# 实验六

读取文件程序

```
# 将读取到的数据定义到cpu内
def define_date(commend: str, s: dict,memory:list,date_ip:int,sn:int):
    commend = commend.split(' ')
    # print(commend)
    c2 = commend[2].split('\'')
    # print(c2)
    # 是数字
    if len(c2) == 1:
        s[commend[0]] = sn
        memory[date_ip+sn] = c2[0]
    # 是字符串
    else:
        s[commend[0]] = sn
        memory[date_ip+sn] = c2[1]



# 从文件中解析数据段
def decode_date(file,date,memory,date_ip):
    s = 0
    line = file.readline()
    line = line.replace('\n', '')
    # 直到读取到DATA SEGMENT
    while line != 'DATA SEGMENT':
        line = file.readline()
        line = line.replace('\n', '')

    command = file.readline()
    command = command.split(';')[0].strip()
    while command != 'DATA ENDS':
        if command != '':
            define_date(command, date,memory,date_ip,s)
            s += 1
        command = file.readline()
        command = command.split(';')[0].strip()


# 定义堆栈段
def define_sta(commend: str,sta:list):
    commend = commend.split(' ')
    c2 = commend[2].split('(')[1].split(')')[0]
    s = [c2] * int(commend[1])
    sta.extend(s)


# 读取文件中的堆栈段
def decode_sta(file,sta):
    line = file.readline()
    line = line.replace('\n', '')
    # 直到读取到STACK SEGMENT STACK
    while line != 'STACK SEGMENT STACK':
        line = file.readline()
        line = line.replace('\n', '')

    command = file.readline()
    command = command.split(';')[0].strip()
    # 直到读取到STACK ENDS
    while command != 'STACK ENDS':
        if command != '':
            define_sta(command,sta)
        command = file.readline()
        command = command.split(';')[0].strip()


# 读取文件中的指令段，将指令读入内存
def decode_code(file,memory,iden,ip):
    line = file.readline()
    line = line.replace('\n', '')
    # 直到读取到CODE SEGMENT
    while line != 'CODE SEGMENT':
        line = file.readline()
        line = line.replace('\n', '')
    # 直到读取到ASSUME CS:CODE, DS:DATA, SS:STACK
    while line != 'ASSUME CS:CODE, DS:DATA, SS:STACK':
        line = file.readline()
        line = line.replace('\n', '').strip()

    command = file.readline()
    command = command.split(';')[0].strip()
    # 直到读取到CODE ENDS
    while command != 'CODE ENDS':
        if command != '':
            identifier = command.split(':')
            if len(identifier) == 2:
                # 将标记存入iden字典
                iden[identifier[0]] = ip
            else:
                memory[ip] = (identifier[0])
                ip += 1
            # define_code(command,memory)
        command = file.readline()
        command = command.split(';')[0].strip()
#主要程序
def decode_file(code,date,sta,memory,ip,date_ip):
    with open(code,'r',encoding='utf-8') as file:
        decode_date(file,date,memory,date_ip)
        decode_sta(file,sta)
        decode_code(file,memory,date,ip)
```

运行程序

```
import queue
import threading
```

```python
import demo

class Cpu:
    def __init__(self):
        self.data_segment = {}     # 定义数据段
        self.iden = {'DATA': '10'} #定义跳转标志
        self.memory = ["0"] * 200    # 内存
        self.queue = queue.Queue()   # 指令队列
        self.stack = []         # 堆栈
        self.address_bus = {}        # 地址总线
        self.data_bus = {}           # 数据总线
        self.control_bus = {}        # 控制总线
        # 专用寄存器
        self.special_registers = {
            "DS": 10,  # 数据段 0 415  地址为DS*16 + AX  (AX <= 255)
            "CS": 0,   # 代码段  416 开始到 1000
            "SS": 0,   # 堆栈段 单独设置
            "ES": 0,   # 附加段
            "IP": 0    # 指令寄存器
        }
        # 通用寄存器
        self.general_registers = {
            "AX": 0,
            "AH": 0,
            "AL": 0,
            "BX": 0,
            "BH": 0,
            "BL": 0,
            "CX": 0,
            "CH": 0,
            "CL": 0,
            "DX": 0,
            "DH": 0,
            "DL": 0,
            "SP": 0,  # 堆栈指针
            "BP": 0,  # 存取堆栈指针
            "DI": 0,  # 目的变址寄存器
            "SI": 0   # 源变址寄存器
        }
        # 标志寄存器
        self.flags = {
            "CF": 0,  # 进位标志位
            "PF": 0,  # 奇偶标志位
            "AF": 0,  # 辅助进位标志位
            "ZF": 0,  # 零标志位
            "SF": 0,  # 符号标志位
            "OF": 0,  # 溢出标志位
            "IF": 1   # 中断标志位

        }
        # 指令集
        self.instructions = {
            'MOV'  : self.mov,
            'PUSH' : self.push,
            'POP'  : self.pop,
            'XCHG' : self.xchg,
            'ADD'  : self.add,
            'SUB'  : self.sub,
            'ADC'  : self.adc,
            'SBB'  : self.sbb,
            'INC'  : self.inc,
            'DEC'  : self.dec,
            'MUL'  : self.mul,
            'IMUL' : self.imul,
            'DIV'  : self.div,
            'IDIV' : self.idiv,
            'AND'  : self.And,
            'OR'   : self.Or,
            'XOR'  : self.Xor,
            'NOT'  : self.Not,
            'TEST' : self.Test,
