CSM Read: RISC-V AutoGrader Implementation

Thomas Boyle, Cole Howard, Priya Mudapalli, Coleson Oliver & Nia Williams

For Prof. Amelia Read

June 16th, 2021

Table of Contents:

1 Introduction

The Computer Organization class at Mines teaches assembly language, to learn about system architecture. The client, Professor Amelia Read teaches several Computer Organization sections and was looking for ways to help students more effectively understand and practice coding in assembly. The Computer Organization course as a whole is also in the process of migrating class material from the existing system which uses the MIPS assembly language to a newer and more relevant assembly language called RISC-V.

The goal of this project was to implement RISC-V on the existing Mines AutoGrader (bartik.mines.edu) server to support bite-sized programming assignments with instant feedback that build towards larger standalone RISC-V programs in the Computer Organization course. The server currently supports assignment creation in both C++ and Python for other Computer Science courses at Mines. To that end, the following are the functional and non-functional requirements that were crucial in terms of delivering a product that meets the client's needs.

2 Requirements

2.1 Functional Requirements

- Write a program to take in code written by students within the AutoGrader interface and pass it into the RISC-V emulator
- Read the internal state of the RISC-V emulator and compare that data with specified assignment solutions to verify the correctness of student code
- Return the results of tests run to the AutoGrader interface to be displayed to the user as correct or incorrect, as well as any error messages if necessary
- Make options to choose RISC-V as the test language as a server admin
- Allow server admins to choose if they want to restrict pseudo-instructions
- Write and make public at least 4 basic RISC-V programming assignments to be deployed on the production server
- Additional code and functionality our service adds must follow existing conventions for style and language

2.2 Non-functional Requirements

- Protect against memory leaks
- Format system output in a readable and intuitive way to the end-user

- Use the existing framework as a base for the new RISC-V support, does not change other functioning parts of the system
- Decide on an appropriate timeout length for RISC-V to catch and manage cases of infinite looping

3 System architecture

Implementing RISC-V within the existing functionality of the AutoGrader server involved modifying much less of the overall code flow than was initially expected. Knowledge of most of the system architecture was unnecessary to add support for a new language, as the files that directly interact with the student code are rather isolated from the rest of the server, to help insulate the system from any malicious code. There were very few parts of the existing code that needed to be edited or added to. The part that was edited was the assignment creation for teachers or administrators, where more options were added to choose RISC-V as the language as well as new input and output types. The parts that were added were a new class, Docker image, and Bash file to run any RISC-V code. The server uses Ruby as the primary language with the Ruby on Rails framework.

