实验四 实现部分指令集

```
import queue
import threading
import time

class Cpu:
    def __init__(self):
        self.memory = ["0"] * 1000   # 内存
        self.queue = queue.Queue()  # 指令队列
        self.stack = [] * 100       # 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": 1,  # 数据段 0 415  地址为DS*16 + AX  (AX <= 255)
            "CS": 11,   # 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
        }
        # 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,
            'HLT' : self.hlt
        }

    # 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}")
            self.special_registers["IP"] += 1
            return component
        else:
            exit(0)

    # 解码指令
    def decodes(self, part):
        if self.queue.qsize() != None:
            t_d1 = time.process_time()
            parts = part.split(' ')
            op = parts[0]
            reg = parts[1] if len(parts) >= 2 else None
            val = parts[2] if len(parts) == 3 else None
            t_d2 = time.process_time()
            if op in ['INC','DEC']:
                print(f"正在解码指令: {part} 为 操作码 {op}, 寄存器 AX  所用时间为{t_d2 -
t_d1}")
            elif op in ['MUL','DIV']:
                print(f"正在解码指令: {part} 为 操作码 {op}, 寄存器 AX  所用时间为{t_d2 -
t_d1}")
```

```
            else:
                print(f"正在解码指令: {part} 为 操作码 {op}, 寄存器 {reg}, 值 {val} 所用时间为
{t_d2 - t_d1}")
            return op, reg, val

    # 执行指令
    def execute(self, operation, register, value):
        if operation in self.instructions:
            t_e1 = time.process_time()
            if operation == 'HLT':
                self.instructions[operation]()
            elif value is not None:
                if value.isdigit():
                    value = int(value)
                self.instructions[operation](register, value)
                print(f"寄存器 {register} 的新值为 {self.general_registers[register]}")
            elif register is not None:
                self.instructions[operation](register)
            else:
                self.instructions[operation]()
                print(f"寄存器 AX 的新值为 {register}")
            t_e2 = time.process_time()
            print(f"正在执行指令 {operation}...所用时间为{t_e2 - t_e1}")

            self.print_registers()
            self.update_buses(operation, register, value)

    # 地址解析
    def address_resolution(self, value):
        # 立即数寻址 100
        value = str(value)
        global address
        if value.isdigit():
            return int(value)
        # 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]
                # 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]
                    return int(self.memory[address][0])
                # DS:[100] 直接寻址
                elif gr.isdigit():
                    address = int(self.special_registers[sr]) * 16 + int(gr)
                    return int(self.memory[address][0])
```

```
            elif len(parts) == 1:
                g = parts[0]
                gr = g[1:len(g) - 1]
                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][0])
                    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][0])
                elif len(string) == 1:
                    # [BX][SI]  基址变址寻址
                    string = string[0]
                    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][0])
                # [BX] 间接寻址
                parts = parts[0]
                s1 = parts[1:len(parts)-1]
                if s1 == 'BP':
                    address = self.special_registers['SS'] * 16 +
int(self.general_registers[s1])
                else:
                    address = self.special_registers['DS'] * 16 +
int(self.general_registers[s1])
```

```
            return int(self.memory[address][0])

    # 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, value):

        if register1 in self.general_registers:
            self.general_registers[register1] = self.address_resolution(value)
        elif register1 in self.special_registers:
            self.special_registers[register1] = 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):
        self.general_registers[register1] += self.address_resolution(value)
        # 给指定寄存器中的值加上一个数
```

```
        a = self.general_registers[register1]
        b = self.address_resolution(value)
        c = a + b
        self.change_flags(a,b,c,'+',1)

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

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

    # 带借位减法
    def sbb(self, register1, value):
        self.general_registers[register1] -= self.flags['CF']
        self.general_registers[register1] -= self.address_resolution(value)
        a = self.general_registers[register1] - 1
        b = self.address_resolution(value)
        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):
        a = self.general_registers['AX']
        self.general_registers['AX'] += 1
        self.general_registers['AH'] = self.general_registers['AX'] // 16
        self.general_registers['AL'] = self.general_registers['AX'] % 16
        if a > 0 and self.general_registers['AX'] > 255:
            self.flags['OF'] = 1
        else:
            self.flags['OF'] = 0
    # 自减
    def dec(self):
        a = self.general_registers['AX']
        self.general_registers['AX'] -= 1
        self.general_registers['AH'] = self.general_registers['AX'] // 16
