虚拟机设计

报告

姓　　名：　　张绍磊

班　　级：　　2016211310

学　　号：　　2016211392

课程名称：　　计算机组成原理课程设计

实验环境：　　C++ & QT

目录

一、任务描述

利用高级语言实现硬件虚拟机。

设计模型机指令系统，小系统基本要求：64 位 CPU、1MB 内存、数据通路、IO。并设计测试程序，进行验收。对计算机的基本组成、部件的设计、部件间的软件连接、指令集设计、程序运行流程有更深的了解，加深对理论课程的理解。

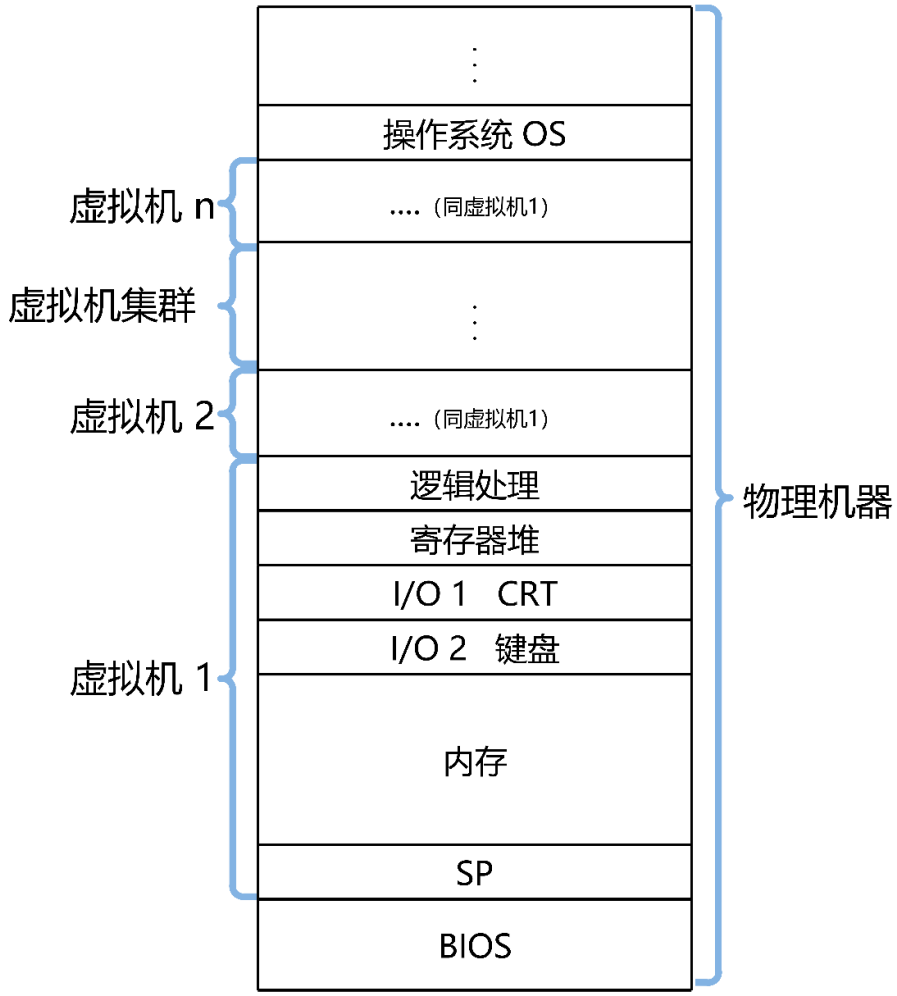
二、虚拟机原理

运行时系统是计算机程序的执行环境，它为分为两种：一种是由处理器自身直接执行该处理器上的程序指令，处理器提供了一种执行程序指令的机制，计算机的操作系统和处理器就构成了程序中指令的运行时系统；另外一种运行系统中，程序的指令不是由处理器直接执行而是完全通过一个软件系统——虚拟机来执行，即指令是通过软件而不是硬件来执行。

虚拟机是仿真实现计算机体系结构的一个软件程序，它也会像处理器那样取出指令、执行指令，但二者的区别在于虚拟机的指令执行过程是发生在软件级而不是硬件级。虚拟机为上层的应用程序提供了一个运行环境，它向所有应用程序提供一致的接口来屏蔽了底层硬件结构的差异。虚拟机在系统中的位置如图所示：

应用程序
操作系统
虚拟机
硬件系统

三、整体构架设计



虚拟机指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。虚拟机的软件从电脑资源中分出一部分的 CPU、内存、硬盘存储....等等，然后虚拟机软件把这些资源整合，组成了一台电脑。

