02135 Assignment 2:
Implementation of an instruction-set architecture (ISA) simulator in Python

Luca Pezzarossa [`lpez@dtu.dk`]

September 17, 2021

1 Introduction

This document describes the second assignment of the course 02135 - Introduction to Cyber Systems. In this assignment, you are required to implement an instruction-set simulator for a simple processor in Python.

An instruction-set architecture (often referred to as ISA) is an abstract model of a computer. The instruction-set architecture is basically machine language, it defines the set of elementary commands that a processor is able to perform. In other words, it defines everything a machine language programmer needs to know in order to program a computer. In general, compilers are used to translate high-level programming languages into machine language. However, for this assignment, we will directly deal with small machine language programs in order to understand the functionality and the structure of a simple processor. In Chapter 3 of the textbook [1], you can find a more complete description of instruction-set and processors architecture.

Similarly to assignment 1, this assignment should be carried out in **groups three people**. The groups can be different to the ones of Assignment 1. In addition to the simulator implementation, you are required to prepare a short report. Further details are provided in Section 6. All the material related to this assignment can be found on the DTU-Learn course page at the following location:

```
DTU-Learn/Course Content/Content/Assignments/Assignment 2
```

The deadline for this assignment is **Friday 15th October 2021 at 23:59**. By this date, you have to hand-in an electronic version of the short report in

PDF format and the source files as ZIP archive using the assignment utility in the DTU-Learn course page at the following location:

```
DTU-Learn/Course Content/Assignments/Assignment 2
```

This document is divided into 6 sections, plus an appendix:

- Section 2 describes the simple processor architecture that we consider in this assignment.
- Section 3 defines the instruction-set architecture to be simulated.
- Section 4 describes the simulator specifications, the format of the program and data memory files, and the three Python classes provided to facilitate the simulator implementation.
- Section 5 describes the tests used to prove the correct functionality of the implemented simulator.
- Section 6 lists the assignment tasks and the report requirements.
- The appendix contains the documentation of the provided Python classes.

As we are not enforcing any particular approach on the simulator implementation, it may be difficult for the teachers to understand your implementation and thus, help to debug your code by taking into account the following hints:

- Structure your code such that it is clear what the different parts are doing.
- Think about testing your code while planning and make sure that you test and debug the individual parts.
- Document your code with comments and, if necessary, a README file with instructions on how to run it.

Feel free to ask or send an e-mail to the assignment author if you have questions regarding practical matters and the assignment in general.

2 Simple processor architecture

The processor architecture that we use in this assignment is very simple and it is shown in Figure 1. In the following, we briefly describe the functionality and the characteristics of the architecture. Please refer to Section 3.2 of the textbook [1] for further explanation of the processor architecture.

The datapath consists of the arithmetic logic unit (ALU) and the register file, which stores temporary data. The arithmetic logic unit performs integer arithmetic and bitwise logic operations on the data stored in the file register. The simple processor we want to simulate has a register file of size 16 registers. The

