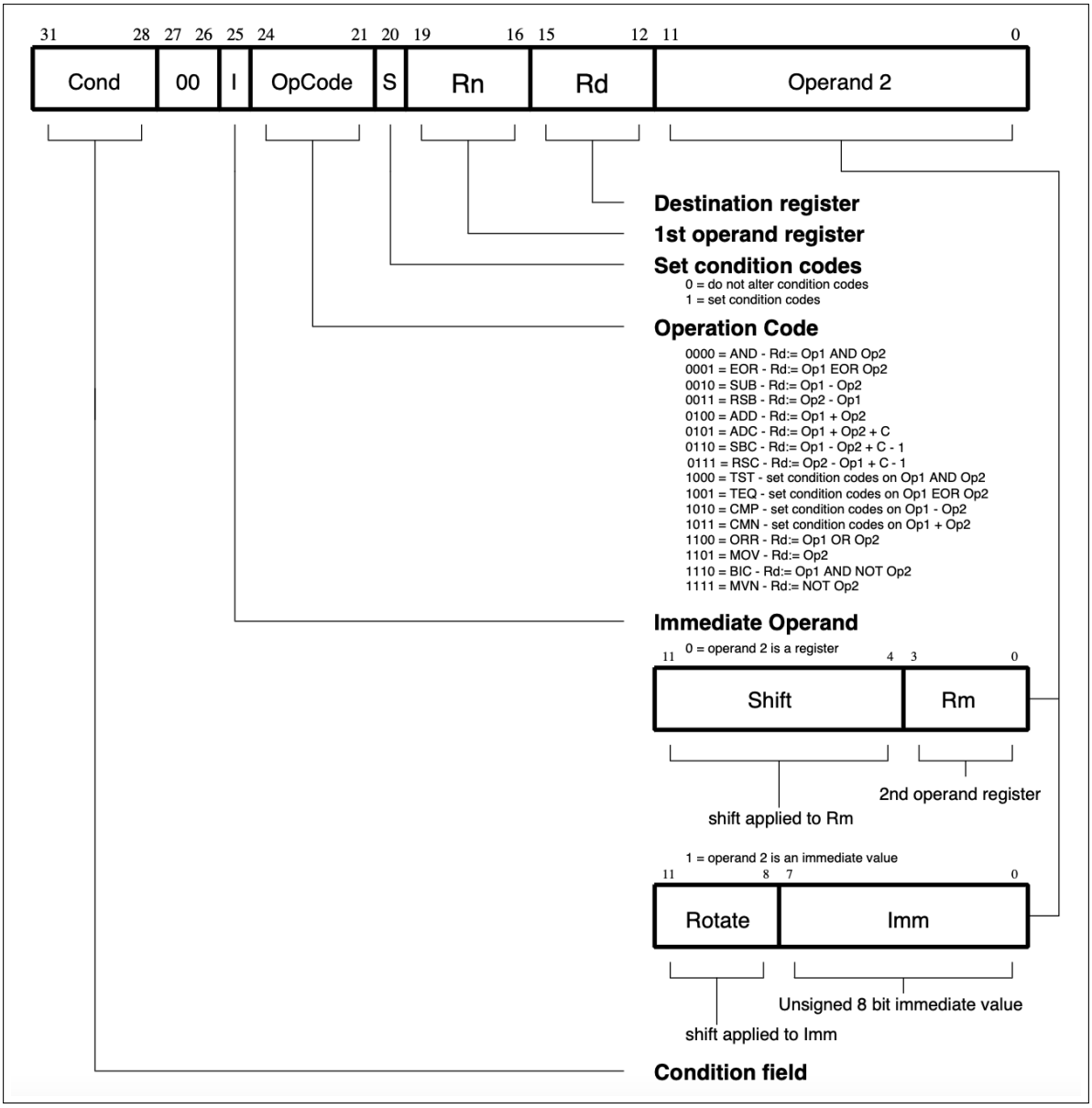

Figure 4.5: The encoding used to represent a data processing instruction in 32-bits, with the meanings of each bit. Diagram from ARM (5)

4.4 Emulator

The emulator component of the application is responsible for taking assembled ARM instructions and running them on an emulated CPU and MMU. This component was programmed in the C language, as it is the most complicated part of the application. Programming it in C enables further work to use this component in other applications, such as a web-based version of the app. A simple Swift wrapper around the C code allowed other Swift-based components of the app to interact with the emulator easily.

