CMPE 230: Systems Programming Project 2
CPU230 Assembler & Execution Simulator

Aral Dörtoğul
2018402108

June 4, 2021

Contents

1 Introduction **2**

2 Assembler **3**

2.1 Implementation 3

3 Execution Simulator **5**

3.1 Implementing CPU230 Instructions 6

4 Conclusion & Future Remarks **8**

1 Introduction

In this project, an assembler and an execution simulator is implemented for a hypothetical CPU called CPU230. Python (3.6.9) is used as the programming language. CPU230 can be summarized with Figure 1:

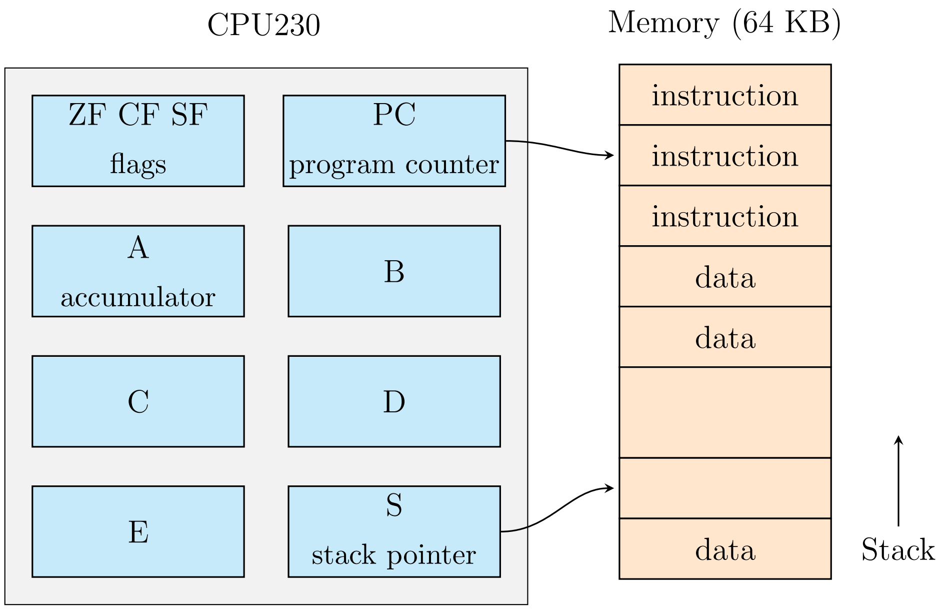

Figure 1: An Illustration of CPU230

There are 3 flags (ZF: zero flag, CF: carry flag, SF: sign flag), 5 registers from A to E, a program counter (PC) register and a stack pointer register (S).

Instructions of CPU230 has fixed length of 3 bytes: the first 6 bits are the operation code, the next 2 bits are the addressing mode, and the last 16 bits (2 bytes) are the operand.

An operand can be immediate data if it's a hexadecimal digit (there is 0 in front of the number if the number starts with a letter), or a character in single quotation marks like `'A'`, or a label name. An operand can be a memory address in a register if the operand is in the format `[r]`, where `r` is the register name. If the operand is in the form of `[xxxx]`, where `x`'s are hexadecimal digits, then the operand is a memory address.

Bits	Addressing Mode
00	immediate data
01	data in register
10	memory address in register
11	memory address

Table 1: Addressing Modes of CPU230

2 Assembler

The assembler's main duty is to convert lines of assembly code to 3-byte hexadecimal machine code. The following steps are used for assembling:

1. .asm file is read line by line form the input file to detect the label names, and the labels in the .asm file are added to the dictionary of labels with their corresponding values.
2. The assembly lines which are **not** label definitions are tokenized and stored in a list of tokenized lines.
3. For each line of tokens, the required hexadecimal code is generated and stored in a list of hexadecimal machine code.
4. If a syntax error is encountered (e.g., undefined operand, operation etc.) in the previous steps, the process is aborted and `Error!` is written in the output file.
5. If all the lines are traversed without encountering any syntax error, the hexadecimal code of the input file is written in a file with the same name but with .bin extension.

2.1 Implementation

When reading the input file line by line, the lines are tokenized using `re` (regular expression) built-in module in Python. The character groups with non-whitespace characters are found using `re.findall(r‘\S+’, line)`. The labels are recorded into the dictionary after converting them to lowercase. The reason behind this is to make the label names case insensitive.

All the instructions of CPU 230 are divided into 5 groups according to the types of operands they can accept. The groups can be seen in Table 2. The abbreviations I, M and R stands for immediate data, memory and register, respectively.

Group	**Instructions in the Group**
No Operand	`HALT`, `NOP`
IMR Operand	`LOAD`, `ADD`, `SUB`, `INC`, `DEC`, `XOR`, `AND`, `OR`, `NOT`, `CMP`, `PRINT`
MR Operand	`STORE`, `READ`
Only R Operand	`SHL`, `SHR`, `PUSH`, `POP`
Only I Operand	`JMP`, `JZ/JE`, `JNZ/JNE`, `JC`, `JNC`, `JA`, `JAE`, `JB`, `JBE`

Table 2: The Groupings of the Instructions of CPU230

In order to generate the hex code of an instruction, the instruction's group is detected and the function for generating hex code for that group is called. The functions are:

- `getHexCode_no_op(command)`
- `getHexCode_IMR_op(command, operand)`
- `getHexCode_MR_op(command, operand)`
- `getHexCode_I_op(command, operand)`
- `getHexCode_R_op(command, operand)`

These functions detect the addressing mode of the operand and combine it with the instruction's hex code and the operand.

