# Software Construction (L+E) HS 2023

*Instructor:* Prof. Dr. Alberto Bacchelli — Assignment 3

*Teaching Assistant:* Konstantinos Kitsios — Week 09

---

To correctly complete this assignment you **must**:

- Carry out the assignment with the team you have formed for assignments 01 and 02. Work with your team only (unless otherwise stated). You are allowed to discuss solutions with other teams, but each team should come up its own personal solution. A strict plagiarism policy is going to be applied to all the artifacts submitted for evaluation.
- Prepare the solutions to the exercises by strictly following this structure:
  - A root folder named: `Group[id on OLAT]-a[AssignmentNumber]`[1]
  - Inside the folder:

    **`README.md`** : a Markdown file explaining the decisions taken in the source code and documents your solution;

    **`vm/`** : a folder that we provide you and contains the working implementation of the Virtual Machine as seen in Chapter 25. You will be asked to add features by modifying the files in this folder in exercise 3;

    **`debugger/`** : a folder that we provide you and contains the working implementation of the Debugger as seen in Chapter 26. You will be asked to add features by modifying the files in this folder in exercise 4;

    **`exercise 1/`** : a folder that you will create containing the necessary files for exercise 1;

    **`exercise 2/`** : a folder that you will create containing the necessary files for exercise 2. It should contain at least a `disassemble.py` and an `input_file.mx` file, plus any files you find useful for the Testing part;

    **`exercise 3/`** : a folder that you will create containing the necessary files for exercise 3, named `example_3_i.as`, $\texttt{i} \in \{1, 2, 3\}$;
- The final structure of the directory should match that in Figure 1 [2]
- Package your root folder into a single ZIP file named: `Group[id on OLAT]-a[AssignmentNumber].zip`[3]
- Upload the solution to the right OLAT task by the deadline (*i.e.*, **Dec 05, 2023 @ 18:00**)

---

[1] *e.g.*, a correct name would be: `Group99-a3`

[2] The 3 dots (...) mean that there is no strict structure in this subfolder, you are free to choose the structure that suits you best

[3] *e.g.*, a correct name would be: `Group99-a3.zip`

Figure 1: Directory Structure

```
Groupg99-a3/
├── README.md
├── vm/
│   ├── architecture.py
│   ├── arrays.py
│   ├── assembler.py
│   ├── count_up.as
│   ├── count_up.mx
│   └── vm.py
├── debugger/
│   ├── architecture.py
│   ├── assembler.py
│   ├── count_up.as
│   ├── count_up.mx
│   ├── vm_base.py
│   ├── vm_break.py
│   ├── vm_extend.py
│   └── vm_step.py
├── exercise 1/
│   └── ...
├── exercise 2/
│   ├── disassembler.py
│   ├── example_input.mx
│   └── ...
└── exercise 3/
    ├── example_3_1.as
    ├── example_3_2.as
    └── example_3_3.as
```

## Goal

The goal of this assignment is to deepen your understanding of Virtual Machines and Debuggers by enhancing the implementation of the book. These notions are daily encountered by software engineering professionals, yet they are usually thought of as black-boxes. Furthermore, you will get your hands dirty with **pytest**, a popular testing framework for Python.

## Prerequisites

Before starting this assignment, you should read and understand the Chapters on Virtual Machines and Debuggers, both theoretically and practically, by playing around with the code provided in the book and/or the lectures.

## Grading Criteria

Your assignment will be graded based on the following criteria:

- Correctness of implementation.
- Correctness of design process: document your design decisions in the `README.md` file.

Assignments will be scored based on two main factors: fulfillment of criteria and evidence of effort:

- **2 out of 2:** For assignments that meet all criteria and clearly demonstrate substantial effort.
- **1 out of 2:** For assignments that meet some criteria but still show evident effort.
- **0 out of 2:** For assignments that lack clear evidence of effort, regardless of criteria met.

# 1 Unit Testing

### Implementation

You are requested to write unit tests for the Assembler and the Virtual Machine implemented in Chapter 25 and Chapter 26 respectively. You will do this by using `pytest` which is a popular, simple yet powerful testing framework for the Python language. More specifically:

(A) To test the Assembler:
  (i) write simple assembly programs of your choice (`.as` files);
  (ii) manually calculate the output VM instructions for each input assembly program;
  (iii) run the assembly programs through the Assembler to produce one `.mx` file for each `.as` file;
  (iv) compare the output of the Assembler to what you calculated in step (ii).

(B) To test the Virtual Machine:
  (i) assume that the Assembler is correct (how much more complicated would things be without this assumption?);
  (ii) run the `.mx` files you produced in step (A)(iii) above through the VM;
  (iii) compare the output of the VM with the expected output of each program you chose in step (A)(i).