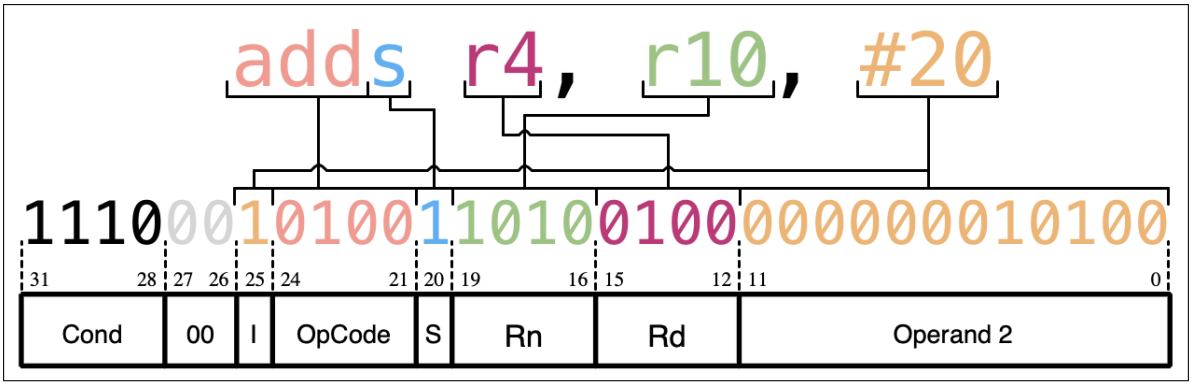

Figure 4.6: The relationship between the components of an instruction and the bits in which they encode to.

The emulated CPU contains 16 variables storing the value of `r0-r15`, and a variable storing the value of the `CPSR`. When executing instructions such as data processing instructions, operations are performed on the register variables and stored in the resulting register. If a data processing instruction has the S-bit set, the negative, zero, carry, and overflow bits of the CPSR variable will be updated based on the result of the instruction. This allows for conditional execution of any instruction. The four highest bits of all instructions in ARMv7 represent the condition code. The emulator parses the condition code of each instruction, reads the CPSR variable, and decides whether or not an instruction should be executed. The instruction `ADDNE`, for example, will only be executed if the value of the Z (zero) bit of the CPSR variable is 0.

The emulator decides whether or not an instruction should be executed by using a function that takes in the condition code and the current CPSR value. A switch statement takes the condition code and for each value checks specific bits of the CPSR value. The function can be seen in Figure 4.7

The emulated MMU has two arrays of bytes: one that represents the MMU's flash memory and one that represents the MMU's SRAM. These were both programmed so that they could be variable size, meaning specific physical ARM development boards can be emulated by setting the emulated memory size to be the same as the physical board. This allows programs to be written that must stay within the memory constraints of a physical board. In this application, however, the programs written are simple enough to easily fit in a small amount of memory. When the emulator loads a program, the program is loaded into the flash memory of the MMU. The CPU then reads instructions from the MMU flash memory based on the program counter register. This register is incremented after each instruction execution and is also modified by branch instructions and any other instruction that directly operates on it. The emulated MMU contains functions to reset the memory, load a program into memory, and read or write memory in byte or word increments. This allows

```c
int cond_is_fulfilled(uint8_t cond, uint32_t cpsr) {
    switch (cond) {
        case 0b0000: // EQ
            return (cpsr >> 30) & 0b1;
        case 0b0001: // NE
            return ((cpsr >> 30) & 0b1) == 0b0;
        case 0b0010: // CS
            return (cpsr >> 29) & 0b1;
        case 0b0011: // CC
            return ((cpsr >> 29) & 0b1) == 0b0;
        case 0b0100: // MI
            return (cpsr >> 31) & 0b1;
        case 0b0101: // PL
            return ((cpsr >> 31) & 0b1) == 0b0;
        case 0b0110: // VS
            return (cpsr >> 28) & 0b1;
        case 0b0111: // VC
            return ((cpsr >> 28) & 0b1) == 0b0;
        case 0b1000: // HI
            return ((cpsr >> 29) & 0b1) & (((cpsr >> 30) & 0b1) == 0b0);
        case 0b1001: // LS
            return (((cpsr >> 29) & 0b1) == 0b0) | ((cpsr >> 30) & 0b1);
        case 0b1010: // GE
            return ((cpsr >> 31) & 0b1) == ((cpsr >> 28) & 0b1);
        case 0b1011: // LT
            return ((cpsr >> 31) & 0b1) != ((cpsr >> 28) & 0b1);
        case 0b1100: // GT
            return (((cpsr >> 30) & 0b1) == 0b0) & (((cpsr >> 31) & 0b1) == ((cpsr >> 28) & 0b1));
        case 0b1101: // LE
            return ((cpsr >> 30) & 0b1) | (((cpsr >> 31) & 0b1) != ((cpsr >> 28) & 0b1));
        case 0b1110: // AL
            return 0b1;
        default:
            printf("Unexpected Condition\n");
            return 0b1;
    }
}
```

Figure 4.7: The function used to determine whether an instruction should be executed based on the condition code and CPSR register value.

instruction such as `LDR` (single data transfer) to read a byte from the MMU and store it in one of the register variables in the emulated CPU. The sandbox uses the memory write function of the MMU to load a number block into emulated memory when it is placed inside the "memory slot" in the sandbox.

