The following report is a description of the implementation of an instruction-set architecture ISA simulator in Python.

Solution approach

While designing the solution we wanted to approach it in a way that the code it well structured, readable and concise. For this reason we created following elements in the program:

- State - class, which is instantiated only once and effectively is a singleteon. It represents state of the program.
- get_cycle_data - function which reads the data for the current address of the program
- execute_instruction - function which executes data read by get_cycle_data. This function has a neat match sattement which is responsible for executing an instruction based on its 'opcode'.
- while loop - calls two above functions as long as certain conditions are met.

Program flow

While loop

The While loop starts with a condition that checks if the number of current cycles is less than or equal to the maximum number of cycles, and that the attribute of state.is_terminated is not True.

```
current_cycle = 0
while (current_cycle <= max_cycles
       and not state.is_terminated):
```

```
    case 'END':
        state.is_terminated = True
```

Reading cycle data

If trhe above conditions are fulfilled, then the get_cycle_data() function is executed. The function returns three pieces of data in total, from instruction memory and state:

- instruction - code of an instruction which should be executed
- operators - list ofparameters passed for the instruction set
- values - values which are represented by the operators

Visulisation of a piece of the running program: Instruction; Operators; Values

```
ADD; ['R2', 'R2', 'R3'];  [19, 19, 1]
JLT; ['R9', 'R2', 'R7'];  [20, 20, 20]
JEQ; ['R8', 'R4', 'R0'];  [17, 0, 0]
LI; ['R1', '10', '-'];  [19, 10]
```

```
current_cycle = 0
while (current_cycle <= max_cycles
       and not state.is_terminated):
    instruction, operators, values = get_cycle_data(instructionMemory, state)
    #  print(f'{instruction}; {operators};  {values}')  # uncomment for step by step debugging
    execute_instruction(registerFile, state, instruction, operators, values)
```

```
def get_cycle_data(instructionMemory: InstructionMemory, state: State):
    instruction = instructionMemory.read_opcode(state.current_address)
    operators = [
        instructionMemory.read_operand_1(state.current_address),
        instructionMemory.read_operand_2(state.current_address),
        instructionMemory.read_operand_3(state.current_address)
    ]
    values = []  # either int values given directly, or values from register
    for operator in operators:
        if operator.startswith('R'):
            values.append(int(registerFile.read_register(operator)))
        elif operator != '-':
            values.append(int(operator))

    return instruction, operators, values
```

The function has two parameters, instruction memory and state. First it reads the next instruction code from instruction memory, stores it under instruction variable and then reads off the operators, to find which register was specified for the instruction. Then, from operators (opcodes) list, the underlying values are extracted. If the operand starts with ‘R’, the operand is a register. If it does not start with ‘R’ , then the operand is an integer to be read directly.

Executing an instruction

Having read the data for the current cycle, the next step is to take an action based on them. The next method, excecute_instruction, has the parameters registerFile (supplied by teacher), state, and the 3 variables: instruction, operators, and values. It checks which instruction to be executed, and then for example for “ADD’, we call the method write_register, and we assign the value of register 2 plus the value of register 3 to register 1. This is similarly implemented for the other arithmetic and logic instructions.

```
def execute_instruction(registerFile: RegisterFile, state: State, instruction: str, operators: list[str], values: list[int]):
    match instruction:
        case 'ADD':
            return registerFile.write_register(operators[0], values[1] + values[2])
```

For storing data, we write to the memory by calling the method write_data, using the first value supplied and the second value supplied.

```
case 'SD':
    return dataMemory.write_data(values[1], values[0])
```

For the opcodes 'SUB','OR','AND' and' NOP' ,which are used in arithmetic and logic instructions to perform various computations on data , the same principle applies as explained above for the opcode 'ADD' and the program counter is updated.

```
case 'ADD':
    return registerFile.write_register(operators[0], values[1] + values[2])
case 'SUB':
    return registerFile.write_register(operators[0], values[1] - values[2])
case 'OR':
    return registerFile.write_register(operators[0], values[1] | values[2])
case 'AND':
    return registerFile.write_register(operators[0], values[1] & values[2])
case 'NOT':
    return registerFile.write_register(operators[0], ~values[1])
```

The opcodes 'LI' , 'LD' and 'SD' are used in data transfer instructions. These instructions allow the program to transfer data stored in memory to and from registers.