虚拟机不包含物理机器中的 BIOS 和操作系统。虚拟机包括逻辑处理、寄存器堆、鼠标和键盘外部设备的 I/O、SP 和内存，其中内存包括虚拟机测试程序、数据区和 I/O 区。

一部物理机器中可以包含多个独立的虚拟机，构成虚拟机集群。各个虚拟机构造相似，公用同一个物理机器但独立工作，互不影响。

四、显示界面设计

显示界面包括：控制区、状态区、程序显示区、寄存器显示区、堆栈、输出区、手动输入区、日志区。

1. 控制区，包含 11 和按钮和一个滑块：

- 开机：开启虚拟机，完成初始化。
- 装载：选择测试程序，并且加载到存储器的程序区。
- 单拍：单步执行 1 条指令。
- 连续：连续执行测试程序。
- 滑块：调整主频，控制连续执行的快慢。
- 停机：关闭虚拟机，清空界面。
- 帮助：显示帮助文档及测试教程。
- 退出：退出程序，关闭界面。
- 清空：清空手动输入区的内容。

- 执行：将手动输入区的指令，加载到中断子程序，执行中断子程序。

2. 状态区，包含虚拟机运行状态和错误及异常：

   - 状态： 0:未开机；1:开机未加载程序；2:程序加载完成；3:程序运行中；4:中断中；5:停机；
   - 异常：0:程序运行正常；1:未加载程序时开始执行；2:指令包含非法操作数；3:寄存器超出规定数量；4:条件跳转时，未比较，但尝试跳转；5:在存储器中寻址时超出预设范围；6:弹栈时，堆栈里无数据；7:堆栈区溢出；8:陷入循环；9:中断中再次申请中断；10:未中断时，中断恢复；11:未中断时，手动输入指令；12:除数为 0；

