Shiraz University

Assembly Project Phase 1

> This Project is coded and this documentation is compiled by **Amirali Kazerooni**

Description

This Project is an assembler written in python which takes assembly code for **x86** architecture and generates the machine code for it.

The following instructions is supported in the project:

- **ADD**
- **SUB**
- **INC**
- **DEC**
- **PUSH**
- **OR**
- **AND**
- **XOR**
- **POP**
- **JMP (Forward and Backward)**

There are four main lists and one in this project:

1. **reg_8**: Contains the string of 8 bit registers
2. **reg_16**: Contains the string of 16 bit registers
3. **reg_32**: Contains the string of 32 bit registers
4. **REG_values_of_registers**: The REG value of registers can be accessed by their name using this list

There is also an `OP table` in which the OP codes of instructions are separated by their **name** and then the different OP codes of that instruction is separated by **the number of bits** of source or destination operand(based on the case) and then based on the **source operand** of that instruction being **memory(True) or register(False)** the correct OP code can be accessed

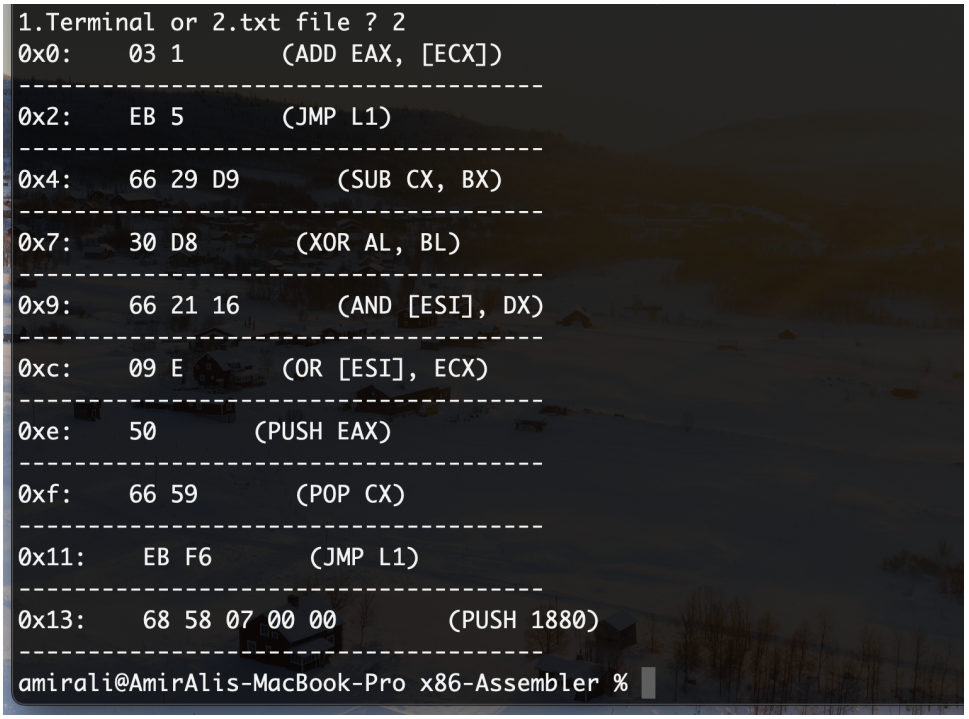
```
opcode_table = {
    "ADD": {8: {False: "00", True: "02"}, 16: {False: "66 01", True: "66 03"}, 32: {False: "01", True: "03"}},
    "SUB": {8: {False: "28", True: "2A"}, 16: {False: "66 29", True: "66 2B"}, 32: {False: "29", True: "2B"}},
    "AND": {8: {False: "20", True: "22"}, 16: {False: "66 21", True: "66 23"}, 32: {False: "21", True: "23"}},
    "OR": {8: {False: "08", True: "0A"}, 16: {False: "66 09", True: "66 0B"}, 32: {False: "09", True: "0B"}},
    "XOR": {8: {False: "30", True: "32"}, 16: {False: "66 31", True: "66 33"}, 32: {False: "31", True: "33"}},
    "INC": {8: "FE 00", 16: "66 FF 00", 32: "FF 00"},
    "DEC": {8: "FE 08", 16: "66 FF 08", 32: "FF 08"},
}
```