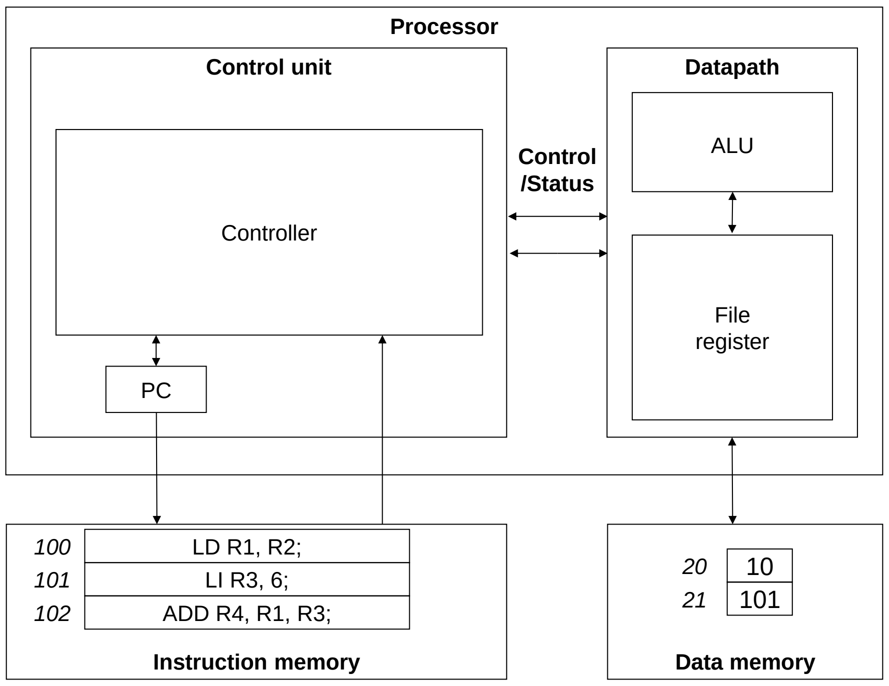

Figure 1: Block diagram of the simple processor we use in this assignment.

registers are named from `R0` to `R15`. Register `R0` is read only and always contains 0 (zero). The size of each register is 8 bits; therefore the range of the value contained in each register is from 0 to 255. For simplicity, we only use natural numbers (unsigned).

The control unit of the processor manages the execution of the program. It fetches the instruction to be executed and controls the datapath according to the instructions. The control unit contains the program counter (often referred to as PC). The program counter is a variable that keeps track of the program execution. It contains the address where the current instruction to be executed is stored in the instruction memory. If no jumps are performed, the program counter is incremented by 1 in each cycle. Thus, pointing to the next instruction.

The data memory stores the data used by the processor. Data can be loaded from memory to register or stored form registers to memory with dedicated instructions. The data memory of our processor has 256 locations addressed from 0 to 255. Similarly to the register file, each memory location contains an unsigned 8-bit value.

The instruction memory stores the program that the processor executes. The instruction memory has also 256 locations addressed from 0 to 255. Each memory location contains one instruction.

3 Definition of the instruction-set architecture

The instruction-set we want to simulate is presented in Table 3. It is a minimal instruction-set, but it offers enough instructions to run a wide range of programs. Table 3 is divided into 3 sections: arithmetic and logic instructions, data transfer instructions, control and flow instructions.

The arithmetic and logic instructions are used to perform operations between the content of the register file. The data transfer instructions are used to move data from memory into the registers and vice-versa. The control and flow instruction are used to control the execution of the program by performing 'jumps' between instructions if certain conditions are met.

Each instruction consists of an OPCODE and a list of operands. The OPCODE specifies the operation to be executed using the operands. Here we use `R1`, `R2`, and `R3` as operands, in general they could be replace by any register `R0 - R15` In the following, we explain the instructions in detail.

Table 1: The complete instruction-set architecture for the simulator.

Instruction	**Syntax (example)**	**Meaning (example)**
Arithmetic and logic instructions		
Addition	`ADD R1, R2, R3;`	`R1 = R2 + R3`
Subtraction	`SUB R1, R2, R3;`	`R1 = R2 - R3`
Bitwise OR	`OR R1, R2, R3;`	`R1 = R2 or R3`
Bitwise AND	`AND R1, R2, R3;`	`R1 = R2 and R3`
Bitwise NOT	`NOT R1, R2;`	`R1 = not(R2)`
Data transfer instructions		
Load immediate	`LI R1, 6;`	`R1 = 6`
Load data	`LD R1, R2;`	`R1 = memory(R2)`
Store data	`SD R1, R2;`	`memory(R2) = R1`
Control and flow instructions		
Jump	`JR R1;`	`go to R1`
Jump if equal	`JEQ R1, R2, R3;`	`if(R2==R3) go to R1`
Jump if less than	`JLT R1, R2, R3;`	`if(R2<R3) go to R1`
No operation	`NOP;`	`do nothing`
End execution	`END;`	`terminates execution`

Arithmetic and logic instructions

All the arithmetic and logic instructions perform operations on the content of the registers. For all the arithmetic and logic instructions the first operand is the target of the operation (i.e., where the result is stored).

