Documentation

Berea Manuela-Mihaela **Popa Bianca-Ştefana**

Neculea Gabriela **Sbârcea Ștefan-Vladimir**

Contents

1 Introduction **2**

2 Terms definitions **2**

3 Main classes **3**

3.1 Processor . . . 3

3.1.1 Attributes: . . . 3

3.1.2 Methods: . . . 4

3.2 Memory . . . 5

3.2.1 Attributes: . . . 5

3.2.2 Methods: . . . 5

3.2.3 Memory validation methods: . . . 6

3.3 Keyboard . . . 6

3.3.1 Attributes: . . . 6

3.3.2 Methods: . . . 7

3.4 Screen . . . 7

3.4.1 Attributes: . . . 7

3.4.2 Methods: . . . 7

3.5 GUI . . . 7

3.5.1 Attributes: . . . 7

3.5.2 Methods: . . . 8

3.6 Custom Exceptions classes . . . 8

3.6.1 DivisionByZeroException . . . 8

3.6.2 InvalidMemoryAddrError . . . 8

3.6.3 MemoryOverflowError 8

4 Brief User Manual **9**

4.1 Installation 9

4.2 Usage 9

4.3 Assembly-Like Language Syntax 10

5 Unit tests **11**

5.1 Setup on Windows PowerShell 11

5.2 Run tests 11

5.3 Coverage 13

6 Assertions **14**

6.1 Methods that check the preconditions 14

7 Contributors **15**

1 Introduction

The project is a simulation software written in **Python 3.12.0** that emulates the behavior of a computer system's main components, including the **processor**, **memory**, **keyboard**, and **screen** peripherals. It provides users with the capability to write assembly-like instructions, manipulate data, perform arithmetic and logical operations, control program flow, and interact with peripheral devices through a graphical interface built using **tkinter**.

2 Terms definitions

- **Register** = Small, fast storage locations directly within the CPU, used to hold temporary data that the processor needs during the execution of programs, such as operands for arithmetic operations, address pointers for accessing memory, or intermediate results and special states of the machine. These registers are much faster to access than main memory, which is why processors use them for immediate operations.

- **Conditional Flags** = Used for decision-making (e.g., comparison results), reflecting how real CPUs manage operations and control flow. They are set or cleared after the outcome of each operation. More about conditional flags

```
self.flags = {
    'CF': False,  # Carry Flag: is set when an arithmetic
       operation generates a carry or borrows from the most
       significant bit; used in testing for overflow in signed
       integer arithmetic
    'PF': False,  # Parity Flag: is set only when the least
       significant byte of the result has an even number of 1
       bits
    'ZF': False,  # Zero Flag: is set when the result of an
       operation is 0
    'SF': False,  # Sign Flag: holds the value of the most
       significant bit of the result; indicates the sign of a
       signed integer (0 = positive, 1 = negative)
    'OF': False   # Overflow Flag: is set when an overflow
```

```
        occurs in signed integer arithmetic
}
```

- **Program Counter (PC)**: Points to the next instruction to execute.
- **Stack Pointer (SP)**: Points to the current top of the stack in memory, used for push/pop operations. More about stack in assembly

3 Main classes

3.1 Processor

- holds data registers and flags
- executes instructions (assignment, addition, subtraction, multiplication, division, boolean op, comparison, jumps, push/pop and function call/return using a part of memory as stack)

The processor reads and parses the entire instruction set from the file and stores it in the instruction memory at the start. During execution, the processor fetches each instruction based on the program counter (PC), processes it, and then moves to the next one.

The keyboard buffer is allocated a single address - this address is part of the data memory and is **checked by the processor** to see if there's any new input (the processor will periodically check this address to see if it contains any new data (non-zero value), process the data if present, and then reset the buffer).

3.1.1 Attributes:

- `data_registers`: List to store the values of the 8 data registers.
- `flags`: Dictionary to store the values of conditional flags (CF, PF, ZF, SF, OF).
- Special registers:
  - `stack_pointer`: List to simulate stack operations. (the stack pointer is in the list so we don't need to store it separately)
  - `program_counter`: Points to the current instruction being executed.