Example usage:

INPUT:

```
add eax, [ecx]
jmp L1
sub cx, bx
xor al, bl
L1:
AND [esi], dx
Or [esi], ecx
push eax
pop cx
jmp L1
push 1880
```

OUTPUT:



References Used for this project:

- http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0210
- https://c9x.me/x86/
- http://sparksandflames.com/files/x86InstructionChart.html

The followings are description on functions and general logics of the code…

Main()

The program starts here:

📌 1. First of all the input file is read and saved in a list called `test_cases` and then we iterate over each element of this list which is each test case(line) in the input file and we split each test case using the space delimiter to have **instruction, source operand and destination operand** separately.
2. Then we check the validity of the test case using `test_case_validity` function and we check to make sure this test case(line) is **not a Label** then we will do below bullets:

- The first thing we do is to see what the instruction of this test case is and based on that we add a value to our `finisher_counter` variable which is the variable that we use to count the number of bytes occupied in memory for later use in printing memory address of machine codes and in calculating the machine code of **JMP** instruction.
  The values added to the counter are per below based on instructions:
  - **JMP**: 2
  - **INC, DEC**: 1 if the operand is reg32 and else 2
  - **PUSH, POP**: 2 if the operand is reg16 , 1 if operand is reg32 and for pushing immediate, `getPushCode` function is called
  - **ELSE**: 3 if first operand is memory and reg16 and the second operand is reg16 else the value is 2
- Then based on the instruction we extract the operands and save in **dstOp**(destination operand) and **srcOP**(source operand)
  - If the instruction is **JMP** then `toMachineCode` function will be called with arguments being the **instruction, srcOp, dstOp, all the test cases, current test case and the finisher counter** but the srcOp is empty string.
  - If the instruction is **PUSH, POP, INC or DEC** then `getPushCode`, `getPopCode`, `getIncCode` or `getDecCode` will be called respectively and the argument of all being the **dstOp**
  - **ELSE**: `toMachineCode` function is called with the same arguments as before, after the src and dst operands are separated.
- The return value of `toMachineCode` function is saved in a variable named `Mcode` and in this last step we check to see if the variable is not

null(None) then we print the result for this test case with it's memory location which is calculated using the **finisher_counter** variable

read_file()

📌 This function basically reads the input file line by line and save each line as a element of the test_cases list

test_case_validity()

📌 This function checks the test cases per below bullets and if one of them was not satisfied this function returns **false**

- If the instructions is in the supported instruction by this project
- If instruction is either of PUSH, POP, INC, DEC then it's operand should not be memory
- If instruction is either of PUSH, POP, INC, DEC then it the number of operands provided for it should be **1** and if it is among other supported instruction then number of operands provided should be **3**
- If the operands are inside brackets then the inside of those brackets should be a valid register or if they are not inside brackets then again they should be a valid register (other than the case of **PUSH imm**)
- If an operand is in the brackets then it must be a 32bit register
- Both operands should nit be register

is_memory()

📌 This function basically checks to see if the operand starts with `[` charachter and ends with `]`

is_reg()

📌 This function basically checks to see if the operand is in one of these lists: **reg8, reg16 or reg32** and then if not returns **False**

toMachineCode()

📌 This function first checks to see if the instruction passed to it is **JMP** and if it is then it calculates the OP code of this test case by providing the the instruction, srcOp and destOp to a function called `opcodeBuilder` then the current test case and the list of all test cases and the destOp of JMP instruction is provided to another function called `label_looking_counter` which searches for the label among all the input file and then the **finisher_counter** is subtracted from the return value of this function which gives us length between label and current JMP test case and then `HEX` of this value will be concatenated to the OP code calculated earlier which gives us the machine code

Else if the instruction is not JMP then this function checks to see if it is either of POP, PUSH, INC, DEC and if is one of these instructions mentioned then it calculates the OP code using `opcodeBuilder` function but this time the srcOp being **None** and then the `REG` value of the destOp which is available in `REG_values_of_registers` list will be concatenated to this OP code