            'MOVSB': self.movsb,
            'MOVSW': self.movsw,
            'CMPSB': self.cmpsb,
            'CMPSW': self.cmpsw,
            'SCASB': self.scasb,
            'SCASW': self.scasw,
            'LODSB': self.lodsb,
            'LODSW': self.lodsw,
            'STOSB': self.stosb,
            'STOSW': self.stosw,
            'NOP'  : self.nop,
            'CLC'  : self.clc,
            'STC'  : self.stc,
            'CMC'  : self.cmc,
            'CLCD' : self.cld,
            'STD'  : self.std,
            'CLI'  : self.cli,
            'STI'  : self.sti,
            'HLT'  : self.hlt,
            'INT'  : self.Int,
            'LEA'  : self.lea
        }


    # 将16进制转10进制
    def issixteen(self,value:str):
        value = str(value).split('h')
        s = value

        # 10进制
        if len(s) == 1:
            num = int(s[0])
            return num
            # 16进制
        elif len(s) == 2:
            num = int(s[0], 16)
            return num
    #解析指令
    def match(self,str):
        c0 = None
        c1 = None
        c2 = None
        c3 = None
        s = str.split(' ')
        c0 = s[0]
        if len(s) == 3:
            s1 = str.replace(s[0],' ').strip()
            ss = s1.split(' ')
            # print(ss)
            # 中间无空格 MOV AL, 15
            if len(ss) == 2:
                c1 = ss[0].replace(',','').strip()
                c2 = ss[1]
            #中间有空格 MOV AL,15
            elif len(ss) == 1:
                ss = ss[0].split(',')
                c1 = ss[0]
                c2 = ss[1]
        elif len(s) == 2:
            c1 = s[1]
        elif len(s) == 4:
            s1 = str.replace(s[0],' ').strip()
            ss = s1.split(' ')
            c1 = ss[0].replace(',', '').strip()
            c2 = ss[2]
            c3 = ss[1]
        return c0,c1,c2,c3
    # 获取指令
    def fetch(self):
        if len(self.memory) > self.special_registers["IP"] and
self.memory[self.special_registers["IP"]] != '0' :
            component = self.memory[self.special_registers["IP"]]
            print(f"正在从内存地址 {self.special_registers['IP']} 获取指令:
{component}")
            op,reg,val,com = self.match(component)
            # print(op,reg,val)
            if op == 'JMP':
                self.jmp(reg)
            elif op == 'CALL':
                self.call(reg)
            elif op == 'RET':
                self.ret()
            elif op == 'IRET':
                self.iret()
            elif op == 'LOOP':
                self.Loop(reg)
            elif op == 'LOOPZ':
                self.Loopz(reg)
            elif op == 'LOOPNZ':
                self.Loopnz(reg)
            elif op == 'HTL':
                self.hlt()
            else:
                self.special_registers["IP"] += 1
            return component
        else:
            exit(0)
    # 解码指令
    def decodes(self, part):
        if self.queue.qsize() != None:
            op,reg,val,com = self.match(part)
            if op in ['INC', 'DEC']:
                print(f"正在解码指令: {part} 为 操作码 {op}, 寄存器 AX  ")
            elif op in ['MUL', 'DIV']:
                print(f"正在解码指令: {part} 为 操作码 {op}, 寄存器 AX ")
            else:
                print(f"正在解码指令: {part} 为 操作码 {op}, 寄存器 {reg}, 值 {val} ")
            return op, reg, val

    # 执行指令
    def execute(self, operation, register, value):
        if operation in self.instructions:
            if operation == 'HLT':
                self.instructions[operation]()
            elif value is not None:
                if value.isdigit():
                    value = int(value)
                self.instructions[operation](register, value)
            elif register is not None:
                self.instructions[operation](register)
            else:
                self.instructions[operation]()
            self.print_registers()

    # 地址解析
    def address_resolution(self, value):
        # 立即数寻址 100
        value = str(value)
        global address
        s = value.split('h')
        if s[0].isdigit() or value.count('h')==1:
            # 10进制
            if len(s) == 1:
                num = int(s[0])
                return num
            # 16进制
            elif len(s) == 2:
                num = int(s[0], 16)
                return num
        # 寄存器寻址 AX
        elif value in self.general_registers:
            return self.general_registers[value]
        else:
            parts = value.split(':')
            if len(parts) == 2:
                sr = parts[0]
                g = parts[1]
                gr = g[1:len(g) - 1] #去掉了[]
                # DS:[BX] 寄存器间接寻址
                if gr in self.general_registers:
                    address = self.special_registers[sr] * 16 +
self.general_registers[gr]
```

```
                print(address)
                return int(self.memory[address])
            # DS:[100] 直接寻址
            elif gr.isdigit():
