Documentation of cpu230 CPU Project

Atilla Türkmen – 2019400216

Assembler part:

I have created three dictionaries at the top of the assembler script. First one maps written instructions to corresponding hex opcodes. Second one maps four addressing modes to instruction codes that can use that addressing mod. The last one maps register letters to their bit representation.

The script goes over the instructions in the file twice. In the first pass, a labels dictionary is populated. When a line with one token that has a colon at the end is reached, that label is mapped to the memory address in that line in labels dictionary.

In the second pass, binary output is generated. First of all opcode of command is looked up from the dictionary that was created previously. If there is an operand after the command it checked whether it is a memory address (inside []), register, label, character or hex number with successive if statements. If operand is none of the above script raises an exception. Character and hex number check is done with regex. Addressing mode is determined in these if statements. An exception is raised if current operation can't take that addressing mode. Finally opcode, addressing mode and converted operand are concatenated and formatted into 6 digit hex numbers.

Executer part:

Only IO operations are done in cmpe230exec file. CPU execution is simulated in MyCPU class in another python file. A function in MyCPU class takes instructions as arguments and returns printed values as list. There is a dictionary in this class which maps hex opcodes to functions. Instruction at PC is executed while PC is less than length of instructions. PC increments by one with every instruction. All registers are string values that contains strings of ones and zeroes. Memory is a dictionary that maps hex numbers (string type) to 16 bit binary values. Stack is a list that is used as stack. Below are explanations for functions of all instructions.

- **Halt:** Sets "halt" variable of class true which breaks the loop that executes instruction functions.
- **Load:** The class has a function named "get_value" that returns the desired value by binary addressing mode and operand values. This function calls that function and assigns the returned value to A register. "get_value" function returns operand without modification if addressing mode is 00. Returns the value in register if mode is 01. Checks memory dictionary with given operand as key if mode is 11. Checks with value in register if mode is 10. All functions that needs to take a value as operand uses "get_value" function to get the data.
- **Store:** "store_in_location" function puts given data to appropriate place with operand and addressing mode. This function calls "store_in_location" function with value in A register. Assigns to register variable if mode is 01. Puts to memory dictionary if mode is 11. Puts to memory dictionary with key as value in register if mode is 10.
- **Add:** This function calls four functions. First gets the operand value with "get_value" method. Then adds the value in A register to operand that was obtained before with "addition" method and assigns to it A register. Finally sets the sign and zero flags. "addition" method takes two 16 bit binary numbers (as string) as arguments. Sums them by converting to decimal and converts to binary string again. If the result is longer than 16 bits carry flag is set to true, otherwise false. "set_zero_flag" function sets ZF to true if all digits in given string is

zero. “set_sign_flag” function sets SF to true if first digit of given string is one. These functions are called with the result after every function that sets these flags.

- **Sub:** This function sums reversed operand with one and calls sum function above with the result (to sum with A register and setting the flags).
- **Inc:** Gets the value with operand and mode (“get_value”). Adds one to it (“addition”). Stores it if mode is not 00 (“store_in_location”) and sets sign and zero flags (“set_sign_flag” and “set_zero_flag”).
- **Dec:** Same with the above except instead of adding one this function adds 16 bits of one to operand (NOT(1) + 1).
- **Xor:** Compares value given with operand and value in A register bit by bit. Does xor operation for every bit and appends it to a string. Then assigns that string to A register. Finally sets sign and zero flags.
- **And:** Same with xor function except does and operation for every bit.
- **Or:** Same with above two function except does or function to bits.
- **Not:** Gets the value with operand and mode (“get_value”). Calls “reverse_bits” function on it. Assigns it to A register. Sets sign and zero flags (“set_sign_flag” and “set_zero_flag”).
- **Shl:** Sets CF to true if first bit is one, otherwise sets it to zero. Deletes first bit and appends a zero at the end. Stores it with function explained previously. Sets zero and sign flags.
- **Shr:** Delete most right bit and put a zero at the most left. Store it. Set zero and sign flags.
- **Nop:** Just a pass keyword.
- **Push:** Get operand value and push it to the stack named S that was initialized in the constructor.
- **Pop:** Pops from S stack and stores it in given location (“store_in_location” function).
- **Cmp:** Assigns the value in A register to a temporary variable. Calls sub function explained above to set the flags with value given in operand. Finally assigns the temporary value to A register again.
- **Jmp:** Sets PC to “operand / 3 - 1” because every instruction is 3 bytes and PC will increment by one after this instruction.
- **Je:** Calls jmp if ZF is true.
- **Jne:** Calls jmp if ZF is false.
- **Jc:** Calls jmp if CF is true.
- **Jnc:** Calls jmp if CF is false.
- **Ja:** Calls jmp if SF and ZF is false.
- **Jae:** Calls jmp if SF is false or ZF is true.
- **Jb:** Calls jmp if SF is true and ZF is false.
- **Jbe:** Calls jmp if SF is true or ZF is true.
- **Read:** Takes input from STDIN with input() function. Gets its ASCII value with ord() function, formats it as 16 bit binary number and stores it as specified in operand and addressing mode. If two letters are entered before pressing enter, only first letter is read.
- **Print:** Gets the value given by operand and addressing mode, converts it to char by chr() function and appends it to output list that is returned at the end of compute function.