Similar to the assembler, the emulator was developed by looking at the ARMv7 Instruction Set and creating a function for each instruction. These functions read each bit of the instruction using C bitwise operators and perform actions on the register variables and simulated memory based on which bits are set. The `BX` (branch and exchange) function, for example, will first check the highest four bits of the instruction and verify this condition against the values in the CPSR register. It will then read the lowest four bits as the number of a register and store the value in that register variable into the program counter register. This closely mimics what a real ARM CPU would do when reading the instruction.

In order to update the sandbox based on the state of the emulator, callback functions were implemented for three actions. The emulator stores three function pointers for these callbacks: one that is called whenever a register value changes, one that is called when a software interrupt instruction is executed, and one that is called after each instruction execution.

The register value change callback is useful for detecting the value of a register at any given point during program execution. This is used to display register values in the sandbox. The starting value of each register is shown, and the callback function is called whenever an instruction in the emulator updates one of the registers, passing the register number and the updated value. The sandbox then updates the displayed value for the register.

The software interrupt execution callback is the most important callback, as it functions as the primary link between the emulator and sandbox. In a typical ARM processor, a software interrupt instruction will cause a software interrupt exception, which is used to call operating system routines. There is no operating system in this application, but the sandbox area is interacted with similarly to how an operating system would be. When a software interrupt instruction is executed by the emulator, the associated callback function will be called, and a number known as a "comment" is passed as input. This allows different software interrupt calls to control different actions in the sandbox, such as moving the character around the sandbox area. The emulator will wait for the callback function to be completed, so actions that take time will pause the emulator. This prevents instruction execution from overlapping and causing instructions to be executed out of order.

The instruction execution callback is called after any instruction is executed by the emulator. This is used to perform updates in the sandbox. When a number block is in the "memory slot" of the sandbox, the sandbox will read the memory address after each instruction to keep the value in the sandbox synchronized with any changes in the emulator.

The emulator implementation provides a simulated ARM MMU and CPU that can execute all relevant ARM instructions identified in Chapter 3.3 as well as callback functions for connecting peripherals to the emulator. The language choice of C allows emulator functions to be called from any language that supports interaction with C.

4.5 Sandbox

The sandbox component of the application forms the other half of what students see on screen. The sandbox takes up the right half of the screen and allows programs from the left-hand editor to perform actions in a visual sandbox. The graphics of the sandbox were implemented using *SpriteKit*, an Apple framework for drawing 2D graphics. SpriteKit uses a tree-based scene structure, with nodes representing each sprite or asset in a scene.

Assets for the sandbox were taken from Kenney.nl (28), a website providing thousands of game assets under a public domain license. These high-quality assets give the sandbox a professional look that can easily be expanded upon in the future using animations.

The background assets of the sprite are rendered as a tilemap, where each tile is either a wall or ground tile. Above the tilemap are nodes representing the player and number blocks. These nodes are translated and scaled around the scene using animations built into SpriteKit, allowing smooth translation over a set duration.

The editor-sandbox interaction described in Chapter 3.6 was implemented using the emulator's callback functions. The `onRegisterChange` callback is used to update the values of the register inspector each time any of them change. This is done after an instruction that operates on a register is executed, after an instruction updates the condition flags register, and after the program counter register is changed. The new register value is reflected in the number next to the register number in the register inspector area of the sandbox.

The `onSoftwareInterrupt` callback function is responsible for performing actions in the sandbox based on the comment field of a software interrupt. When a software interrupt instruction is executed by the emulator, the `onSoftwareInterrupt` callback is called with the comment passed as input. A switch statement parses the comment and determines what action should be performed by the sandbox. These actions can consist of moving the character to the input, picking up numbers, and placing them in memory or output.

The memory-mapped component of the sandbox was implemented using the `afterInstructionExecution` callback. After each instruction is executed, the sandbox checks whether or not the number block node is in the position of the memory slot. If it is, the sandbox will tell the emulator to store the value of the number block into the memory location specified. If the number in memory is changed, the new value will be reflected in the visual number block in the sandbox. This provides the illusion of numbers being physically placed in memory locations by the character.

