Artun Akdoğan
2020400327

Zeynep Gültuğ Aydemir
2019502276

CMPE 230 - SYSTEMS PROGRAMMING
Spring 2021
Project 2

For this project, we implemented an assembler and an execution simulator for a hypothetical CPU named CPU230. It has two layers, in which it produces the binary code first with cpu230assemble.py converting the Assembly code into binary and writing them into , and then executing the binary with cpu230exec.py. The code complains with the output of the Assembly code written in the output text file.

Description

Addressing mode bits are as follows:

Bits(binary)	Addressing mode
00	operand is immediate data
01	operand is in given in the register
10	operand's memory address is given in the register
11	operand is a memory address
Note that registers are represented as bit patterns (here given in hex): PC=0000, A=0001, B=0002, C=0003, D=0004, E=0005, S=0006.	

Instructions are as follows:

Instruction	Instruction code (hex)	Operand	Meaning	Flags set
HALT	1		Halts the CPU.	
LOAD	2	immediate memory register	Loads operand onto A .	
STORE	3	memory register	Stores value in A to the operand.	
ADD	4	immediate memory register	Adds operand to A. Perform the addition by treating all the bits as unsigned integer.	CF,SF, ZF
SUB	5	immediate memory register	Subtracts operand (OPR) from A. Implement it as ADD instruction as follows: A - OPR = A + (-OPR) = A + not(OPR) + 1	CF,SF, ZF
INC	6	immediate	increments operand (equivalent to add 1)	SF, ZF, CF
		memory register		
DEC	7	immediate memory register	decrements operand (equivalent to sub 1)	SF, ZF, CF
XOR	8	immediate memory register	Bitwise XOR operand with A and store result in A.	SF, ZF
AND	9	immediate memory register	Bitwise AND operand with A and store result in A.	SF, ZF

OR	A	immediate memory register	Bitwise OR operand with A and store result in A.	SF, ZF
NOT	B	immediate memory register	Take complement of the bits of the operand.	SF, ZF
SHL	C	register	Shift the bits of register one position to the left.	SF, ZF, CF
SHR	D	register	Shift the bits of register one position to the right.	SF, ZF
NOP	E		No operation.	
PUSH	F	register	Push a word sized operand (two bytes) and update S by subtracting 2.	
POP	10	register	Pop a word sized data (two bytes) into the operand and update S by adding 2.	
CMP	11	immediate memory register	Perform comparison with A-operand and set flag accordingly., i.e. A-OPR	SF, ZF, CF
JMP	12	immediate	Unconditional jump. Set PC to address.	
JZ JE	13	immediate	Conditional jump. Jump to address (given as immediate operand) if zero flag is true.	
JNZ JNE	14	immediate	Conditional jump. Jump to address (given as immediate operand) if zero flag is false.	
JC	15	immediate	Conditional jump. Jump if carry flag is true.	
JNC	16	immediate	Conditional jump. Jump if carry flag is false.	
JA	17	immediate	Conditional jump. Jump if above.	
JAE	18	immediate	Conditional jump. Jump if above or equal.	
JB	19	immediate	Conditional jump. Jump if below.	
JBE	1A	immediate	Conditional jump. Jump if below or equal.	
READ	1B	memory register	Reads a character into the operand.	
PRINT	1C	immediate memory register	Prints the operand as a character.	

Assembler

The assembler mainly consists of two parts as two separate for loops. The goal of the first loop is to detect the address of labels in the assembly code and add them to a dictionary named "label" respectively. It also checks whether the label has been defined before and prevents those types of errors. The relevant code block is given below.

*Label check and double definition detection (cpu230assemble.py, line 47)

```
    elif value.__contains__(":"):               # Found the label
        value=value.replace(":", "")
        if value in label:                      # If defined before
            print("Label defined twice!")
        operand = hex((counter) * 3)[2:]
        label[value] = operand
```

After this loop finishes, the second one starts to parse the assembly code line by line once again. This one, the purpose is to convert every single line into the binary code and write it to the .bin file in each step. Meanwhile, it also looks out for possible syntax errors using regex patterns for the input lines. If the input format doesn't match with syntax rules listed in the pattern list, it will quit the code and print the error message and the line number.

*Syntax error handling, (cpu230assemble.py, line 73)

```
        if not syntax:
            print("Syntax Error at Line " + str(counter))
            break
```

To do this, we start with a basic if block, detecting whether the line is just a simple instruction or a label definition. The line is divided into separate integers with split() function, and it's first index is assigned to the variable value. (For "LOAD [A], value is LOAD) If the line is empty, value is defined as -1 and it skips to the next line without printing any binary code. If the value is found in the instruction dictionary, their opcode is assigned from the dictionary.