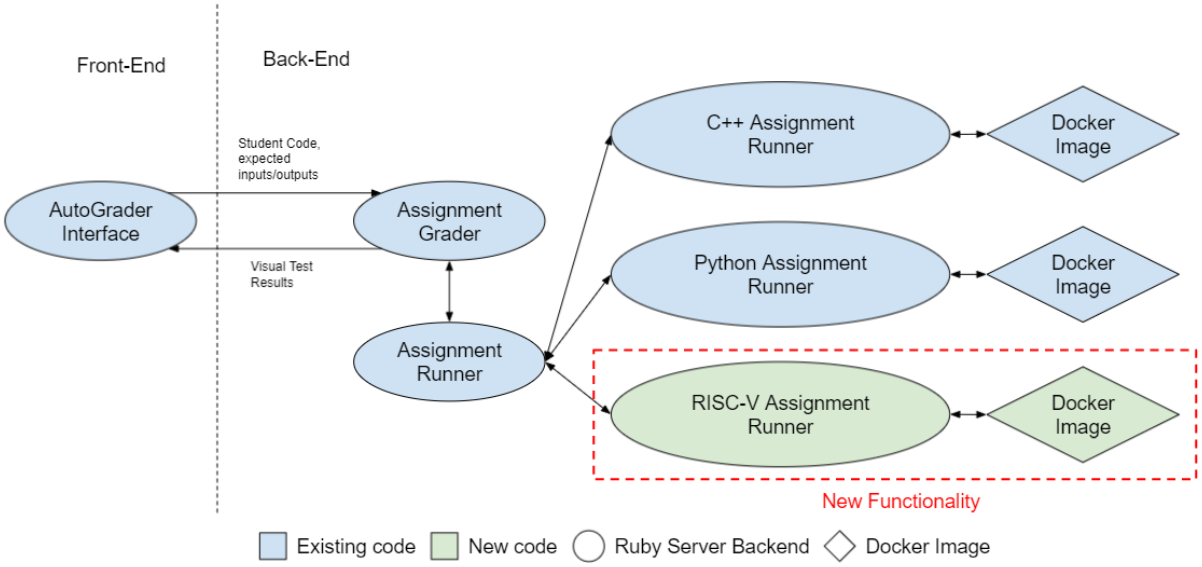

Figure 1: Overall System Architecture

The processing of student code starts with the AutoGrader web interface. The interface will take the student code and bundle it with the specific assignment data, such as the specified inputs and outputs for the assignment. This is then passed unmodified through a few controllers, which did not need to be edited to include RISC-V, to the AssignmentGrader, the start of Figure 1. The AssignmentGrader is the class that will

later handle passing the test results to the class that displays them on the web page. It gives the student code and assignment parameters to the general language AssignmentRunner, which starts to split up the assignment parameters into separate variables. It determines which specific language runner needs to be run, then creates an instance of it and pulls the appropriate file extension to which to write the code.

The C++ and Python assignment runners work in very similar manners; both take in the student code file and write a unit testing file in their respective languages to test the student code against the expected output. This testing suite and the student code are run inside a Docker image, which helps to isolate the student code even more from the server, as well as help parallelize the workload. The results of the unit tests are written to an XML file to be interpreted by the Ruby assignment runner as either a pass, fail, runtime error, or a timeout error. Compilation errors are handled before any of the actual tests are run.

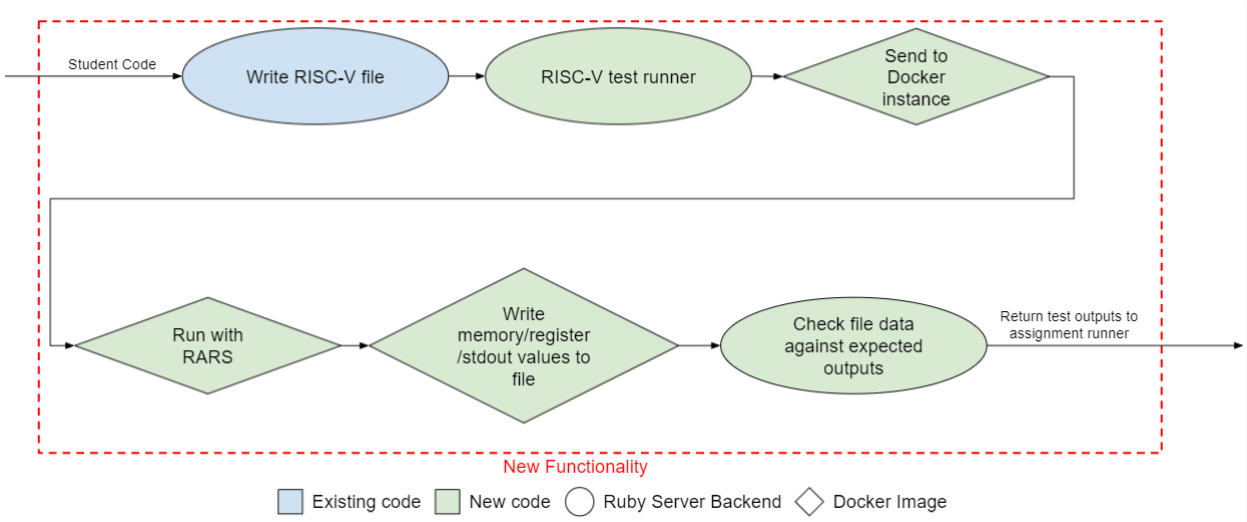

Figure 2: RISC-V Test Runner Detailed View

The RISC-V assignment runner works differently than the other language runners, as shown in Figure 2, due to the assembly language requiring an interpreter and lacking a native unit testing suite suitable to the needs of this project. The student code is given to the interpreter, called RARS, which is run inside a Docker image. The memory locations and standard out of the RISC-V program are written to files, which are then parsed by the Ruby class and compared to the expected outputs to produce the same types of test results as the other classes. All of these results are passed back to the general AssignmentRunner, which passes it up to the AssignmentGrader. The AssignmentGrader parses the test results into a form that can be used to display the results to the student on the webpage.

