Granule

ARM Thumb simulator

Author:
Joshua Worth
(u5012808)

Supervisor:
Dr. Eric McCreairth
Abstract

The Introduction to Computer Systems course at The Australian National University (ANU) currently has a pseudo-assembly simulator which is used for teaching the fundamental underlying principles of computer processors and programming. The goal of this project was to implement a similar simulation program (named “Granule”) which instead uses an instruction set used in real processors, specifically ARM® Thumb. This program closely replicates the functionality of the factory assembler for Thumb chipsets, and the way it interprets instructions. It is implemented to be extensible and robust so that future modifications will be logical and keep the code in a clean, readable state.
Acknowledgements

I would like to thank Dr. Eric McCreath for supervising this project, helping with the program design, and meeting with me on a regular basis to discuss its progress. I would also like to thank Weifa Liang for running the project courses, consistently reminding us of approaching deadlines, and generally wanting us to succeed.

Contents

1 Introduction 3
  1.1 Background ............................................. 3
  1.2 Similar Projects ........................................... 4
  1.3 Contribution ............................................. 4
  1.4 Report Structure ......................................... 5

2 Project Milestones 5
  2.1 Assembler and Virtual Machine Layout ...................... 5
  2.2 GUI .................................................... 5
  2.3 Finding and Reading Documentation .......................... 5
  2.4 Assembler ............................................... 7
  2.5 Virtual Machine and Memory ................................ 7
  2.6 Virtual Devices ......................................... 7

3 Design 8
  3.1 Architecture ............................................ 8
  3.2 Language Specifics ....................................... 8
  3.3 Version Control .......................................... 8
  3.4 Project Licence .......................................... 9
  3.5 Code .................................................. 9

4 Implementation 10
  4.1 GUI and Terminal Interface ............................... 10
  4.2 Assembler ............................................... 17
  4.3 Virtual Machine ......................................... 20
  4.4 Virtual Devices ......................................... 23
List of Figures

1  Diagram of all classes in the main package .......................... 12
2  Diagram of all classes in the editor package .......................... 15
3  Screenshot of Granule’s main window ................................. 16
4  Diagram of all classes in the compile package ....................... 19
5  Diagram of all classes in the machine package ....................... 22
6  Detailed diagram of all classes in the compile package ............. 34
7  Detailed diagram of all classes in the editor package ............... 35
8  Detailed diagram of all classes in the machine package ............ 36
9  Detailed diagram of all classes in the main package ............... 37

List of Acronyms

ANU  The Australian National University
GUI  Graphical User Interface
IDE  Integrated Development Environment
ISA  Instruction Set Architecture
JIT  Just In Time
JVM  Java Virtual Machine
1 Introduction

“Learning only high level languages without any assembly would be like skipping Newton’s physics models and going straight to quantum mechanics.”
– Jason Ernst (2010)

1.1 Background

When learning how to program a computer, one of the most important things to learn is how to write in assembly language. This is so that when you write in higher level languages, from those which are very similar to assembly such as C, to those which abstract the procedural nature of assembly away such as Haskell, you will understand how they work and what is really going on underneath. With this knowledge, you can understand exactly why some code is faster than other code, and know why a computer really is “dumb”: because underneath everything that happens on a computer is just millions of very simple, built-for-purpose commands running in succession.

The Introduction to Computer Systems course at ANU aims to teach these concepts. To do so, a program called PeANUt was used for many years. This simulator had an extensive instruction set with a 30-page specification manual. A new program called rPeANUt was written by Eric McCreath in 2011. This was a “RISC” version of PeANUt, meaning that its instructions worked with a load-to-register, perform operation, store-to-memory architecture in mind, as opposed to PeANUt, which had many instructions which performed several operations in one line of assembly. RPeANUt had a much simpler design, and was intended to make both teaching and learning a faster process.

Both of these simulators used a pseudo-assembly language, which meant that they did not apply directly to real-life. The aim of this project was to implement a new simulator for an Instruction Set Architecture (ISA) used in real chipsets. For this, the ARM® Thumb ISA is used, which in many aspects is similar to the PeANUt simulator.

This document covers the design process behind creating this simulator, and the details of its final implementation. COMP3740 (The course for this project) is an implementation course, and as such the goals of the project are not centred on mathematics and research, but on the resulting application and its design.
1.2 Similar Projects

There have been many projects in the past to create assembler simulators for both educational and professional purposes. Here are two of the many which already exist.

**GNUSim8085 (Ratnakumar (2003))** – This is a cross-platform (Linux and Windows) program designed to assemble to and simulate the Intel® 8085 microprocessor. It is a mature project with its latest code updates committed in 2011. However, for the purpose of Introduction to Computer Systems, the simulator may be too complicated, and does not support Thumb. It also does not have Mac OS X® support, which would be an issue because a large number of students use Mac laptops for development.

