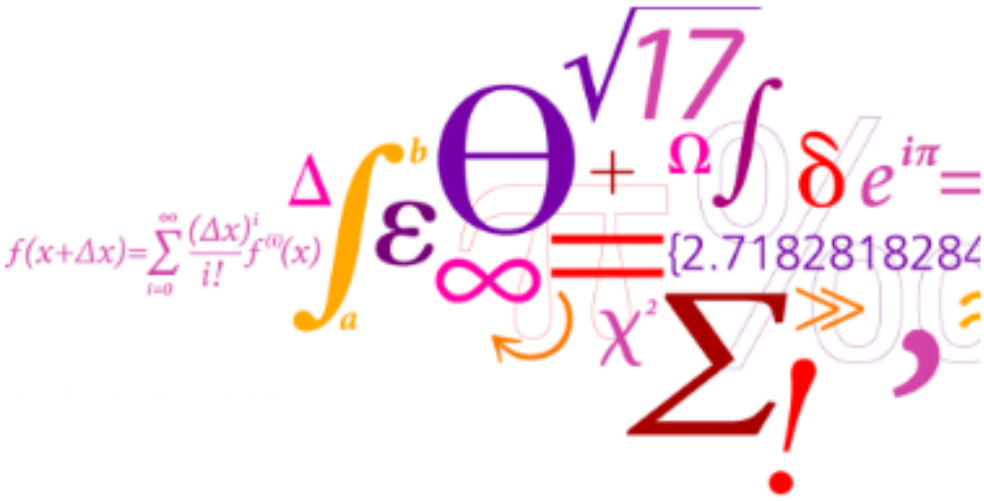
Technical University of Denmark

Implementation of an ISA simulator in Python

Simon Tobias Lund - s184459
Sara Maria Guijarro Heeb - s184484
Thomas Fayad - s184440

October 21, 2019



Technical University of Denmark
Course: 02135 - Introduction to Cyber Systems

Introduction

An instruction-set architecture (often referred to as ISA) is an abstract model of a computer. An instruction-set architecture is basically machine language, it defines the set of elementary commands that a processor is able to perform. ISA is the part of the processor that is visible to the programmer or compiler writer. It serves as the interface between software and hardware.

For this project, we had to create a program that can execute a simple assembly language. This is a language that consists of simple operations that can be implemented as logic gates and transistors. In this case, the language used corresponds to instruction-set architecture.

1 Constructing an ISA simulator in Python

1.1 General overview of the simulator

Similarly to the first project, much of the simulator relies on a prewritten code given by the responsibles of the course *Introduction to Cyber Systems*. The program starts by creating a simulation object. This object contains object of the other classes necessary for the stimulation. So the different objects, which are sub-objects of simulation, initiate themselves by reading the data from the memory file and the program from the program file. Then sorting it into suitable data-types. For instance, registers and operations performed on these, are in a dictionary in a object of the class `RegisterFile`.

To simulate instruction-set architecture tests, currying functions were used so that we can use the same code for calling functions taking different numbers of parameters. Currying is used to transform multiple-argument function into single argument function by evaluating incremental nesting of function arguments. Basically when applying the function, instead of returning a value, it returns another function which can take new parameters into account. As a matter of fact, we can feed the function an undefined number of parameters, and have it return a value as soon as the innermost function is reached.

Furthermore, operations of type `SUB` takes three parameters : the register containing the value to be subtracted from, the register containing the value to subtract, and the register to store the result in. On the other side, operations of type `NOP` takes no parameters, as it does not compute any calculations. Consequently, by defining the operations as functions using currying, we can use the same code for an arbitrary number of parameters. Thus, it permits to execute all operations as `ADD`, `LI` or `SD`.