4 Technical Design

4.1 RISC-V Implementation

Since RISC-V is an assembly language which is the code that runs directly on a CPU, it operates differently from the existing languages implemented on the AutoGrader system. The first major difference is the lack of an existing unit testing suite; the method that the C++ and Python runners use is by writing unit testing code for each test and running the student code using the unit tests. This means a custom unit testing suite had to be written in Ruby to check the correctness of the outputs from the student code. The second major difference is that if the assembly language does not natively run on a given CPU, then it must have an emulation layer to allow it to run. The existing Linux environment that runs the AutoGrader server runs on an x86-64 CPU, therefore the RISC-V code cannot be run natively and must use such an emulation layer. To allow this the client determined that RARS (RISC-V Assembler and Runtime Simulator) was the best emulator to use, due to it both being a capable interpreter and a port of the existing MIPS emulator the Computer Organization class already used, MARS (MIPS Assembler and Runtime Simulator).

4.2 RARS/Ruby interface

To link RARS and Ruby, the test runner writes a series of files that specify what inputs and outputs should be given to RARS. Since RARS has a command-line interface there is a file called run.sh that does all the direct interaction with RARS. It first checks to make sure that the given student code has no compile errors, if that passes then it constructs a Linux command to both put the desired inputs into RARS and to return the desired outputs. It also defines a timeout period for the student code and writes a file for any test that timed out to signify the student code took too long to run. Figure 3 shows all the files that are written by the test runner and the files that are returned by the shell script, the # symbol represents the test number and shows that there is a unique file for each test (e.g. with 3 tests: code1.s, code2.s, code3.s).