`ADD R1, R2, R3;` Addition. This instruction executes the unsigned addition between the content of the registers `R2` and `R3` and places the results in register `R1`. Overflows are not reported and the result stored in `R1` is modulo 256.

`SUB R1, R2, R3;` Subtraction. This instruction executes the unsigned subtraction between the content of the registers `R2` and `R3` and places the results in register `R1`. Underflows are not reported and the result stored in `R1` is modulo 256.

`OR R1, R2, R3;` Bitwise OR. This instruction executes the bitwise OR operation ('|' in Python) between the content of the registers `R2` and `R3` and places the results in register `R1`.

`AND R1, R2, R3;` Bitwise AND. This instruction executes the bitwise AND operation ('&' in Python) between the content of the registers `R2` and `R3` and places the results in register `R1`.

`NOT R1, R2;` Bitwise NOT. This instruction executes the bitwise NOT operation ('~' in Python) on the content of the register `R2` and places the results in register `R1`.

Data transfer instructions

The data transfer instructions perform operations on the content of the registers and of the data memory. All the data transfer instructions work with two operands.

`LI R1, 6;` Load immediate. This instruction loads the value 6 (or any other specified value) into the register `R1`.

`LD R1, R2;` Load data. This instruction loads the content of the data memory at the address specified as content of the register `R2` into the register `R1`.

`SD R1, R2;` Store data. This instruction stores the content of the register `R1` in the data memory at the address specified as content of the register `R2`.

Control and flow instructions

The control and flow instructions affect the execution of the program, unconditionally or depending on the content of registers.

`JR R1;` Jump. This instruction jumps, in the execution of a program, to the instruction stored in the instruction memory at the address specified as content of the register `R1`. In other words, the program counter assumes the value of the register `R1` content.

`JEQ R1, R2, R3;` Jump if equal. This instruction jumps, in the execution of a program, to the instruction stored in the instruction memory at the address specified as content of the register `R1` only if the content of the register `R2` is equal to the content of the register `R3`.

`JLT R1, R2, R3;` Jump if less than. This instruction jumps, in the execution of a program, to the instruction stored in the instruction memory at the address specified as content of the register `R1` only if the content of the register `R2` is less than the content of the register `R3`.

`NOP;` No operation. This instruction performs no operations. It is used as a controlled way to wait for 1 cycle.

`END;` End execution. This is a pseudo-instruction (not actually executed by the processor) and it indicates the end of the program execution. In our case, the simulator should stop.

4 Simulator specifications

In this section, we provide the simulator specifications and we describe the format of the files used as input and the expected output. We also give an overview of some additional code we provide to facilitate the simulator implementation.

4.1 Overview

The simulator should be implemented in Python 3 and named `isa-sim.py`. Similarly to the FSMD simulator, the instruction-set architecture simulator should be cycle based, which means that in each cycle a single instruction is executed. A cycle counter variable should keep track of the progression of time.

The simulator should accept the following arguments:

```
python isa-sim.py <max_cycles> <program_file> <data_mem_file>
```

The `<max_cycles>` is an integer number that specifies for how many cycles the simulator should run. Therefore, the simulator terminates when the cycle counter reaches the value specified as `<max_cycles>` or if the `END` instruction is executed. The `<program_file>` is a text file that contains the program, written in machine language, that needs to be executed. The `<data_mem_file>` is a text file that contains the initial content that should be loaded in the data memory before starting to execute the program. The structure of these two files is described in the following.

For example, to simulate a maximum of 50 cycles using the files `program_1.txt` and `data_mem_1.txt` located in the same folder of the simulator, the command is:

```
python isa-sim.py 50 ./program_1.txt ./data_mem_1.txt
```

4.2 Program file

The program file is a text file that contains the program, written in machine language, to be executed by the simulator. Each line of the file contains an instruction. Comments are ignored and start with the symbol #. The instruction format is:

```
<instruction mem. address>: <OPCODE> <COMMA SEPARATED OPERANDS>;
```