                address = int(self.special_registers[sr]) * 16 + int(gr)
                return int(self.memory[address])
        elif len(parts) == 1:
            g = parts[0]
            gr = g[1:len(g) - 1] #去掉了[]
            string = gr.split('+')
            if len(string) == 2:
                s1 = string[0]
                s2 = string[1]
                if s2.isdigit():
                    # [SI+CNT] 相对寻址
                    if s1 == 'BP':
                        address = self.special_registers['ss'] * 16 +
self.general_registers['BP'] + int(s2)
                    elif s1 == 'BX':
                        address = self.special_registers['DS'] * 16 +
self.general_registers['BX'] + int(s2)
                    elif s1 == 'SI':
                        address = self.special_registers['DS'] * 16 +
self.general_registers['SI'] + int(s2)
                    elif s1 == 'DI':
                        address = self.special_registers['DS'] * 16 +
self.general_registers['DI'] + int(s2)
                    return int(self.memory[address])
                else:
                    # [BX+SI] 基址变址寻址
                    if s1 == 'BX':
                        address = self.special_registers['DS'] * 16 +
self.general_registers['BX'] + self.general_registers[s2]
                    elif s1 == 'BP':
                        address = self.special_registers['SS'] * 16 +
self.general_registers['BP'] + self.general_registers[s2]
                    return int(self.memory[address])
            elif len(string) == 1:
                string = string[0]
                if string.count('[') == 1:
                # [BX][SI]  基址变址寻址
                    s1 = string[1:3]
                    s2 = string[5:7]
                    if s1 == 'BX':
                        address = self.special_registers['DS'] * 16 +
self.general_registers['BX'] + self.general_registers[s2]
                    elif s1 == 'BP':
                        address = self.special_registers['SS'] * 16 +
self.general_registers['BP'] + self.general_registers[s2]
                    return int(self.memory[address])
                elif string.count('[') == 0:
                    # [BX] 间接寻址
                    s1 = string
                    if s1 == 'BP':
                        address = self.special_registers['SS'] * 16 +
int(self.general_registers[s1])
```

```
                        elif s1 in self.data_segment:
                            address = self.special_registers['DS'] * 16 +
int(self.data_segment[s1])
                        else:
                            address = self.special_registers['DS'] * 16 +
int(self.general_registers[s1])
                        print(address)
                        return int(self.memory[address])


    # 当低和高位寄存器改变时，同时调整整个寄存器
    def change_hl(self,register):
        if register in ['AH', 'AL', 'BH', 'BL', 'CH', 'CL', 'DH', 'DL']:
            self.general_registers['AX'] = int(self.general_registers['AH']) * 16
+ int(self.general_registers['AL'])
            self.general_registers['BX'] = int(self.general_registers['BH']) * 16
+ int(self.general_registers['BL'])
            self.general_registers['CX'] = int(self.general_registers['CH']) * 16
+ int(self.general_registers['CL'])
            self.general_registers['DX'] = int(self.general_registers['DH']) * 16
+ int(self.general_registers['DL'])

    # 改变标志寄存器
    def change_flags(self, a, b, c, op, hl):
        if bin(c).replace('0b', '').count('1') % 2 == 0:
            self.flags['PF'] = 1
        else:
            self.flags['PF'] = 0
        if  c == 0 :
            self.flags['ZF'] = 1
        else:
            self.flags['ZF'] = 0
        if c < 0:
            self.flags['SF'] = 1
        else:
            self.flags['SF'] = 0
        if c>255 :
            self.flags['OF'] = 1
        else:
            self.flags['OF'] = 0
        if op == '+' :
            sa = a & 3
            sb = b & 3
            sc = sa+sb
            if sc & 4 == 1:
                self.flags['AF'] = 1
            else:
                self.flags['AF'] = 0
            ta = a & 127
            tb = b & 127
            tc = ta+tb
            if tc & 128 == 1:
                self.flags['CF'] = 1
            else:
                self.flags['CF'] = 0
        elif op == '-':
            sa = a & 3
            sb = b & 3
            if sa-sb < 0:
                self.flags['AF'] = 1
            else:
                self.flags['AF'] = 0
            if c < 0:
                self.flags['CF'] = 1
            else:
                self.flags['CF'] = 0

        elif op == '*' :
            if hl == 1:
                if c > 255:
                    self.flags['AF'] = 1
                    self.flags['CF'] = 1
                else:
                    self.flags['AF'] = 0
                    self.flags['CF'] = 0
            elif hl == 2:
                if self.general_registers['DX'] > 0:
                    self.flags['AF'] = 1
                    self.flags['CF'] = 1
                else:
                    self.flags['AF'] = 0
                    self.flags['CF'] = 0

    # 调整通用寄存器
    def adjust_register(self):
        self.general_registers['AH'] = self.general_registers['AX'] // 16
        self.general_registers['AL'] = self.general_registers['AX'] % 16
        self.general_registers['BH'] = self.general_registers['BX'] // 16
        self.general_registers['BL'] = self.general_registers['BX'] % 16
        self.general_registers['CH'] = self.general_registers['CX'] // 16
        self.general_registers['CL'] = self.general_registers['CX'] % 16
        self.general_registers['DH'] = self.general_registers['DX'] // 16
        self.general_registers['DL'] = self.general_registers['DX'] % 16

    # 转移指令
    # 将值移动到指定寄存器
    def mov(self, register1:str, value):
        if register1 in self.general_registers:
            if value in self.data_segment:
                self.general_registers[register1] = int(self.data_segment[value])
            elif value in self.iden:
                self.general_registers[register1] = int(self.iden[value])
            else:
                self.general_registers[register1] =
int(self.address_resolution(value))
            self.change_hl(register1)

        elif register1 in self.special_registers:
            if value in self.data_segment:
                self.general_registers[register1] = int(self.data_segment[value])
            elif value in self.iden:
                self.general_registers[register1] = int(self.iden[value])
            else:
                self.special_registers[register1] =
int(self.address_resolution(value))
        else:
            re = register1.replace('[','').replace(']','')
            re = int(self.data_segment[re])
            add = self.special_registers['DS'] * 16 + re
            if value in self.data_segment:
                self.memory[add] = int(self.data_segment[value])
            elif value in self.iden:
                self.memory[add] = int(self.iden[value])
            else:
                self.memory[add] = int(self.address_resolution(value))

    def lea(self, register1:str, value):
        if register1 in self.general_registers:
            if value in self.data_segment:
                self.general_registers[register1] = int(self.data_segment[value])
            elif value in self.iden:
                self.general_registers[register1] = int(self.iden[value])
            else:
                self.general_registers[register1] =
int(self.address_resolution(value))
            if register1 in ['AH','AL','BH','BL','CH','CL','DH','DL']:
                self.general_registers['AX'] = int(self.general_registers['AH'])
* 16 + int(self.general_registers['AL'])
                self.general_registers['BX'] = int(self.general_registers['BH'])
* 16 + int(self.general_registers['BL'])
                self.general_registers['CX'] = int(self.general_registers['CH'])
* 16 + int(self.general_registers['CL'])
                self.general_registers['DX'] = int(self.general_registers['DH'])
* 16 + int(self.general_registers['DL'])

        elif register1 in self.special_registers:
            if value in self.data_segment:
                self.general_registers[register1] = int(self.data_segment[value])
            elif value in self.iden:
                self.general_registers[register1] = int(self.iden[value])
            else:
                self.special_registers[register1] =
int(self.address_resolution(value))
        else:
            re = register1.replace('[', '').replace(']', '')
            re = int(self.data_segment[re])
            add = self.special_registers['DS'] * 16 + re
            if value in self.data_segment:
                self.memory[add] = int(self.data_segment[value])
            elif value in self.iden:
                self.memory[add] = int(self.iden[value])
            else:
                self.memory[add] = int(self.address_resolution(value))

    # 入栈
    def push(self, register1):
        if register1.isdigit():
            self.stack.append(register1)
        elif register1 in self.general_registers:
            self.stack.append(self.general_registers[register1])

    # 出栈
    def pop(self, register1):
        self.general_registers[register1] = self.stack.pop()

    # 交换值
    def xchg(self, register1, value):
        self.general_registers[register1], self.general_registers[value] =
self.general_registers[value], self.general_registers[register1]

    # 算数运算指令
    # 加法
    def add(self, register1, value):
        num = self.issixteen(value)
        self.general_registers[register1] += num
        self.change_hl(register1)
        # 给指定寄存器中的值加上一个数
        a = self.general_registers[register1]
        b = num
        c = a + b