- Additional attributes:
  - `memory`: Reference to the memory class to access memory locations.
  - `instruction_types`: Dictionary mapping instruction names to their respective methods.
  - `is_file_parsed` (bool): Indicates whether the file has been parsed.
  - `file_name` (str): Name of the file containing instructions.
- Keyboard input handling:
  - `is_reading_input` (bool): Indicates whether the processor is waiting for keyboard input.
  - `input` (str): Keyboard input string.
  - `input_destination` (str): Destination operand for keyboard input.

3.1.2 Methods:

- `set_file_name`: Set the name of the file containing instructions to be executed.
- `execute_instruction(instruction)`: Executes a single instruction.
- `execute_program(file_name)`: Execute instructions from a file sequentially.
- `parse_instruction(instruction)`: Parses a single instruction from the program file and adds it to the instruction list. (in case of a label, it stores the label and its corresponding instruction index in the labels dictionary).
- `parse_file(file_name)`: Reads the program file and parses each instruction.
- `parse_memory_operand`: Parses a memory operand to determine its address.
- `check_register_index(index)`: Checks if the given index is a valid register index.
- `get_operand_value(operand)`: Returns the value of an operand (register, memory location, or constant value).
- `store_result(destination, result)`: Stores the result of an operation in the destination operand.

- `assert_16_bit(value)`: Truncates a value to fit within 16 bits.
- `reading_input`: Handles keyboard input.
- `get_memory_data(operand, destination)`: Gets the value of a memory operand. (starts reading input if keyboard buffer is accessed)
- `convert_keyboard_input(keyboard_input)`: Converts keyboard input to a numerical value or ASCII value.
- Methods for various instruction types such as `mov`, `add`, `sub`, `mul`, `div`, `cmp`, `jmp`, `je`, `jne`, `jg`, `jl`, `jge`, `jle`, `push`, `pop`, `call`, `ret`, `not_op`, `and_op`, `or_op`, `xor_op`, `shl`, `shr`.

3.2 Memory

- stores both instruction and data memory
- handles read/write operations

3.2.1 Attributes:

- `MAX_MEMORY_SIZE`: Maximum memory size in bytes.
- `MIN_MEMORY_SIZE`: Minimum memory size in bytes.
- `instruction_memory`: List to store program instructions.
- `data_memory`: List to store data, including special areas for peripherals.
- `keyboard_buffer_address`: Address of the keyboard buffer.
- `video_memory_start`: Start address of video memory.
- `video_memory_end`: End address of video memory.
- `labels`: Dictionary to store labels and their corresponding addresses.

3.2.2 Methods:

- `set_keyboard_pointer(ptr)`: Sets the pointer to the Keyboard instance in the keyboard buffer.

- `get_keyboard_pointer`: Gets the Keyboard instance from the keyboard buffer.
- `read_video_memory()`: Reads the content of video memory.
- `get_instruction(address)`: Retrieves instruction from the given address.
- `add_instruction(instruction, label=None)`: Adds an instruction to the instruction memory.
- `set_data(address, value)`: Sets data at the specified address in data memory.
- `get_data(address)`: Gets data from the specified address in data memory.
- `goto_label(label)`: Jumps to the address associated with the specified label.

3.2.3 Memory validation methods:

- `check_instruction_memory_overflow(address)`: Checks for overflow in instruction memory.
- `check_instruction_memory_address(address)`: Checks if the address is within bounds of instruction memory.
- `check_data_memory_overflow(address)`: Checks for overflow in data memory.
- `check_memory_address(address)`: Checks if the address is within bounds of data memory.
- `validate_memory_size(size)`: Validates the memory size.

3.3 Keyboard

Represents a simulated keyboard peripheral device.

3.3.1 Attributes:

- `key_queue` (deque): A queue to store characters pressed on the keyboard.