3. 程序显示区，包含主程序和中断子程序：

   - 测试程序：选择加载的测试程序，为主程序
   - 中断子程序：手动输入的中断子程序。

4. 寄存器显示区：

   - 特殊寄存器：显示 PC、IAR、IR、SP、PSW 的值
   - 通用寄存器：显示 16 个同用寄存器 AX——PX 的值。

5. 堆栈区：显示堆栈中的数据。

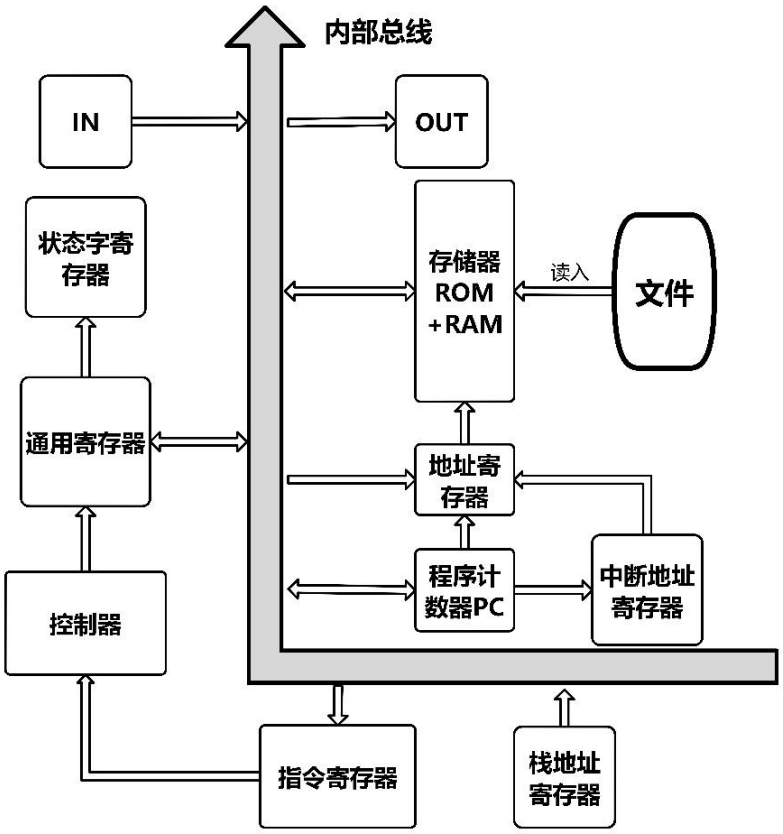
6. 输出区：将寄存器的值输出到屏幕上。

7. 手动输入区：中断时，通过命令行手动输入中断子程序。

8. 日志区：显示并记录每一条指令的操作过程。

这种显示界面设计，保证虚拟机内部透明，可以时刻掌握虚拟机当前运行状态，以及各部件的数据。方面调试，和验证正确性。

五、硬件框架



存储器（ROM+RAM）	存储测试程序及数据
地址寄存器	用来记录即将访问的内存地址
程序计数器 PC	用来记录当前执行的程序地址
指令寄存器	记录当前执行的指令
控制器	通过识别指令，控制数据通路执行程序
通用寄存器	记录 64 位数据
状态字寄存器	记录上一次比较的结果
栈地址寄存器	记录当前栈顶地址
中断地址寄存器	记录中断前程序运行地址
IN/OUT	输入输出

六、软件框架

寄存器实现：

通用寄存器：共 8 个通用寄存器。AX、BX、CX、DX……HX，其中 AX 为累加器。每个通用寄存器通过 int 型整数实现，其中数据为 64 位，用 16 位 16 进制数表示。

数据缓冲寄存器：通过一维数组 DR[16]实现，其中数据为 64 位，用 16 位 16 进制数表示。

程序计数器、地址寄存器、中断地址寄存器、栈地址寄存器：通过 int 型整数实现，大小为 0x0000-0xFFFF。

指令寄存器：

状态字寄存器：通过 int 型整数实现，数值为 1、2、3。

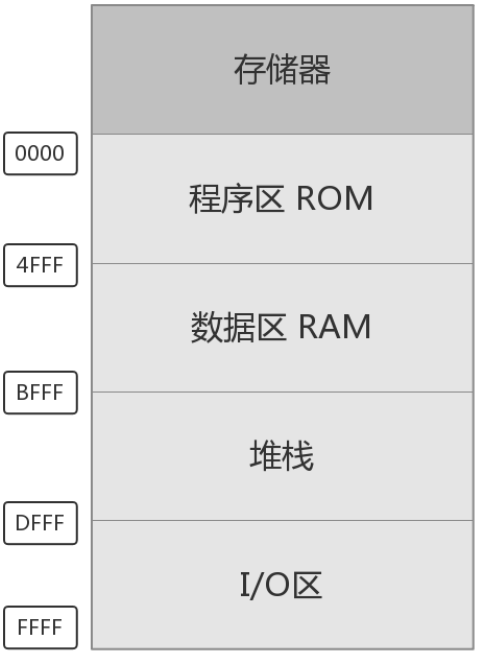
存储器实现：

存储器：通过二维数组 memory[0xFFFF][16] 实现。其中地址为 0x0000-0xFFFF，数据为 64 位，用 16 位 16 进制数表示。

存储器地址划分成 4 个部分：

- 程序区 ROM：地址为 0000-4FFFF。通过文件导入测试程序，只读。
- 数据区 RAM：地址为 5000-BFFF。存储数据，可读可写。
- 堆栈区：地址为 C000-DFFF。存放堆栈中的数据。
- I/O 区：地址为 E000-FFFF。存放通过手动输入的中断子程序。

寄存器	
PC	程序计数器
AR	地址寄存器
IR	指令寄存器
DR	数据缓冲寄存器
IAR	中断地址寄存器
PSW	状态字寄存器
SAR	栈地址寄存器



七、指令集设计

指令集使用 RISC 精简指令。参考 x86 汇编指令，在其基础上做了一些简化，有所改动。

指令集包含了开机指令、停机指令、显示指令、算术运算指令、逻辑运算指令、移位指令、数据传送指令、程序控制类指令、中断指令、I/O 指令，能完成计算机基本操作。

每一条指令不超过 12 位，前 3 位（4 位）表示操作数，后面是参与操作的寄存器或者内存地址。立即数、寄存器、内存地址表示方法如下：

- **立即数**：一个 16 位的 16 进制常数，XXXXXXXXXXXXXXXX。不会省略前导零，字母使用大写，如 000002C002C002C0；

- **寄存器**：共 8 个通用寄存器。用 AX、BX、CX、DX……HX 表示，字母皆为大写。其中 AX、BX、CX 为累加器，中断时需要保护累加器中的数据。
- **内存地址**：采用 ‘‘立即数直接寻址’’。通过 4 位 16 进制数 XXXX 表示，如 02C0，表示内存地址 02C0。

操作指令：

开机	RUN	标识着程序的开始。如无特殊说明，内存和寄存器均已初始化为 0。
停机	STOP	标识着程序的正常结束。
显示	ECHO AX	将寄存器 AX 中的值输出。

