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

School of Computer Science and Statistics

High-Level Tests of Assembly Language Programs using PyTest

Cian Mawhinney

Supervisor: Dr. Jonathan Dukes

April 15, 2024

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MCS Computer Science

Declaration

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

I consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

I agree that this thesis will not be publicly available, but will be available to TCD staff and students in the University's open access institutional repository on the Trinity domain only, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Signed: _____ Date: _____

Abstract

Within the education sector, unit testing is commonly used to assess the correctness of programs, often to automatically grade students assignments and also offer rapid feedback to students. Within the context of teaching ARM Assembly, a low-level language, this is no different, though the tooling to test Assembly programs can often be cumbersome when writing complex tests.

This project addresses this issue by allowing tests to be written for ARM Assembly programs in a higher level language, in this case, Python. This involves designing and implementing a test harness which is suitable for use by novice programmers in an educational setting. In order to implement the test harness, an intermediary layer was built that provides an abstraction around an emulator so that the ARM Assembly programs could be executed on common hardware.

Being built for students and novice programmers, the test harness exposes intuitive APIs for invoking subroutines and supports interacting with registers and memory. As the goal of unit testing is to uncover and fix bugs in the code being tested, care was taken to ensure that any error messages that arise during execution are clear, readable and actionable.

Finally, a code coverage tool has also been developed to help assess the quality of a test suite.

Acknowledgements

I would like to thank my supervisor Dr. Jonathan Dukes for his invaluable help and guidance on this project. Without his support this project would not have been possible, and I'm grateful for all his constructive advice and feedback that has gone into improving my work.

I'd also like to thank my friends and family, who have supported and encouraged me not only during this project but throughout my degree.

Contents

List of Figures

List of Tables

Nomenclature

AAPCS	Procedure Call Standard for Arm Architecture
ABI	Application Binary Interface
ARM	Advanced RISC Machine
Link Register	Register used to store the address where execution should branch back to after a subroutine finishes
Program Counter	Register used to store the address of current point of execution

1 Introduction

1.1 Context

When learning to program, students want fast feedback on programs they write, so they can iterate quickly and improve their solutions. To enable this in an educational setting, unit testing and auto-grading are commonly used to automate this rapid feedback process. As well as allowing this rapid feedback for students, potentially instructors also benefit from unit testing, as assignments can be graded more easily.

Within Trinity College Dublin, first year computer science students are taught ARM Assembly language in the Introduction to Computing modules [1]. For assembly languages, as they are low-level languages, there are no compiler-enforced structures in place within the code. This can often pose a challenge to unit testing frameworks, which will typically rely on a particular structure for a "unit".

1.2 Motivation

For high-level languages, unit tests can be written using existing frameworks, allowing instructors to quickly write a comprehensive suite of tests that students can use to check their solution against. However, for Assembly languages, there are very few of these tools publicly available. At present, existing solutions can involve writing tests in Assembly itself, which can be cumbersome for more complex tests. Other solutions such as using a GDB (GNU Debugger) script to manipulate an emulator tend to be inflexible, requiring a large amount of boilerplate code to initialise a test scenario. Within Trinity College Dublin, the current solution is based on using a custom YAML file to declare input and output values for tests. In this case, custom validator functions must be written for each required complex data type. For example, when dealing with sets the order of the elements does not matter, whereas for a list of elements the order likely would, depending on the exact scenario.

Therefore, the main aim of the project is to investigate how best to write tests for ARM Assembly language programs in a higher-level language, then design and implement a test harness for these programs.

Figure 1.1 shows a GDB script for controlling QEMU's execution of a test procedure written in ARM assembly. This was previously used in Trinity College Dublin's Introduction to Computing module. The subroutine being tested calculates the intersection between two sets of numbers.

```
advance Main
set *(signed int [11] *) 0x08080000 = {10, 299, 6, 342, 12, 0,
↪  -100, 22, 88, -5, 50}
set *(signed int [7] *) 0x08080100 = {6, 342, -5, 6, 81, -200,
↪  7}
set $r0 = 0x20000000
set $r1 = 0x08080000
set $r2 = 0x08080100
advance Eval_Test
set logging on
x/128dw 0x20000000
set logging off
quit
```

Figure 1.1: A GDB script which writes data into memory and registers, then executes instructions until it reaches the label Eval_Test.

By contrast, Figure 1.2 shows a proposed syntax for how this project might achieve interact with the same program being tested in a much more intuitive way. Note that this syntax is for illustration purposes only.

1.3 Goals and Methodology

The main objective of this project is to allow testing of students' ARM Assembly assignments using a test suite written in a high level language. To do this, research and a review of the existing literature will be conducted. Following this, a proof-of-concept unit testing framework will be designed which is suitable for use within an educational setting. This framework will be implemented to demonstrate its effectiveness and validate the design decisions made. Lastly, a qualitative evaluation of the framework which has been implemented will be completed.

1.4 Contribution

The largest contribution made by this project is the design and implementation of a test harness for ARM Assembly language programs, written in Python. This required designing an intuitive API for interacting with an intermediary layer that emulates the ARM

```
import pytest
from constants import SlotLocation
from decorators import input_map
from machine import Machine

@pytest.fixture
def machine():
    return Machine("firmware.elf")

custom_argument_ordering = [SlotLocation.R1, SlotLocation.R2]
@input_map(custom_argument_ordering)
def test_intersection(machine):
    set_1 = {299, 6, 342, 12, 0, -100, 22, 88, -5, 50}
    set_2 = {342, -5, 6, 81, -200, 7}
    assert machine.Main(set_1, set_2) ==
    ↪  set_1.intersection(set_2)
```

Figure 1.2: A proposed Python syntax for the same subroutine as Figure 1.1. For illustration purposes only.

architecture. During the implementation of this test harness a new, streamlined API for comparing the contents of an emulator's memory with a Python object was developed. This reduces the amount of boilerplate code necessary in the unit test itself. Finally, a new tool for assessing the quality of the test suite has been contributed, in the form of generating a coverage analysis report.

1.5 Outline

This report covers the development and implementation of a library designed to allow the unit testing of ARM Assembly Language programs from Python, a much higher-level language. Within the background chapter, an overview of the technologies and existing work that surround the project are discussed. In the Design and Implementation chapter, the design decisions and and the implementation of this project will be discussed, covering the project's requirements, design goals and features. Finally, the project will be evaluated in the last two chapters, based on the project's requirements and other prior work in the field.

2 Background

2.1 Unit Testing

Unit testing is an approach for testing software where small units of code are tested individually to ensure they exhibit the correct behaviour. While the definition of what constitutes a unit can differ between projects, a function or subroutine is commonly chosen. To be rigorous in verifying the functionality of the unit, test suites are constructed made up of various test cases covering different scenarios.

