EXPERIENCE TEACHING RISC-V ASSEMBLY PROGRAMMING WITH MOODLE

Idilio Drago[1], Sergio Rabellino[1], Enrico Cassano[2], Saverio Accurso[2]

[1] University of Turin, Computer Science Department
{idilio.drago,sergio.rabellino}@unito.it

[2] University of Turin, Course degree in Computer Science
{enrico.cassano,saverio.accurso}@edu.unito.it

— FULL PAPER —

***TOPIC**: Higher Education*

Abstract

RISC-V is an open-source Instruction Set Architecture (ISA) designed to be simple, modular, and extensible, making it a versatile choice for a wide range of applications. It is, therefore, an excellent ISA option for teaching computer architectures and assembly programming, and indeed it is starting to be adopted by a range of textbooks on the topic. Given the middle-level nature of the language, it is usually challenging for students to test the correctness of their code and run their programs in practice. This high entry barrier often results in a difficult learning experience, reducing the effectiveness of the learning process.

We here describe our experience in teaching computer architectures using Moodle. In addition to creating a vast amount of material in the form of classic self-evaluation quizzes, we have developed a pipeline to simplify the preparation of programming exercises using the RISC-V language. Our pipeline allows instructors to prepare questions and unit tests to verify the questions. We have been using this environment at the University of Turin for two academic years, with more than 1000 students already using it for both self-study and exams. We describe our tools and initial experience using them. We contribute with our source code and a large database of questions open to the community as open source.

Keywords – RISC-V, Moodle quiz, Code Runner, adaptive teaching, automated exams.

1 INTRODUCTION

The RISC-V Instruction Set Architecture (ISA) [1] has garnered significant recognition in recent years as an open-source, versatile, and modular computing framework. Designed to accommodate a wide array of applications, ranging from resource-constrained embedded systems to high-performance supercomputers, RISC-V stands out as a choice for teaching computer architectures and assembly programming. The fact the assembly code is simple and extensible makes it particularly well-suited for educational purposes, leading to its adoption in various textbooks [2].

Yet, learning assembly programming in general is a challenging task, primarily due to the inherently middle-level nature of this type of computer language, which uses predefined words called mnemonics that describe actions and operands like cpu registries or memory addresses, composing algorithms that can be directly translated to the low-level machine language [11]. Students often find it daunting to validate the correctness of their code and execute their programs effectively, creating a high entry barrier that can hinder the learning experience. Traditionally, courses in assembly programming have turned to simulators and emulators to mitigate these complexities, enabling students to gain hands-on experience without the intricacies of real machine execution. Nonetheless, these tools often lack the means to provide timely and informative feedback to students regarding their projects.

We here share our experiences in addressing these challenges by utilizing Moodle as an instructional platform for RISC-V assembly programming. We have generated an extensive repository of educational materials, including selfassessment quizzes as well as developed a pipeline for simplifying

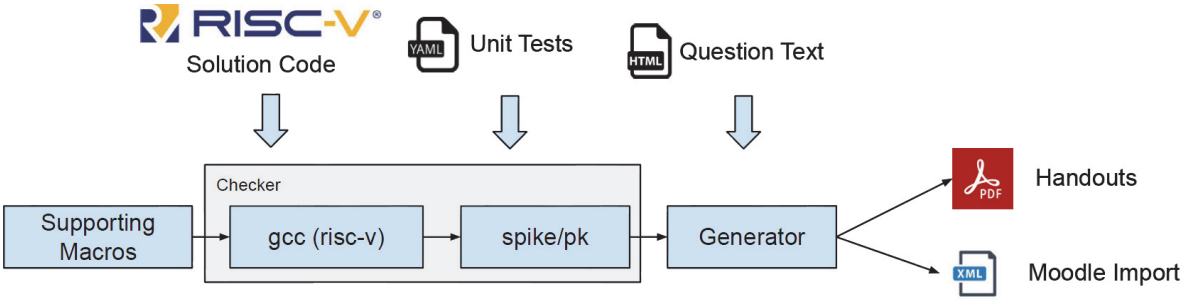


Figure 1 – Question preparation pipeline. The "Checker" module is also installed in the Moodle CodeRunner back-end (Jobe) to validate students' submissions

the creation of programming exercises that use the RISC-V assembly language. Our pipeline helps instructors design meaningful and engaging questions, along with unit tests to evaluate student responses using CodeRunner [7]. Over the course of two academic years, the solution has been used at the University of Turin, engaging over 1000 students in self-study and exams.

