CMPE230 - Project 2 Report

Hüseyin Türker Erdem - 2017400213

Halil Burak Pala - 2019400282

Overview

In this project, we are supposed to design an assembler and an executer, which are supposed to convert pre-defined assembly file to a binary file and execute the binary file. Language of the assembly file is defined in the project description.

General Approach

We are given a hypothetical system with a hypothetical CPU that has several registers. Other than the CPU, we are given a 64 KiB byte-addressable memory.

Our instruction set is 3B and we have the following format:

000000000000000000000000

opcode addr. mode operand

We interpreted each instruction in the "assempler.py" and we get hexadecimal instructions that refers to binary instructions as above.

Assembler

In "cmpe230assemble.py", we did string operations and conversions very much. First of all, we defined the main python script as a process line-by-line in the input file. Each line is tokenized by words at this point, and then we iterate over lines in order to seperate labels and instructions into two different lists.

```
    for line in lines:
        # This for line is for detecting and saving labels in 'labels' dictionary.
        if len(line) == 1 and line[0][-1] == ":":
            # If line is made up of one word and #this word's last character is ':', then it means
            # this line is a label. We save this label as key and its corresponding address as value
            # in dictionary.
            labels[line[0][0:-1].lower()] = format((i - noflabels)*3, 'x')
            noflabels += 1
        else:
            # If the line is not in the format of a label, the we save it to our instructions list.
            instructions.append(line)
            linenumbers.append(i+1)
        i += 1
        # At the end of previous if-else block, we eliminated all the labels in our code file
        # and saved all of the labels and their corresponding address in memory.
```

After successfully reading each instruction and label, conversion stage begins. In order to convert each instruction, we created a loop on it and on each iteration we set opcode, addressing mode and operand accordingly. At this point, there are many if conditions with string operations. These conditions are controlled by mnemonic string, which is the first word of the instruction. Then the possible addressing modes are set according to the project description.

```python
            elif mnemonic == "store":
                opcode = "3"
                memory = True
                register = True
            elif mnemonic == "add":
                opcode = "4"
                immediate = True
                memory = True
                register = True
            elif mnemonic == "sub":
                opcode = "5"
                immediate = True
                memory = True
                register = True
            elif mnemonic == "inc":
                opcode = "6"
                memory = True
                register = True
            elif mnemonic == "dec":
                opcode = "7"
```

After these control blocks, only two things left to implement, which are the convert method and operand check.

In convert method, we utilized Python built-in functions “format()” and “int()” that allows us to switch between strings and integers. Then, after a few lines of string and integer operations, we return the instruction in hexadecimal format.

```python
def convert(binarycode):
    '''
        This function converts a binary code given in the
        format "opcode addr_mode operand"
        into a hexadecimal code that we want. e.g:
        1c 1 3 ---> 710003
        2 0 41 ---> 080041
    '''
    result = []
    for code in binarycode:

        opcode   = int(code[0],16)
        addrmode = int(code[1],16)
        operand  = int(code[2],16)

        bopcode = format(opcode, '06b')
        baddrmode = format(addrmode, '02b')
        boperand = format(operand, '016b')
        bin = '0b' + bopcode + baddrmode + boperand
        ibin = int(bin[2:],2)
        instr = format(ibin, '06x')
        result.append(instr)
    return result

```

In operand check method, we take 3 flags that defines whether this operand can be an immediate data, a register or a memory location (direct or indirect). Other than the flags, the function recieves the operand (string), which is the second element of the instruction, and the labels (dictionary), which contains the addresses (string of hexadecimal digits) of labels.

In this method, there are again too many if conditions with string operations. At the end of the method, hexadecimal value of the operand and the addressing mode are returned, which are strings.

A few examples about this method are as follows:

input (operand	imm	reg	mem))	:return (operand	addresssing mode)	
PC	0	1	0	: 0	1	
PC	0	0	0	: x	x	(error)
Label	1	0	0	: 0AB5	0	(operand is an address)
Label	0	0	0	: x	x	(error)
0A41	1	0	0	: 0A410		
0A411	1	0	0	: A4110		
A411	1	0	0	: A4110		
“A”	1	0	0	: 61	0	(61 is ASCII value of “a”)
A	0	1	0	: 1	1	
[A]	0	0	1	: 0F2D2	(operand is the value in A)	
[“0412”]	0	0	1	: 04123		

Finally, this convert method is used in the main process. Each instruction is prepared to be printed to the output file as a single hexadecimal instruction.

The output of “assembly.py” is a single prog.bin (if the input is prog.asm). If “cmpe230assemble.py” gets an error at some point, it prints an error message to the terminal.