What makes unit testing unique when compared to other testing methods is its focus on testing units without any dependence on other components. This allows the programmer to have a fine-grained view of where the error might have originated. In contrast, integration testing, by definition, tests multiple components together. This is typically done to test the interactions between a subset of an application's components or services. End-to-end testing is a form of test that tests the application as a whole, looking at the system from the point of view of the user. Since all components are tested together at the same time, this type of testing is at the other end of the spectrum to unit testing.

2.1.1 Industry

Unit testing is commonplace within industry, primarily being used to verify software is operating correctly, and will stay correct in the future after it is updated and maintained. As unit tests are a fast way to assess whether the application is functioning as intended, it gives the programmer confidence when working in the codebase, allowing more rapid progress when working on new features or when refactoring code.

2.1.2 Education

Within the education sector, unit testing is still commonly used to assess the correctness of programs, though the purpose differs slightly from its use in industry. Here, the primary role of unit testing tends to focus on how it can be used to automatically grade student submissions for assignments. This is particularly useful for large classes with many

assignments, as instructors can spend less time manually grading programming assignments. [2]

A further benefit is that when students have the opportunity to run their code against a test suite, Edwards found that students scores increased and the number of defects decreased. [3]

Various software systems and frameworks to facilitate auto-grading have been developed for use in universities and colleges. These allow students to upload their submissions and have their programs graded against test suites written by the instructors.

Some common auto-grading systems include:
Web-CAT: is one of the most common auto-grading systems, designed to encourage students to write their own tests so they can better understand how their program functions. One of the key features of the system is that students can be graded on how well they write their own tests, as well as how correct the program is according to a test suite provided by the instructor. [4]

Submitty: is a more modern open source auto-grading system compared to Web-CAT. Originally developed for the Rensselaer Polytechnic Institute, it supports a variety of programming languages as well as some more advanced features such as code coverage and testing networked applications. [5, 6]

Check50: is a custom auto-grading framework developed for Harvard University's CS50: Introduction to Computer Science. This framework is language agnostic as it uses standard input and output to interact with programs, and provides other features such as a Python API for interacting with test cases and support for compiling code. [7]

GitHub Classroom: a hosted solution, builds upon GitHub's offering for Git repositories and uses GitHub Actions, GitHub's CI/CD (Continuous Integration and Continuous Development) system, to execute the tests on the code committed to students' repositories. [8]

Within Trinity College Dublin, modules run by the School of Computer Science and Statistics (SCSS) have made use of Web-CAT and, more recently, Submitty.

2.2 Unit Testing Frameworks

For each programming language, there is usually a variety of testing frameworks to choose from, each with their own approaches and feature set. This section will highlight a few of them to provide context for the design decisions made in the next chapter.

2.2.1 JUnit

JUnit is the most commonly used unit-testing framework for Java. Since its creation in 1997 by Kent Beck and Erich Gamma, [9] it has become very influential for the design of subsequent unit testing frameworks. Borrowing concepts from SUnit, a testing framework designed by Kent Beck for the Smalltalk programming language, JUnit includes features like automatic setup and teardown of test scenarios and tools for asserting that values are what they should be.

2.2.2 Unittest

Unittest is the built-in library for writing unit tests in the Python programming language. [10] It describes itself as being influenced by JUnit, and provides utilities for setting up and tearing down test scenarios before and after running tests.

2.2.3 PyTest

PyTest is a third-party Python unit testing framework that aims to make it easier to write tests for applications. Like the built-in `unittest` library, PyTest also provides utilities for setting up and tearing down test scenarios, although it does so in a declarative manner rather than a procedural one. It does this by introducing new tooling for working with fixtures, which allow the programmer to define how a resource should be constructed, and PyTest will initialise this resource and pass it to the test that needs it automatically. Once the test routine has completed, the library will also handle the cleanup for the resource. A further feature of PyTest is that the results of tests are asserted using Python's built-in assert keyword without any PyTest specific assert* methods.

2.2.4 Cargo Test Runner

Cargo is the standard toolchain for the Rust programming language, including a package manager, build system and test runner. As Cargo is widely used among Rust developers, basic unit-testing capabilities are effectively built into the language.

2.3 Assessing Test Suite Quality

Since a test suite will indirectly influence the quality of the piece of software it tests, it is also necessary to be able to evaluate how good the test suite itself is. This section will provide a brief overview of some common techniques for determining the quality of a test suite.

2.3.1 Coverage Analysis

Coverage Analysis is the practice of measuring how much of a program's source code has been executed as a result of tests being run. The idea being that if sections of the codebase have not been executed by a test, these sections of code are either not necessary for the desired behaviour of the program or the test suite is not fully complete.

To measure code coverage there are a few different approaches including:

Instruction coverage: the percentage of CPU instructions executed by the test suite.

Line coverage: the percentage of source code lines executed by the test suite.

Branch coverage: the percentage of possible code paths executed by the test suite.

While a high test coverage metric is usually seen as a good thing, Daka et. al (2014) found that code coverage was not the only aspect that developers focus on when writing new tests. Instead they found that creating realistic test scenarios was the highest priority of the respondents to their survey. [11]

2.3.2 Mutation Testing

Mutation testing is a type of fault injection, though instead of testing how an application handles external errors, in this case the errors themselves are injected into the application and it is the test suite itself which is being tested. The goal is to quantify how well the test suite would perform when bugs are introduced to the codebase it is testing. To do this, errors in the source code are simulated and run against the test suite. Errors such as, off by 1 errors, incorrect constants, incorrect branch conditions etc. can be introduced for the test suite to identify.

2.4 Low-Level Software Testing

In this section, some of the tools used to test low-level software, particularly software that may not be written for the same architecture as the host machine, will be explored.

2.4.1 QEMU

QEMU, short for Quick Emulator was first developed by Fabrice Bellard in 2003 to execute x86 binaries on non-x86 platforms [12]. Since then, it has had 8 major releases, and grown to be very mature [13]. Having initially been developed to emulate single binaries in a "User Mode", QEMU also supports full system emulation, including BIOS and peripherals. QEMU

supports execution of binaries from a variety of common architectures including x86, ARM, MIPS, RISC-V. [14]

2.4.2 Unicorn Engine

Unicorn Engine is a CPU emulator based on the core of QEMU [15], commonly used for malware analysis and reverse engineering. A key feature of Unicorn is its inclusion of APIs to interact with the emulator, something that QEMU lacks. The main codebase is written in C, though has bindings for a variety of other popular programming languages, including Python, Rust and Java. As it is based on QEMU, Unicorn Engine supports execution of binaries from many architectures including x86, ARM, MIPS, RISC-V.

2.4.3 GDB

GDB (GNU Debugger) [16] is a command-line debugging tool that supports manipulating values in registers and memory, starting and stopping execution, and adding breakpoints to programs.