Here we provide a brief description of the developed tools and share our initial experiences with their application. This analysis has required new instruments to extract data about the students' engagement with the platform. We thus have developed a new Moodle plugin to allow us to export detailed statistics on the students' tentatives history when engaging with the CodeRunner questions. To broaden the impact of our work, we make the source code of our question preparation pipeline, our database of RISC-V questions (https://github.com/idrago/ArchI-exercise), and our Moodle plugin available to the community as open-source (plugin will be submitted to the Moodle plugin database). By doing so, we aim to contribute to the teaching of RISC-V and assembly programming, ultimately facilitating a more effective and accessible learning experience for students and educators alike.

2 ARCHITECTURE AND SYSTEM DESIGN

2.1 The pipeline from the teacher' point of view

Figure 1 shows the pipeline used for preparing the RISC-V programming questions in our system. The main components are:

- Checker: a Python module that takes as input
    - a RISC-V source code containing the function to be tested and a test driver (main) that calls the particular function,
    - a source file with supporting macros (common to all questions), and
    - a set of unit tests. It runs all unit tests on the code using the test driver and produces a report of the results in textual form. The Checker is also installed in the Moodle CodeRunner backend (Jobe) to validate students' submissions.
- Generator: a second Python module that takes as input the same files as above and an additional HTML file containing the question text. It produces PDF handouts (using pandoc) and the XML file to import the question into Moodle.

The Checker is the key module of the pipeline. We use a state-of-the-art compiler (GCC) for cross-compiling RISC-V code and run it in a simulation environment [3]. We use specifically Spike [9] as a simulator, with the RISC-V's Proxy Kernel (pk)[10] to proxy system calls and execute programs in a Linux kernel over a different architecture. Given the above setup, the process of preparing a question involves the writing of three input files:

5. *the question text;*

6. *the solution code;*
7. *the unit tests.*

The question text is an ordinary HTML document that can contain any HTML tags, from font formatting to links and embedded images. Figure 2 provides an example of a solution file that must be provided by the instructor. The solution code is a RISC-V assembly program that contains the function to be tested and a test driver (main). The code must contain a standalone program ready to run on Spike, and it may use several macros that are available to help verify the correctness of the solution.

```
1         #include "support.S"
2
3         ###GLOBALEXTRA###
4         # Driver
5         _start:
6             # puts numbers on the saved registers
7             start_saved_registers_leaf
8
9             # Call the student function
10            la   a0, array
11            la   a1, x
12            lw   a1, 0(a1)
13            la   a2, y
14            lw   a2, 0(a2)
15            jal  ra, equal
16
17            # print results
18            printreg a0, __ra0
19
20            # check whether the saved registers are preserved
21            check_saved_registers
22
23        ###ANSWER###
24        # Procedure equal(array, x, y)
25        # a0 -> address of array
26        # a1 -> index x, a2 -> index y
27        # return 1 if array[x] == array[y], 0 otherwise
28        equal:
29            # load array[x] in t0
30            slli t1, a1, 2
31            add  t1, a0, t1
32            lw   t0, 0(t1)
33
34            # load array[y] in t1
35            slli t1, a2, 2
36            add  t1, a0, t1
37            lw   t1, 0(t1)
38
39            # compare t0 and t1
40            bne  t0, t1, equal_false
41            li   a0, 1
42            jr   ra
43        equal_false:
44            li   a0, 0
45            jr   ra
46
47        ###TESTCASE###
48        .section .data
49            array: .word 1,1,2,2,3,4,4,1
50            x:     .word 0
51            y:     .word 1
52
```

Figure 2: Example of source code provided during the preparation of a question. Here students are asked to write a function that compares two elements of an array. The driver uses macros present in the support.S file to initialize the registers with arbitrary values, print the content of

the register a0 that must contain the function return value, and check whether the student solution respects the RISC-V calling conventions. Annotations are used to mark the solution, the driver, and the position where the unit tests will be inserted by the Checker.