In other cases this function will calculate the OP code per before and then calculates `MOD` and `REG/RM` by calling `modBuilder` and `REG_RM_builder` functions respectively and providing destOp and srcOp as their arguments then the machine code will be `HEX` of MOD and REG/RM concatenated to each other like **MOD + REG_RM** code concatenated to OP code

Then the machine code calculated based on each case will be returned

opcodeBuilder()

📌 This function returns the OP code based on each instruction per below:

- **JMP**: the OP code will be `EB` in hex
- **PUSH, POP, INC, DEC**: the OP code will be extracted from `opcode_table` based on the instruction and the number of bits of it's operand
- **If srcOp is a reg16 or reg8 and destOp is memory**: the OP code will be extracted from `opcode_table` based on the instruction and the number of bits of **srcOp** and wether the srcOp is a memory or not
- **In other cases**: It's just like the before bullet but the number of bits is calculated based on **destOP** instead of srcOp

Note that the number of bits is calculated using `x_bits` function and providing the register as it's argument

x_bits()

📌 This function basically determines the number of bits the register passed to it contains by checking wether the register is in either of **reg8, reg16, or reg32** lists

label_looking_counter()

📌 Basically this function iterates over all the test cases until it finds the `label` of JMP instruction that it is looking for and by iterating over all the test cases it updates a counter based on the instructions it passes on and the values added for each time is based on the ones described in main function for **finisher_counter**

modBuilder()

📌 This function is used to distinguish between register or memory of an instruction's operands and if one of the operands of an instruction is memory it returns “00” else it returns `“11”`

REG_RM_builder()

📌 This function finds REG & R/M byte with the help of REG values of registers

- **If destOp is memory**: **REG** value will be the REG value of srcOp and **RM** value will be the REG value of destOp register
- **If srcOp is memory**: The values above will switch places
- **Other cases**: REG value will be the REG value of srcOp register and RM will be the REG value of destOp register

And finally this function returns **RM concatenated to REG** value

getDecCode()

📌 It returns the OP code for DEC instruction per below:

- **If destOp is reg32**: The HEX value of destOp register concatenated to 1001
- **If destOp is reg16**: The HEX value of destOp register concatenated to 1001 and then concatenated to `66` HEX(because of 16 bit registers)
- **Other cases**: The HEX value of destOp register concatenated to 11001 and then concatenated to `FE` HEX

getIncCode()

📌 It returns the OP code for DEC instruction per below:

- **If destOp is reg32**: The HEX value of destOp register concatenated to 1000
- **If destOp is reg16**: The HEX value of destOp register concatenated to 1000 and then concatenated to `66` HEX(because of 16 bit registers)
- **Other cases**: The HEX value of destOp register concatenated to 11000 and then concatenated to `FE` HEX

getPopCode()

📌 It returns the OP code for DEC instruction per below:

- **If destOp is reg32**: The HEX value of destOp register concatenated to 1011
- **If destOp is reg16**: The HEX value of destOp register concatenated to 1011 and then concatenated to `66` HEX(because of 16 bit registers)

getPushCode()

📌 It returns the OP code for DEC instruction per below:

- **If destOp is reg32**: The HEX value of destOp register concatenated to 1010
- **If destOp is reg16**: The HEX value of destOp register concatenated to 1010 and then concatenated to `66` HEX(because of 16 bit registers)
- **If destOp is immediate**:
  - If the value of immediate is greater than **127** then the `littleEndian` function is called with destOp argument and then the return value of it gets concatenated to `"68"` HEX
  - If the value of immediate is less or equal to 127 then it is just like before bullet but the retun value of little endian is concatenated to `"6A"` HEX

littlEndian()

📌 This function takes a number as it's input and returns the HEX of it but in little endian representation

decimalToHexConvertor() and binaryToHexConvertor()

📌 The first one takes a decimal number and returns the HEX of it in string format and the second one takes a binary number as input and then returns the HEX of it in string format