GDB also supports debugging programs on simulators (e.g. QEMU) or on remote hardware through a gdbserver implementation, which the GDB CLI can then communicate with. For embedded devices or resource constrained devices, another option is to configure GDB to communicate over a JTAG interface. Using a JTAG interface means that the gdbserver does not have to run on the hardware being debugged. [17]

Finally, in addition to a command-line interface, GDB also has a machine interface, GDB/MI, which aims to make it easier to interact with GDB programmatically. [18] Libraries for communicating with GDB using GDB/MI have been written in a variety of programming languages, including Python and Rust.

2.5 Prior Work

There are a few examples of work in similar areas with similar aims to this project. All of the following tools are focused on education, for teaching assembly language programming, and therefore require require various simulators to execute code.

2.5.1 MIPSUnit

MIPSUnit is a testing framework for MIPS Assembly, developed by Zachary Kurmas [19]. It was developed at Grand Valley State University for their Computer Architecture course, where MIPS Assembly is taught.

A key feature of the project is allowing the programmer to use one of two different unit testing frameworks, JUnit for the Java programming language, and RSpec for Ruby. What the author noted, however, was that the RSpec interface would only show one failure at a time, and this behaviour was frustrating to work with. On the other hand, JUnit does not have this problem, and students were subsequently advised to use the JUnit interface.

Once initialised, the source code being tested is executed on the MARS (MIPS Assembler and Runtime Simulator) MIPS Simulator [20]. MIPSUnit provides methods for invoking subroutines, passing optional parameters to be placed in registers. Writing to memory is achieved using methods that dynamically allocate space in the emulator's memory. The methods to read from memory return arrays of values in the size requested.

The above paper describes a system that is most similar to this project in terms of the outcome it desires, though there are some important differences. The largest of which is that MIPSUnit is a framework designed for the MIPS Architecture, whereas this project is focused on the ARM architecture. As a result MARS cannot be used as the emulator for this project.

2.5.2 AOCO

AOCO, short for Automatic Observation and (grade) Calculation for (subroutine) Operations, is a tool for automatic grading of ARM assembly language assignments, developed for the University of Porto's Microprocessors and Personal Computers course. [21]

To specify tests, AOCO uses YAML configuration files, where the type and values of each parameter and input are listed. AOCO supports subroutines that return numeric results, array based results, a combination of the two, or subroutines that return no result. Execution of the tests is achieved by invoking a Python wrapper that builds a C program around the students ARM assembly code. Once compiled, this is then executed using QEMU.

3 Design and Implementation

This chapter covers the design decisions and and the implementation of this project. The project's design goals, proposed architecture and features will be discussed in the following sections.

3.1 Requirements

In this section, the core requirements for this project will be laid out. These requirements will serve as the primary way this project is to be evaluated in Chapter 4.

As outlined previously, the main objective of this project is to design and implement a test harness for ARM assembly programs using a high-level language. Since this project is intended to be used by novice programmers in an educational setting, when design decisions are made, they should take this fact into account. As is often the goal for most unit-testing frameworks, a clear and intuitive API should be provided, and clear feedback should be delivered to the user when errors or exceptions occur.

This project should abstract a lot of the complexity of interacting with low-level concepts. This is to allow the programmer to focus on the desired behaviour of the tests instead. To minimize the amount of cognitive overhead needed when using this project, this should be done in an intuitive manner, following the "Principle of Least Surprise".

The project should deliver on the following features:
Execution of ARM assembly code: assembled ARM machine code must be able to be executed on hardware owned by students. This will require incorporating a mechanism to emulate the ARM architecture processors on x86-based CPUs. Assembling the ARM Assembly Source code into a firmware file is out of scope for this project, and it will be assumed the firmware file already exists.

Support interacting with registers: the simplest way to pass values to a subroutine is to do so using registers. Setting values into registers, and reading any return values back after a subroutine has been executed must be possible in an intuitive manner.

Support interacting with memory: when pieces of data are too large to fit inside

registers they are placed in memory, which therefore should be supported by the project.

Exception Handling: a key part of a unit testing framework is how it handles errors and exceptions. When displaying an error, this must always be accompanied by relevant, actionable feedback to the programmer.

Utilities for Assessing Tests: the project should also provide utilities to measure the quality of the test suite, such as generating code coverage reports or mutation testing.

3.2 Overall Architecture

Early on in the design process Python was chosen as the high-level language to write tests in because of its readable syntax and ease of defining custom functionality.

When designing the overall architecture for this project, the first challenge encountered is that ARM machine code cannot natively be executed on x86 platforms. Therefore an emulator is required to execute the students' programs, and the majority of work involved with this project centres around how an existing emulator can be packaged in a manner that makes writing tests in Python for ARM assembly language intuitive.

Considering the architecture as a whole, since the goal is to have high-level tests for low-level code written for another platform, there is a need for an intermediary layer to sit between the two. This intermediary layer, which wraps the emulator and exposes it as a single object will be called a `Machine` in this report. Using this design allows this object to be created as a fixture using PyTest in our test suite.

In Figure 3.1, the test suite is depicted on the left hand side, and the firmware file built from the student's source code is depicted on the right. To allow the tests to interact with the ARM assembly code, the intermediary layer exposes APIs for invoking subroutines and reading or writing to the emulator's memory.

3.3 Building Blocks

To fulfill the requirements set out in 3.1, this project will build upon a number of core technologies. This section will introduce these technologies, briefly explain their features, and how they will be utilized within the context of this project.

3.3.1 Unicorn Engine

Unicorn was chosen for this project because it allows execution of ARM machine code on x86 CPUs, having been built on top of QEMU. Another major benefit of Unicorn for this