'LI' is used to load an immediate values[1] into the register specified by operators[0].

```
case 'LI':
    return registerFile.write_register(operators[0], values[1])
```

'LD' reads the data from the memory location specified by values[1] in the dictionary data memory and writes it into the register specified by operators[0].

```
case 'LD':
    return registerFile.write_register(operators[0], dataMemory.read_data(values[1]))
```

'SD' writes the value from the register specified by value[0] into the memory location specified by value[1].

```
case 'SD':
    return dataMemory.write_data(values[1], values[0])
```

The three following instructions are used as control and flow instructions:

- 'JR' - the instruction jumps to the address specified by value[0], which is in the instruction memory
- 'JEQ' - the instruction jumps to the address specified by values[0] if the values at values[1] and values[2] are equal.
- 'JLT' - the instruction compares the values at values[1] and values[2], if the value at values[1] is less than the value at values[2] , the instruction jumps to the address specified by values[0].

For the three cases, state.is_instruction_jump_in_current_cycle is set to ‘True’, indicating that a jump has occurred in the current cycle. In such cases, the current address is explicitly set to the desired address specified by the jump instruction and the current cycle is not incremented because the program counter(current_adress) is explicitly set to a different value.

```
    case 'JR':
        state.current_address = values[0]
        state.is_instruction_jump_in_current_cycle = True
    case 'JEQ':
        if values[1] == values[2]:
            state.current_address = values[0]
            state.is_instruction_jump_in_current_cycle = True
    case 'JLT':
        if values[1] < values[2]:
            state.current_address = values[0]
            state.is_instruction_jump_in_current_cycle = True
```

In case the opcode is ‘NOP’, no operation should be performed in this cycle and the program moves to the next instruction and ‘END’ marks the end of the program :

```
case 'NOP':
    pass
case 'END':
    state.is_terminated = True
```

The self.is_instruction_jump_in_current_cycle is responsible for indicating if a jump to a different instruction was executed. When it is set to true , it signifies that a jump has occurred ,the cycle will therefore not be incremented. Because if it is incremented , the program would not go to the wanted instruction, it would however go to the specified instruction +1(the program would advance the next instruction after the one specified in the jump). When self.is_instruction_jump_in_current_cycle is set to False, it indicates that no jumps have been executed in the current cycle , and therefore, the cycle is incremented as usual. The instruction therefore allows jumps to occur without disrupting the cycle incrementation process.

```
    if not state.is_instruction_jump_in_current_cycle:
        state.current_address += 1
    state.is_instruction_jump_in_current_cycle = False
    current_cycle += 1
```

At the end of the code , we directly print the content of the entire register file from the RegisterFile dictionary and the data memory from the dataMemory dictionary that has been used in any way, whether the data has been initialised, read or written at least once.

```
print('')
registerFile.print_all()
print('')
dataMemory.print_used()
print('')
print(f"Executes in {current_cycle-1} cycles.")

print('\n---End of simulation---\n')
```

Custom test case - selection sort

We decided to implement a sorting algorithm. Initially we were thining about implementing something ambitions, however 16 registers is a serious constrain and requires an in-place algorithm with a constant space complexity. We planned to implement merge sort, however, it's most recomended implementation is not done in place. Therefore, we thought about quick sort, however, it would still take too much space in the memory, which we did ont have. Even though it is in place, its recursive nature and passing first and last indexes by values, would quickly fill a tiny 16 registers array. Therefore, the algorithm of choice was selection sort, which is in place and has a constant space complexity.

We created the algorithm in python in file selectionSort.py and tested it. Then we rewrote it into assempy language. We followed the exact code we write and did everyhing step by step.

There are the following steps which we represented in the assemply in order to create selection sort:

1. Specifying first index and last index+1 elements of the array in the momory file.
2. Staring the first 'i' loop.
    a. End i loop if i == last array index + 1
3. Starting the second 'j' loop.
    a. End j loop if j == last array index + 1
    b. Check if current index represents smaller value in the array. If it does store the current index in a variable.
4. Go to the begining of the 'j' loop.
5. Swap the smallest element with current element
6. Go to the begining of the 'i' loop.

At the end of the execution, printed memory is sorted in ascending order and contains all the values given at the beginning of the programm in data_mem_3.txt.