The use of these three callback functions provides a realistic and easy-to-use way to interact with the sandbox using code. SpriteKit made it easy to add a visual 2D scene that can engage students more than basic register values while still closely following a real-world peripheral architecture.

4.6 Lessons

Lessons form the educational content of the application and contain instructions and hints to guide students through an assembly language concept. Lessons were implemented as a Swift struct, with fields for lesson title, instructions, image, and verification function. This simple struct allows for easy creation of new lessons, so the rest of the app was designed to display any number of lessons. Instructors can create new lessons using the "Add Lesson" page, which allows lesson information to be input and saved. New lessons are added to the list of existing ones.

To allow students to choose which lesson they want to work on, the initial view of the application is a lesson selection view. This displays all the default lessons and any custom lessons added to the app and shows a title and image for each one. The images for the default lessons were designed to be simple gradients with an icon representing the lesson topic. This gives students an idea of what the lesson is about at first glance. Selecting a lesson will transition the application into the editor and sandbox area, and successfully completing a lesson will return students to the lesson selection screen.

The lesson struct was set to conform to the Swift *codable* protocol, giving it smooth conversion to external types with no extra code. For example, this allows lessons to be encoded and decoded into JSON, which lets instructors and students share a JSON file containing custom lessons. These lessons can be imported and exported, allowing instructors to create custom lessons on advanced topics for students to complete.

A verification function is used to test the output of a student's program. The function takes the input and output numbers from the sandbox and verifies that the output is correct based on the current lesson. If the student has written a program that follows the specification of the lesson, the output will match the expected output for a lesson and the verification function will return true. To allow custom lessons to be created from the app, the verification function had to be code that was not pre-compiled and could be run using Just-In-Time compilation. The easiest way to do this was using the JavaScriptCore framework built into iPadOS.

JavaScriptCore is the JavaScript engine used by many Apple apps such as Safari and