Here is an example of a program (`test_1/program_1.txt`).

```
0: LI R1, 1;  # Load the value 1 into R1
1: LD R2, R1;  # Load mem(1) into R2
2: LD R1, R0;  # Load mem(0) into R1
3: ADD R3, R1, R2;  #  R3 = R1 + R2

... omitted ...

24: JLT R1, R3, R0;  # Jump if R3<0
25: LI R14, 255;  # Load the value 255 into R14 (executed)
26: NOP;  # Do nothing
27: END;  # Terminate execution
```

4.3 Data memory initialization file

The data memory initialization file is a text file that contains initial values of the data memory. Each line of the file contains one location of the data memory. Comments are ignored and start with the symbol #. The data memory initialization format is:

```
<data mem. address>: <memory value>;
```

Here is an example of a data memory initialization file (`test_1/data_1.txt`). After initialization, address 0 contains the value 11 and address 1 contains the value 9.

```
0: 11;
1: 9;
```

4.4 Simulator output

The simulator should have a textual output in the Python shell. For each cycle, the simulator should report at least: the current cycle, the current program counter value and the instruction to be executed. Optionally you can also print the content of the file register, and the data memory (only used location). At the end of the simulation, show the full content of the data memory and of the register file.

4.5 Provided Python classes

In order to facilitate the implementation of the simulation, we provide a template for the `isa-sim.py` that includes the definition of three Python classes to model the register file, the program memory, and the data memory. In the appendix A, B, and C we explain the functionality and the methods offered by these classes. Feel free to use the provided code or implement your own.

5 Testing the simulator

In order to test that your simulator is functional, you should perform the three test described in this section. The first two tests are provided by us and the third test should be developed by you.

5.1 Test 1

This is a simple test where all the instructions are used. The program does not perform any specific function other than testing the simulator with all the instructions. The files for this test can be found in the `test_1` folder. In the folder, you can also find a file named `expected_1.txt`, where you can see the expected values of the register file and the data memory after running the program. Check the expected output and compare it with the one of your simulator.

5.2 Test 2

In this test, the provided program orders (from small to large) an array of 10 values stored in the data memory starting from address 0. The result is placed in the data memory stating form address 10. The files for this test can be found in the `test_2` folder. In the folder, you can also find a file named `expected_2.txt`, where you can see the expected values of the register file and the data memory after running the program.

5.3 Test 3

For this test, you are required to write your own program and test the simulator with it. The program should use data from the data memory initialized from the data memory initialization file. Try to use a wide variety of instructions. Please document the expected output of the program you write.

6 Assignment tasks and report requirements

In the following, we list the task that you should perform in this assignment and the requirements for the short report. The simulator will be tested as described in Section 5. Therefore, remember to hand-in all the source code needed to test the simulator and, if necessary, a README file with instructions on how to run the simulator.

These are the tasks you should perform in this assignment:

1. Read this document carefully, especially the instruction-set definition and the simulator specifications, and take a look at all the provided source files, including the tests.
2. Design and implement your simulator.
3. Test the simulator with the provided programs of Test 1 and Test 2.
4. Write your own program and test the simulator with it.
5. Prepare a short report according to the instruction provided in the following.

Similarly to the Assignment 1, the report should not be longer than 6 pages (everything included) and should provide an overall description of your simulator focusing on the main implementation ideas and decisions. There are no specific requirements on the template to be used for the short report. Do not include the full code in the report. You can include some code snippets if these are relevant to explain certain aspects of the implementation.

References

[1] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/-Software Introduction.* John Wiley & Sons, Inc., 2002.

Appendix: Documentation for the provided Python classes

A The `RegisterFile` class

This class models the register file of the processor. It contains 16 8-bit unsigned registers named from `R0` to `R15` (the names are strings). `R0` is read only and reads always 0 (zero). When an object of the class `RegisterFile` is instantiated, the registers are generated and initialized to 0.

Object instantiation

`registerFile = RegisterFile()`

Methods:

- **`write_register(register_name, register_value)`**

Description:
This method writes the content of the specified register.

Parameters:
- **`register_name`** - (String) Name of the register to be written (e.g., `'R3'`).
- **`register_value`** - (Integer) Value to be written in the register.

Return:
- None.

