Sabri Gökberk Yılmaz – Şefika Akman
2017400144-2017400192

The problem:

We are asked to simulate a hypothetical CPU. This cpu has an instruction set that has at most 1 operand for each instruction. There are 4 addressing modes: immediate, register, memory address in register and memory address. Some of the instructions takes the addressing mode into consideration. For instance store may store into memory or a register etc.

We have the instruction set, but also we need these instructions to be converted into binary codes of which a machine can work. That's the reason we implemented the assembler for cpu230.

After converting assembly code into binary code, we need to execute the operations and produce the desired result. For this, we need a CPU simulation. Of course, we are not directly utilizing CPU or memory, but instead we let python do that.

Our implementation:

1- Assembler:

We first implemented the assembler because without the machine code, a CPU cannot work. Assembler reads from an input file and disregards the empty lines. It also detects the labels, and maps the label's address with its name (case insensitive). Then, whenever the label is called, we check the dict structure and do the rest of the operation with its value. Lines that are not label or empty are stored in a list, and converted into hex.

While converting a line into hex, we need 3 components. 3rd is given directly by the assembly language, the operand. 1st and 2nd were a little tricky to deal with. 6 bits represent the instruction and 2 bits represent the addressing mode. We evaluated the addressing mode by looking up if the operand is register, if it is a label, if it is an immediate, if it is an address. The opcode was easier because, the operation name is directly mapped to a 6-bit value. Afterwards, we concatenated these 8 bits and converted into hex.

The final output is put into a .bin file, this is the input of our executor python file.

2- Executor:

The executor was harder to implement. At first we had to decide how to implement the memory, because, everything should be related to it. Even the instructions. We decided to keep a 2^16 sized list of string type. Every index represents a byte, and keeps 2 hex digits.

First, the input is read from the .bin file and hex numbers are stored in memory, each took 3 indexes. Then, we created the registers and flags. Registers all have int value and flags, though they keep integer, they took either 0 or 1.

Afterwards, we analyzed a given line by line_analyzer function. It returns the opcode, address mode and operand of a 24-bit (6 hex digits) instruction. Then, for each instruction in the memory, the PC's address is fetched and analyzed. For each instruction we implemented a separate function.

Sabri Gökberk Yılmaz – Şefika Akman
2017400144-2017400192

Each instruction has different ways of working, some has all 4 address modes, some has only 1. Those which take more than one address mode has if statements according to address mode and, operations are done according to this address modes.

Some instructions set some flags, and the way our CPU works is dependent on these flags. For instance if an instruction that is able to set ZF is evaluated to 0, ZF is set to 1. If it is non-zero than to 1. If an instruction that can set CF has 17 bits as result and the leftmost bit is 1, CF is 1, else 0. If an instruction is able to set SF has 1 at its leftmost bit, the SF is 1 else 0. Especially jumps are dependent on this. And, jumps are essential for an assembly language, because it allows loops and conditional statements.

Some interesting instructions:

-ADD: it adds A and the operand, It treats them as unsigned binary values but then, while setting the flags SF and CF are also handled, and 16 bits are stored in A.

-SUB: it negates the operand, adds 1 to it(twos complement form) then it calls the ADD instruction to handle the flags. By calling add, it also handles the storing.

-CMP: it calls the SUB function but since it alters the A register, to avoid mistakes, we keep the A's value in a temp variable, after the subtraction is done and the flags are set, it restores the A

-PUSH: it stores the operand register in stack, at the place that S and S+1 points. And increases S by 2.

-POP: it takes the value from S's pointed address and decreases it by 2.

-jumps: they all work as desired in the description, they check the flags while operating. There are plenty of them!

Final Comments:

We completed the project, and it passes all the input tests. Wish we had something that is not similar to project1 :)

We think python is much more comfortable than C++ or Java, because it is much easier to manipulate variables and most of the code we wrote worked fine, while coding in CPP, we take errors even when trying to implement the simplest thing :)

This project has made us more familiar with concepts like assembly, memory, binary codes, some topics in cmpe240 and 344.

*We did not do any syntax checks since it is not the point of this project, and it consumes too much time. It works all right when given correct syntax.

We implemented the project in 3.9 because we couldn't downgrade our python version, however we tested it in online 3.6.9 compiler and it worked.