算术运算指令：

加法	ADD AX BX	将操作数 AX 中的值与 BX 中的值相加，结果存回 AX。相加产生溢出时，直接将溢出部分丢弃即可（截断）
减法	SUB AX BX	将操作数 AX 中的值减去 BX 中的值，结果存回 AX。
乘法	MUL AX BX	将操作数 AX 中的值与 BX 中的值相乘，结果存回 AX。相乘产生溢出时，直接将溢出部分丢弃即可（截断）
除法	DIV AX BX	将操作数 AX 中的值除以 BX 中的值，结果整数部分存回 AX。

模	MOL AX BX	将操作数 AX 中的值对 BX 中的值取模，结果存回 AX。
加 1	INC AX	将操作数 AX 中的值加 1，结果存回 AX。同样忽略溢出。
减 1	DEC AX	将操作数 AX 中的值减 1，结果存回 AX。
比较	CMP AX BX	比较操作数 AX 和 BX 中的值的大小，结果将存入 PSW，作为条件跳转指令的依据。

逻辑运算指令：

逻辑与	AND AX BX	将寄存器 AX 中的值与寄存器 BX 中的值按字节相与，结果存回寄存器 AX。
逻辑或	OR AX BX	将寄存器 AX 中的值与寄存器 BX 中的值按字节相或，结果存回寄存器 AX。
逻辑非	NOT AX	将寄存器 AX 中的值按字节取反，结果存回寄存器 AX。

移位指令：

左移	SAL AX	算术左移。把寄存器 AX 中数据的低位向高位移，空出的低位补 0，再存回 AX。
右移	SAR AX	算术右移。把寄存器 AX 中数据的高位

		向低位移，空出的高位用最高位（符号位）填补，再存回 AX。

数据传送类指令：

转移	MOV AX BX	将寄存器 BX 中的值写入寄存器 AX。
取数	LAD AX XXXX	将地址 XXXX 中的值写入寄存器 AX。
存数	STO XXXX AX	将寄存器 AX 中的值写入地址 XXXX。
置入存储器	SET XXXX XXXX	将立即数 XXXX 中的值置入内存地址 XXXX。
置入寄存器	SAVE AX XXXX	将立即数 XXXX 中的值置入寄存器 AX。
压栈	PUSH AX	将寄存器 AX 中的值压入堆栈顶。
弹栈	POP AX	将栈顶数据弹出，存进寄存器 AX 中。

程序控制类指令：

无条件跳转	JMP XXXX	直接跳转。该指令执行完后，将去执行第 XXXX 条指令。
大于时跳转	JG XXXX	a 大于 b 时跳转（PSW=1）。
小于时跳转	JL XXXX	a 小于 b 时跳转（PSW=2）。
等于时跳转	JE XXXX	a 等于 b 时跳转（PSW=3）。

中断指令：

中断允许	EI	中断允许。

中断返回	IRET	中断返回。

I/O 指令：

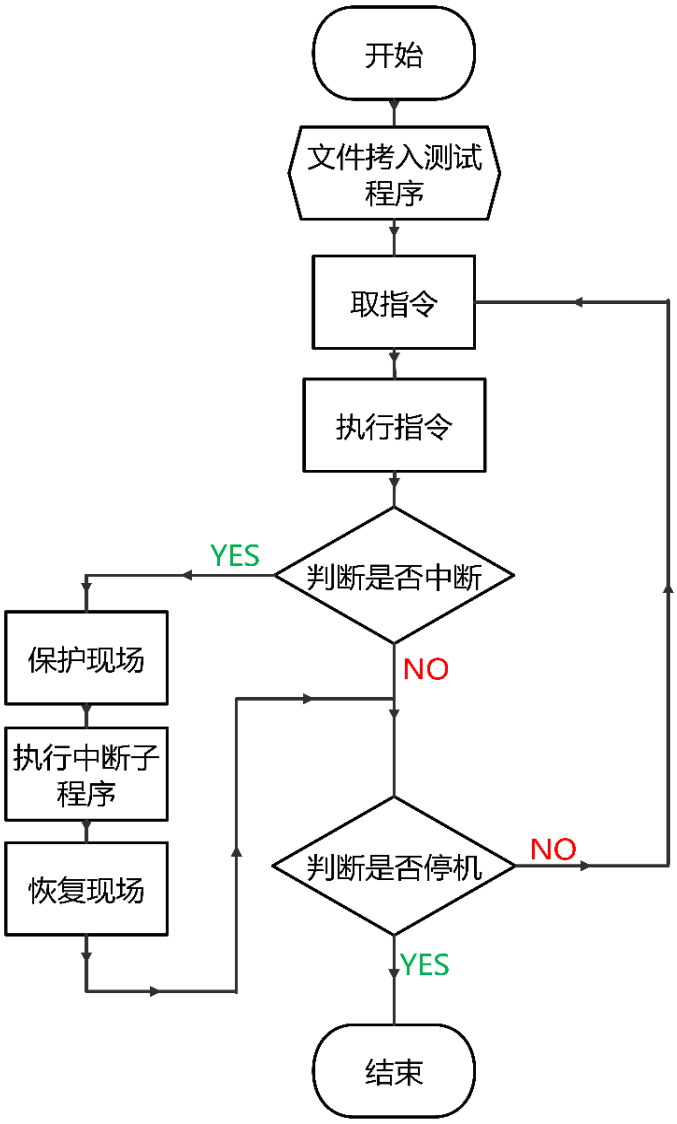
输入	IN XXXX	从输入区直接输入数据到地址 XXXX 中。
输出	OUT XXXX	直接输出 XXXX 中数据到显示区。

