x86 Assembler Design

Raha Rahmanian
CSE student at Shiraz University

Table of contents

Instructions

This projects' goal is to assemble a few of the Assembly x86 instructions into the corresponding machine code. I will briefly explain what each instruction does in assembly before we get to the code:

- ADD: add instruction takes two operands and adds their values and stores it in the first operand.
- SUB: sub instruction takes two operands and subtracts their values and stores it in the first operand.
- AND: and instruction takes two operands and performs bitwise and operation and stores it in the first operand.
- OR: or instruction takes two operands and performs bitwise or operation and stores it in the first operand.
- XOR: xor instruction takes two operands and performs bitwise xor operation and stores it in the first operand.
- INC: inc instruction takes one parameter, adds one to its value and stores it in the operand.
- DEC: dec instruction takes one parameter, subtracts one from its value and stores it in the operand.
- PUSH: push instruction takes one parameter and pushes the value to the runtime stack.
- POP: pop instruction takes one parameter; it pops the last value that has been pushed to stack and stores it in the parameter.
- JMP: jmp instruction transfers control to the offset of the label that comes in front of it.

You can read more about these instructions in kip Irvines Assembly x86 book. (you can get the book from this link)

Referral:

In this documentation I'm going to explain the core logic of the program and explain some of the key methods you can see the full project on my github.

Testing Tool:

I used shellstorm on x86(32) mode to debug and check my code.

Instruction Format

Instructions can be assembled to an encoding up to 16 bytes long but the implementations in this project only target instructions with an encoding without SIB, Displacement and Immediate bytes.

In this project I divided the instructions different parts and assessed them individually to avoid confusion:

- One Operand Instructions
    - Registers as input operand
    - Indirect addressing or Immediate as input operand
- Two Operand Instructions
    - Registers as input operands
    - Register/Indirect addressing as input operands

To construct the operation code of each instruction there are a couple of things we need to know

Firstly, we should know that each register has an individual code. Each register also has a size (1 byte 2 bytes or 4 bytes), this information is used in all of the instruction's encodings in different ways. We also should know that instructions might get prefixes for registers of different sizes.

Having this information, we can move on to how the op code for each instruction is encoded.

One Operand Instructions

Let's begin with one operand instructions, the encoding for these instructions is constructed of:

Prefix(optional) + op code byte + MODR/M byte

But there is an easier way than constructing the MODR/M byte, if you can find the hexadecimal value that is added to the register codes almost all one operand instructions can get encoded:

- INC:
    - 8 bit registers → prefix: feh / c0h + register code
    - 16 bit registers → prefix: 66h 40h + register code
    - 32 bit registers: 40h + register code

- DEC:
  - 8 bit registers → prefix: feh / c8h + register code
  - 16 bit registers → prefix: 66h 48h + register code
  - 32 bit registers: 40h + register code
- PUSH
  - Immediate → 68h / little endian 32 bit hexadecimal representation of the immediate
  - Indirect addressing → prefix: ffh / 30h + register code
  - 32 bit registers: 40h + register code
- POP
  - 16 bit registers → prefix: 66h / 58h + register code
  - 32 bit registers: 58h + register code

Two Operand Instructions

Now for the two operand instructions there is no shortcuts and the op code and the MODR/M byte should both be individually constructed:

OP code 8bits

Instruction code	d	s
The first 6 bits are the instruction code of that are unique for each instruction	1 if register is source 0 if register is destination	1 if the register size is 32 0 if register size is 8/16

MODR/M 8bits

MOD	REG	R/M
11 if operands are two registers 00 if one of the operands is indirect address	Destination register code if operands are two registers register code if one of the operands is indirect addressing	Source register code if operands are two registers Indirect addressing register code if one of the operands is indirect addressing

Knowing these, makes possible for us to construct the op code and the MODR/M for all of our instructions. I should also note that two operand instructions always get the ‘66h’ prefix when working with 16 bit registers.