**MARS (Sanderson, Vollmar (2013))** – The “MIPS Assembler and Runtime Simulator” is an Integrated Development Environment (IDE) used in the Missouri State University for teaching assembly. It is regularly updated, with the latest update in 2013, and is released under the MIT open-source licence. It is written with Java and Swing much like rPeANUt and Granule. Again, this instruction set and chipset may be too complicated for teaching purposes.

However, there appear to be none for ARM® Thumb. As assemblers are generally simple to implement, making a new simulator just for the purpose of Introduction to Computer Systems is a reasonable project. This allows for the desired target chipset to be supported, and for the program to be specifically tailored toward the way the course will be run.

1.3 Contribution

This project benefits the course convenor for the Introduction for Computer Systems course, currently Eric McCreath, by giving him a simulator for an instruction set which can run on cheap prototype boards. Eric already has boards which are capable of executing Thumb instructions, so development for these will be made simpler by using Granule to write the code first. It also allows the course to be taught how it was previously because the program is similar to rPeANUt.
1.4 Report Structure

The rest of this report is structured as follows. First, the milestones which needed to be reached along the course of the project will be listed. Then the design of the application will be described, to give an overview of how it works. Next, the implementation will be detailed in-depth. After that, there is a description of benchmarks which were used. Finally, there are conclusions and future directions.

2 Project Milestones

The following describes the different milestones for the design and implementation process. These are in roughly the order of which they were implemented, but there was a lot of interleaving of tasks.

2.1 Assembler and Virtual Machine Layout

This task involved laying out the basis for an assembler and virtual machine in Java code. By this stage it did not have any code specific to the ARM® platform. This was the first task because it gave an idea of how everything would fit together.

2.2 GUI

The Graphical User Interface (GUI) was written next, as it gave a framework for visualising the rest of the development. Although the interface was made to be similar to the open-source rPeANUt, the backing code was written entirely from scratch to keep the code base consistent, and add more extensibility for later improvements.

2.3 Finding and Reading Documentation

The original plan for the project was to make a simulator for an ARM® Thumb-2 based chipset, but this turned out to be far too complicated as it consists of important 16-bit and 32-bit instructions, and complicated CPU features to go with them. The goal instead was to implement part of the Thumb ISA. As the Thumb-2 ISA is a
superset of the Thumb ISA, this turned out to be a better alternative. However, the documentation for the ISA and the binary encoding are purposely made to be largely separate.

Granule uses only 16-bit ARM® Thumb instructions because they are simple, but extensive. Thumb-based processors have a total of 16 registers:

- **R0-R7** are the “low” registers, which can be referenced by instructions that have a 3-bit register field.
- **R8-R15** are the “high” registers, which can be also referenced by instructions with a 4-bit register field.

There is also something called the APSR (Application Program Status Register), which contains four flags which are set by several instructions to be used for conditional branching (jumping between locations in the program). These are the flags:

- **N** is set if the result of an instruction was negative.
- **Z** is set if the result of an instruction was zero.
- **C** is set if an instruction required a carry. For example if you add two binary numbers 1 and 1, this will result in 10, which required a bit to be carried.
- **V** if an instruction causes an overflow.

The **CMP** (Compare) instruction takes two numbers and sets these flags according to the difference between them. They can then be used to check a certain if a condition is met. For example, if after running `CMP R0, R1` the **Z** flag is set, then the two numbers are equal.

For the implementation of this project, the main reference was the ARM® v7-M Architecture Reference Manual (ARM (2010)). This describes in great detail each instruction’s syntax, its binary encoding, and the operation it performs. However, it purposely does not describe specifics of the compiler, such as the formatting of labels and comments. For these, a style similar to the GCC ARM® assembler was chosen (TIGCC (2003)).

A label can start with a dot and must end with a colon, like this:

`.F0:`

And a comment starts with an “@” symbol, like this:

`add r0, r1, r2 @ This is a comment`

There are also assembler directives, which define specific control mechanisms during compile time. These have the same syntax as a normal instruction, except the name begins with a dot. For example, loading an ASCII string into a program at location 0x123 looks like this:
Another reference document used for this project was the Thumb® 16-bit Instruction Set Quick Reference Card (ARM (2007)). This defined in simple terms what each 16-bit instruction was, what inputs it could take, and what action it performed.

2.4 Assembler

Writing the assembler was a large part of the project. It involved writing a tokeniser for the assembly, then parsing the tokens into commands, then converting the commands into their binary representation. Further description on this process is described in the Assembler (4.2) section of this report.