Executer

In “cmpe230exec.py”, we defined the system with 6 registers and a memory of 65536 addresses, each has 1B storage. Initial values in the memory, registers (except SP) and SP are 00, 0000 and FFFF in hexadecimal values. Output file is defined as “prog.txt” where “prog.bin” is the input file.

```
# 64 KB byte-addressable memory. Every entry is 1 byte.
memory = ["00"]*65536

# 1-bit flags, initially all of them set to zero (False).
ZF = False
CF = False
SF = False

# 2-byte registers PC, A, B, C, D, E, SP.
registers = ["0000", #PC
             "0000", #A
             "0000", #B
             "0000", #C
             "0000", #D
             "0000", #E
             "FFFF"] #SP

```

In this python scirpt, we have functions that do the operations and the main process that assigns values to the appropriate places.

In the main process, input file is read and lines are again set as a list. Each element in the list is a hexadecimal value, like “10FAD2”.

Before fetching and decoding the instructions from this list of instructions, each instruction is loaded into memory (each instruction occupies 3 addresses in the memory). After this operation, executer starts to read instructions from memory until it recieves a HALT instruction.

```python
    # load instructions into the memory:
    temp_pc = 0
    for line in lines:
        instruction = line[0]
        # notice that each instruction is 3B whereas each memory location can store 1B
        firstbyte = instruction[0:2]
        secondbyte = instruction[2:4]
        thirdbyte = instruction[4:6]
        memory[temp_pc] = firstbyte
        memory[temp_pc+1] = secondbyte
        memory[temp_pc+2] = thirdbyte
        temp_pc += 3

```

Then, at each instruction, current instruction is read from memory, which means we do not work with that older “lines” list anymore. Each instruction is converted to binary and the opcode, the addressing mode and the operand are decoded here. The addressing mode is stored as binary string whereas the opcode and the operand are hexadecimal strings.

```python

    # instruction fetch and decode:
    while True:
        current_pc = int(registers[0],16)
        firstbyte = memory[current_pc]
        firstbinary = format(int(firstbyte,16),"08b")

        # '00', '01', '02', ... , '0B', '0C', ... , or '1C'
        opcode = format(int(firstbinary[0:6],2),"02x")

        # '00','01', '10' or '11'
        addressingmode = firstbinary[6:8]
        secondbyte = memory[current_pc+1]
        thirdbyte = memory[current_pc+2]
        operand = secondbyte + thirdbyte

```

After these conversions, instructions can be easily executed. The main process continues with if conditions that controls the opcodes, so we can easily call the appropriate function for that particular instruction.

If the instruction is an arithmetic operation, then related function is called, such as ADD instructions call ADD function. Functions for arithmetic operations works very similar to each other. They take the operand, addressing mode and the necessary flags from the main process. These functions find the actual value that the operand actually points to, i.e. addressing mode is used in order to find the original value. Then the arithmetic operation is done and finally these functions set the flags and return result and the flags.

Below, we can see how ADD function works:

```python
# The function that takes the operand, addressing mode and flags.
# We first calculate the actual value that the operand  implies.
# Then the the value on the accumulator is prepared and both values are converted to int.
# After the addition of these integers, the result is set as binary by format function.
# Finally, the flags are set based on the binary result.
# The function returns hexadecimal result and the flags.
def ADD(operand, addressingmode, CF, ZF, SF):
    hexaddend1 = getoperand(operand, addressingmode)
    hexaddend2 = registers[1]
    binaryresult = format(int(hexaddend1,16) + int(hexaddend2,16),"016b")
    if len(binaryresult) == 16:
        CF = False
    elif len(binaryresult) == 17:
        CF = True
        binaryresult = binaryresult[1:]
    if binaryresult[0] == "1":
        SF = True
    else:
        SF = False
    if int(binaryresult,2) == 0:
        ZF = True
    else:
        ZF = False
    hexresult = format(int(binaryresult,2),"04x").upper()
    return (hexresult,CF, ZF, SF)
```

Here is the SHR function:

```python
# This function gets the binary value of the operand at first.
# Then it shifts all bits one to the right.
# The flags are set accordingly.
def SHR(operand, addressingmode, ZF, SF):
    hexoperand = getoperand(operand, addressingmode)
    binaryoperand = format(int(hexoperand,16),"016b")
    binaryresult = ("0" + binaryoperand)[:-1]
    hexresult = format(int(binaryresult,2),"04x")
    if binaryresult[0] == "1":
        SF = True
    else:
        SF = False
    if int(binaryresult,2) == 0:
        ZF = True
    else:
        ZF = False
    return (hexresult,ZF,SF)
```

As stated above, these functions are too similar except the arithmetic operation. We carefully controlled how to determine the flags in each function. Flags represent the same thing and their control is same, but in some particular functions we do not need to change them.