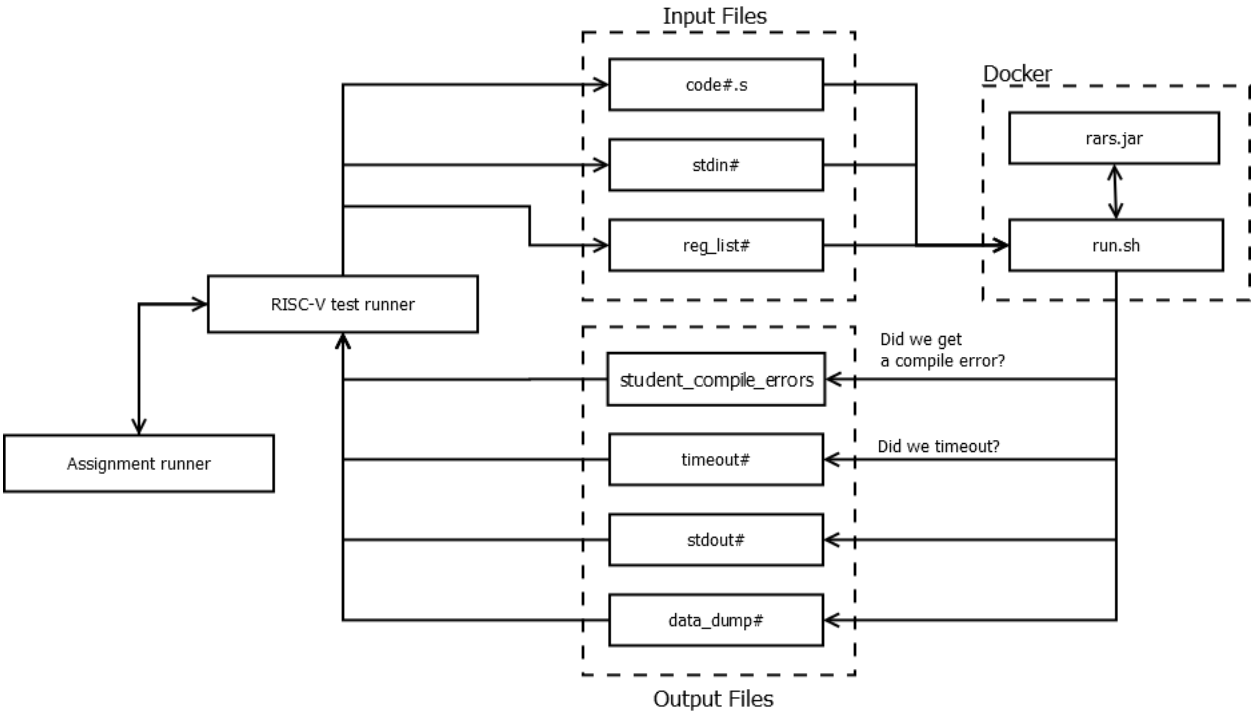

Figure 3: Input/output File Diagram

code#.s	Contains the student code as well as register value assignments specified as the input for a given test
stdin#	The file that is piped into the std input for the student code to grab from
reg_list#	Passing this file as an argument to RARS tells it to print out specific register values in the right order to be checked
student_compile_errors	This file exists if there were any compile errors when compiling the non-modified student code, tests will not run if this is generated
timeout#	If this file exists for a given test # it specifies that that test timed out
stdout#	If the student file writes to std output then it is captured in this file for each test
data_dump#	Contains register information as well as the memory dump of the system

Table 1: Intermediate File Details

4.3 Running a test

To run a given test, if the desired input is the standard in stream then a stdin file for each test is written and is then piped into stdin for RARS. If the desired input is instead through writing individual registers, then the student code file must be prepended for each test to set the register values for that test. The Ruby code then invokes Docker and run.sh to check if the code successfully compiles, and then it runs the tests. The following is a flowchart of how the server runs each test.