3.3.2 Methods:

- `input_character(character)`: Simulates inputting a character into the keyboard buffer.
- `get_next_character()`: Retrieves the next character from the keyboard buffer.
- `has_characters()`: Checks if there are characters in the keyboard buffer.

3.4 Screen

Represents a screen simulated by video memory.

3.4.1 Attributes:

- `width` (int): The width of the screen.
- `height` (int): The height of the screen.

3.4.2 Methods:

- `__init__(width, height)`: Initializes the Screen object with the specified width and height.

3.5 GUI

Represents the graphical user interface for simulating peripheral devices.

3.5.1 Attributes:

- `processor`: Instance of the Processor class.
- `keyboard`: Instance of the Keyboard class.
- `screen`: Instance of the Screen class.
- `buttons_per_row` (int): Number of buttons per row in the keyboard frame.
- `interval` (int): Time interval in milliseconds for updating the screen.
- `root`: Tkinter root window.
- `screen_frame`: Frame containing the screen display.

- `screen_text`: Text widget displaying the screen content.
- `keyboard_frame`: Frame containing the keyboard buttons.

3.5.2 Methods:

- `__init__(memory, keyboard, screen, buttons_per_row=16)`: Initializes the GUI with memory, keyboard, and screen instances.
- `select_asm_file()`: Opens a file dialog to select an assembly file.
- `run_program()`: Executes the program loaded in the processor and updates the screen.
- `create_keyboard_buttons()`: Creates keyboard buttons for character input.
- `create_button(char_str, row, col, name=None, width=5, height=2)`: Creates a button with the specified character, row, column, name, width, and height.
- `key_press(char)`: Handles keyboard button press events.
- `update_screen()`: Updates the screen display with the content from video memory.

3.6 Custom Exceptions classes

3.6.1 DivisionByZeroException

- Raised when a division operation is attempted with a divisor of zero.

3.6.2 InvalidMemoryAddrError

- Raised when an invalid memory address is accessed.

3.6.3 MemoryOverflowError

- Raised when memory overflow occurs.

4 Brief User Manual

4.1 Installation

- Clone the repository: `git clone https://github.com/Vladimir-SS/CSS-Project.git`
- Install Python 3.12.0 or higher. Download Python
- Install graphical user interface library `tkinter` (usually included in Python distributions). If not installed, run `pip install tk`.
- Run the program: `python ./src/main.py`

4.2 Usage

- The program reads instructions from a file and executes them sequentially.
- The instructions are written in an assembly-like language syntax. To execute instructions, select a file containing the instructions by pressing the "Select Assembly File" button. (The file must have `.asm` extension to be visible in the file selection dialog). See below for supported instructions.
- You can change the `./src/config.cfg` file to modify the memory sizes and special memory locations. Make sure to maintain the same format:
  - **instruction_memory_size:** *<number multiple of 1 KB, not exceeding 65536>*
  - **data_memory_size:** *<number multiple of 1 KB, not exceeding 65536>*
  - **keyboard_buffer:** *<number multiple of 1 KB, not exceeding 65536, not between **video_memory_start** and **video_memory_end***>
  - **video_memory_start:** *<number, not exceeding data_memory_size>*
  - **video_memory_end:** *<number, not exceeding data_memory_size>*
- The program will show on the screen any data found in the video memory.
- The UI keyboard saves the input and whenever a read instruction (e.g., `MOV R0, M<keyboard_buffer_memory_location>`) is executed, the program will read the input from the keyboard waiting for an enter key press.