        self.change_flags(a, b, c, '+', 1)

    # 带进位加法
    def adc(self, register1, value):
        num = self.issixteen(value)
        self.general_registers[register1] += self.flags['CF']
        self.general_registers[register1] += num
        a = self.general_registers[register1] + 1
        b = num
        c = a + b
        self.change_flags(a, b, c, '+', 1)

    # 减法
    def sub(self, register1, value):
        num = self.issixteen(value)
        self.general_registers[register1] -= num
        self.change_hl(register1)
        a = self.general_registers[register1]
        b = num
        c = a - b
        self.change_flags(a, b, c, '-', 1)

    # 带借位减法
    def sbb(self, register1, value):
        num = self.issixteen(value)
        self.general_registers[register1] -= self.flags['CF']
        self.general_registers[register1] -= num
        a = self.general_registers[register1] - 1
        b = num
        c = a - b
        self.change_flags(a, b, c, '-', 1)

    # 乘法
    def mul(self, register1):
        if register1 in ['AL', 'AH', 'BL', 'BH', 'CL', 'CH', 'DL', 'DH']:
            a = self.general_registers[register1]
            b = self.general_registers['AL']
            c = a * b
            self.change_flags(a, b, c, '*', 1)
            self.general_registers['AX'] = self.general_registers[register1] *
self.general_registers['AL']
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16
        elif register1 in ['AX', 'BX', 'CX', 'DX']:
            a = self.general_registers[register1]
            b = self.general_registers['AX']
            c = a * b
            self.change_flags(a,b,c,'*',2)
            self.general_registers['DX'] = self.general_registers[register1] *
self.general_registers['AX'] // 256
            self.general_registers['DH'] = self.general_registers['DX'] // 16
            self.general_registers['DL'] = self.general_registers['DX'] % 16
            self.general_registers['AX'] = self.general_registers[register1] *
self.general_registers['AX'] % 256
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16
        elif register1.isdigit():
            a = int(register1)
            b = self.general_registers['AX']
            c = a * b
            self.change_flags(a, b ,c, '*', 2)
            self.general_registers['DX'] = a * self.general_registers['AX'] //
256
            self.general_registers['DH'] = self.general_registers['DX'] // 16
            self.general_registers['DL'] = self.general_registers['DX'] % 16
            self.general_registers['AX'] = a * self.general_registers['AX'] % 256
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16

    # 带符号乘法
    def imul(self, register1):
        if register1 in ['AL', 'AH', 'BL', 'BH', 'CL', 'CH', 'DL', 'DH']:
            a = self.general_registers[register1]
            b = self.general_registers['AL']
            c = a * b
            self.change_flags(a, b, c, '*', 1)
            self.general_registers['AX'] = self.general_registers[register1] *
self.general_registers['AL']
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16
        if register1 in ['AX', 'BX', 'CX', 'DX']:
            a = self.general_registers[register1]
            b = self.general_registers['AX']
            c = a * b
            self.change_flags(a, b, c, '*', 2)
            self.general_registers['DX'] = self.general_registers[register1] *
self.general_registers['AX'] // 256
            self.general_registers['DH'] = self.general_registers['DX'] // 16
            self.general_registers['DL'] = self.general_registers['DX'] % 16
            self.general_registers['AX'] = self.general_registers[register1] * 
self.general_registers['AX'] % 256
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16

    # 除法
    def div(self, register1):
        if register1 in ['AL', 'AH', 'BL', 'BH', 'CL', 'CH', 'DL', 'DH']:
            self.general_registers['AL'] = self.general_registers['AX'] // 
self.general_registers[register1]
            self.general_registers['AH'] = self.general_registers['AX'] % 
self.general_registers[register1]
            self.general_registers['AX'] = self.general_registers['AH'] * 16 + 
self.general_registers['AL']
        elif register1 in ['AX', 'BX', 'CX', 'DX']:
            self.general_registers['DX'] = (self.general_registers['DX'] * 256 + 
self.general_registers['AL']) // self.general_registers[register1] // 256
            self.general_registers['DH'] = self.general_registers['DX'] // 16
            self.general_registers['DL'] = self.general_registers['DX'] % 16
            self.general_registers['AX'] = (self.general_registers['DX'] * 256 + 
self.general_registers['AL']) % self.general_registers[register1] % 256
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16
        elif register1.isdigit():
            self.general_registers['AL'] = self.general_registers['AX'] // 