2.5 Virtual Machine and Memory

Writing the virtual machine was also a large part of the project, which was too complicated to be completed in the time given for the project as there are a lot of instructions to implement. However, it does have support for some simple instructions, and the main focus was on the design of the machine. It also handles program running and device memory mapping. More details on this can be found in the Virtual Machine (4.3) section of this report.

Along with the machine there needed to be memory, which was a much simpler task because it only needed to support storing and loading words.

2.6 Virtual Devices

The virtual devices needed to be made so that there could be some interaction with the programs written with Granule. These memory-mapped devices can be read-from and written-to by a program as if it were part of the machine’s native memory.

The devices were implemented using a single universal Java interface, so that in the future more devices can be easily implemented.
3 Design

3.1 Architecture

The instruction architecture chosen for the simulator was the ARM® Thumb instruction set. The reason for this is that it is a subset of the Thumb-2 instruction set (Or more correctly, Thumb-2 is a superset of the Thumb instruction set).

Since Thumb instructions can run on Thumb-2 processors, it is suitable for use in the Cortex-M0 chip, which was chosen by Eric McCreath as the most suitable for teaching a Computer Systems course.

3.2 Language Specifics

Java was the chosen language for the development of this project. The main advantage for this choice is that it is mostly cross-platform, so in a class of students using Windows/Mac/Linux etc. there wont be any issues getting it to work, as long as they have the JRE (Java Runtime Environment) installed.

For the GUI, the Swing toolkit was used for a similar reason: it comes with the normal JRE installation. This is a commonly used GUI toolkit, but is becoming redundant as newer GUI libraries are developed like JavaFX (Oracle (2013)). However, JavaFX is not factory supplied, so Swing is still the best option.

3.3 Version Control

Mercurial was chosen for version control. The repository was kept on a personal server because it also contained the project report.

After this project has been submitted, the code base will be loaded into a Google® Code repository and released as open source. This then allows Eric McCreath and me to continue development after the project is over so that it will eventually be in a sufficient state for teaching Computer Systems.
3.4 Project Licence

The “zlib” (Gailly and Adler (2013)) licence was chosen because of its simplicity and permissibility. Unlike the GNU GPL licence used by rPeANUt, it allows for use in other projects with almost any licence, as long as it keeps the licence and the original author’s name intact. Here is the full text of the licence:

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

3.5 Code

All of the code for this project was designed with flexibility and maintainability in mind. It was written in a modular fashion which makes code easier to read and understand.

The Machine class uses a structure of switch statements which represent the order in which they are described in the ARM® v7-M Architecture Reference Manual (ARM (2010)). This means that the code is fit for implementing more of the instructions described in the manual without needing to reorganise code. It also defines load() and store() functions which first check for device mappings before forwarding the request to the Memory class’s equivalent methods.

To make new memory-mapped devices easy to include in the program, a dedicated device interface was made. This completely separates the development of new devices from the rest of the program, aside from one line after a new Machine is created which “plugs” the device in. At the same time, doing this made debugging much
simpler because a change of one line can move, add or remove entirely any device from the running program.

The ARM® Thumb assembler was written in such a way that allows the code to be entered in roughly the same syntax it is written in the reference manual, with the occasional minor modification. If there is an issue with the compilation process, it will usually be in the underlying parser for the instruction definitions, rather than the definitions themselves. More information on this can be found under the Assembler (4.2) section of this report.

4 Implementation

This section describes in detail the implementation of each part of the Granule program. In most cases this includes a description of each class’ purpose, and a class diagram to clearly show the relations between them.

4.1 GUI and Terminal Interface

The GUI is the Graphical User Interface, which is completely separate from the Virtual Machine. This works by using Java interfaces to make “listeners”, which are used to send update requests to an attached interface. This gives a robust way of allowing classes such as the Terminal device to switch between GUI interactions and raw terminal output, hence the “Terminal Interface”. It also has the advantage of uncluttering the code, and removing redundant object allocations.

As shown in Figure 1, the GUI consists of the following classes in the main package:

MainWindow This describes the interface for the main Granule window. It contains the menu bar, the status bar, the editor, and the machine view. It also defines methods which perform important operations such as sending the code to the assembler when the Assemble button is clicked. It is also responsible for asking the user if they would like to save the program before closing the window, or loading a new program.
Granule  This class contains the main function for Granule, which is responsible for initialising the Machine, inserting devices, and deciding whether to create and attach the GUI or just run a specified file on the command line once through with its output redirecting to the terminal.

It also defines globally accessible static fields such as the program name, its version, and a Preferences object which keeps stored values persistently even after the program is terminated and restarted.