其他指令：

空指令	EMP	空操作
清零	CLR AX	将寄存器 AX 中数据清零

八、程序与指令流程图

- **程序运行流程：**



通过文件读入测试程序。当收到开机 RUN 指令时，程序开始运行。

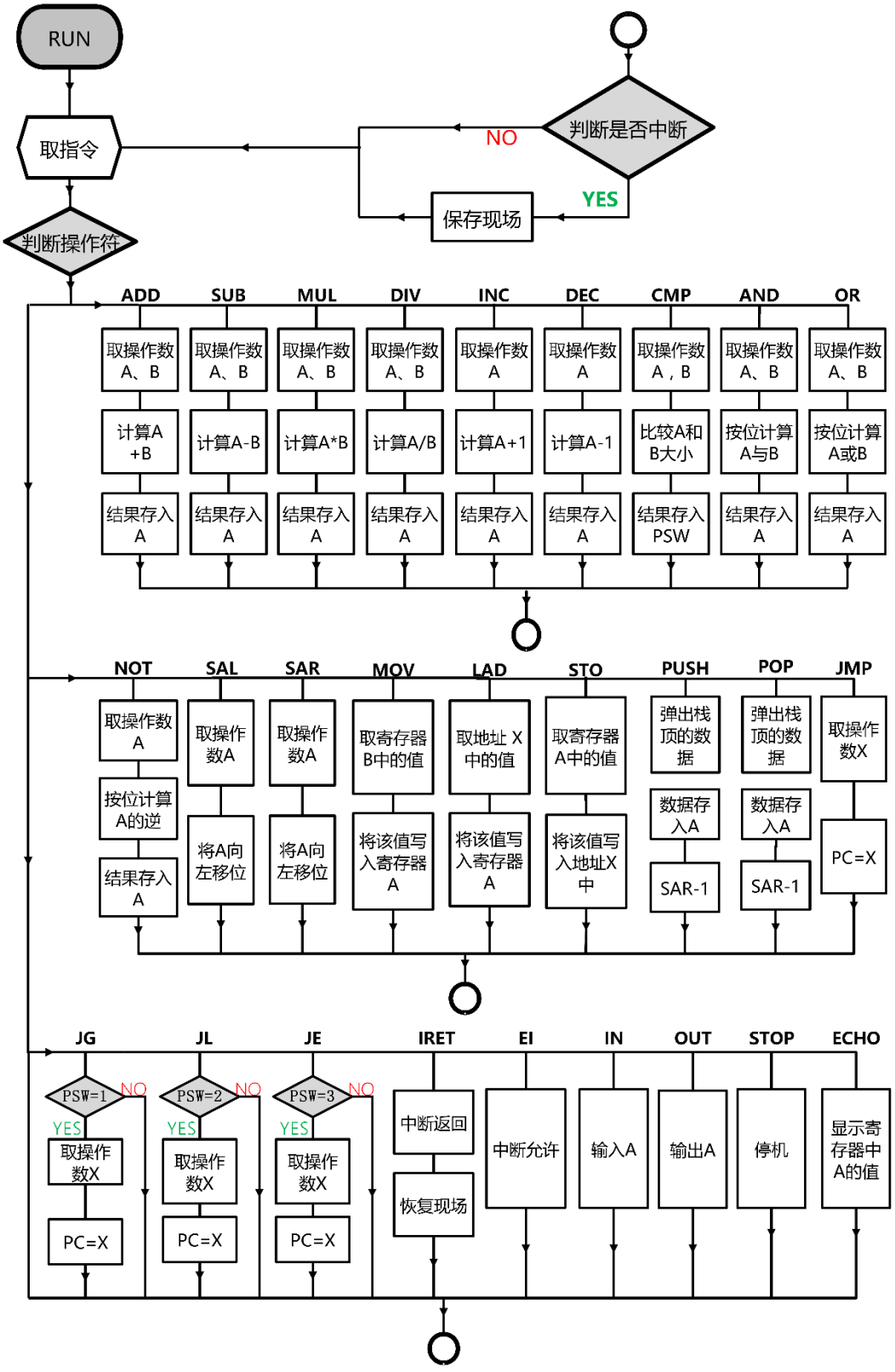
程序执行时在取指令、执行指令两个状态间转移。

每一条指令执行后，判断缓冲区是否存在中断信号。若存在中断信号，则保护现场，执行中断子程序，执行到中断返回指令时，恢复现场。继续取主程序下一条指令。

当收到停机 STOP 指令时，程序停止运行。

- **指令执行流程：**

共 28 条指令，指令流程图如下：



九、代码框架

将虚拟机封装成一个 VM 类，主函数中负责文件读取、图形化界面显示、中断接收。

```
struct VM
{

    char memory[0xFFFF][16];        //64 位存储器  0000-FFFF
    unsigned long long reg[8];        //8 个 64 位通用寄存器

    void init();      //初始化
    void load();        //装载测试程序
    void read();        //读一条指令
    bool operate();        //执行一条指令
    int check_error();      //检查是否出错或存在异常

}vm;

int main()
{
     vm.fp=fopen("测试程序.txt", "r");  //通过文件读取测试程序
     vm.read();
     fclose(vm.fp);

     while(!vm.terminated)  //未结束
     {
        do();        //   虚拟机运行
        show();        //图形化显示
        is_interrupt();      //判断是否中断
     }
    return 0;
}
```

九、变量声明

1. 指令集合

```
//指令操作名称
const char *CMD[35] = { "RUN", "STOP", "ECHO", "ADD", "SUB", "MUL", "DIV",
                        "MOL" ,"INC", "DEC", "CMP","AND","OR","NOT","SAL",