4.3 Assembly-Like Language Syntax

The program supports a simplified assembly-like language with the following instructions:

- **Assignment:** `MOV destination, source`
- **Arithmetic operations:**
  - `ADD destination, source`
  - `SUB destination, source`
  - `MUL destination, source`
  - `DIV destination, source`
- **Comparison:** `CMP operand1, operand2`
- **Jumps:**
  - `JMP label`
  - `JE label`
  - `JNE label`
  - `JG label`
  - `JL label`
  - `JGE label`
  - `JLE label`
- **Stack operations:**
  - `PUSH source`
  - `POP destination`
  - Function call/return:
    - `CALL label`
    - `RET`
- **Boolean operations:**
  - `NOT destination`

- AND destination, source
- OR destination, source
- XOR destination, source
- SHL destination, source
- SHR destination, source

- **Data types:**
  - Data registers: R0, R1, R2, R3, R4, R5, R6, R7
  - Memory locations: M<value>/<register>
  - Constant values: #<value>

There are also 2 example files in the ./src directory asm-example-file.asm and asm-example-file-2.asm that you can use to test the program.

5 Unit tests

- The unit tests are written using the unittest library.
- The test files are located in the tests directory. And the test files are named after the classes they test (e.g., test_processor.py for the Processor class).

5.1 Setup on Windows PowerShell

```
$env:PYTHONPATH += ";C:\Users\path\to\project\src"
$env:PYTHONPATH # to check if the path is added
```

5.2 Run tests

```
python -m unittest # in the root directory
# or
pip install pytest
pytest # in the root directory

# For coverage
```

```
coverage run -m pytest
coverage report
coverage html
# or
pip install pytest-cov
pytest --cov=src tests/ # to check coverage
```

5.3 Coverage

Table 1: Coverage

Name	Stmts	Miss	Branch	BrPart	Cover
src\GUI.py	68	53	16	0	18%
src\Keyboard.py	12	1	2	1	86%
src\Memory.py	89	10	30	8	83%
src\Processor.py	329	76	136	35	71%
src\Screen.py	4	0	0	0	100%
src_ _init_ _.py	0	0	0	0	100%
src\exceptions\DivisionByZeroException.py	3	0	0	0	100%
src\exceptions\InvalidMemoryAddrError.py	3	0	0	0	100%
src\exceptions\MemoryOverflowError.py	3	0	0	0	100%
src\exceptions_ _init_ _.py	0	0	0	0	100%
src\tests_ _init_ _.py	0	0	0	0	100%
src\tests\test_gui.py	44	16	8	2	65%
src\tests\test_keyboard.py	15	1	2	1	88%
src\tests\test_memory.py	75	5	28	3	88%
src\tests\test_processor.py	290	2	24	1	99%
src\tests\test_screen.py	9	1	2	1	82%
TOTAL	944	165	248	52	78%

Name: Python file name, including the relative path.

Stmts: Total executable statements (lines of code).

Miss: Number of statements not executed during tests.

Branch: Number of branch points (if-else conditions, loop exits).

BrPart: Number of partially tested branches (some paths not executed).

Cover: Percentage of executed statements out of total statements.

6 Assertions

- Since most of the program already had custom assertions to check the preconditions and assure the correct behavior, we added only a few more to check the correctness of the program execution.

6.1 Methods that check the preconditions

- Processor class - `check_register_index(index)`: Checks if the given index is a valid register index.
- Processor class - `assert_16_bit(value)`: Truncates a value to fit within 16 bits.
- Memory class - `check_instruction_memory_overflow(address)`: Checks for overflow in instruction memory.
- Memory class - `check_instruction_memory_address(address)`: Checks if the address is within bounds of instruction memory.
- Memory class - `check_data_memory_overflow(address)`: Checks for overflow in data memory.
- Memory class - `check_memory_address(address)`: Checks if the address is within bounds of data memory.
- Memory class - `validate_memory_size(size)`: Validates the memory size.

7 Contributors

- Berea Manuela-Mihaela
  - **Contribution:** Processor - cmp, jumps, functions, and modified Memory to support labels
- Neculea Gabriela
  - **Contribution:** Memory, Keyboard and Processor mostly the data saving in memory
- Popa Bianca-Ştefana
  - **Contribution:** Processor - arithmetic, boolean operations and instruction parsing
- Sbârcea Ștefan-Vladimir
  - **Contribution:** GUI, Screen and modified Processor instruction execution to run synchronously with the GUI

Note: Each team member contributed to the documentation and the tests based on the classes they implemented.