*Filtering instruction strings (cpu230assemble.py, line 83)

```
        if value == -1:
            continue
        elif value in instr:
            opcode = instr[value]
            isInstr = True
        else:
            print(value)
            isInstr = False
```

The next part of the second loop checks the second part of the line and decides its addressing mode and operand. There are 5 separate if blocks checking for the input formats such as LOAD A, LOAD [A], LOAD 'A', LOAD 0b69f, label definitions and HALT/NOP. The addressing mode and operands are assigned accordingly and this concludes this step.

In the last part of the assembler, the opcode, addressing mode and operand to the binary code, using the convert.py, given in the project description. They are formatted according to their bits, and finally converted into binary code altogether. Thus, the assembler finishes writing the binary code to the .bin file.

*Binary code conversion (cpu230assemble.py, line 134)

```
            bopcode = format(opcode, '06b')
            baddressing_mode = format(addressing_mode, '02b')
            boperand = format(operand, '016b')

            bin = '0b' + bopcode + baddressing_mode + boperand
            ibin = int(bin[2:],2) ;
            result = format(ibin, '06x')
```

Artun Akdoğan
2020400327

Zeynep Gültuğ Aydemir
2019502276

Execution Simulator

At first, the code checks if the file ends with the .bin suffix and raises an exception if it is not. Then, the execution simulator reads the bin file and writes it onto the memory. At last, it gets into an infinite loop that only will be exited by the “halt” opcode.

In this loop, 3 bytes are divided into the opcode, address mode, and operand. Then, the opcode is checked if it is in the allowed range. If there were no errors so far, the PC register will be increased by 3, and the appropriate function will be executed.

*Execution Loop (cpu230exec.py, line 293)

```
while True:
    opcode = virt_mem[regs[0]] >> 2
    addr_mode = virt_mem[regs[0]] & 3
    operand = (virt_mem[regs[0]+1] << 8)+virt_mem[regs[0]+2]

    if not 0 < opcode <= len(executionArr):  # < 30:
        raise OpcodeError()

    # Instructions have been read
    regs[0] += 3

    # Index on array + 1 = Instruction code
    executionArr[opcode-1](addr_mode, operand)
```

The appropriate function is decided from the Execution array’s index which starts from 0 and ends at 28, representing the opcodes when they are subtracted by 1.

*Execution Array (cpu230exec.py, line 293)

```
# Instructions are numbered from 1 to 29.
# In this array, they are indexed from 0 to 28.
executionArr = [
    lambda addr_mode, operand: halt(),
    lambda addr_mode, operand: load(addr_mode, operand),
```

Those functions are dependent on 4 vital; read_operand, store_operand, not_op_16, and sub_op_16. "not_op_16" and "sub_op_16" functions do not and subscription operations accordingly; however, they are optimized for 16-bit memory. "read_operand" reads and returns data from the relevant address, register, or the operand itself according to address mode. "store_operand" writes data to the memory according to the address mode, and raises an exception if the address mode is 1 as it is not possible to write data to a constant value. Note that “read_operand” and “store_operand” functions read and write 2 byte value; therefore, the operand they get cannot be the maximum possible value (2^16-1). If this value is present in operand or the according register when address mode is 2 or 3, an exception will be raised. Besides, there are 27 registers in total (from PC, A to Z), so any value greater than 26 in operand when address code is 1 or 2 will result in an exception.

Other than these functions, all other functions are made compatible with 16 bit because of the virtual memory's style. On the other hand, the appropriate functions modify 3 flags; the carry flag (CF), the zero flag (ZF), and the sign flag (SF). These flags are used on conditional jump operations

*unconditional jump operation (cpu230exec.py, line 227)

```
    # Unconditional jump
    def jmp(addr_mode, operand):
        regs[0] = read_operand(addr_mode, operand)
        if regs[0] >= MAX_VAL-1:
            raise EndOfMemory
```

Besides those functions, print and read operations are the only operations that interact with the program's users. "read" operation gets the first character from the console before the enter key is pressed and stores the character in the "A" register. "print" operation is in contrast to the read operation as "print" operation prints a character from "A" register to the file.

*print and read operations (cpu230exec.py, line 282)

```
    # Read character from stdin
    def _read(addr_mode, operand):
        store_to_operand(addr_mode, operand, ord(input()[0]))

    # Write character to file
    def _print(addr_mode, operand):
        print(chr(read_operand(addr_mode, operand)), file=outFile)
```

Testing and Result

The program can be run in Python 3 with "python3 cpu230assemble.py" command plus a file with ".asm" extension. This will produce the binary input in the current directory with ".bin" extension. In the second pass, "python3 cpu230exec.py test1.bin" command will execute the binary code and produce the output in a text file.

In case of a syntax error, the program will not be producing the binary code at all and the user will be prompted with an error message including the line that syntax error occurs. Also definition of the same label twice in the same code will result in an error message as well.

Regarding these exceptional situations and our implementation of this hypothetical CPU called CPU230, the program runs and passes all the given test cases successfully.