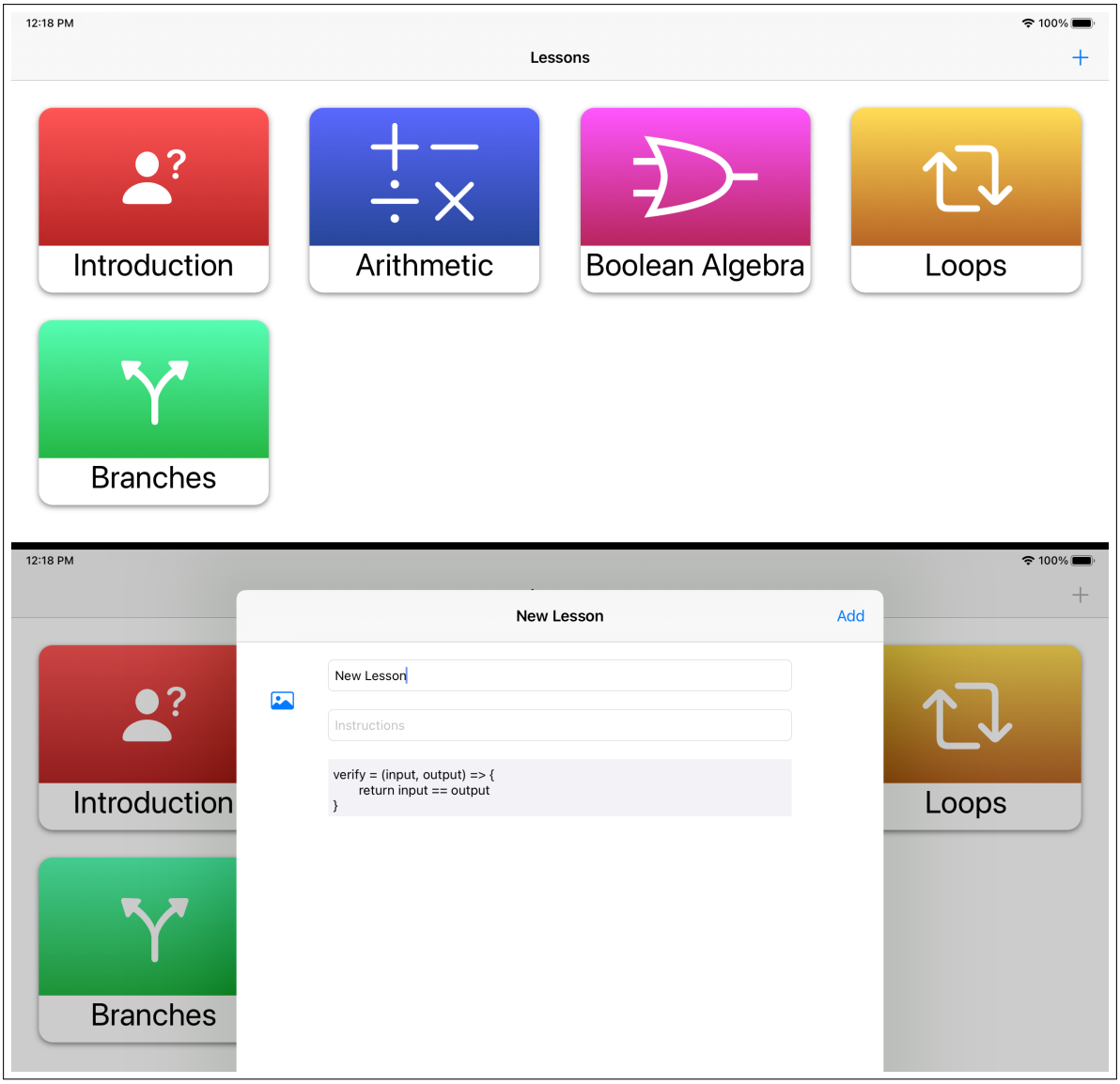

Figure 4.8: A screenshot of the lesson selection screen and lesson creation screen.

is available to use as a library in iPadOS apps. JavaScriptCore allows the lesson verification functions to be evaluated from within the Swift app by evaluating them using the JavaScript virtual machine and casting the return type to a native Swift value. Instead of bundling compiled code into the application, lesson verification functions can be simple strings that are loaded into JavaScriptCore right before verification is needed, allowing the creation of new lessons from within the app.

The lesson architecture designed for this application enables an easily expandable application where students and instructors can create and share lessons beyond the material included in the app. This implementation also allows for further work to easily add more advanced lessons for covering assembly language topics not included in the current application.

4.7 Security Considerations

This application was designed with security in mind and contains a very small attack surface. The application does not communicate with the Internet and does not have any user login, which greatly limits the opportunity for attacks. Because the application contains multiple places where users can write code that is executed, the main security concern for this application is the arbitrary execution of malicious code by an attacker. A proactive security approach was taken to limit the risk of any information being exposed by an attack, and no third-party code was used in order to make sure all code met high security standards. Many significant security breaches have been the result of companies missing vulnerabilities in third-party applications (29), so instead of auditing each dependency, it was decided that all components of the application would be written from scratch.

The ARM emulator allows for the execution of any ARM program written by students, and therefore has associated security concerns. In order to prevent unauthorized access to any data, the emulator CPU and MMU are completely self-contained and have limited emulated memory. The MMU contains simulated SRAM and flash memory which programs are loaded into, meaning any executed code is only able to access SRAM memory, which is zeroed out before any code is executed. The emulator contains very limited access to any other parts of the application and is only able to interact with the sandbox area. No user information can be exposed through this emulator, and all code is confined to the emulated memory.

The intended distribution method of this application is through the Apple App Store, which imposes sandboxing restrictions on all applications. Sandboxing is an OS-level feature that runs all applications in separate "sandboxes", meaning applications will not have access to data created by other applications and cannot make changes to the device they are running on. This adds an extra layer of security, as any security exploit will still only have access to files and data created by this application, which do not contain any sensitive information.

5 Evaluation

5.1 Results

A software tool for use in teaching ARM assembly language and fundamental computing concepts has been presented in this paper. In order to evaluate the effectiveness of this tool, the ability of the tool to teach the learning objectives identified in Chapter 2.3 was investigated and the general application feature set was discussed and compared to current simulators. This provides a summary of how well the original goal was achieved.

5.1.1 Learning Objectives

In Chapter 2.3, seven core learning objectives were identified for use in teaching assembly language. These learning objectives were used to implement educational lessons in the application that taught various computing concepts. An evaluation of how well each objective was achieved is shown in Table 5.1 and described in detail below.

Instruction Set Architecture	✓ Fully Achieved
Arithmetic and Data Processing	✓ Fully Achieved
Boolean Logic and Arithmetic	✓ Partially Achieved
Memory Management	✓ Partially Achieved
Branches and Loops	✓ Fully Achieved
Subroutines	✗ Not Achieved
Peripherals	✓ Partially Achieved

Table 5.1: Evaluation of how well each original learning objective was achieved by the final application.

Instruction Set Architecture

A variety of ARM instructions were introduced and both instructions that operate on registers and instructions that read and write memory were included in the material

covered by the lessons.

Arithmetic and Data Processing

Multiple lessons introduced arithmetic and data processing instructions, providing a good foundation for teaching this learning objective.

Boolean Logic and Arithmetic

The boolean logic lesson introduced boolean instructions, however further work could expand on this with boolean arithmetic lessons.