There is a package named "images" in the project which contains images rather than code, so that they are automatically stored in the project’s .jar file. The Granule class supplies a helper function getImageRes() for loading these files and returning them as a BufferedImage.

Status  This allows global messages to be printed to the console, or if there is a listener added using the Status.addListener method, the messages can be intercepted by the GUI and displayed in the status bar.

Console  This is a simple Swing window which listens to the Status class and prints messages to a text box.

Benchmark  This allows for simple and accurate timing of named tasks. Its contents are entirely static so that it can be called globally throughout the program. It works by calling begin(String name) with the name of a benchmark, then end(String name) with the same name. It then prints out the name of the benchmark, along with the number of seconds between the start and end. It was used to get the values used in the Benchmarks (5) section near the end of this document.

SpinnerDialog  This is a custom Swing dialog which makes a nice interface for changing a number and applying its effect in real-time. It is used to change the font size of the editor.

FileUtils  This defines useful functions for dealing with files to reduce the amount of code required to perform simple tasks.
Figure 1: Diagram of all classes in the main package
As part of this project, a custom editor widget was developed. This editor allows syntax highlighting, change-sets to efficiently keep track of undo and redo events, and advanced features such as indentation and unindentation on several lines at once.

Since the code editing widget for Granule is so complicated, it has its own package in the project. As shown in Figure 2, it consists of the following classes in the editor package:

**CodeEditor**  This is the JComponent for the editor which renders the highlighted text and the line numbers. It also sends keypress events from the EditDoc and resizes the scroll area when the document changes.

**EditorPanel**  Since the CodeEditor is a JScrollPane, the drawing and most of the event handling code is actually defined in this subclass that extends JPanel, which is contained and scrolled by the CodeEditor.

**EditDoc**  This is a class made specifically for the CodeEditor but separates all the GUI and raw event handling. It tracks changes on a line-by-line basis and can revert and apply them to make undo and redo possible. It also keeps track of the cursor position, whether there is a text selection taking place, and where the selection started.

**Change**  This class describes a single change to the EditDoc. It must have one of the following modes:
- REPLACE: This replaces a line of the document with the specified text.
- DELETE: This removes a line from the document.
- INSERT: This inserts the specified text as a line in the document.
- APPEND: This adds the specified text as lines at the end of the document.
- CARET: This changes the position of the cursor (and the selection-starting cursor).

**Line**  This defines a single line of the EditDoc. It contains some helper methods, and caches the syntax highlighting and what literal editor column each character in the line is in (Because tab characters take up more than one column).

**Highlight**  This defines a bit of text and its colour for the purpose of syntax highlighting.
SyntaxRender  This class gets each line of the editing document and colours it using a simple algorithm. For example, if it sees the "@" character (The comment character), it will get the rest of the line and colour it grey. You can see the results of the highlighting in Figure 3 (Note: This document must in colour for the highlighting to be noticeable).

The changes defined by the Change class are tracked in groups. For example, when the user presses a key to insert a character where the cursor is, the following code is run:

```java
public void addChar(char c) {
    // Delete any text that might be selected:
    deleteSelection();
    int li = getCaretLine(); // Get the cursor’s line
    int cc = getCaretCol(); // Get the cursor’s column
    // Get the cursor line’s text:
    String l = getLine(li).getText();
    Change change = new Change(Change.Type.REPLACE, li);
    // Set the change’s replacement text:
    change.setText(l.substring(0, cc) + c + l.substring(cc));
    apply(change); // Apply the change
    // Make a new Change to change the cursor position (Pass
    // in the EditDoc so that the change can record the
    // current cursor position):
    change = new Change(Change.Type.CARET, this);
    // Set the new cursor column:
    change.setCaretCol(cc + 1);
    apply(change); // Apply the change
    // If the character typed is not in the specified range,
    // then commit the current change - set as an undo event:
    if (!((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
        || (c >= '0' && c <= '9'))) {
        commit();
    }
}
```
Figure 2: Diagram of all classes in the editor package
Figure 3 shows the layout of Granule’s main window. On the left is the **CodeEditor** widget as described previously. When the Assemble button near the top left of the window is clicked, the machine is reset and the code is assembled directly into its memory.

On the right is the **MachineView**, which has control buttons at the top, a **MemoryView** on the left, and **RegisterViews** on the right. The view in the bottom right, which reads “Hello World!” is separate from the **MachineView** because it is a **TerminalView**, which visualises a **Terminal** device which may be plugged into the **Machine**. If there is no terminal to display, the **TerminalView** will not be added to the **MainWindow**.

The **MemoryView** is a table in which each row contains a break-point flag (Represented by a red dot), the location in memory the row refers to, the value at that location, and a label which may have been inserted by the assembler to help the user track what is happening relative to the code.

