CmpE 230 Spring 2021

Project II Report

Nurlan Dadashov 2019400300

Aziza Mankenova 2018400387

1. Problem description

The given problem requires us to implement an assembler and an execution simulator for a hypothetical CPU called CPU230. One of the main purposes is to read the assembly code and produce an assembled binary output in “prog.bin” file, where each line is a 24-bit instruction. The first 6 bits represent an operation code, next 2 bits represent an addressing mode, and the last 16 bits represent an operand. After obtaining the assembled binary file, the output of the program should be printed by executing each instruction in the “.bin” file. Both programs are written in Python.

2. Problem Solution

The first thing we started to consider is creating a “.bin’ file. To read the assembly code, we make two passes through the code. During the first one, we find all the labels and assign their addresses taking into account the blank lines. In the next pass, we analyze each line, except the ones that are empty and contain a label. This way we record the operation code, the addressing mode, and the operand for each instruction and print it to the “.bin” file.

The execution of the assembled binary file starts by reading the input and storing all the lines in the list, so that later we could easily parse them. Before the start we define dictionaries of flags, registers, and bit_to_register. The bit_to_register dictionary maps the binary bit patterns to the corresponding registers. All the flags are initially mapped to zero. The “PC” register maps to the beginning of the memory, “S” register maps to the end of memory, while all the other registers are initialized as zero. Also, we create a memory as a list of size 2^{16}, which is 64kB. And each memory cell is initialized to zero at the beginning. But, before the execution every 3 memory cells are filled with every 8 bits of the 24-bit instruction sequence because each memory cell can contain only 1 byte. After these initializations, the execution starts by reading a line which “PC” maps to, which is zero at the beginning. Next, we store the values of operation code(the first 6 bits), and operand(the last 16 bits) in hexadecimal and the value of the addressing mode(the 6^{th} and 7^{th} bits) in decimal. After that, depending on the operation code the necessary operations will take place. Frequently used operations were written as methods. We defined such methods like *setFlags*, *set_flags2, store_data, bitwise*. The first two methods set necessary flags by analyzing the given parameter. The *bitwise* method is used for bitwise operations depending on the type of bitwise operator. The *store_data* is used to store the given data either in a register if the address mode is 1 or in memory if address mode is either 2 or 3. However, it does nothing if the address mode is 0, which means that the given data is an immediate data.

For the addition and increment instructions, we perform unsigned binary addition and set the necessary flags, while for the subtraction, compare and decrement instructions, we perform signed two’s complement subtraction and also set the necessary flags. For the bitwise operations, which are “XOR”, “AND”, “OR”, “NOT”, “SHL”, and “SHR”, we use the built-in bitwise operators.

The “NOP” operation does not perform any operation, and continues execution with a next line, but the “HALT” operation stops the execution. The “PUSH” operation pushes a word-sized operand (two bytes) into the memory and updates the “S” register by subtracting 2 because the operand’s 16 bits are written into two memory cells as 8 bits each. The “POP” operation pops a word-sized data (two bytes) into the operand and updates the “S” register by adding 2 because the operand’s 16 bits are written into two memory cells as 8 bits each. All the jump operations except the unconditional jump operation perform a jump to the specified memory cell if the necessary flags are set or not set. The unconditional jump “JMP” operation performs a jump irrespectively of the flags. The “READ” operation accepts a user input as a character and stores the ASCII code of that character either into a memory or a register. The input is read line-by-line. And the “PRINT” operation writes out the character which corresponds to the ASCII code of the operand into the output file.

3.Conclusion

The program executes and gives the expected output. We successfully implemented both programs. Overall, the given problem was solved. In the future the program could be further developed to do multiplication and division, consider some other flags, too.