- **`read_register(register_name)`**

Description:
This method reads the content of the specified register.

Parameters:
- **`register_name`** - (String) Name of the register to be read (e.g., `'R3'`).

Return:
- (Integer) Content of the register.

- **`print_register(register_name)`**

Description:
This method prints the content of the specified register.

Parameters:
- **`register_name`** - (String) Name of the register to print (e.g., `'R3'`).

Return:
- None.

- **`print_all()`**

Description:
This method prints the content of the entire register file.

Parameters:
  - None.

Return:
  - None.

B The `DataMemory` class

This class models the data memory of the processor. When an object of the class `DataMemory` is instantiated, the data memory model is generated and automatically initialized with the memory content specified in the file passed as second argument of the simulator. The memory has 256 location addressed from 0 to 255. Each memory location contains an unsigned 8-bit value. Uninitialized data memory locations contain the value zero.

Object instantiation

```
dataMemory = DataMemory()
```

Methods:

- **`write_data(address, data)`**

Description:
This method writes the content of the memory location at the specified address.

Parameters:
  - **`address`** - (Integer) Address of the memory location to be written.
  - **`data`** - (Integer) Value to be written at the specified address.

Return:
  - None.

- **`read_data(address)`**

Description:
This method writes the content of the memory location at the specified address.

Parameters:
  - **`address`** - (Integer) Address of the memory location to be read.

Return:
  - (Integer) Content of the memory location at the specified address.

- **`print_data(address)`**

Description:
This method prints the content of the memory location at the specified address.

Parameters:
  - **`address`** - (Integer) Address of the memory location to print.

Return:
  - None.

- **`print_all()`**

Description:
This method prints the content of the entire data memory.

Parameters:
  - None.

Return:
  - None.

- **`print_used()`**

Description:
This method prints the content only of the data memory that has been used (initialized, read, or written at least once).

Parameters:
  - None.

Return:
  - None.

C The `InstructionMemory` class

This class models the instruction memory of the processor. When an object of the class `InstructionMemory` is instantiated, the instruction memory model is generated and automatically initialized with the program specified in the file passed as first argument of the simulator. The memory has 256 location addressed from 0 to 255. Each memory location contains one instruction. Uninitialized instruction memory locations contain the instruction `NOP`.

Object instantiation

`instructionMemory = InstructionMemory()`

Methods:

- **`read_opcode(address)`**

Description:
This method returns the OPCODE of the instruction located in the instruction memory location in the specified address. For example, if the instruction is `ADD R1, R2, R3;`, this method returns `ADD`.

Parameters:
 - **`address`** - (Integer) Address of the memory location to be read (e.g., the program counter).

Return:
 - (String) OPCODE of the instruction located in the specified address.

- **`read_operand_1(address)`**

Description:
This method returns the first operand of the instruction located in the instruction memory location in the specified address. For example, if the instruction is `ADD R1, R2, R3;`, this method returns `R1`.

Parameters:
 - **`address`** - (Integer) Address of the memory location to be read (e.g., the program counter).

Return:
 - (String) First operand of the instruction located in the specified address.

- **`read_operand_2(address)`**

Description:
This method returns the second operand of the instruction located in the instruction memory location in the specified address. For example, if the instruction is `ADD R1, R2, R3;`, this method returns `R2`.

Parameters:
 - **`address`** - (Integer) Address of the memory location to be read (e.g., the program counter).

Return:
 - (String) Second operand of the instruction located in the specified address.

- **`read_operand_3(address)`**

Description:
This method returns the third operand of the instruction located in the instruction memory location in the specified address. For example, if the instruction is `ADD R1, R2, R3;`, this method returns `R3`.

Parameters:

`- address -` (Integer) Address of the memory location to be read (e.g., the program counter).

Return:

- (String) Third operand of the instruction located in the specified address.

• **`print_instruction(address)`**

Description:
This method prints the instruction located at the specified address.

Parameters:

`- address -` (Integer) Address of the instruction memory location to print.

Return:

- None.

• **`print_program()`**

Description:
This method prints the content of the entire instruction memory (i.e., the program).

Parameters:

- None.

Return:

- None.