![Screenshot of Granule’s main window](image-url)
4.2 Assembler

As shown in Figure 4, the assembler consists of the following classes in the `compile` package:

**Scan** This class is used to scan the tokens from a given file.

**Token** This class stores the information about a Token. It must have one of the types from its `Type` enumeration (enum). Several of these tokens are later parsed into assembly commands.

**Compile** This class will read `Token`s from the `Scan` class, then turn them into machine code.

**InstDef** This describes the definition of an instruction, parsed from a hard-coded string format. This format is described below.

- **CodePart** This describes a single part of an `InstDef`'s bitcode.

- **Arg** This describes an argument in an `InstDef`.

**Cmd** This describes a single instruction of ARM® Thumb assembly code.

- **Arg** This is a single argument that has been passed into a `Cmd`.

**Context** This keeps track of information relating to a single file’s compilation, such as labels and current memory pointer to load new instructions. There are also methods provided for loading instructions into memory.

The assembler was written using a list of `addInst(String inst, String code)` calls. In this, `inst` is the definition string copied directly from the reference manual. The `code` is in a format designed to easily enter in the bitwise representation of each instruction.

One of the reasons for writing the assembler this way is that it makes debugging much simpler, and the code neater. As you can see in the listing under Benchmarks (5) “Compiler.addInstructions()”, the time required to run this operation is negligible, and is only a small fraction of the time required to configure the GUI (MainWindow). Also, because the text is hard-coded, it will be interpreted exactly the same way every time. This means that the parser does not need much error checking, as there
is a fixed set of data to parse every time.

An interesting observation about the construction time of the assembler is that after running several times, the time running `addInstructions()` reduces from about 0.06 seconds to about 0.02 seconds. This is because of the Java Virtual Machine (JVM)’s built in Just In Time (JIT) compiler, which attempts to compile repeatedly running byte code to native machine code.

A Thumb instruction definition looks like this:

```
STR<c> <Rt>,[<Rn>,#<imm5>]
```

Reading from left to right, this says that:

- The instruction name is `STR`
- It is conditional, ie. can also be compiled from `STREQ`, `STRLT` etc.
- Its first argument must be a register, named `Rt`
- The last two arguments must be in square brackets: `[]`
- The second argument must be a register named `Rn`
- The last argument is optional, named `imm5`, and must be an immediate value (one beginning with a “#”)

This is used to do pattern matching with input instructions to check if the number of arguments match, if the argument types are the same, and if any number passed in as an argument can fit in the number of bits in the instruction’s output. The bitwise output definition is a comma separated list that looks like this:

```
b011,b0,b0,imm5,v3 Rn,v3 Rt
```

Any part which starts with a “b” represents the binary bits which go at that location in the bitcode. Any part that starts with a “v” represents a variable at that location. The first number after the “v” is the number of bits the variable translates to, and the second value is the variable name, respective to the instruction definition string.

Every instruction is defined like this in the `Compile.addInstructions()` method, and are parsed when Granule first starts.
Figure 4: Diagram of all classes in the compile package
To compile a string of assembly code, a Context must first be instantiated. Here is an example of how the MainWindow class compiles the code from the editor widget:

```java
// Get the editor's text
String text = editor.getText();
// Make a new compiler (This really only happens once when
// Granule starts)
Compiler comp = new Compiler();
// Create a scanner for the code string
Scan scan = new Scan(text);
// Create a new context based on the machine
Context cxt = new Context(machine);
// Compile the text from the scanner into the context
comp.run(scan, cxt);
```

### 4.3 Virtual Machine

The Virtual Machine is Granule’s ARM® processor simulator. This takes instructions and parses them bit-by-bit to find the exact operation to execute, then runs it. The GUI interface for this can be found on the right side of the main window (Figure 3).

As shown in Figure 5, the virtual machine consists of the following classes in the machine package:

- **Machine** This simulates ARM® Thumb instructions. The implementation is essentially a giant switch statement which selects bits and jumps to the appropriate code. This is basically how a real CPU works, so it is acceptable. It also allows the Java compiler to easily optimise the code into a set of jump-tables, which are very efficient.

- **MachineListener** This is the listener interface used to update the MachineView if it is attached.

- **MachineView** This is the GUI used to display the Machine’s state and controls. It consists of Play/Step/Stop buttons, a set of RegisterViews, and a MemoryView.

- **Memory** This keeps track of information about the virtual machine memory. It also allows for listeners to track modifications.
Cell

This describes a single cell of memory, and any extra details used in the simulator. This includes the cell’s value, how many times it has been accessed (hits), if it is a breakpoint (bp), and an optional label string (label) which is inserted by the assembler. All this information is displayed by MemoryView.