For example, we provide macros to initialize the registers with arbitrary constants – e.g., useful to check whether students' solutions assume that registers would contain particular values when the program starts – to print the content of the registers as well as to check RISC-V calling conventions. The source code must be annotated with special tags that mark what should be considered i) the solution of the exercise, ii) the test driver, and iii) the position where unit tests should be inserted when verifying the solution. In the source code, the test cases are positioned at the end of the file. The Checker will replace the static test with the test cases provided by the instructor.

```
---
name: "equal"
preload: "equal:"
testcases:
  - testcode: |
      .section .data
          array: .word 1,1,2,2,3,4,4,1
          x:     .word 0
          y:     .word 1
    expected: "a0: 1"
    display: "HIDE"
  - testcode: |
      .section .data
          array: .word 1,1,2,2,3,4,4,1
          x:     .word 1
          y:     .word 2
    expected: "a0: 0"
    display: "HIDE"

```

(a) Check register value

```
---
name: "arraycpy"
preload: "arraycpy:"
testcases:
  - testcode: |
      .section .data
        array1: .half 8,5,3,7,2,6,4,1
        array2: .half 0,5,3,7,2,6,4,0
        size:  .word 8
    expected: "array = 8 5 3 7 2 6 4 1"
    display: "HIDE"
  - testcode: |
      .section .data
        array1: .half -8,-5,-3,-7
        array2: .half -8,-5,-3,-7
        size:  .word 4
    expected: "array = -8 -5 -3 -7"
    display: "HIDE"

```

(b) Check array content

Figure 3: Examples of unit tests checking different properties.

The test cases are written in YAML and contain a piece of code for the test, the expected output of the driver, and the display mode, which controls whether the case should be shown as feedback to students or not.

Figure 3 shows two examples of unit tests. In the first, relative to the program seen in Figure 3(a), the instructor has specified different arrays and indexes to the function, and the tests verify whether the function returns on the register a0 the expected value. In Figure 3(b) the tests for a more complex example are shown, in which the the driver prints the contents of an array, which are verified against the expected values.

2.2 Technical details on implementation

For the purposes of this experimentation, we had two goals:

- having all the RISC-V environment directly available from Moodle to simplify the interaction for teachers and students
- having a simple way to extract quiz attempts history information for each quiz in a manageable format for extracting statistics informations on RISC-V tools usage.

The first goal was achieved enhancing the yet running coderunner cluster [6], by adding to the dockered "JobeInABox" [8] the needed packages to build and run RISC-V programs: the Spike RISC-V ISA Simulator [9] and the RISC-V Proxy Kernel and Boot Loader [10].

An example of added code near the end of the jobeinabox dockerfile based on Ubuntu 22.04 follows:

```
#
# For RISC-V purposes
#
RUN apt-get install -y gcc-riscv64-unknown-elf
RUN apt-get install -y gcc-riscv64-linux-gnu
WORKDIR /root
RUN git clone https://github.com/riscv-software-src/riscv-isa-sim
RUN git clone https://github.com/riscv-software-src/riscv-pk
RUN apt-get install -y device-tree-compiler
RUN cd riscv-isa-sim && \
        mkdir build && \
        cd build && \
        ../configure --prefix=/opt/RISCV && \
        make && \
        make install
RUN cd riscv-pk && \
        mkdir build && \
        cd build && \
        ../configure --prefix=/opt/RISCV --host=riscv64-linux-gnu && \
        make && \
        make install
ENV PATH="/opt/RISCV/bin:/opt/RISCV/riscv64-linux-gnu/bin:${PATH}"
RUN cd /usr/bin; ln -s /opt/RISCV/riscv64-linux-gnu/bin/pk .
```

These additions gives the RISC-V functionalities to CodeRunner server without the complexity of adding a new language; deploying the docker image built using the modified dockerfile to each of the cluster's nodes it's done with standard docker image save/load and subsequently jobeinabox instances run with standard configuration.

Our cluster was proven to support 3/400 contemporary users doing exams with optimal performances and no interruptions of service [6].