(C) In (B) you tested the VM from a machine code perspective. Here, you will also test 2 common errors that occur during software development:
  (i) Out-of-memory error; since our VM only has a finite amount of RAM, it is expected that if a user requests to allocate space for an array that exceeds this amount, the program should crash. Write a test case for this scenario.
  (ii) Instruction-not-found error; write a test case for the scenario where a given `.mx` file contains a code that corresponds to unknown instruction.

(D) Measure and report the **test coverage** in % for the above tests. Measuring test coverage with `pytest` is very simple and plenty of online resources are written on this, you can take a look here for example. Whichever syntax you choose for measuring line coverage, describe it in the `README.md` file for reproducibility.

Do not try to write too complex assembly programs, or step (A) will get complicated. Aim at small programs that test specific groups of instructions **and make sure you test each one of the 11 instructions once**. The number and the depth of tests is up to you, as long as all 11 instructions are covered.

### Use

Place your file(s) in the `exercise 1/` subfolder (see Figure 1). The `pytest` framework allows for a certain degree of freedom in the directory and file structure, so the only requirement here is that when you run the terminal command

```
pytest
```

from the `exercise 1/` subfolder, all the designated tests should run and print their output in the terminal.

# 2 Disassembler

## Implementation

Write a disassembler that takes 2 inputs:

1. a `.mx` file containing VM instructions
2. a `.as` file where the output will be written

and converts the VM instructions to assembly code.

## Use

You should be able to run your code from command line with the below command

```
python disassemble.py input_file.mx output_file.as
```

Write your code in `disassemble.py` and showcase its functionality by implementing an `input_file.mx`.

## Testing

Finally, write test cases in `pytest` to ensure the correctness of your disassembler. As in exercise 1, the number of tests and the file structure is up to you, as long as the terminal command

```
pytest
```

when executed from the `exercise 2/` subfolder, runs all the designated tests and prints the output in the terminal.

# 3 New features and Problems - Assembler

## 3.1 Increment and Decrement

### Implementation

Add instructions `inc` and `dec` to our assembly language that add one to the value of a register and subtract one from the value of a register respectively.

### Use

To do so, modify the `architecture.py` and `assembler.py` files in the `vm/` directory. To showcase the functionality, write `example_3_1.as`, an assembly program of your choice that runs with the following commands from the terminal

```
python assembler.py example_3_1.as output_3_1.mx

python vm.py output_3_1.mx -
```

## 3.2 Swap values

### Implementation

Add instruction `swp R1 R2` that swaps the values in R1 and R2 without affecting the values in other registers.

### Use

To do so, modify the `architecture.py` and `assembler.py` files in the `vm/` directory. To showcase the functionality, write `example_3_2.as`, an assembly program of your choice that runs with the following commands from the terminal

```
python assembler.py example_3_2.as output_3_2.mx

python vm.py output_3_2.mx -
```

## 3.3 Reverse array in place

### Implementation

Write an assembly language program that starts with:

- the base address of an array in one word
- the length of the array N in the next word
- N values immediately thereafter

and reverses the array in place

### Use

To showcase the functionality, write `example_3_3.as`, an assembly program of your choice that runs with the following commands from the terminal

```
python assembler.py example_3_3.as output_3_3.mx

python vm.py output_3_3.mx -
```

# 4 New features - Debugger

Implement the following features for the debugger by modifying necessary files in the `debugger/` directory (see Figure 1). You are not required to showcase these functionalities in separate files because of their interactive nature, so all you have to do is change the files in the `debugger/` folder and provide instructions in the `README.md` file on how to test each new feature interactively. For you development and testing needs, you can use the `count_up.as` and `count_up.mx` file that we provide or create your own ones.

## 4.1 Show Memory Range

Modify the debugger so that if the user provides a single address to the ”memory” command, the debugger shows the value at that address, while if the user provides two addresses, the debugger shows all the memory between those addresses.

## 4.2 Breakpoint Addresses

Modify the debugger so that if the user provides a single address to the ”break” or ”clear” command, it sets or clears the breakpoint at that address.

## 4.3 Command Completion

Modify the debugger to recognize commands based on any number of distinct leading characters. For example, any of ”m”, ”me”, ”mem”, and so on should trigger the `_do_memory` method. Programmers should not have to specify all these options themselves; instead, they should be able to specify the full command name and the method it corresponds to, and the VM’s constructor should take care of the rest.

## 4.4 Watchpoints

Modify the debugger and VM so that the user can create watchpoints, i.e., can specify that the debugger should halt the VM when the value at a particular address changes. For example, if the user specifies a watchpoint for address 0x0010, then the VM automatically halts whenever a new value is stored at that location.