MemoryListener

This is the listener interface used to update the MemoryView if it is attached.

MemoryView

This is the GUI used in Granule to display the current Memory state. It consists of a table, which only displays a cell’s value and location if it has been modified by a running program. It also allows the user to set locations as breakpoints by clicking in the first column.

RegisterView

This is the GUI text-box used to display a single register’s value.

Device

This is the base class for a description of an I/O device which memory-maps onto the virtual Machine.

There are two helper functions in Machine defined as follows:

```java
// Grab a bit from n
private int b(int n, int bit) {
    return (n >> bit) & 1;
}

// Grab bits from n
private int b(int n, int from, int len) {
    return (n >> from) & ~(0xffffffff << len);
}
```

These are used to select bits from the Instruction Register (IR) and jump to the relevant code. For example:

```java
switch (b(IR, 14, 2)) {
    case 0b00: // So far: 00
        switch (b(IR, 11, 3)) {
            case 0b011: // So far: 00 011
                switch (b(IR, 9, 2)) {
                }
        }
    }
```
case 0b00: // So far: 00 011 00
    // ADD Register definition
    rd = b(IR, 0, 3);
    rn = b(IR, 3, 3);
    rm = b(IR, 6, 3);
    ...
    break;
}
4.4 Virtual Devices

Virtual Devices are I/O-based devices which map onto memory. These were designed to be dynamic, so that any device can be mapped onto anywhere in memory with one line of code that looks like this:

```java
machine.addDevice(0xFF, new Terminal(machine));
```

Where `machine` is an instance of `Machine`. This maps a `Terminal` device to location 0xFF in memory, which means that when an instruction wishes to read from or write to location 0xFF, it will instead be redirected to the terminal device.

A Virtual Device may also have a size, so for example to create a screen device, it can map each pixel to its own memory location as an offset from a starting point. It also defines two methods: `send(int offset, int value)` and `recv(int offset)`. Both of these methods take an offset value, which represents the offset from the smallest address the device is mapped onto.

Here is the implementation for the `Terminal` device in Granule, which maps onto one word of memory and can do read and write operations to a terminal:

```java
// Device which prints out characters written to the allocated memory location.
public class Terminal extends Device {
    // Listener for making callbacks to the GUI
    public static interface TerminalListener {
        public void out(char c);
        public void reset();
    }

    private TerminalListener listener;

    // Buffer to keep track of key presses made by the GUI
    private ArrayList<Integer> keyBuffer;

    public Terminal(Machine mach) {
        super(mach, 1);
        name = "Terminal";
        keyBuffer = new ArrayList<Integer>();
    }

    // Get the character width of the terminal
    public int getWidth() {
        return 40;
    }
```
public int getHeight() {
    return 5;
}

// Set the listener for the GUI to connect to
public void setListener(TerminalListener listener) {
    this.listener = listener;
}

// Write a character
public void send(int loc, int val) {
    if (listener == null) {
        // Write character to the real terminal
        System.out.print((char)val);
    } else {
        // Send character to the GUI
        listener.out((char)val);
    }
}

// Get a character code from a key press, or return 0
// when no key is pressed
public int recv(int loc) {
    if (listener == null) {
        try {
            // Get a character from the real terminal, or
            // a file which may be piped in
            return System.in.read();
        } catch (IOException e) {
            // Fall back to no key press
            return 0;
        }
    } else {
        // Check the key buffer to see if there were any
        // key presses
        if (keyBuffer.isEmpty()) {
            // There were no key presses
            return 0;
        } else {
            // Pop the oldest key press from the key
            // buffer
            return keyBuffer.remove(0);
        }
    }
}
This terminal device, like rPeANUt, maps one memory address to be the input and output of the device. Any instruction which writes a character code to that address in memory will print the corresponding character to either the terminal, or if there is a GUI attached (TerminalView), it will send it through a listener and get appended onto the contents of a text box. It stores key presses from a GUI in a buffer, and returns them one-by-one every time an instruction reads from the mapped memory location.

5 Benchmarks

This section contains a table of benchmarks used to test the usability of different parts of the program. Since this is an application with a user interface, speed is important. This was done with a simple class called Benchmark.java (7.2), which can start and stop a timer from anywhere in the program and prints out the time difference in seconds.

When unsure about a specific method and its efficiency, the Benchmark class was used to check running time against other factors. For example, if the program Swing interface itself takes a second to start, then a 5% increase on that time from the assembler initialisation is acceptable.
Below are two identical “Hello World!” programs for Granule and rPeANUUt. They both run a total of 7399 instructions, and print the string "Hello World!\n" on the screen 255 times. The time required to start the JVM, compile and run these programs using the terminal mode of both programs is recorded in the benchmarks table above under “Time to run program”.