The second goal was fulfilled by creating a quiz report plugin capable of extracting the attempts history of each student for a quiz. This task could be also done with the "Configurable Reports" plugin, but stated that having a downloadable format of the attempts histories could be a remarkable feature not only for us, but for every teacher doing exams, we opted for building a quiz report that export the history data in CSV format, which it's automatically available to all teachers.

The teacher could process the downloaded attempts history in CSV format to extract useful information on effective path followed by the students doing a quiz, focusing on particular phenomena and exploiting autonomous learning analytics on quiz attempts.

The plugin mimics the "export quiz attempts" plugin behaviour, by adding a new menu item to the quiz results items called "Attempts export CSV" (figure 4), which in turn presents a classical quiz attempts choice where the action button starts the history download.

At the time of writing the plugin works with minimal optional features, but we would like to discuss with the people at the MoodleMoot about plugin functionalities to be added/removed, giving a chance to enhance this little but, we think, useful plugin.

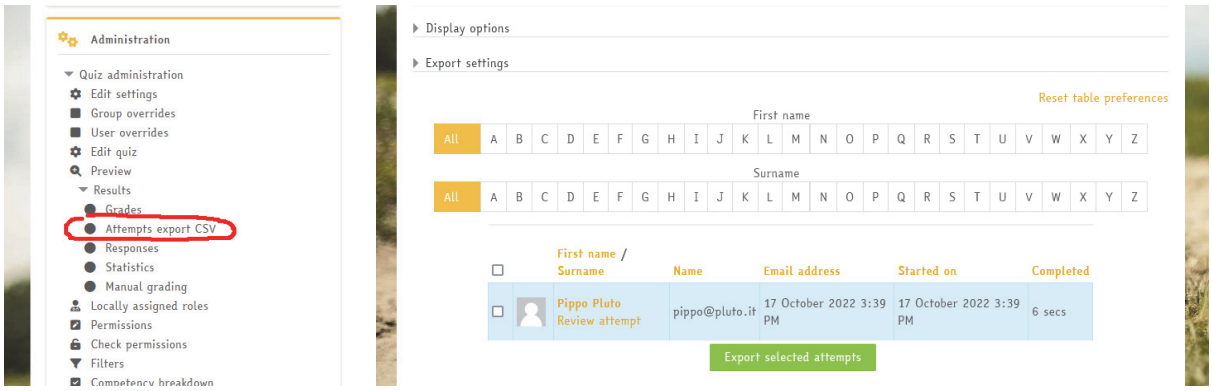


Figure 4: some screenshot of the Attempts export CSV Moodle plugin

3 EARLY DEPLOYMENT EXPERIENCE

3.1 Question Dataset

Our CodeRunner RISC-V solution and our question preparation pipeline have been used in two academic years at the University of Turin to support the course of Computer Architecture in the Computer Science Bachelor's degree. In the first year, the system was used primarily for exams, while in the second year, the question dataset was extended and self-assessment quizzes were put online during the semester. In total, the system has supported more than 1000 students in both exams and self-assessments. The dataset contains both classic quiz questions to test theoretical aspects and more than 50 programming questions produced with our pipeline. Our programming questions cover a wide spectrum of skills, including arithmetic operations (addition, subtraction, division), logic operations; memory load and store of different data types, control flow instructions (branches and jumps), stack manipulation, bit-wise operations, and addressing (e.g., immediate).

Most of the questions are written in Italian, but our goal is to extend and translate all questions to English soon. All programming questions follow the same pattern in which a main function is provided in the text of the question, and students are asked to write a function that is called by this main. The main function works as the driver and calls the student solution. Follow-up questions are also present in which students must develop more advanced functions, using their previously answered ones, to test their abilities to call nested functions.

3.2 Deployment

The Moodle plugin developed allows for the seamless export of students' complete attempt history in an exercise, capturing every action undertaken by both students and teachers for each attempt. Without this plugin, one would need to resort to direct database export [4]. The information exported by our plugin includes the initial pre-checking steps and any subsequent manual corrections made by teachers during the evaluation process. We have deployed the CodeRunner RISC-V solution in our production Moodle system. We evaluate results from three computer architecture exam seasons, which involved over 900 students' solutions to questions within approximately 500 quizzes. Students have access to an external simulator during the exams, known as RARS [5], offering them the freedom to develop and test their solutions independently. The use of Moodle CodeRunner to check their answers thus remains optional, while all submissions are later manually evaluated by teachers.