Memory Management

Memory management was introduced in the arithmetic lesson to enable loading and storing numbers from the sandbox. More complex memory management, such as the use of stacks, was not explained in the current lessons.

Branches and Loops

The branches lesson and the loops lesson both served to teach this learning objective as they required students to implement programs using conditional branches.

Subroutines

The concept of subroutines was not taught in any of the existing lessons.

Peripherals

The sandbox acted as both a memory-mapped peripheral and software interrupt triggered peripheral, however more work could be done to relate these more directly to real-world peripherals.

5.1.2 Feature Set

The final application can be evaluated against past work described in Chapter 2.5 to compare and contrast the feature set of each application. Two distinct categories of simulators were found in Chapter 2.5.2: industry-oriented and education-oriented simulators. The final application designed for this project lacked many of the complicated features of industry-oriented products such as advanced debugging, breakpoints, and specific simulated hardware, however these features were determined to be beyond the scope of an educational tool. The final application had a much simpler setup process than tools like Keil μVision and students can download

the application and immediately start writing assembly language. The application does lack some features that the education-oriented applications included, such as a visual program counter, however it improved upon these applications in other areas. The application contains instructional content as well as automatic evaluation of programs, meaning, unlike other educational simulators it can be used independently of a college course. It can also be used in conjunction with a college course, as instructors can add their own lessons to the application.

5.2 Limitations

While the application was able to largely achieve the original goal of this project, there are still limitations to the final outcome. The most limiting aspect of the final application is the sandbox interaction. The sandbox acts as a memory-mapped and software interrupt triggered peripheral, and both aspects have limitations in the current application.

Software interrupt triggered actions in the sandbox are similar to system calls used in assembly language programs running on operating systems, however the meaning of these software interrupts can be difficult to interpret. A software interrupt instruction is passed a number to specify which interrupt to call, and in this application these numbers are used to specify different actions (1 = move to input, 2 = pickup, etc.). These numbers are arbitrary and difficult to associate with an action, so in the current application the number for each action is explained in the lesson instructions. This limitation could be improved upon by injecting labels into the assembly source before assembling. A textual label such as `pickup` could be added to the top of the source with a numerical value corresponding to the software interrupt. This would enable programs to instead perform actions using these textual labels. For example, instead of using `SWI #1` to trigger a pickup, students could write `SWI "pickup"`. This would greatly improve the readability of programs.

The memory-mapped slot in the sandbox closely simulates a memory-mapped peripheral, where changing values at a specific memory location will have some effect on the peripheral. The current implementation is limited to one memory location. This works for lessons that load in the number from memory and perform an action on it, however more advanced lessons with two numbers are not currently possible. A lesson that instructs students to pick up two numbers, add them together, and place them in the output, for example, is not possible with the current sandbox implementation. Enabling more advanced memory mapping would allow more interactive lessons and may improve engagement.

5.3 Further Work

There are many opportunities for further work to contribute to this project. Several improvements were discussed in the previous chapters that were beyond the scope of this project, and these could be easily incorporated into the app in the absence of time constraints.

The first area of improvement for this application is the user interface and experience. The current user interface offers a simple and easy-to-use environment for writing ARM assembly, however some features were excluded which would improve the programming experience. Syntax highlighting is not as important for assembly language as it is for higher-level languages due to the reduced instruction set, however implementing syntax highlighting in the editor would allow a better understanding of programs at a glance. Register operations would be clear and distinguishable which could aid in debugging. Another interface improvement to assist debugging could be a visual program counter indicator which would show which instruction is currently being stepped through. This is a common feature included in many of the simulators discussed in Chapter 2.5.2. The sandbox assets could also be improved upon to provide a more cohesive visual look, which may improve engagement as discussed in Chapter 2.6.

Another area of improvement for the application is the included lessons. The current application contains lessons for teaching basic assembly language concepts and touches on six of the seven learning objectives identified in Chapter 2.3, however some lessons could be improved upon and more advanced lessons could be developed. Subroutines, for example, were identified to be an important concept for teaching assembly language, however due to time constraints, a subroutine lesson was not included in the application. A lesson on subroutines and any other fundamental concepts could be the target of future work and could be easily implemented using the lesson editor included in the application. There is also potential for a "lesson catalog" where students and instructors can upload lessons they have designed for anyone to download and complete.

The peripheral limitations can be improved upon in the future by improving the sandbox component of the application. More simulated memory slots open up the possibility of more advanced lessons, and there is also potential for more interaction in the sandbox. An interesting area of future work on this application would be to experiment with interactive graphics in the sandbox, such as visual switches and buttons that could have a direct effect on the assembly language program currently running.