int(register1)
            self.general_registers['AH'] = self.general_registers['AX'] % 
int(register1)
            self.general_registers['AX'] = self.general_registers['AH'] * 16 + 
self.general_registers['AL']

    # 带符号除法
    def idiv(self, register1, value):
        if register1 in ['AL', 'AH', 'BL', 'BH', 'CL', 'CH', 'DL', 'DH']:
            self.general_registers['AL'] = self.general_registers['AX'] // 
self.general_registers[register1]
            self.general_registers['AH'] = self.general_registers['AX'] % 
self.general_registers[register1]
            self.general_registers['AX'] = self.general_registers['AL'] * 16 + 
self.general_registers['AH']
        if register1 in ['AX', 'BX', 'CX', 'DX']:
            self.general_registers['DX'] = (self.general_registers['DX'] * 256 + 
self.general_registers['AL']) // self.general_registers[register1] // 256
            self.general_registers['DH'] = self.general_registers['DX'] // 16
            self.general_registers['DL'] = self.general_registers['DX'] % 16
            self.general_registers['AX'] = (self.general_registers['DX'] * 256 + 
self.general_registers['AL']) % self.general_registers[register1] % 256
            self.general_registers['AH'] = self.general_registers['AX'] // 16
            self.general_registers['AL'] = self.general_registers['AX'] % 16
    # 自增
    def inc(self,register1):
        a = 1
        if register1 in self.special_registers:
            a = self.general_registers[register1]
            self.general_registers[register1] += 1
        elif register1 in self.general_registers:
```

```python
            a = self.general_registers[register1]
            self.general_registers[register1] += 1
            self.change_hl(register1)
        if a > 0 and a+1 > 255:
            self.flags['OF'] = 1
        else:
            self.flags['OF'] = 0
    # 自减
    def dec(self,register1):
        a = -1
        if register1 in self.special_registers:
            a = self.general_registers[register1]
            self.general_registers[register1] -= 1
        elif register1 in self.general_registers:
            a = self.general_registers[register1]
            self.general_registers[register1] -= 1
            self.change_hl(register1)
        if a < 0 and a-1 < -255:
            self.flags['OF'] = 1
        else:
            self.flags['OF'] = 0

    # 逻辑运算指令
    # 与
    def And(self, register1, value):
        s = str(value).split('h')
        if s[0].isdigit() or value.count('h') == 1:
            num = self.issixteen(value)
            self.general_registers[register1] &= num
        else:
            self.general_registers[register1] &= self.address_resolution(value)
        self.change_hl(register1)
        c = self.general_registers[register1]
        if bin(c).replace('0b','').count('1') % 2 == 0:
            self.flags['PF'] = 1
        else:
            self.flags['PF'] = 0
        self.flags['CF'] = 0
        self.flags['OF'] = 0

    # 或
    def Or(self, register1, value):
        s = str(value).split('h')
        if s[0].isdigit():
            num = self.issixteen(value)
            print(num)
            self.general_registers[register1] |= num
        else:
            self.general_registers[register1] |= self.address_resolution(value)
        self.change_hl(register1)
        c = self.general_registers[register1]
        if bin(c).replace('0b','').count('1') % 2 == 0:
            self.flags['PF'] = 1
        else:
            self.flags['PF'] = 0
        self.flags['CF'] = 0
        self.flags['OF'] = 0

    # 异或
    def Xor(self, register1, value):
        s = str(value).split('h')
        if s[0].isdigit():
            num = self.issixteen(value)
            self.general_registers[register1] ^= num
        else:
            self.general_registers[register1] ^= self.address_resolution(value)
        self.change_hl(register1)
        c = self.general_registers[register1]
        if bin(c).replace('0b', '').count('1') % 2 == 0:
            self.flags['PF'] = 1
        else:
            self.flags['PF'] = 0
        self.flags['CF'] = 0
        self.flags['OF'] = 0
    # 测试指令
    def Test(self, register1, value):
        if str(value).isdigit():
            c = self.general_registers[register1] & int(value)
        else:
            c =self.general_registers[register1] & self.address_resolution(value)
        if bin(c).replace('0b','').count('1') % 2 == 0:
            self.flags['PF'] = 1
        else:
            self.flags['PF'] = 0
        self.flags['CF'] = 0
        self.flags['OF'] = 0

    # 取反
    def Not(self, register1):
        if register1 in ['AH','AL','BH','BL','CH','CL','DH','DL']:
            self.general_registers[register1] = 15-
self.general_registers[register1]
        elif register1 in ['AX','BX','CX','DX']:
            self.general_registers[register1] = 255-