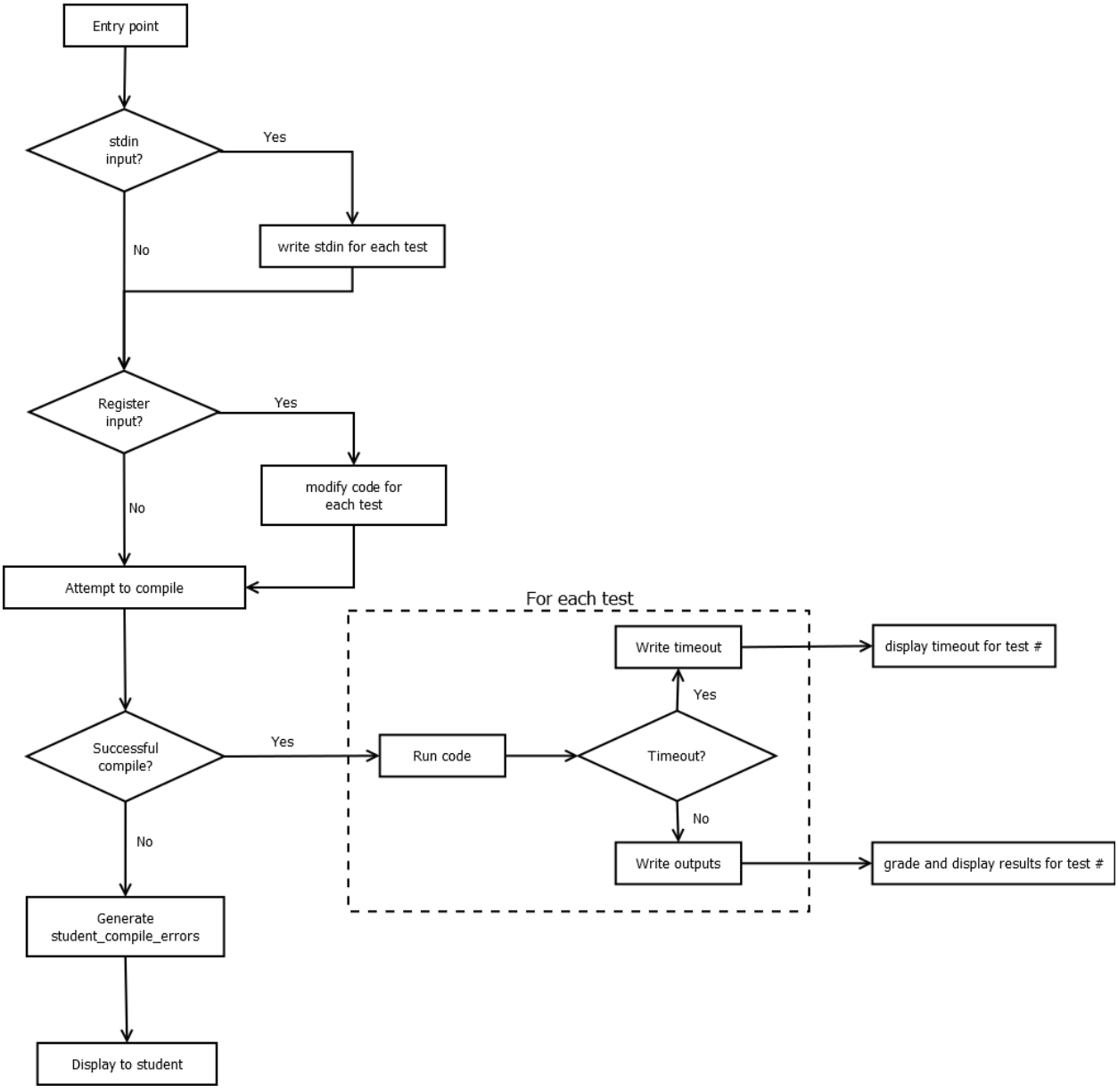

Figure 4: Test Running Flowchart

5 Quality Assurance

5.1 Testing

In the current implementation of the AutoGrader server backend, there is no general testing framework included with the existing codebase. To this end, there is no way to robustly confirm that the code that has been modified does not affect previously implemented functionality via the use of an existing unit testing suite or similar platform. To test the system without needing to add a fully fleshed out and functioning unit testing suite, it was crucial that the functionality of the other language runners was not compromised. This was checked by running programming assignments on the development server in both pre-existing languages (C++ and Python) to make sure that their respective language runners perform as expected with our additional changes.

Additionally, to test the functionality of the backend, a collection of RISC-V "unit tests" were created to run using the new RISC-V extension to the Mines AutoGrader server. These tests take various inputs/outputs to test that the system can handle all different forms of input/output methods appropriately. This is to make sure that both the output and state of the system are correct, in terms of the input presented to it. These tests verify that the new RISC-V extension to the AutoGrader server functions as intended and catches any errors if necessary. For example, Figure 5 below gives a sample of what a simple unit test looks like. This code accepts two user inputs and stores one of them in register `t0` and the other in stdout.

```
# simple echo int input program
.text

addi a7, x0, 5          # take user input
ecall

add t0, a0, x0          # store user input in t0

ecall                   # take user input again

addi a7, x0, 1          # print out user input
ecall
```

Figure 5: Sample Unit Test

5.2 Coding Standards & Documentation

In terms of coding standards, since the server and two similar language implementations were already written, the new implementation mostly follows the naming, commenting, and formatting standards established by the existing codebase. Due to inadequate comments within the code and the lack of documentation, it was quite difficult to understand what was happening initially. Consequently, any unclear previous coding practices were avoided and it was a priority to make more meaningful comments. The existing documentation related to getting development on the server started was also inadequate, so this document was added onto as well for future developers. Additionally, there were new formatting requirements that were put into place by the team, and documentation was added accordingly so that teachers or anyone else that needs to be able to create assignments will have the resources to do so. The overall goal is to make sure that the code is thoroughly understandable and that future developers will have no issues understanding and extending the new code.

6 Results

Since there was no existing unit testing framework implemented throughout the server, there was not an easy way to check what parts of the code were broken by new features. To solve this, a few specific RISC-V assignments were written to test certain microscale data paths and ensure that each different possible input and output scenario was accounted for in the code. These assignments gave some insight into what specific data paths were producing errors and helped in debugging the code as new features and data paths were implemented. Several other larger scope test assignments were created to ensure that all implemented functionality is handled properly. The macroscale data path for the server is the most difficult to test, so these tests are essential to ensuring that all facets of the student code are handled as expected.