Finally, and perhaps most importantly, there is a clear need to test this application in the real world. Due to time and scope constraints, the final application was not tested by any students. Doing so could provide useful and meaningful feedback on areas of improvement. Similar to studies carried out by Lee et al. (30), the application could be evaluated by performing a "think-aloud" study, where students with no assembly language experience could use the application and their experience could be recorded. The application could also be evaluated against a group of students using an existing simulator with no visual sandbox to study the impact of visual gamified feedback on engagement and learning. Observing both students and instructors as they use the application could reveal previously unknown areas of improvement and offer valuable insights into how people will use the application.

5.4 Conclusions

The original goal of this project was to create an ARM simulator that can be easily set up and installed and provides self-contained educational content and visual feedback while keeping students engaged. This goal was further developed in Chapter 3, resulting in the idea for an iPad application containing an ARM IDE, emulator, and sandbox with self-contained lessons that teach students basic computing concepts.

During the process of this project...

- The usage of ARM assembly language in teaching core computing concepts was investigated. A case was made for using a simulator instead of a development board based on the research done in Chapter 2. Existing ARM simulators were evaluated and areas of improvement were found. Research was done into how motivation and engagement can be encouraged in a teaching tool.
- A design was created for an iPad application consisting of five lessons with instructions and automatic grading. Instead of simulated peripherals, a visual sandbox was designed based on principles extracted from past research.
- ARM assembly language lessons were created based on core computing curriculum and a mechanism for automatically grading each lesson was developed.
- The final design was implemented as separate components for reusability and language choice, resulting in a fully self-contained iPad application consisting of a code editor, assembler, emulator, and sandbox.

The application contains several lessons that teach assembly language concepts, a

simple IDE for programming solutions to each lesson, and a visual sandbox that updates according to executed code. Behind the scenes, an assembler and emulator build and run these programs. ARM assembly language was chosen due to it's prevalence in education and ease of learning, however because of the modular design of the application, the assembler and emulator can be replaced with an assembler and emulator for any assembly language and the editor and sandbox will not need to be modified. The core assembly language concepts will remain the same.

The memory-mapped slots in the sandbox offer a way to interact with the sandbox that closely mimics real-world peripherals, however looking back upon the project the software interrupt-based actions could be revised. Software interrupts were used to perform actions in the sandbox because of the simplicity they provided. Using them enabled students to learn how to use a software interrupt in the first lesson and be able to perform actions in the sandbox with one line, however this method ended up taking away from the gamified experience. Using software interrupts to interact with the sandbox does not result in the feeling of direct control over the sandbox that was intended at the beginning of the project. Alternative ways to interact with the sandbox can be developed in the future that offer more direct control, such as controlling the exact movement of the character using assembly language. This would significantly inflate the upfront cost of learning to interact with the sandbox, however it would likely result in a more coherent experience, leading to improved student engagement.

Although a number of limitations have been identified, each one can be addressed relatively easily through future work on this project. Further time and experimentation can improve the educational material of the application and offer a better experience to users.

The final result of this project offers a unique ARM simulator unlike any existing tools. While the advanced feature set of industry-oriented simulators may surpass the feature set of this application, the application supports many of the features identified in existing educational simulators. Unlike current simulators, however, this application contains educational content capable of teaching ARM assembly language independently of a college-level course.

While a simulator is not a complete replacement for a real ARM microprocessor, this application provides a useful tool for beginners. The application is free, easy to install, and provides a visual sandbox analogous to peripherals that may not be accessible to students due to financial constraints. For students with access to ARM development boards, this tool can still be useful for learning how to write programs before needing to learn how to connect a commercial IDE to a development

board.

Although there is a benefit to having a dedicated instructor while learning, this tool can simplify the teaching process and free up instructors to focus on more specific challenge areas students have. Students are able to self-guide their learning and can complete lessons at their own pace.

The end result of this project is a novel iPad application which is of a standard suitable for submission to the iOS app store. It provides a solid foundation for achieving the original goal of this project and improving upon it in the future.

Bibliography

[1] William Yurcik. Editorial. *Journal on Educational Resources in Computing*, 1(4):1–3, December 2001. ISSN 1531-4278. doi: 10.1145/514144.514700. URL https://doi.org/10.1145/514144.514700.

[2] Randall Hyde. *The art of assembly language*. No Starch Press, San Francisco, 2003. ISBN 9781886411975.

[3] Arm Ltd. The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter. URL https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter.

[4] A. Clements. Arms for the poor: Selecting a processor for teaching computer architecture. pages T3E–1, 11 2010. doi: 10.1109/FIE.2010.5673541.