self.general_registers[register1]
        self.change_hl(register1)

    # 字符串指令
    # DSI -> ESI
    def movsb(self):
        str1 = self.special_registers["DS"] * 16 + self.general_registers["SI"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        self.memory[str2] = self.memory[str1]
        print(f"内存中地址为{str2}的值变为{self.memory[str1]}")
        self.general_registers['SI'] += 1
        self.general_registers['DI'] += 1
    # DSI -> ESI (两位)
    def movsw(self):
        str1 = self.special_registers["DS"] * 16 + self.general_registers["SI"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        self.memory[str2],self.memory[str2 + 1] =
self.memory[str1],self.memory[str1 + 1]
```

```
            print(f"内存中地址为{str2}的值变为{self.memory[str1]}")
            print(f"内存中地址为{str2+1}的值变为{self.memory[str1+1]}")
            self.general_registers['SI'] += 2
            self.general_registers['DI'] += 2

    # 比较 ESI 和 DSI 改变标志位
    def cmpsb(self):
        str1 = self.special_registers["DS"] * 16 + self.general_registers["SI"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        if self.memory[str2] == self.memory[str1] :
            print(f"内存中地址为{str2}的值和内存中地址为{str1}的值相等，ZF变为1")
            self.flags['ZF'] = 1
        else:
            print(f"内存中地址为{str2}的值和内存中地址为{str1}的值不相等，ZF变为0")
            self.flags['ZF'] = 0
        self.general_registers['SI'] += 1
        self.general_registers['DI'] += 1

    # 比较 ESI 和 DSI 改变标志位（两位）
    def cmpsw(self):
        str1 = self.special_registers["DS"] * 16 + self.general_registers["SI"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        if self.memory[str2] == self.memory[str1] and self.memory[str2 + 1] ==
self.memory[str1 + 1] :
            print(f"内存中地址为{str2}的值和内存中地址为{str1}的值相等，并且内存中地址为
{str2+1}的值和内存中地址为{str1+1}的值也相等，ZF变为1")
            self.flags['ZF'] = 1
        else:
            self.flags['ZF'] = 0
        self.general_registers['SI'] += 1
        self.general_registers['DI'] += 1

    # 比较 ESI 和 AL 改变标志位
    def scasb(self):
        str1 = self.general_registers["AL"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        if self.memory[str2] == str1:
            print(f"内存中地址为{str2}的值和寄存器AL的值相等，ZF变为1")
            self.flags['ZF'] = 1
        else:
            self.flags['ZF'] = 0
            print(f"内存中地址为{str2}的值和寄存器AL的值不相等，ZF变为0")
        self.general_registers['DI'] += 1

    # 比较 ESI 和 AX 改变标志位
    def scasw(self):
        str1 = self.general_registers["AX"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        if self.memory[str2] == str1:
            print(f"内存中地址为{str2}的值和寄存器AX的值相等，ZF变为1")
            self.flags['ZF'] = 1
        else:
            print(f"内存中地址为{str2}的值和寄存器AX的值不相等，ZF变为0")
            self.flags['ZF'] = 0
        self.general_registers['DI'] += 1
```

```python
    # 将地址值存储到AL中
    def lodsb(self):
        str = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        print(f"将寄存器AL的值变为内存中地址为{str}的值")
        self.general_registers["AL"] = int(self.memory[str])
        self.general_registers['DI'] += 1
    # 将地址值存储到AX中
    def lodsw(self):
        str = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        print(f"将寄存器AX的值变为内存中地址为{str}的值")
        self.general_registers["AX"] = int(self.memory[str])
        self.general_registers['DI'] += 1
    # 将AL存储到ESI中
    def stosb(self):
        str1 = self.general_registers["AL"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        print(f"将内存中地址为{str2}的值变为寄存器AL的值{str1}")
        self.memory[str2] = str1
        self.general_registers['DI'] += 1
    # 将AX存储到ESI中
    def stosw(self):
        str1 = self.general_registers["AX"]
        str2 = self.special_registers["ES"] * 16 + self.general_registers["DI"]
        print(f"将内存中地址为{str2}的值变为寄存器AX的值{str1}")
        self.memory[str2] = str1
        self.general_registers['DI'] += 1
    def nop(self):
        print("执行了NOP指令")
    def clc(self):
        self.flags['CF'] = 0
    def stc(self):
        self.flags['CF'] = 0
    def cmc(self):
        self.flags['CF'] = 0 if self.flags['CF'] == 1 else 1
    def cld(self):
        self.flags['DF'] = 0
    def std(self):
        self.flags['DF'] = 1
    def cli(self):
        self.flags['IF'] = 0
    def sti(self):
        self.flags['IF'] = 1
    def jmp(self,value):
        ip = self.special_registers['IP']
        if value.isdigit():
            self.special_registers['IP'] = int(value)
            print(f"程序IP由{ip}跳到{int(value)}")

        elif value in self.data_segment:
            self.special_registers['IP'] = int(self.data_segment[value])
            print(f"程序IP由{ip}跳到{int(self.data_segment[value])}")
        else:
            self.flags['IP'] = self.general_registers[value]
            print(f"程序IP由{ip}跳到{self.general_registers[value]}")
    def call(self,value):
        ip = self.special_registers['IP'] + 1