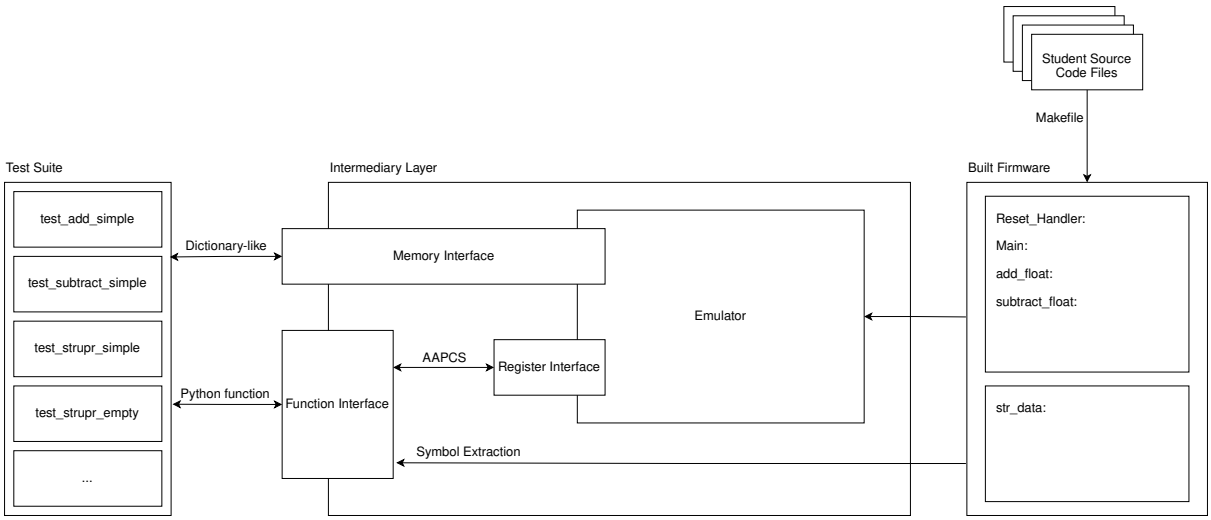

Figure 3.1: Overall Architecture Diagram

project is that it has a convenient API for interacting with registers, memory, and starting emulation. This section will give a brief introduction into these features, providing examples using the bindings for Python, our chosen language.

To begin, the emulator must be initialised. Initialising the emulator using the `unicorn.Uc` constructor sets the CPU architecture, as well as particular modes the emulated CPU can operate in, for example, little/big endian mode.

To be able to read or write from regions within memory, first it must be mapped into the emulator's address space using the `Uc.mem_map` methods. After this, the `the Uc.mem_read` or `the Uc.mem_write` methods can be used to read or write bytes to the emulator's memory, respectively.

Register sized values can also be programmatically read from or written to registers using the `the Uc.reg_read` or `the Uc.reg_write` methods.

A further feature of Unicorn Engine is the ability to execute a callback function after particular events that occur such as a memory read, an instruction being executed, or an invalid instruction found.

A minimal example showing how these methods can work together can be found below in Figure 3.2. For demonstration purposes, the assembled machine code multiplies the value in register R0 by 10, saving the result back to R0.

3.3.2 PyTest

While the core functionality of PyTest and unittest libraries are relatively similar, PyTest was chosen for this project because test suites written using PyTest typically have less boilerplate, resulting in more understandable code.

```python
from unicorn import unicorn
from unicorn.arm_const import *
from unicorn.unicorn_const import *

emulator = unicorn.Uc(UC_ARCH_ARM, UC_MODE_ARM)

# R0 = R0 * 10
code = b"\x0a\x10\xa0\xe3\x91\x00\x00\xe0"
CODE_ADDRESS = 0x8000000
MEMORY_SIZE = 0x20000

emulator.mem_map(CODE_ADDRESS, MEMORY_SIZE)
emulator.mem_write(CODE_ADDRESS, code)

# As an example for using `mem_read`, read back the code section
print(emulator.mem_read(CODE_ADDRESS, len(code)))
# >>>  bytearray(b'\n\x10\xa0\xe3\x91\x00\x00\xe0')


# Function to be executed after an instruction is processed
def code_hook(uc, address, size, user_data):
    print(f"Executing instruction from {hex(address)}")


emulator.hook_add(UC_HOOK_CODE, code_hook)


emulator.reg_write(UC_ARM_REG_R0, 99)
emulator.emu_start(CODE_ADDRESS, CODE_ADDRESS + len(code))
# >>>  Executing instruction from 0x8000000
# >>>  Executing instruction from 0x8000004

print(emulator.reg_read(UC_ARM_REG_R0))
# >>>  990
```

Figure 3.2: A minimal example showing how the Unicorn Engine emulator can be set up to execute ARM machine code.

A major feature of PyTest is how it manages dependencies for tests. PyTest's fixtures are an alternative to setup and teardown methods that would ordinarily be configured to execute before and after each test. Instead, fixtures are used as a way to declare which component or data item a test depends on.

A further feature of PyTest is its ability to use the built-in `assert` keyword while still providing detailed error messages. This is in contrast to libraries like unittest, which instead

use specialised assert* methods.

To execute a PyTest test suite, the `pytest` command will discover tests beginning with `test_*` in files matching `test_*.py`, by default. Alternatively, the files containing the tests can be run directly if PyTest's `pytest.main` function is invoked.

To demonstrate how unit tests and fixtures can be written using Pytest, Figure 3.3 shows a minimal example involving two tests which depend on a pre-initialised array `arr`.

```
import pytest

@pytest.fixture
def arr():
    return [1, 2, 3, 4]

def test_append(arr):
    arr.append(5)
    assert arr == [1, 2, 3, 4, 5]

def test_remove(arr):
    # remove the 0th element
    arr.pop(0)
    assert arr == [2, 3, 4]

if __name__ == "__main__":
    pytest.main()
```

Figure 3.3: A minimal example showing how unit tests and fixtures can be written using PyTest.

3.3.3 The ELF File Format

The Executable and Linkable Format (ELF) is a file format for executable files. Originally introduced as the System V Unix ABI, the format is cross-platform, though is most commonly used in Unix environments.

Of particular interest to this project is the Symbols Table (called .symtab within the ELF file itself). Here, labels are given to particular points within the code, including labels for subroutines and other pieces of data. Figure 3.5 compares how the labels defined in the assembly source code get added to the ELF file symbols table, when the source code is assembled to an ELF file.

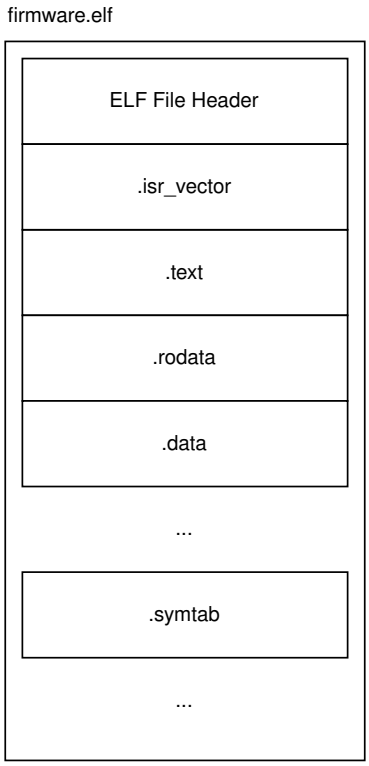

Figure 3.4: The internal structure of an ELF file showing the relevant sections for this project contained within it.

3.3.4 Building the Code

For Trinity College Dublin's Introduction to Computing Module, students' code is built into the ELF file format using a Makefile, a file which specifies how source code should be built. In this case these files specify the commands for how ARM Assembly source files should be assembled into object files (file extension *.o), and how these object files are then in turn linked together using a linker script to form the fully-built ELF file.