There was also no standard method of performance testing as there was no access to the server the software will run on. Performance depends mostly on the student's code as dispatching the test cases does not take very long. Performance to dispatch the student code was comparable to the other languages that were already implemented in the software. In the case that the student code does not return a value after a preset amount of time, the software will terminate the program and give a timeout error. Also, the memory usage was restricted to only 32 kilobytes in the RISC-V simulator, as the problems are planned to be short and simple. This memory size is enough for the scope of the assignments that will later be added to the server, and it prevents the possibility of excessive memory usage.

7 Future Work

There are a few ways that the current RISC-V AutoGrader framework could be extended, in terms of future work. One of these includes finishing writing a full suite of assignments for Computer Organization as each of the large assignments that students normally take on all at once needs to be broken down into smaller, more manageable, problems that can give feedback immediately to the student. Also, assembly languages can often be hard to read as instructions all start on the same column with no color coding to distinguish code from comments. Therefore, adding to the editor's functionality on the front end would be helpful for the students to improve the readability of their code as syntax highlighting for RISC-V is currently not supported.

Due to the requirements of the course, one feature that turned out to not be possible to implement without extensive additional work was the dynamic restriction of pseudo-instructions. Because the student is required to load the starting address of any string or array that needs to be compared against an expected value, the use of the load address pseudo-instruction is unavoidable in most assignments. Due to the nature of reduced instruction set architectures the RARS feature that allows the restriction of all pseudo-instructions would force the students to use complex and unhelpful workarounds for the load address instruction which is counterproductive to our goal of making assembly programming more accessible. Overall, the assignments will be less effective to students who skip the work and use pseudo-instructions, however, restricting them could be implemented by a post-processing procedure and searching through the student's code for any disallowed pseudo-instructions given a list of acceptable ones.

Another possible extension to the server backend as a whole as previously mentioned is the addition of an overarching unit testing suite. Because the server has no existing testing implemented, errors created by any new code that inadvertently affects existing functionality will not be caught until it crashes the server. Implementing an overarching unit testing suite for the Ruby backend would make it easier to identify and eliminate unintentional datapath errors and other code interaction problems during development, creating an overall more robust service and additionally simplifying general debugging and server maintenance. An existing implementation of unit tests would also aid the future development of new features as the process of unit testing documents the intended usages of existing code by providing example cases. Therefore, even where documentation may be lacking, there is a separate repository of code usage examples in the form of unit tests to more completely show the intended use of different functional elements.

8 Appendices

8.1 Development environment setup instructions

You'll need Ruby and Rails setup on your computer. If you're using Linux or Mac, you should be able to install the nix tool and use the `shell.nix` file to create a development environment with all dependencies installed. If you're not using nix, then install rails and ruby using your package manager, then run `bundle install` to grab all of the dependencies.

You'll also need docker setup on your computer. Install docker using your package manager and add yourself to the docker group. You can test that it works by running `docker build common/cpp` and `docker build common/python` from the root of the project.

Once the dependencies are installed, you'll need to create a database.yml file. Here's an example of what that might look like:

```
development:

  adapter: sqlite3

  database: db/development.sqlite3

  pool: 5

  timeout: 5000

test:

  adapter: sqlite3

  database: db/test.sqlite3

  pool: 5

  timeout: 5000
```

Finally, you need a secrets.yml file. It should look like:

```
development:

  secret_key_base: randomString

test:

  secret_key_base: randomString
```

The command rake secret will output secret strings that you can copy into the file.

That should be it! Just run rails s to start the server

8.2 Feedback

- Fixed proper noun capitalization
- Removed mentions of RARS from section 2 for readability and clarity
- Edited sections 5.1 and 5.2 to correct for verb tense errors and clarity
- What is "bartik"- clarified reference to the bartik server
- Updated introduction to better introduce the project and segway into requirements
- Clarified edits and additions to the existing server
- Elected to not elaborate exactly how the student code is delivered to the AssignmentGrader, as we did not edit that whatsoever and treated it as a black box for the purposes of the project