As you can see, the time required for the rPeANUUt program is almost 30 times longer. Much of what slows down rPeANUUt is that it stores regularly accessed values such as registers directly in a Swing JTextField widget, which means that a lot of string manipulations and drawing actions may take place, even when the GUI isn’t running. Granule solves this by having separate GUI and data models, and listeners to send updates between them. If there is no GUI attached to a listener, then no action will be taken. RPeANUUt also uses a long chain of if-statements to parse machine instructions, which would be far slower than Granule’s method of using switch-statements which jump directly to the next nested switch-statement or instruction definition, without needing to check intermediate values first.

See next page for benchmarking code
Granule code

0x100:
  The terminal is mapped onto 0x0F0
  movs R0, #0xF0
  movs R3, #1
  movs R5, #255

print:
  add R3, R4, R4
  movs R1, #’H’
  str R1, [R0]
  movs R1, #’e’
  str R1, [R0]
  movs R1, #’l’
  str R1, [R0]
  movs R1, #’l’
  str R1, [R0]
  movs R1, #’o’
  str R1, [R0]
  movs R1, #’ ’
  str R1, [R0]
  movs R1, #’W’
  str R1, [R0]
  movs R1, #’o’
  str R1, [R0]
  movs R1, #’r’
  str R1, [R0]
  movs R1, #’l’
  str R1, [R0]
  movs R1, #’d’
  str R1, [R0]
  movs R1, #’!’
  str R1, [R0]
  movs R1, #’\n’
  str R1, [R0]
  cmp r4, r5
  blt print
  bkpt #0

rPeANUt code

0x100:
  The terminal is mapped onto 0xFFF0
  load #0xFFF0 R0
  load #1 R3
  load #255 R5

print:
  add R3 R4 R4
  load #’H’ R1
  store R1 R0
  load #’e’ R1
  store R1 R0
  load #’l’ R1
  store R1 R0
  load #’l’ R1
  store R1 R0
  load #’o’ R1
  store R1 R0
  load #’ ’ R1
  store R1 R0
  load #’W’ R1
  store R1 R0
  load #’o’ R1
  store R1 R0
  load #’r’ R1
  store R1 R0
  load #’l’ R1
  store R1 R0
  load #’d’ R1
  store R1 R0
  load #’!’ R1
  store R1 R0
  load #’\n’ R1
  store R1 R0
  ; Loop 255 times
  sub R4 R5 R6
  jumpn R6 print
  halt
6 Conclusion and Future Directions

Granule was written to be used in the Introduction to Computer Systems course at The Australian National University as a replacement for the rPeANUt simulator. It was designed to have a simple user interface and a clean, modular implementation. These goals were fulfilled by the means of a Java application. It also had the advantage of running considerably faster than rPeANUt.

There will be further development required after the project’s submission. This became apparent after researching the ARM® Thumb documentation and finding that it is far too complicated to be completed in the time frame of the project, which is why Granule was written to be extensible.

Future developments will include making it download the compiled code onto a real ARM® Thumb based chipset and having it run. The Virtual Machine also currently only implements a subset of the instructions which are supported be the Assembler, so there will also need to be more work in this area. However, this is just the task of reimplementing the large quantity of pseudo-code from the reference manual into Java code.

There was also the possibility discussed with Eric McCreath to use all of the features and optimisations provided by Granule to improve rPeANUt, which would be entirely possible with a few modifications to the assembler and remaking the `exec()` function in the virtual machine so that it runs rPeANUt instructions instead.
References

ARM 2007, ARM, viewed 1 October 2013,
\[
\text{http://infocenter.arm.com/help/topic/com.arm.doc.qrc0006e/index.html}
\]

ARM 2010, ARM, viewed 27 August 2013,
\[
\text{http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403c/index.html}
\]

Jason Ernst 2010, Why Learning Assembly is Still Important for CS Undergrads,
Jason’s Computer Science Blog, viewed 26 October 2013,
\[
\text{http://www.jasonernst.com/2009/12/02/}
\quad\text{why-learning-assembly-is-still-important-for-cs-undergrads/}
\]

Jean-Loup Gailly and Mark Adler 2013, Zlib, viewed 11 August 2013,
\[
\text{http://zlib.net/zlib_license.html}
\]

Oracle 2013, Oracle, California, United States, viewed 28 October 2013,
\[
\text{http://www.oracle.com/us/corporate/contact/index.html}
\]

Pete Sanderson and Kenneth Vollmar 2013, Missouri State University, Missouri,
United States, viewed 30 October 2013,
\[
\text{http://courses.missouristate.edu/kenvollmar/mars/}
\]

Sridhar Ratnakumar 2003, Sridhar Ratnakumar, viewed 15 October 2013,
\[
\text{http://gnusim8085.org}
\]

TIGCC Team 2003, TIGCC Team, viewed 6 October 2013,
\[
\text{http://tigcc.ticalc.org/doc/gnuasm.html}
\]
7 Appendix

7.1 README

This is the full text of Granule’s README file.