For the purposes of this project, it will be assumed that this process of building the source code into a firmware ELF file has already been completed and the original source code remains in place.

3.4 Basic Program Execution

One of the key requirements of the project is to be able to execute ARM code on a non-ARM computer so it can be tested. Since the majority of personal computers today are based on the x86 architecture, the code built cannot be run natively on most hardware. Furthermore, the assembled code also expects that it is the only thing being run by the hardware. That is, the code should not be run on top of an existing operating system.

As was described in Section 3.3.1, to solve this problem, Unicorn Engine was chosen as an

```
Main:
  MOV     R0, #1
While:
  CMP     R2, #0
  BEQ     EndWh
  MUL     R0, R0, R1
  SUB     R2, R2, #1
  B       While
EndWh:
End_Main:
  BX    LR
```

(a) ARM assembly code annotated with the following labels: Main, While, EndWh, End_Main

```
Symbol table '.symtab' contains 90 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
                        ...
26: 080001d4     0 NOTYPE  LOCAL  DEFAULT    2 While
27: 080001e2     0 NOTYPE  LOCAL  DEFAULT    2 EndWh
28: 080001e2     0 NOTYPE  LOCAL  DEFAULT    2 End_Main
                        ...
45: 080001d0     0 NOTYPE  GLOBAL DEFAULT    2 Main
                        ...
```

(b) The resulting symbols generated by the labels

Figure 3.5: A figure showing how the labels defined in the assembly source code get added to the ELF File symbols table.

emulator, which is an emulator built on top of QEMU. Unicorn supports interacting with the emulator using APIs written in a variety of programming languages, including Python, making it easy to integrate with and build on top of. At a high-level, the APIs Unicorn provide that are relevant for this project largely center around manipulating memory and starting/stopping execution of instructions.

With this design decision made and Unicorn chosen as an emulator, the first major milestone to accomplish was the execution of an ARM subroutine using Unicorn.

Within this, the first challenge to solve was loading the contents of the ELF file into the memory. Here, a few things need to be achieved. The first of these is initialising memory pages for Unicorn to use. This is necessary to limit the amount of memory consumed by Unicorn. Only accesses to memory locations which have been mapped will succeed, and those which have not been mapped will result in an error being thrown.

The code being run will assume that two memory address ranges are valid: 0x08000000 -

0x801FFFF and 0x20000000 - 0x20001FFF. The first of these ranges is the location the code and other read-only data is expected to be found at, and should therefore be readable, executable, but not writable. The second range is intended to be used for the stack and other data the program would need to write to RAM.

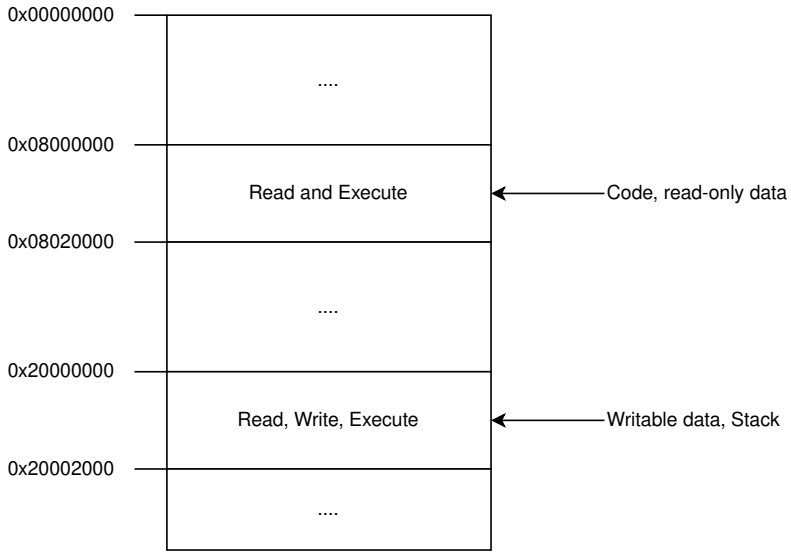

Figure 3.6: A diagram showing the memory locations expected to be present, the expected permissions, and the intended data to be stored in those locations. Ranges marked with '...' are not relevant for this project.

To initialise the Read-only Memory (ROM) section of the address space, the .text section of the ELF file is copied to it using Unicorn's `mem_write` function.

```
# Copy the .text section of the elf file to the emulator
code = self.elf_sections[".text"]
self.emulator.mem_write(code.address, code.data)
```

Figure 3.7: Code snippet to initialise ROM

In principle, this approach could have also been used to initialise the data for the RAM section of memory, where the contents of the .data section of the ELF file could have been copied directly to memory range 0x20000000 - 0x20001FFF. However, a different approach was chosen to initialise the data to keep more in line with how the initialisation process would happen on real hardware. Instead, a subroutine within the startup script, written in ARM assembly, copies the data that should be in RAM from where it is initally stored in ROM.

This subroutine therefore gets executed each time the emulator is initialised. For this project, this subroutine is assumed to be located at label `Reset_Handler`.

At this point, after the Reset Handler has been run, the emulator is considered to be fully initialised, and therefore is ready to execute the student's code. Unicorn Engine provides an API to execute instructions from a memory address range, and can also be bounded by the number of instructions executed.

To call a subroutine, all of the labels from the source code get exposed as methods on the `machine` object. This means that a subroutine can be invoked just by calling a method with the same name. Figure 3.8 shows how two subroutines that begin with the labels `fp_add` and `fp_sub` can be invoked from Python.

```
# Floating point addition subroutine
assert machine.fp_add(3.4, 1.2) == 4.6

# Floating point subtraction subroutine
assert machine.fp_sub(3.4, 1.2) == 2.2
```

Figure 3.8: Code snippet showing that subroutines with different names can be invoked through the Python function interface.

Starting execution at a particular point is relatively easy by looking up the start address of the subroutine using the ELF file symbols table, though stopping execution is a little bit more involved. One potential solution would be to define a label right after the subroutine in the ARM assembly source code. This comes with the advantage that it is very easy to implement, especially if the "end label" is just a prefixed version of the "start label". For example, `Main` until `End_Main`.

Unfortunately, this also comes with a major disadvantage that this approach relies on the programmer remembering to include the label in their source code. As this label is not necessary for the program itself to function, students will often remove this label. Therefore it would be desirable to use another mechanism to stop execution of the subroutine.

The mechanism that was ultimately settled on instead was to stop execution after the subroutine returns. What this means in practice is that before execution starts, the value that is stored in the Link Register is saved. This mechanism solves the potential problem of a label being missing, though on its own, it assumes that a subroutine will eventually return. Therefore execution is also bounded by a maximum number of instructions that can be executed, after which the program should give up.