The SUB function is a little different. Instead of implementing a new arithmetic operation, we utilized our functions to find the result. As the instructor explained, we know that A - OP = A + (-OP) = A + NOT(OP) + 1. However, we are not supposed to change the flags at NOT and INC functions in SUB function. So, the resulting SUB function is as follows:

```python
# This function uses other functions to subtract operand from A.
# Since we know that the operand is unsigned, we don't need sign bit for it.
# First we use NOT and INC functions to convert the value of the operand to
# its negative form. But these functions do not change flags.
def SUB(operand,addressingmode,CF,ZF,SF):
    (temp1,_,_) = NOT(operand, addressingmode, ZF, SF)
    (temp2,_, _, _) = INC(temp1, "00", CF, ZF, SF)
    return ADD(temp2, "00", CF, ZF, SF)
```

Now, all the arithmetic operations can be assumed to be working. Remaining instructions are CMP, READ, PRINT and all jump instructions.

Read and print instructions are very simple. In read instruction, we read a string from the terminal and get the use its first character so as not to throw exception. Then, this character is converted to its ASCII value and this value is stored in the operand. Print instruction gets the ASCII character equvalent of the hexadecimal value on the operand. This character is printed to the output file.

```python
        elif opcode == "1b": #READ
            try :
                rawstr = input("enter:")
            except TypeError:
                print("exception")
            hexresult = format(ord(rawstr[0]),"04x")
            if addressingmode == "00":
                # error
                print("error2")
            elif addressingmode == "01":
                registers[int(operand,16)] = hexresult
            elif addressingmode == "10":
                memory[int(registers[int(operand,16)],16)] = hexresult[0:2]
                memory[int(registers[int(operand,16)],16)+1] = hexresult[2:4]
            elif addressingmode == "11":
                memory[int(operand,16)] = hexresult[0:2]
                memory[int(operand,16)+1] = hexresult[2:4]

        elif opcode == "1c": #PRINT
            outputfile.write(chr(int(getoperand(operand,addressingmode),16)))
            outputfile.write("\n")
```

At jump instructions, we utilized 2 sources:

"The *Hypothetical CPU in the project* is NOT 8086 - so do not confuse the two" at https://piazza.com/class/km84om1nme367e?cid=149

https://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/Lecture%2018%20Conditional%20Jumps%20Instructions.pdf

These sources are used in order to determine the conditions for the jump instructions. As we implemented in the main process, JA instruction in this project works exactly same as JG in 8086.

```python
        elif opcode == "12": #JMP
            registers[0] = getoperand(operand,addressingmode)
            continue

        elif opcode == "13": #JZ - JE
            if ZF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

        elif opcode == "14": #JNZ - JNE
            if not ZF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

```

```python
        elif opcode == "15": #JC
            if CF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

        elif opcode == "16": #JNC
            if not CF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

        elif opcode == "17": #JA
            if (not SF) and (not ZF):
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

```

```python
        elif opcode == "18": #JAE
            if not SF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

        elif opcode == "19": #JB
            if SF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

        elif opcode == "1a": #JBE
            if SF or ZF:
                registers[0] = getoperand(operand,addressingmode)
                continue
            else:
                pass

```

As we can see, only thing that the jump operations do is that PC is updated if flags are as expected. As stated above, we utilized those sources to determine the flags for each jump operation.

Finally, the CMP operation is the easiest one even if it looks much harder. Only difference between the SUB and CMP is that SUB writes the result onto A but CMP only changes flags.

```
        elif opcode == "05": #SUB
            (hexresult,CF,ZF,SF) = SUB(operand,addressingmode,CF,ZF,SF)
            registers[1] = hexresult

```

```
        elif opcode == "11": #CMP
            (hexresult,CF,ZF,SF) = SUB(operand,addressingmode,CF,ZF,SF)

```

Problems and Summary

There are a few problems we met during the implementation of the project. Some of these problems are minor problems and some other are major problems. We assumed the implementation that gives the correct output for the testcases as the solution for that particular problem.

We encountered case sensitivity problem in many different subprograms, so we solved this minor problem by using “String.lower()” function. So our submission is case insensitive.

In the assembly step, we are supposed to read following lines successfully:

load A	: load register A to register A.
load ‘A’	: load ASCII value of ‘A’ to register A.
load 0A	: load hexadecimal value of A to register A.

We forgot to read the third possibility, but when we noticed it we solved it easily.

A major problem was to determine how to set flags on each arithmetic operation. Then, after a research on the internet, we figured it out and solved the problem easily.

Hardest part in this project was to check each edge case from Piazza to get the answer of the instructor for that particular edge case. These edge cases can be listed as case sensitivity, 0A411-A411 case, JA ambiguity, how to set flags in SUB instruction, etc.

In summary, we learned very much during the project. We learned another detail about the CPUs and the instructions sets at each edge case. We finally managed to recieve valid outputs in the end.