Granule ARM(R) Thumb Simulator by Joshua Worth
----------------------------------------------

This program provides a friendly interface for writing assembly with the ARM Thumb instruction set, and compiling and simulating it.

Running
------

The current folder is an Eclipse project, so you should get Eclipse (eclipse.org), then import the project into your workspace. At this point you can either export the project into a Runnable JAR file, or just press play.

Supported Instructions
----------------------

Although many instructions can be compiled, the simulator only currently supports a subset of these. Here is a list:

ADD (registers)
ADD (SP plus immediate)
AND (register)
B (conditional, immediate)
B (conditional, register)
BKPT (This is used as a way to stop the simulation)
CMP (immediate)
CMP (registers)
MOV (immediate)
MVN (register)
LDR (immediate)
LSL (immediate)
LSR (immediate)
ORR (register)
STR (immediate)
SUB (SP minus immediate)

Device Mappings
---------------

There are devices in Granule which are mapped onto certain memory locations. These mappings are displayed when Granule is first started, and the interfaces for them are in the bottom right of the window. Here is a description of the devices currently included in Granule:

Terminal - This device maps onto one word of memory. It allows for printing characters by writing to its mapped location, and reading key presses by reading from it.

Screen - This device maps bits of memory to corresponding screen pixels. 1 means white, 0 means black. Simply write the mapped memory locations to update the screen pixels. Reading is also supported.

Editor
------

Granule comes with a syntax-highlighting editor. It also allows for block indentating and unindenting, as well as line duplication and moving. Here are the shortcuts for the editor:

Ctrl+A: Select all
Ctrl+X: Cut
Ctrl+C: Copy
Ctrl+V: Paste
Tab: Indent selected or current line
Shift+Tab: Unindent selected or current line
Alt+Up: Move line up
Alt+Down: Move line down
Ctrl+Alt+Up: Duplicate line up
Ctrl+Alt+Up: Duplicate line down
Double click: Select word
Triple click: Select line

Simulator Interface
-------------------
The simulator is displayed and controlled by the interface in the right of the main window. The table displays parts of memory which have been modified, everything else can be assumed to be zero. To the right of that are the current register values, and below them are the current APSR (Application Program Status Register) flags. These turn a colour other than white when they are set.

Console
-------

Messages that appear in Granule’s status bar are also printed to the console. This can be opened by either the Code menu, or by pressing F3. It may also show messages you didn’t see because they were replaced too quickly. For example, if you pass incorrect arguments to an instruction, it will show valid options.

Command-line options
---------------------

Granule can simulate programs directly in the terminal without the need for a GUI. Assuming you have exported the Granule to a JAR file, to do this simply type:

java -jar granule.jar program_to_run.s

The full usage is as follows:

Usage: granule OPTIONS [filename]

OPTIONS can be:
--help  Print out this message.
--count Print out the total count of instructions run after the program halts.
--check Test if the specified file compiles properly.
--bench Print out timings for various parts of Granule.

Licence
-------

See file LICENCE
7.2 Benchmark.java

```java
public class Benchmark {
    // Allows all benchmarks to be enabled and disabled at once
    private static final boolean ENABLED = true;

    // Keeps track of the start time of each benchmark
    private static HashMap<String, Long> times = new HashMap<String, Long>();

    // Begins a benchmark
    public static void begin(String name) {
        if (ENABLED) {
            System.out.println("Benchmark: Begun " + name);
            times.put(name, System.nanoTime());
        }
    }

    // Ends a benchmark, and prints out the result
    public static void end(String name) {
        if (ENABLED) {
            Long time = System.nanoTime();
            Long benchtime = times.get(name);
            if (time != null) {
                Long off = time - benchtime;
                Double seconds = (double)off / 1000000000L;
                System.out.println("Benchmark: Ended " + name + " with time " + seconds);
            } else {
                System.out.println("Benchmark: No existing benchmark " + name);
            }
        }
    }
}
```
7.3 Detailed Class Diagrams

Figure 6: Detailed diagram of all classes in the `compile` package
Figure 7: Detailed diagram of all classes in the editor package
Figure 8: Detailed diagram of all classes in the `machine` package
Figure 9: Detailed diagram of all classes in the main package