        print(ip)
        self.stack.append(int(ip))
        if value.isdigit():
            self.special_registers['IP'] = int(value)
            print(f"程序IP由{ip}跳到{int(value)}")
        else:
            self.flags['IP'] = self.general_registers[value]
            print(f"程序IP由{ip}跳到{self.general_registers[value]}")
    def ret(self):
        ip = self.stack.pop()
        self.special_registers['IP'] = ip
        print(f"程序IP跳回{self.special_registers['IP']}")
    def Int(self,value):
        if self.flags['IF'] != 1:
            print("中断程序未开启")
        else:
            # 保存寄存器状态，压入栈
            #还未写
            str = value.replace('h','')
            if int(str) == 21:
                if self.general_registers['AH'] == 2:
                    print(f"DL的值为{self.general_registers['DL']}")
                elif self.general_registers['AH'] == 76:
                    self.hlt()

    def iret(self):
        #无中断返回
        print("执行了IRET指令")
        self.special_registers["IP"] += 1
    def Loop(self,value):
        ip = self.special_registers['IP']
        self.general_registers['CX'] -= 1
        if self.general_registers['CX'] > 0:
            if value.isdigit():
                self.special_registers['IP'] = int(value)
                print(f"程序IP由{ip}跳到{int(value)}")
            else:
                self.flags['IP'] = self.general_registers[value]
                print(f"程序IP由{ip}跳到{self.general_registers[value]}")
        else:
            self.special_registers['IP'] += 1

    def Loopz(self,value):
        ip = self.special_registers['IP']
        self.general_registers['CX'] -= 1
        if self.general_registers['CX'] != 0 and self.flags['ZF'] == 1:
            if value.isdigit():
                self.special_registers['IP'] = int(value)
                print(f"程序IP由{ip}跳到{int(value)}")
            else:
                self.flags['IP'] = self.general_registers[value]
                print(f"程序IP由{ip}跳到{self.general_registers[value]}")
        else:
            self.special_registers['IP'] += 1
    def Loopnz(self,value):
        ip = self.special_registers['IP']
```

```python
            self.general_registers['CX'] -= 1
            if self.general_registers['CX'] != 0 and self.flags['ZF'] == 0:
                if value.isdigit():
                    self.special_registers['IP'] = int(value)
                    print(f"程序IP由{ip}跳到{int(value)}")
                else:
                    self.flags['IP'] = self.general_registers[value]
                    print(f"程序IP由{ip}跳到{self.general_registers[value]}")
            else:
                self.special_registers['IP'] += 1

    # 停机指令
    def hlt(self):
        # 停止执行
        print("停止执行")
        exit(0)

    def print_registers(self):

        self.adjust_register()
        # 输出所有寄存器的状态
        print("通用寄存器状态:")
        for reg, val in self.general_registers.items():
            print(f"{reg}: {val} ", end='')
        print("\n专用寄存器状态:")
        for reg, val in self.special_registers.items():
            print(f"{reg}: {val} ", end='')
        print("\n标志寄存器状态:")
        for reg, val in self.flags.items():
            print(f"{reg}: {val} ", end='')
        print('\n')


    # 运行biu
    def biu_run(self):
        print("biu开始执行")
        while True:
            com = self.fetch()
            self.queue.put(com)

    # 运行eu
    def eu_run(self):
        print("\neu开始执行")
        while True:
            op, reg, val = self.decodes(self.queue.get())
            self.execute(op, reg, val)
            if op == 'HLT':
                break

if __name__ == "__main__":
    cpu = Cpu()
    code = './input'
    ip = 0 #指令段起始地址
    date_ip = 160 #数据段起始地址
    demo.decode_file(code,cpu.data_segment,cpu.stack,cpu.memory,ip,date_ip)
    eu = threading.Thread(target=cpu.eu_run)
    biu = threading.Thread(target=cpu.biu_run)
    biu.start()
    eu.start()
    biu.join()
    eu.join()
```

```
设计思路
1.定义结构体Cpu
    属性:
            内存，指令队列，堆栈，三条总线，通用寄存器，标志寄存器，专用寄存器，指令集
    函数:
            fetch() 获取指令
            decode() 解码指令
            execute() 执行指令
            address_resolution() 地址解析
            change_flags() 改变标志寄存器
            adjust_register()  调整通用寄存器
            print_registers() 输出所有寄存器状态
            update_buses()  输出所有总线状态
            指令函数 :
                    mov(),puch(),pop(),xchg(),

add(),adc(),sub(),sbb(),mul(),imul(),div(),idiv(),inc(),dec(),
                    and(),or(),xor(),test(),not(),
                    hlt()
2.总线接口单元(BIU)
    定义函数biu_run():
            调用fetch()函数从内存中读取指令到指令队列
            输出指令所用时间
3.执行单元（EU）
    定义函数eu_run():
            从指令队列中获取指令，解码指令decode()，执行指令execute(),打印cpu状态
print_registers()，update_buses()
            输出执行每条任务所用时间
4.主函数
    创建两个线程:
            eu = threading.Thread(target=cpu.eu_run)
            biu = threading.Thread(target=cpu.biu_run)
    同时执行，模拟BIU，EU同时工作
```