Notice that, during exams, we set up CodeRunner to provide only a semaphore indicating whether the solution is completely correct or not. No other feedback is given, and thus students have an incentive to use primarily RARS as a debugging environment.

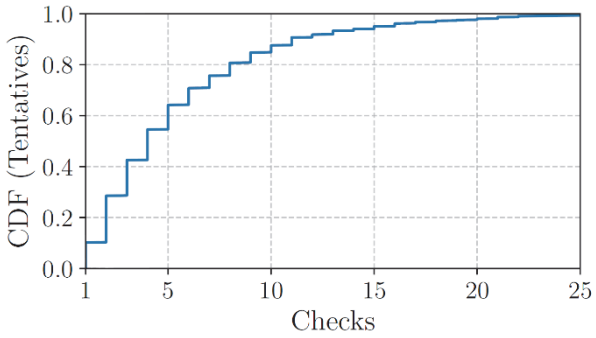
Next, we delve into usage patterns, tracking how students have engaged with our CodeRunner RISC-V solution. We scrutinize how their grading evolves from the initial submission to the final assessment, getting initial insights into the effect of feedback from CodeRunner usage in their exam trajectory.

4 EXPERIENCE ON EXAMS

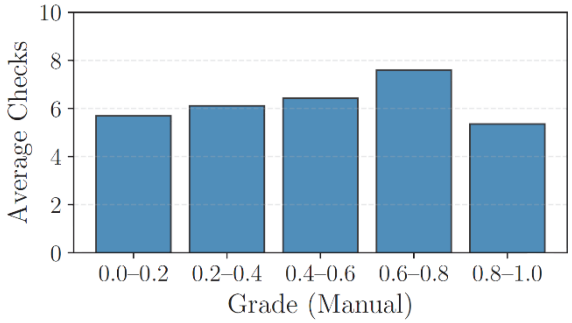
We characterize the use of our CodeRunner RISC-V solution. Figure 5a shows the number of corrections for each question in the evaluated exams. We observe that only a minor fraction of the attempts (around 10%) are closed with a single correction. These cases typically involve situations in which only the automatic system correction has been applied, and we manually confirm that they are mostly related to instances where students have not engaged with the given question.

In approximately half of the attempts, the system has been invoked 4-5 times. However, we also notice attempts in which students resort to the system dozens of times, with some attempts accumulating more than 40 corrections. Based on usage logs, these cases appear to be situations in which some students decide to skip the external simulator and develop their solutions directly within CodeRunner, even without detailed feedback.

To delve further into this data, Figure 5b displays the average number of checks performed by our system in relation to the final grade manually assigned by the teacher. While differences are subtle, interesting patterns emerge. Firstly,students who achieve the highest grades tend to rely less on our CodeRunner solution. These cases include several students who produce perfect solutions in RARS, using CodeRunner for a final check before closing the quiz. More interestingly, we see that students with partial but good grades are the ones relying the most on CodeRunner. The number of checks per attempt increases.



(a) Students' corrections per tentative



(b) Average checks per final grade ranges

Fig. 5: General statistics of students' manual checks observed during exams.

from approximately 6 for students receiving the lowest grades (0.0-0.2) to about 8 for those obtaining near-perfect grades (0.6-0.8).