3.5 Program State - Registers

Following on from basic execution of code, a way is needed to pass parameters to subroutines. The simplest way to achieve this in assembly is to use registers. The ARM Calling Convention (AAPCS) specifies that these parameters should be passed in registers R0-R3, with the computed result being returned in R0. Should more than 4 parameters be necessary, these should be placed on the stack, though manipulating the stack is out of scope for this project.

Keeping the "Principle of Least Surprise" in mind, this project will, by default, assume that the ARM calling convention is being followed, and use the Python function parameters to set these register values for the subroutine. Again, keeping in line with the ARM calling convention, the return value of the function is the value in R0 at the end of the subroutine.

Should the ARM calling convention not be followed for a particular subroutine, the register order for the arguments being passed to the function can be overridden, either on a per-test basis, or when the `machine` object is created. As an example, the code snippet in Figure 3.9 uses a program to calculate the result of raising the value in R1 to the power of the value in R2, placing the result in R0. Here, both approaches to overriding this order have been demonstrated.

One thing to note is that all parameters and return values are assumed to be Two's Complement integers for the sake of simplicity in this project. In the future should this assumption not be sufficient, multiple data types could be supported through a concept this paper will refer to as Lazy Deserialisation. This will be elaborated on further in Section 3.6.2.

```python
import ...

# Use _ as a placeholder value
_ = None
custom_argument_ordering = [SlotLocation.R1, SlotLocation.R2]

@pytest.fixture
def machine():
    return Machine(
        "firmwares/1410-power/build/firmware.elf",

        # Argument order can be specified when initialising the
        ↪  machine...
        # argument_mapping=custom_argument_ordering,
    )

# ... or for a specific test
@input_map(custom_argument_ordering)
def test_power_simple(machine):
    assert machine.Main(3, 4) == 81
```

Figure 3.9: Code snippet showing how the register mapping can be overridden either on a per-machine basis or per-test basis.

3.6 Program State - Memory

To go beyond what can be achieved using only registers, it is a requirement to also support interacting with memory - both reading and writing - ideally in a way that is intuitive to programmers who are familiar with Python. Square-bracket notation has been chosen as the syntax for accessing memory. In Python this can be achieved by defining `__setitem__()` and `__getitem__()` methods, which can be overridden for a class to implement custom functionality. This notation was chosen as it is familiar to students, and because of its similarities to ARM Assembly's square bracket notation for interacting with memory. An example of the notation can be found in Figure 3.10.

To specify memory addresses, this project will make heavy use of labels which point to pre-allocated regions in the emulator's RAM. These memory regions will have be statically allocated when building the student's source code into an ELF file.

In the rest of this section, reference will be made to the functions `to_bytes(value)` and `from_bytes(value, value_type)`. As their name implies, these functions allow for converting Python objects to bytes, and reconstructing a Python object from raw bytes. Support for integers, strings, lists, and sets have been added to these functions, though can

be extended to support more data types in the future.

```
def test_string_upper(machine):
    machine["testString"] = "hello"
    machine.Main()
    assert machine["testString"] == "HELLO"
```

Figure 3.10: Code snippet showing the syntax for reading and writing to memory within a unit test.

As will be discussed later in Sections 3.6.2 and 3.6.3, the main challenge when trying to assign meaning to bytes at a particular memory location is that there is usually not enough context to interpret the bytes and recover the information about an object's type.

3.6.1 Writing to Memory

When comparing both reading and writing to memory, writing is the more straightforward of the two operations. This is because when serialising a Python object to bytes all of the information about the type is present. The main steps to writing to memory are first converting the object into an agreed byte format, and then writing those bytes to the emulator's memory, starting at a chosen address. In this project, the address can be specified using a label that has been specified in the student's program's source code.

These steps can be seen in the code shown in Figure 3.11, which is the implementation for the `__setitem__` method used to support using square-bracket notation for accessing memory.

```
def __setitem__(self, key, value):
    if key not in self.elf_symbols:
        raise KeyError(f"Label `{key}` not found in elf file")
    memory_address = self.elf_symbols[key]

    value_bytes = to_bytes(value)
    self.emulator.mem_write(memory_address, value_bytes)
```

Figure 3.11: Code snippet showing implementation for writing to memory using square-bracket notation.

3.6.2 Lazy Deserialisation

At this stage, before considering the specific details of memory reads, it is necessary to introduce a concept that will be used to streamline the process of assigning a datatype to

bytes in the emulators memory. As previously noted, the biggest problem with reading from memory and interacting with those bytes in Python, is that there is usually no type information present in the bytes themselves. What this normally means is that the programmer must provide the missing length and type information, usually taking the form of invoking purpose-built methods to parse the data from a specific format.

For example, in Figure 3.12, to be able to determine whether the string "Hello World!" is what is actually stored at memory location `example_string` an exact number of bytes from the emulator's memory must first be read, then function to reconstruct a Python object must then be used. Only at this point can Python's equality operator be used to compare the two strings to determine equality. Here in this hypothetical example, the programmer must pass the number of bytes that should be be read to the `read_bytes` function, and the desired reconstructed data type to the `from_bytes` function.

```
# Assume "Hello World!" is a string at label "example_string"
bytes = machine["example_string"].read_bytes(12)
reconstructed_string = from_bytes(bytes, str)
assert reconstructed_string == "Hello World!"
```

Figure 3.12: A minimal example for potential syntax to manually assign an interpretation to bytes read from the emulator's memory.

In Python, as is common for many other methods, the equality operator (==) can be overridden to provide custom functionality. This is achieved by defining a function with the signature `def __eq__(self, value: object) -> bool` on the class whose equality operator is being overridden. A key insight for being able to change the style of code being demonstrated in Figure 3.12 is that the equality operator actually has enough information itself to be able to determine both the datatype and length it should take up in memory. The desired type of the bytes in memory can be inferred from the other object's type, and the number of bytes to read from the emulator's memory can also be determined from the other object too. Since the equality operator itself has the information needed to perform a memory read, this allows the actual read from the emulator's memory to be deferred until the comparison step. A natural consequence of using this technique is that the bytes in the emulator's memory will only have meaning when being compared with something else.

Using this technique allows our previous example to instead be written in a single line, as shown in Figure 3.13

```
# Assume "Hello World!" is a string at label "example_string"
assert machine["example_string"] == "Hello World!"
```

Figure 3.13: The example from Figure 3.12 rewritten using Lazy Deserialisation.

3.6.3 Reading from Memory

Now that a mechanism for assigning meaning to bytes in memory has been established, it becomes a lot easier to reconstruct a Python object from raw bytes.