3 Execution Simulator

The execution simulator's duty is to run the assembled program line by line and simulate CPU230's actions. The simulator's input is the output of the assembler, which is the .bin file consisting of hexadecimal machine code. This code can be interpreted and executed directly by CPU230. The steps of executing the .bin file is as follows:

1. `PC`, `A`, `B`, `C`, `D`, and `E` registers are set to "`0000`" in hexadecimal. The stack pointer `S` is initialized to "`FFFE`". It is not initialized to "`FFFF`" because the space required for stack pushing is two bytes, not one.
2. The flags `SF`, `ZF`, and `CF` are created and initialized to `False`.
3. The 64KB memory is constructed as a list of 2^{16} bytes. The bytes are initialized to "`00`" in hexadecimal.
4. The lines of 6-digit hexadecimal codes in the .bin file are read line by line and converted into 24-digit binary strings. Each string is divided into three parts: instruction (6 bit), addressing mode (2 bit) and operand (16 bit).
5. Until encountering `HALT` instruction or an error, an infinite loop is executed which reads the partitioned binary strings that `PC` is pointing and calls the required function by checking the instruction part of the string.
6. If the instruction is not a jumping instruction, PC is incremented by 3 before the next iteration of the loop. If it is a jumping instruction, jumping is executed and this step is skipped.
7. The output file is generated after the termination of the loop. If any error is encountered, `Error!` is written in the output file. If the program is error-free, the appropriate output lines are written in a file with the name of the input file but with .txt extension.

3.1 Implementing CPU230 Instructions

The instructions of CPU230 are executed with the functions below. Similar instructions are executed in the same function (e.g., `PUSH` and `POP`, `SHR` and `SHL` etc.).

- `execute_load(addressMode, operand)`
- `execute_store(addressMode, operand)`
- `execute_add_sub_cmp(instr, addressMode, operand)`
- `execute_inc_dec(instr, addressMode, operand)`
- `execute_xor_and_or_not(instr, addressMode, operand)`
- `execute_shr_shl(instr, addressMode, operand)`
- `execute_push_pop(instr, addressMode, operand)`
- `execute_read(addressMode, operand)`
- `execute_print(addressMode, operand)`

The full list of the instructions and the way they are executed (in the functions above) is in Table 3:

Table 3: Instructions of CPU230

Instruction	**Explanation**
`HALT`	terminates the execution
`LOAD`	Loads operand onto register `A`
`STORE`	Stores value in `A` to the operand
`ADD`	Performs `A` + `operand` and stores the result in `A`. Sets the flags `CF`, `SF`, `ZF`.
`SUB`	Performs `A` + `NOT(operand)` + 1 and stores the result in `A`. Sets the flags `CF`, `SF`, `ZF`.

`INC`	Increments the operand and stores the result in the operand if it's not immediate data. Sets the flags `CF`, `SF`, `ZF`.
`DEC`	Decrements the operand and stores the result in the operand if it's not immediate data. Sets `CF`, `SF`, `ZF`.
`XOR`	Performs bitwise `A XOR operand` and stores the result in `A`. Sets the flags `SF`, `ZF`.
`AND`	Performs bitwise `A AND operand` and stores the result in `A`. Sets the flags `SF`, `ZF`.
`OR`	Performs bitwise `A OR operand` and stores the result in `A`. Sets the flags `SF`, `ZF`.
`NOT`	Performs bitwise `NOT(operand)` and stores the result in the operand if it's not immediate data. Sets the flags `SF`, `ZF`.
`SHL`	Shifts the bits of the register to the left. Sets the flags `CF`, `SF`, `ZF`.
`SHR`	Shifts the bits of the register to the right. Sets the flags `SF`, `ZF`.
`NOP`	No operation
`PUSH`	Pushes the two byte operand to the stack and updates `S` by subtracting 2.
`POP`	Pops two byte from the stack into the operand and updates `S` by adding 2.
`CMP`	Performs `A - operand` and sets the flags `CF`, `SF`, `ZF`.
`JMP`	Sets the `PC` to the operand.
`JZ/JE`	Sets the `PC` to the operand if `ZF` is true. Increments it by 3 otherwise.
`JNZ/JNE`	Sets the `PC` to the operand if `ZF` is false. Increments it by 3 otherwise.
`JC`	Sets the `PC` to the operand if `CF` is true. Increments it by 3 otherwise.
`JNC`	Sets the `PC` to the operand if `CF` is false. Increments it by 3 otherwise.

`JA`	Sets the `PC` to the operand if `SF` is false. Increments it by 3 otherwise.
`JAE`	Sets the `PC` to the operand if `SF` is false or `ZF` is true. Increments it by 3 otherwise.
`JB`	Sets the `PC` to the operand if `SF` is true. Increments it by 3 otherwise.
`JBE`	Sets the `PC` to the operand if `SF` is true or `ZF` is true. Increments it by 3 otherwise.
`READ`	Reads a character from user and stores it in the operand (word size). If more than one character (string) is given as input, the characters other than the first one are basically ignored.
`PRINT`	Prints a word size character in the operand.

4 Conclusion & Future Remarks

The most crucial part of any computer, the CPU receives, directs, and processes the computer's data, and my implementation of CPU230 works decently. Nevertheless, it can even be improved further. For example, in the future, it can be reasonable to add more instructions such as `MUL`, `DIV` to add more functionality to the CPU. Another possible future improvement is the support of different data types with different sizes. Currently, only word size (2 byte) operands are supported. Support of other data sizes can greatly increase the utility of the CPU.

To sum up, in this project, an assembler is developed in order to transform low level assembly language into machine code and a hypothetical CPU is constructed considering the essential components of a central processing unit: Registers, flags, arithmetic and logic unit etc. In my opinion, this project was important in terms of grasping the way the operations are interpreted and handled in a CPU. Because of this, I believe that this project was educative.