Assembler for x86 processors

Author: MohammadYasin Farshad

Shiraz University

Introduction

This is a project on the Assembly course at Shiraz University. Here we tried to design a simple assembler that is capable of converting low-level code of assembly language into machine language or computer 1s and 0s bits.

Reading and Compiling the Inputs

First, our program starts getting the inputs and saving them for later through this part of the code:

```
label = {}
for i in range(len(asm_code)):
    string = asm_code[i].split(":")
    string_instruction = string[len(string)-1].split()[0].lower()
    string_reg = string[len(string)-1].split()[1].split(",")
    for j in range(len(string_reg)):
        string_reg[j] = string_reg[j].lower()
    asm_code[i] = [string_instruction , string_reg]
    if len(string) > 1:
        label[string[0].lower()] = i
```

Note: we use dictionary to save labels and their addresses for later when we encounter the jump instruction.

Opcode

With the use of the dictionaries below, we were able to find the Opcode for instructions and registers needed.

```
reg32Bit = {
  "eax" :"000", "ebx" :"011", "ecx" :"001", "edx" :"010",
  "esp" :"100", "esi" :"110", "edi" :"111", "ebp" :"101"
}

reg16Bit = {
  "ax" :"000", "bx" :"011", "cx" :"001", "dx" :"010",
  "sp" :"100", "si" :"110", "di" :"111", "bp" :"101"
}

reg8Bit = {
  "al" :"000", "bl" :"011", "cl" :"001", "dl" :"010",
  "ah" :"100", "bh" :"111", "ch" :"101", "dh" :"110"
}

opCodesInstruction = {
    "add" :"000000", "sub" :"001010", "inc" :"111111", "dec" :"111111",
    "or"  :"000010", "and" :"001000", "xor" :"001100"
}
```

We also used this dictionary to access each register list to be able to simulate an assembly-like environment to run and examine assembly language (*this part is still a work in progress and not finished but we will discuss more about it later*).

```
all_registers = {"eax": eax ,"ebx": ebx ,"ecx": ecx ,"edx": edx ,"esp": esp
,"esi": esi ,"edi": edi ,"ebp": ebp ,"ax": ax ,"bx": bx ,"cx": cx ,"dx": dx
,"sp": sp ,"si": si ,"di": di ,"bp": bp,"al": al ,"ah": ah ,"bl": bl ,"bh": bh
,"cl": cl ,"ch": ch ,"dl": dl ,"dh": dh}
eax = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
ax = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
al = [0,0,0,0,0,0,0,0]
```

,etc.

Our output has different formats from one to another. In our project, the format of the output mostly comes back to the number of operands its instruction takes, for example, for **ADD**, **SUB**, **AND**, **OR**, and **XOR** the opcode breaks to 8 bits where the first part is the left four bits represent the instruction we used.

Right after that, we have another bit which we call **'d'** that shows which part is the destination or source of our line of code.

After 'd' we have **the size bit** that is either 0 or 1 depending on the size of our used register.

Now for the second byte for the first two bits, we have mod that specifies the type of process we are performing (*in this project we only encounter the 00 mod and 11 mod since we were either working with registers or their indirect addressing*).

After that, we have 6 bits where each 3 represents the register and/or the memory used in code (*the first 3 bits MUST be registers*).

Now for one operand instructions such as **INC, DEC, PUSH** and **POP**, the case is different. For these instructions, their Opcode is a summation of their value and the represented number of the register used in that line of code. Also, it is important to note for different sizes of registers their Opcode could become a bit different for example for **INC** and **DEC** when they are used on an 8bit register the Opcode is 2 bytes, not 1 which is because it receives an extra "FEh" byte on the left side.

As for **JMP** instruction Opcode is calculated differently yet again. To calculate the jmp Opcode you need to first calculate the address of the current line bitwise plus the size of that line itself and then minus the

result from the label address the line of code is pointing to and put the result next to JMP representation which is “EBh”.

Output and Functions

Our program calculates each of the mentioned Opcodes and prints them each but it’s also good to mention it has the function to actually run the codes but then again it is not finished YET.