When requesting a read from a location in the emulator's memory, instead of directly returning a byte array, now only a placeholder object gets returned immediately. This placeholder object stores the location to start reading from, as well as a custom equality operator implementing Lazy Deserialisation. Again, it is important to note that on its own this class does not contain any way of interpreting the bytes in memory, and instead must be compared to another Python object for the meaning of the bytes to be realised. Figure 3.14 shows the relatively simple implementation of the `__getitem__` method, which returns a `ProxyResult` placeholder object.

```
def __getitem__(self, key):
    if key not in self.elf_symbols:
        raise KeyError(f"Label `{key}` not found in elf file")

    return ProxyResult(self, key)
```

Figure 3.14: Code snippet showing implementation for reading from memory using square-bracket notation.

Due to the fact the actual read from memory is deferred until the `ProxyResult` object is compared with another object, the bulk of the complexity from the read is contained within the `ProxyResult`'s equality operator, as seen in Figure 3.15. The first stage is to read the same number of bytes that are necessary to store the comparison object. For simplicity this is done by converting the comparison object to bytes and measuring the length of this, though in principle this could be calculated directly. After this, the `from_bytes` function can be used to reconstruct the bytes into a Python object, again, using the same data type as the comparison object. To support different datatypes within generic data structures, we also pass the type of the inner elements to the `from_bytes` function. Currently, this supports 1 layer of nesting, though there is potential to change this to support arbitrary levels of nesting in future work. The final step is to use the native equality operator to compare the two objects.

```python
class ProxyResult:
    def __init__(self, machine, label: str) -> None:
        self._machine = machine
        self._label = label

    def __eq__(self, value: object) -> bool:
        comparison_object_bytes = to_bytes(value)
        machine_bytes = self._machine.read_memory_location(
            self._label, len(comparison_object_bytes)
        )

        # for generic data structures, get the inner type
        inner_type = None
        if type(value) in [list, set]:
            inner_type = type(next(iter(value)))

        reconstructed_object = from_bytes(
            machine_bytes,
            element_type=type(value),
            inner_element_type=inner_type
        )
        return value == reconstructed_object
```

Figure 3.15: Code snippet showing the implementation for Lazy Deserialisation when comparing a region in memory with Python object.

3.7 Handling Errors and Exceptions

A key part of a unit testing framework is how it presents exceptions to the user. These should be clear and actionable for the programmer, which in this case will be a student.

By default, the exception that is thrown by Unicorn Engine provides little meaningful context as to why the error has occurred. In the screenshot in Figure 3.16 we see that when an exception is thrown, the context shown is internal to the Unicorn Engine library, and is of little use to a programmer who is developing a program written in ARM Assembly.

```
self = <unicorn.unicorn.Uc object at 0x7f3a1c975fd0>, begin = 134218229, until = 134218192, timeout = 0, count = 0

    def emu_start(self, begin: int, until: int, timeout: int=0, count: int=0) -> None:
        self._hook_exception = None
        status = _uc.uc_emu_start(self._uch, begin, until, timeout, count)
        if status != uc.UC_ERR_OK:
>           raise UcError(status)
E           unicorn.unicorn.UcError: Invalid memory write (UC_ERR_WRITE_UNMAPPED)
```

Figure 3.16: Screenshot showing a bare exception from Unicorn Engine.

What would be more useful to the programmer would be if they knew where in their code

the exception occurred, had more context as to why that error might be happening. More specifically, this project shows the exact instruction that caused the exception in the source code, and also provides a hint to the programmer on what might need to be changed to solve the error.

To achieve this, the `unicorn.UcError` exception is caught and a new exception is rethrown. During the process of throwing this new error, the current address present in the Program Counter register is inspected, and then passed to addr2line from GNU binutils to look up the line in the source code where the exception comes from. This uses the debugging information present in the ELF file to map addresses to line numbers in the source code. Naturally, this assumes the ARM source code is still available in the same location as when the firmware was built. From here we are able to read from the source code files around the location of that line number to provide better context to the programmer. Finally, if the exception thrown is a common error, hints for what might be going wrong are also included. All of these hints have been listed in table 3.1 below.

Error Name	Hint
Invalid instruction being executed	Ensure nothing has overwritten the contents of this address
Invalid memory read	Your program is trying to read from a memory location that doesn't exist on the emulator
Invalid memory write	Your program is trying to write to a memory location that doesn't exist on the emulator
Invalid memory fetch	Your program is trying to fetch an instruction from a memory location that doesn't exist on the emulator
Memory address not writable	Cannot write data to this memory location, as it is not writable. (Are you trying to write data to the read-only memory instead of RAM?)
Memory address not readable	Cannot read data from this memory location, as it is not readable. (Are you reading from a protected region of memory?)
Memory address not executable	Cannot fetch instruction from this memory location, as it is not executable

Table 3.1: A table describing the hints provided to common exceptions caused by a user's program.

The final result is an exception that is much more useful for being able to debug the ARM Assembly program. An example of this new exception can be seen below in Figure 3.17.

```
E               exceptions.ExecutionError: Memory address not writable
E               Extract from /workspaces/dissertation-python/firmwares/1410-power/src/power.s:
E                 17 |    @ Simulate trying to write to ROM
E                 18 |    MOV R0, #0x8000000
E               > 19 |    STRB R1, [R0]
E                 20 |
E                 21 |
E                 22 |    MOV     R0, #1          @ result = 1
E               Cannot write data to this memory location, as it is not writable. (Are you trying
 to write data to the read-only memory instead of RAM?)
```

Figure 3.17: Screenshot showing the rewritten exception with a clear error name, a source code snippet, and a hint for how to solve the error.

3.8 Code Coverage

As Code Coverage is the most common and widely used metric for evaluating how good a test suite is, a utility to generate a report for showing this has been developed as part of this project. It is important to note here that this feature is only relevant for people who are writing the tests. For example, within Trinity College Dublin's Introduction to Computing module, this will be the instructors rather than the students.

The decision around what type of coverage the report should show was largely pragmatic, based on ease of implementation. When comparing instruction coverage and branch coverage, branch coverage is often considered a much more faithful metric, especially when represented as a percentage. However, branch coverage requires that the branching points first be found in the program, which requires decompiling the program and calculating the branch targets. Whereas, instruction coverage does not require this extra step, so therefore this was the metric chosen for the code coverage report.

The mechanism chosen to enable coverage is a `@coverage` decorator that can be placed before the definition of a test. At present this will generate a coverage report specific to that test, though there is scope that this can be developed further to produce an aggregate report across many tests, showing how effective the test suite is as a whole.

```python
@coverage
def test_power_simple(machine):
    assert machine.Main(_, 2, 3) == 8
```

Figure 3.18: Code snippet showing @coverage decorator being used to enable coverage for the test_power_simple test.