The two methods in my code, twoOperandAssembler and twoOperandInstructionMODRM find these codes. You can trace the code below to get a better understanding:

```
public static String twoOperandInstructionMODRM(Dictionary<String, String>
regDict,String[] instruction){
    //I am assuming that it already is checked if both are registers ( so I
only have 11 as MOD )
    String answer = new String();
    int indirectAddressingRegister = whichIsIndirectAddressing(instruction);
    if(indirectAddressingRegister==0){
        answer += "11";
        //the REG bit is the destination register
        answer += regDict.get(instruction[2]);
        //the R/M bit is the source register
        answer += regDict.get(instruction[1]);
       }
    else{
        //the mod is always 00 for indirect addressing without displacement
        answer += "00";
        if(indirectAddressingRegister==1){
            answer += regDict.get(instruction[2]);
            answer +=
regDict.get(getIndirectAddressingRegister(instruction[1]));
        }
        else {
            answer += regDict.get(instruction[1]);
            answer +=
regDict.get(getIndirectAddressingRegister(instruction[2]));
        }
    }
    //turning the string hexadecimal
    return
beautifulHex(Integer.toHexString(Integer.valueOf(Integer.parseInt(answer,
2))));
}

public static String twoOperandAssembler(Dictionary<String,Integer>
regSize,Dictionary<String, String> opDict,Dictionary<String, String>
regDict,String[] instruction){
    String answer = new String();
    String tempAnswer = new String();
    int registerSize=0;
    int indirectAddressingRegister = whichIsIndirectAddressing(instruction);
    //checking if the instruction has two registers or one register and one
indirect address
    if(indirectAddressingRegister==0) {
        if (opDict.get(instruction[0]) != null) {
            if (regDict.get(instruction[1]) == null ||
regDict.get(instruction[2]) == null) {
                return "Error: The operation isn't between two registers";
            }
            if ((registerSize = regSize.get(instruction[1])) !=
regSize.get(instruction[2])) {
                return "Error: The operation is between two registers of
different size";
            }
            //adding prefix if the register is 16 bit
            if (registerSize == 2)
                answer += "66 ";
            tempAnswer += opDict.get(instruction[0]);
            //finding d bit of op code
            //because we are supporting the shell-storm format I put the d bit
0
            tempAnswer += "0";
            //finding s bit of opcode
            tempAnswer += findS(regSize, instruction[1]);
            answer +=
beautifulHex(Integer.toHexString(Integer.valueOf(Integer.parseInt(tempAnswer,
2))));
            answer += " " + twoOperandInstructionMODRM(regDict, instruction);
        }
        //Indirect addressing
    } else {
        //checking if the given string is actually a register or a number or
indirect address
        if(indirectAddressingRegister==1){
            if(regDict.get(instruction[2])==null ||
regDict.get(getIndirectAddressingRegister(instruction[1]))==null)
                return "Error: the given instruction operands are not supported
registers";
        }
        else{
            if(regDict.get(instruction[1])==null ||
regDict.get(getIndirectAddressingRegister(instruction[2]))==null)
                return "Error: the given instruction operands are not supported
registers";
        }

        //this part is for two operand instructions with one indirect
addressing op code
        tempAnswer += opDict.get(instruction[0]);
        if(indirectAddressingRegister==1){
            tempAnswer+="0";
            registerSize = regSize.get(instruction[2]);
        }
        if(indirectAddressingRegister==2){
            tempAnswer+="1";
            registerSize = regSize.get(instruction[1]);
        }
        //determining d bit of the opcode 8bits based on the size of the
register
        if(registerSize==1)
            tempAnswer+="0";
        else if (registerSize==2){
            answer += "66 ";
            tempAnswer+="1";
        }
        else
            tempAnswer+="1";
        answer +=
beautifulHex(Integer.toHexString(Integer.valueOf(Integer.parseInt(tempAnswer,
2))));
        //now finding the modr/m byte
        answer += " " + twoOperandInstructionMODRM(regDict, instruction);
    }
    return answer;
}
```

JMP instruction

Lastly I will discuss the jump instruction. The challenge in encoding the short jump instruction is not in construction of the opcode or MODRM bytes because the op code is the same for all short jump instructions (ebh).

The real challenge of jump instruction is finding the bytes between the instructions offset and the label that it is jumping to. In order to implement this you should first know how to find the offset of each instruction.

The offset of each instruction in a listing file starts from 0000h (virtual offset) this offset is later replaced with the real place that the program is in the memory.

So starting from 0000h we can find each instructions offset by adding the previous lines' offset to the number of bytes that the instruction encoding of that line has.

After finding the offsets of each line the jump instruction encoding can be completed:

Forward jump
Label offset – 2 – jmp offset

Backward jump
Label offset – jmp offset + 2

Note: the numbers should be written in hexadecimal and the backward jump should change to two's complement because its value is negative.

Java helpful methods:

Here are two of the methods that I used repeatedly in my code:

```
Integer.toHexString(int);
```

This method will change integer to hexadecimal, It also automatically gives the two's complement representation for negative numbers given to it.

```
Integer.parseInt(String,radix);
```

This method will change a given string of digits to a number in whatever radix that you give it.

Part 2

The second part of this project is showing how the data is stored in the memory witch is basically the explanation of part 2 documentation and there is no further explanation needed on the matter.