        self.general_registers['AL'] = self.general_registers['AX'] % 16
        if a < 0 and self.general_registers['AX'] < -255:
            self.flags['OF'] = 1
        else:
            self.flags['OF'] = 0
```

```
    # 逻辑运算指令
    # 与
    def And(self, register1, value):
        if str(value).isdigit():
            self.general_registers[register1] &= int(value)
        else:
            self.general_registers[register1] &= self.address_resolution(value)
        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):
        if str(value).isdigit():
            self.general_registers[register1] |= int(value)
        else:
            self.general_registers[register1] |= self.address_resolution(value)
        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):
        if str(value).isdigit():
            self.general_registers[register1] ^= int(value)
        else:
            self.general_registers[register1] ^= self.address_resolution(value)
        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):
        self.general_registers[register1] = ~self.general_registers[register1]

    # 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')

    # 更新总线状态
    def update_buses(self, operation, register, value):
        if operation == 'MOV':
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = self.address_resolution(value)
            self.control_bus['read'] = True
            self.control_bus['write'] = False
        elif operation == 'PUSH':
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = self.general_registers[register]
            self.control_bus['read'] = False
            self.control_bus['write'] = True
        elif operation == 'POP':
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = self.general_registers[register]
            self.control_bus['read'] = True
            self.control_bus['write'] = False
        elif operation == ['XCHG', 'TEST', 'HTL']:
            self.address_bus['source'] = ''
            self.address_bus['destination'] = ''
```

```
            self.data_bus['data'] = 0
            self.control_bus['read'] = False
            self.control_bus['write'] = False
        elif operation in ['ADD', 'ADC', 'SUB', 'SBB']:
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = self.address_resolution(value)
            self.control_bus['read'] = True
            self.control_bus['write'] = True
        elif operation == ['INC','DEC']:
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = 1
            self.control_bus['read'] = False
            self.control_bus['write'] = False
        elif operation == ['MUL', 'IMUL']:
            self.address_bus['source'] = f"{register} AL"
            self.address_bus['destination'] = f"AX"
            self.data_bus['data'] = self.general_registers['AX']
            self.control_bus['read'] = True
            self.control_bus['write'] = True
        elif operation == ['DIV', 'IDIV']:
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"AH AL"
            self.data_bus['data'] = str(self.general_registers['AH']) + ' ' +
str(self.general_registers['AL'])
            self.control_bus['read'] = True
            self.control_bus['write'] = True
        elif operation == ['AND', 'OR', 'XOR']:
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = str(self.general_registers[register]) + ' ' +
str(self.address_resolution(value))
            self.control_bus['read'] = False
            self.control_bus['write'] = False
        elif operation == 'NOT':
            self.address_bus['source'] = f"{register}"
            self.address_bus['destination'] = f"{register}"
            self.data_bus['data'] = self.general_registers[register]
            self.control_bus['read'] = False
            self.control_bus['write'] = True

        print("总线状态:")
        print("地址总线:")
        for key, val in self.address_bus.items():
            print(f"{key}: {val}")
        print("数据总线:")
        for key, val in self.data_bus.items():
            print(f"{key}: {val}")
        print("控制总线:")
        for key, val in self.control_bus.items():
            print(f"{key}: {val}")
```

```
        print('\n')

    # 运行biu
    def biu_run(self):
        print("biu开始执行")
        while True:
            t1 = time.process_time()
            com = self.fetch()
            self.queue.put(com)
            t2 = time.process_time()
            print(f"将指令传入指令队列所用时间为{t2 - t1}秒")
    # 运行eu
    def eu_run(self):

        print()
        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()
    cpu.memory = ['0'] * 1000  #定义内存大小
    cpu.memory[116] = ['111'] #地址段
    cpu.memory[0:7] = ["MOV AX 1","MOV BX 3","TEST AX BX","AND AX 0","OR AX 1","XOR BX
3","NOT AX","HTL"] #指令段

    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
```

运行结果

寻址方式

立即数寻址 ["MOV AX 1"]

寄存器寻址 ["MOV AX 3","MOV BX AX"]

直接寻址 DS为1 指令为["MOV AX DS:[100]",'HLT'] 地址存储值为111

寄存器间接寻址 DS为1 BX为100 指令为["MOV BX 100","MOV AX DS:[BX]",'HLT'] 地址存储值为111

基址加变址寻址 DS为1 BX为84 SI为16 指令为["MOV BX 84","MOV SI 16","MOV AX [BX+SI]",'HLT'] 地址存储值为111

相对寻址 DS为1 SI为16 指令为["MOV SI 16","MOV AX [SI+84]",'HLT'] 地址存储值为111

MOV PUSH POP XCHG 指令 ["MOV AX 88","MOV CX 99","PUSH AX","POP BX","XCHG AX CX",'HLT']

ADD,SUB,INC,DEC,MUL,DIV 指令 ["MOV AX 3","MOV BX 1","ADD AX BX","SUB AX BX","INC","DEC","MUL 2","DIV 2","HTL"]

AND OR XOR TEST NOT 指令 ["MOV AX 1","MOV BX 3","TEST AX BX","AND AX 0","OR AX 1","XOR BX 3","NOT AX","HTL"]