The implementation for generating the report operates in a similar manner to how exceptions are displayed. When the `@coverage` decorator is added to the test, it triggers a hook function to get executed after each instruction executed. This hook stores the address

of the executed instructions, which can then be passed to addr2line to look up what line number the address corresponds to in the source code, computing a set of line numbers that have been executed. To generate the report itself, the source code is read, and each line is marked according to whether that line is present in the set of line numbers determined by the previous step.

The final result is a text-based report denoting which lines of the source code have been executed by that test, an example of which can be seen below in Figure 3.19

```
/workspaces/dissertation-python/firmwares/1410-power/src/power.s
================================================================
1  ❌ .syntax unified
2  ❌ .cpu cortex-m3
3  ❌ .fpu softvfp
4  ❌ .thumb
5  ❌
6  ❌ .global  Main
7  ❌
8  ❌ Main:
9  ❌
10 ❌ @ Write a program to compute x^y
11 ❌
12 ✅ MOV     R0, #1          @ result = 1
13 ❌ While:
14 ✅ CMP     R2, #0
15 ✅ BEQ     EndWh           @ while (y != 0) {
16 ✅ MUL     R0, R0, R1      @  result = result × x
17 ✅ SUB     R2, R2, #1      @  y = y - 1
18 ✅ B       While           @ }
19 ❌ EndWh:
20 ❌
21 ❌ @ End of program ... check your result
22 ❌
23 ❌ End_Main:
24 ✅ BX    LR
25 ❌
26 ❌ .end
```

Figure 3.19: An example code coverage report, showing line coverage.

3.9 Summary

In this chapter, execution of ARM assembly code has been demonstrated, allowing the programmer to set register values using parameters in a function-based API. Data can be read from and written to memory using an intuitive interface, abstracting underlying complexity. Finally, code coverage reports have also been implemented.

4 Evaluation

In this chapter, the test harness as implemented in Section 3 will be discussed and evaluated by comparing it to requirements set out in Section 3.1 and the prior work identified in Section 2.5. The limitations of the project and potential future work will also be identified.

4.1 Discussion

Throughout this project the main objective has been to design a test harness for ARM Assembly programs, and has been achieved successfully.

In terms of the implemented features, the first feature the test framework supports is executing ARM assembly code through its use of Unicorn Engine emulator. The wrapper around the emulator exposes an intuitive function interface which will cause subroutines to be invoked within the emulator. These exposed functions take the name of the labels found in the ARM source code, with the parameters to these functions setting the standard registers.

The next feature successfully implemented was interacting with memory. Both reading and writing is supported, using labels instead of addresses. An aspect which this project performs particularly well is the way it is able to streamline memory reads whenever they are being used for comparisons against another Python object. Streamlining this API reduces the volume of code required in each test, making the tests easier to understand.

When comparing this project with MIPSUnit on memory allocation, further improvement is possible here, as this project supports allocating memory when the firmware file is created. By contrast, MIPSUnit, can allocate memory dynamically through an API.

Another feature this project has successfully achieved is handling exceptions thrown by the code. This project will rewrite any exceptions thrown to include the exact ARM assembly instruction where the exception was thrown. Hints for how the error might be fixed are also included. This is particularly important for novice programmers, who will often need guidance or a hint on how to resolve issues.

Finally, a code coverage utility has been implemented as part of this project that can generate reports detailing which lines of the student's program have been executed by a particular test. There is room for improvement here, as in other test frameworks, branch coverage would often be the metric of choice, and the generated report would show the sections of code that have been executed by any test within the suite.

4.2 Limitations and Future Work

While this project is successful in meeting its requirements, it still has some limitations, and there is room to improve in some areas.

The following areas have been identified as potential areas for future work:
Dynamic Memory Allocation: to support operating on an arbitrary number of distinct data items, a dynamic memory allocation system could be developed. This feature would be able to allocate memory programmatically, rather than the current system of pre-allocating memory statically in the ARM Assembly source code.

Support for more data types: expanding on the collection of data types beyond integers, strings, lists and sets would enable tests for more involved subroutines to be written.

Branch Coverage: often viewed as a more useful metric than line coverage, branch coverage was left out due to the added complexity of implementing it, as well as time constraints on the project.

Mutation Testing: while mutation testing was originally planned to be implemented, time constraints meant that this feature could not be completed within the time window allocated to the project.

5 Conclusion

This project has successfully demonstrated that tests for ARM Assembly programs can be written in a higher-level language, in this case, Python. A test harness that is suitable for use in an educational setting by novice programmers has been designed and developed. For the implementation of the test harness, an intermediary layer was developed, providing an abstraction around the Unicorn Engine emulator so that the ARM Assembly programs can be executed on x86-based CPUs.

This intermediary layer supports invoking ARM Assembly subroutines from an intuitive function-based interface, where the parameters set register values for the subroutine. Also successfully demonstrated, is a memory interface that allows data to be written to and read from pre-allocated sections of memory.

Finally, utilities including exception handling and code coverage reports have been achieved.

Bibliography

[1] "CSU11021 – introduction to computing i – teaching and learning."

[2] C. Wilcox, "The role of automation in undergraduate computer science education," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pp. 90–95, Association for Computing Machinery.

[3] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs," vol. 3, no. 3, pp. 1–es.

[4] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT: automatically grading programming assignments," in *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ITiCSE '08, p. 328, Association for Computing Machinery.

[5] M. Peveler, J. Tyler, S. Breese, B. Cutler, and A. Milanova, "Submitty: An open source, highly-configurable platform for grading of programming assignments (abstract only)," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, p. 641, Association for Computing Machinery.

[6] "Features | submitty."

[7] C. Sharp, J. van Assema, B. Yu, K. Zidane, and D. J. Malan, "An open-source, API-based framework for assessing the correctness of code in CS50," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, pp. 487–492, Association for Computing Machinery.

[8] "Use autograding - GitHub docs."

[9] M. Fowler, "Xunit."

[10] "unittest — unit testing framework — python 3.12.2 documentation."

[11] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp. 201–211. ISSN: 2332-6549.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator,"

[13] "Download QEMU - QEMU."

[14] "Emulation — QEMU documentation."

[15] N. A. Quynh and D. H. Vu, "Unicorn: Next generation CPU emulator framework,"

[16] "GDB: The GNU project debugger."

[17] A. Brown and G. Wilson, *The Architecture of Open Source Applications, Volume II*. Lulu.com.

[18] "Debugging with GDB - the gdb/mi interface."

[19] Z. Kurmas, "MIPSUnit: A unit testing framework for MIPS assembly," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 351–355, ACM.

[20] D. K. Vollmar and D. P. Sanderson, "MARS: An education-oriented MIPS assembly language simulator,"

[21] J. Damas, B. Lima, and A. J. Araújo, "AOCO - a tool to improve the teaching of the ARM assembly language in higher education," in *2021 30th Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE)*, pp. 1–6. ISSN: 2472-7687.