1.2 Steps of the ISA simulator written in Python

1. Create a object sim of the class simulation and execute `_init_(self)` method:

```
class simulation:
    def __init__(self):
        #initiate variables
        self.PC = 0
        self.cycle = 0
        self.end = False
        self.max_cycles = int(sys.argv[1])

        #initiate object of other classes
        self.registerFile = RegisterFile()
        self.dataMemory = DataMemory()
        self.instructionMemory = InstructionMemory()
```

2. `registerFile` creates a dictionary called registers of the form `{"R0:0, "R1":0, ..., "R15":0}`, and a dictionary called functions that define which function to call when given a string of the functions name as `"NOP" : self.NOP`

3. `dataMemory` creates a dictionary initialized to the values given in a `.txt` file, with the value 0 for all uninitialized memory slots

4. `instructionMemory` parses the program defined in the `.txt` file, and turns it into the bit complicated dictionary `instruction_memory`. The first key is the cycle number at which to execute a given operation. The value at this key is another dictionary containing the information about the instruction. `"opcode"` is the operation to be executed, thus we can have for example : `instruction_memory = {0: {"opcode":"SUB","op_1":"R3", "op_2":"R2, "op_3":"R5"}}` which can be translated to "subtract the value at R5 from the value at R2 and store it in R3 (all executed at cycle 0)"

5. This was the last step for the initialization, so the simulation can start running:

```
while not sim.end:
    sim.executeCycle()
```

6. Then is ran the method defined in sim, called `executeCycle` until `sim.end` is set to true. `executeCycle` prints a display and runs the method `executeOperation`. `executeOperation` calls the method in its contained object.

7. `instructionMemory`, called `read_opcode` takes `PC` as a parameter and returns the operation contained at the cycle given by `PC`. Additionally, it checks if the operation is contained in the dictionary containing functions with an arbitrary number of parameters. It calls the right method and feeds it the desired number of parameters.

8. Finally throughout the process, `incrementPC()` keeps track on the steps that the code runs, and in parallel `incrementCycle()` checks if `max_cycle` has been reached (if `max_cycle` is reached then `end = True`)

Overall, the most important methods used in the operations are `read_register`, `write_register`, `write_data` and `read_data`. They edit the content of the register dictionary and data dictionary.

Last but not least, `JR`, `JEQ` and `JLT` are significant operations as they set `PC` to a value defined in a register, making it possible to create both loops and to use it for multiple purposes in the rest of the code. `JEQ` only jumps if two registers have equal value, and `JLT` only jumps if one register has a smaller value than another.

2 Developing an ISA - Factorizing numbers

First of all, the code checks if the initial number given is even or odd. Indeed, it divides the given integral by two as long as the last bit of our number is zero (which means it is even).

We factorized by testing the divisibility of all numbers smaller than the square root. Thus, we used the Babylonian method to approximate the square root of the initial number input. The idea is to start by an initial reasonable guess x_0 to the given number S (the number that we want to approximate it's root). Then, the guess is improved by applying the following formula:

$$x_1 = \frac{(x_0 + \frac{S}{x_0})}{2} \tag{1}$$

Indeed, x_1 becomes a better approximation to the square root. Henceforth, the recursive function should be applied until the process converges. A general formula to the algorithm can be described as follow:

$$x_{n+1} = \frac{(x_n + \frac{S}{x_n})}{2} \tag{2}$$

Convergence is achieved when the digits of x_{n+1} and x_n agree to as many decimal places one desires.

As a matter of fact, to perform division, `R4` is loaded with number to divide, `R5` is loaded with divider, and `R7` is loaded with line of code to go back to after division. Finally, the quotient is stored in `R6` and the remainder in `R4`. Factorization happens if the remainder is equal to zero. We then update the number to be the result of the division. On the opposite, if this condition is not satisfied, then the code increments the number that divides the number we're factorizing.

To conclude, the program outputs such that all the numbers after the one saved at zero in memory are prime, and multiplied together they will give the number stored at zero.

Conclusion

In conclusion, the third test went quite well. It is expected that the resulting memory first contains the number factorized and then all prime factors. As a matter of fact, it has to be taken into consideration that we simulate a computer with a memory of 256 bytes, and 16 registers of 1 byte. Thus, we're operating with numbers from 0 to 255. Consequently, there could be errors related to the fact that the numbers can't be larger than 255.

Indeed, the only downside is that it isn't really practical with numbers as small as ours. We could handle bigger numbers if we divided them up into multiple storage spaces, and wrote more to memory than we have done now. However, that would also make our code virtually lot longer.

Finally, factorization was a task that allowed us to show off a lot of the assets of the assembly language, from code modules and loops to bit operations.