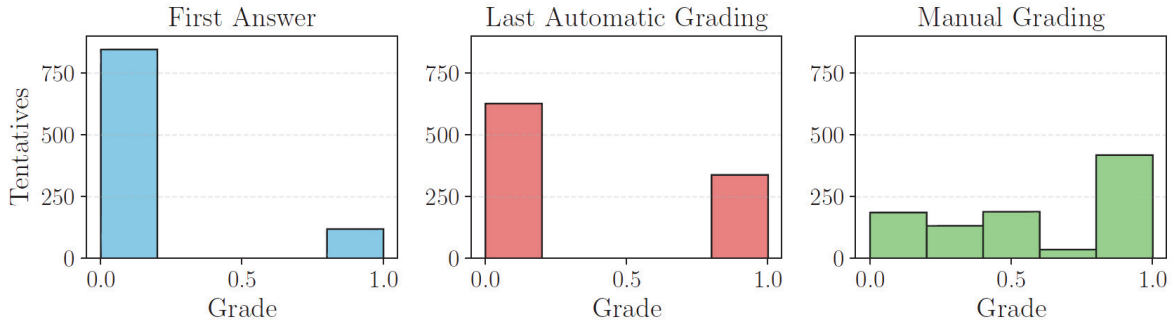


Fig. 6 - Grades i) in the first delivery; ii) in the last automatic grading; and iii) manually assigned by the instructor. Students use an external simulator during exams and CodeRunner checks are offered as an additional resource.

Figure 6 complements the analysis by showing the number of tentatives per grade range in three different moments: i) after the correction of the first students' delivery; ii) after the automatic correction of the last students' delivery; iii) after manual correction.

Comparing the two left-most plots, we see that the number of solutions with the best grades more than doubles from the first to the last delivery. This is somehow expected and can be partly explained by the curiosity of students who start to deliver solutions in CodeRunner even before properly debugging it in RARS. Yet, it demonstrates the interest of students in obtaining early feedback during exams, which is precisely the outcome we expected when building the CodeRunner RISC-V solution.

The right-most plot shows how the tentatives are distributed according to the final question grades manually assigned by teachers. As our CodeRunner solution cannot assign partial grades, it is expected that grades get distributed over the range when manually evaluated. It is interesting to observe how the number of students achieving near-perfect grades (0.6-0.8) is actually the smallest among all ranges. Recall that those are the students relying the most on the CodeRunner RISC-V solution to check their answers. That is, we observe students who heavily resort to the system – while also debugging in RARS – to try to improve their answers. This behavior again points to a positive feedback loop in terms of learning outcomes, since it pushes students to think more thoroughly about their answers.

5 CONCLUSIONS

We have outlined our efforts in integrating a CodeRunner RISC-V environment into Moodle, encompassing both a question writing pipeline and the Moodle back-end for automating the correction of students' submissions. Our solution has been utilized to support students in the Computer Architecture course at the University of Turin. We have successfully implemented this solution in our production Moodle installation for two academic years. Additionally, to facilitate the analysis of the impact of our solution, we have introduced a Moodle plugin to export detailed statistics regarding question attempts. Our data reveals that students actively engage with our CodeRunner RISC-V solution, both for self-assessment throughout the semester and during examinations. We consider this a positive outcome since students have access to an external RISC-V simulator for debugging (RARS). While it is premature to draw precise conclusions regarding its impact on learning outcomes, the observed level of engagement suggests a positive influence. In the future, we plan to expand our analysis by more effectively tracking progress from the beginning of the semester and incorporating students' qualitative feedback into the evaluation process.

BIBLIOGRAPHY

[1] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, D. Patterson, The risc-v instruction set manual, Volume I: User-Level ISA', version 2 (2014).

[2] D. Patterson, Computer organization and design risc-v edition: the hardware (2017).

[3] B. Goossens, Installing and using the risc-v tools, in: Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis, Springer, 2023, pp. 87–103.

[4] Moodle, Overview of the Moodle question engine (2023). https://docs.moodle.org/dev/Overview_of_the_Moodle_question_engine#Detailed_data_about_an_attempt

[5] RARS, RISC-V Assembler and Runtime Simulator (2023). https://github.com/TheThirdOne/rars

[6] Un assetto Moodle per l'esame online di un corso di programmazione, F. Cardone, S. Rabellino, L. Roversi, Atti del MoodleMoot Italia 2021 (2021), AIUM, 2021, pp. 45-52

[7] Moodle plugins directory: CodeRunner, https://moodle.org/plugins/qtype_coderunner

[8] Jobe In a Box, https://github.com/trampgeek/jobeinabox

[9] Spike RISC-V ISA Simulator, https://github.com/riscv-software-src/riscv-isa-sim

[10] RISC-V Proxy Kernel and Boot Loader, https://github.com/riscv-software-src/riscv-pk

[11] Computer Languages, https://www.cs.mtsu.edu/~xyang/2170/computerLanguages.html