[5] *ARM7TDMI-S Technical Reference Manual*. ARM Limited, 3 2004. Rev. B.

[6] ACM. Association for computing machinery curricula recommendations. 2016.

[7] UK DFE. National curriculum in england: Computing programmes of study. 2013.

[8] William Hohl and Christopher Hinds. *ARM assembly language: fundamentals and techniques*. 2015. ISBN 9781482229882 9781482229875 9781482229868 9781482229851. URL http://www.crcnetbase.com/isbn/9781482229868. OCLC: 900192804.

[9] R. N. Horspool, D. Lyons, and M. Serra. Armsim# - a customizable simulator for exploring the arm architecture. In *FECS*, 2009.

[10] Weiying Zhu. Teaching assembly programming for arm-based microcontrollers in a professional development kit. pages 23–26, 05 2017. doi: 10.1109/MSE.2017.7945077.

[11] Nurettin Topaloglu and Osman Gürdal. A highly interactive pc based simulator tool for teaching microprocessor architecture and assembly language programming. *Elektronika ir Elektrotechnika*, 02 2010.

[12] C. Ardito, M. De Marsico, R. Lanzilotti, S. Levialdi, T. Roselli, V. Rossano, and M. Tersigni. Usability of e-learning tools. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, page 80–84, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138679. doi: 10.1145/989863.989873. URL `https://doi.org/10.1145/989863.989873`.

[13] Kathryn Kasmarik and Joe Thurbon. Experimental evaluation of a program visualisation tool for use in computer science education. pages 111–116, 01 2003.

[14] Geetika Malhotra, Namita Atri, and Smruti R. Sarangi. emuarm: A tool for teaching the arm assembly language. In *2013 Second International Conference on E-Learning and E-Technologies in Education (ICEEE)*, pages 115–120, 2013. doi: 10.1109/ICeLeTE.2013.6644358.

[15] G. Surendeleg, Violet Murwa, Han Yun, and Y.S. Kim. The role of gamification in education - a literature review. *Contemporary Engineering Sciences*, 7:1609–1616, 01 2014. doi: 10.12988/ces.2014.411217.

[16] Michael Lee and Amy Ko. Investigating the role of purposeful goals on novices' engagement in a programming game. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 09 2012. doi: 10.1109/VLHCC.2012.6344507.

[17] M. Lee and A. Ko. Personifying programming tool feedback improves novice programmers' learning. In *ICER*, 2011.

[18] Kim Buckner. A non-traditonal approach to an assembly language course. *J. Comput. Sci. Coll.*, 22(1):179–186, October 2006. ISSN 1937-4771.

[19] Michael J. Lee, Andrew J. Ko, and Irwin Kwan. In-game assessments increase novice programmers' engagement and level completion speed. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, page 153–160, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322430. doi: 10.1145/2493394.2493410. URL `https://doi.org/10.1145/2493394.2493410`.

[20] Kathleen M. Swigger and Layne F. Wallace. A discussion of past programming errors and their effect on learning assembly language. *Journal of Systems and Software*, 8(5):395–399, 1988. ISSN 0164-1212. doi:

https://doi.org/10.1016/0164-1212(88)90030-1. URL https://www.sciencedirect.com/science/article/pii/0164121288900301.

[21] Lin Shan. Exploration of education reform based on 32-bit assembly language programming. In *2011 6th International Conference on Computer Science Education (ICCSE)*, pages 595–599, 2011. doi: 10.1109/ICCSE.2011.6028709.

[22] Apple classroom. URL https://www.apple.com/education/docs/getting-started-with-classroom.pdf.

[23] Swift Playgrounds. URL https://www.apple.com/swift/playgrounds/.

[24] The developer's guide to the Human Interface Guidelines - Discover - Apple Developer. URL https://developer.apple.com/news/?id=yyz8lqtw.

[25] Unicorn. Unicorn – The Ultimate CPU emulator. URL https://www.unicorn-engine.org/.

[26] Tree-sitter - Introduction. URL https://tree-sitter.github.io/tree-sitter/.

[27] *Strings and Characters — The Swift Programming Language (Swift 5.4)*. Apple Inc. URL https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html.

[28] Kenney • Home. URL https://kenney.nl.

[29] 4 Risks to consider when implementing third-party code. URL https://about.gitlab.com/blog/2019/07/16/third-party-code-risks/.

[30] Michael J. Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, Sheridan Long, Margaret Burnett, and Andrew J. Ko. Principles of a debugging-first puzzle game for computing education. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 57–64, 2014. doi: 10.1109/VLHCC.2014.6883023.

A1 Source Code

The source code for this project can be found at
`https://github.com/Finnvoor/ARMSandbox`.