                        "SAR","MOV","LAD","STO","SET","SAVE","PUSH","POP",
                        "JMP","JG","JL","JE","EMP","CLR","EI","IRET","IN","OUT"
                        };
```

2. 汇编程序

```
//测试程序指令条数
int lines;
//程序（汇编代码）存储区
char code[105][20];
```

3. 中断子程序

```
//中断子程序指令条数
int inter_lines;
//中断子程序存储区
char inter_code[105][20];
```

4. 64 位存储器（地址 0x0000-0xFFFF）

```
//64位存储器 0x0000-0xFFFF
char memory[0xFFFF][16];
```

5. 16 个 64 位通用寄存器

```
//16个64位通用寄存器
unsigned long long int reg[16];
//用于中断时，保护现场的16个通用寄存器
unsigned long long int reg_tmp[16];
```

6. 专用寄存器

```
//程序计数器PC
int PC;
//地址寄存器
int AR;
//指令寄存器
char IR[20];
//中断地址寄存器
int IAR;
//栈地址寄存器
int SP;
//状态字寄存器 记录上一次CMP结果，1：大于 2：小于 3：等于
int PSW;//运行前初始化为0表示从未执行过CMP
//用于中断时，保护现场的状态字寄存器
int PSW_tmp;
```

7. 虚拟机当前状态

```
int state;
//0:未开机
//1:开机未加载程序
//2:程序加载完成
//3:程序运行中
//4:中断中
//5:停机
```

8. 错误与异常

```
int error;
//0:程序运行正常
//1:未加载程序时开始执行
//2:指令包含非法操作数
//3:寄存器超出规定数量
//4:条件跳转时，未比较，但尝试跳转
//5:在存储器中寻址时超出预设范围
//6:弹栈时，堆栈里无数据
//7:堆栈区溢出
//8:陷入循环
//9:中断中再次申请中断
//10:未中断时，中断恢复
//11:未中断时，手动输入指令
//12:除数为0
```

6. 其他变量

```
//指令编号
int cmd;
//指令中两个地址码
char X[10], Y[10];
//是否终止
int terminated = 0;
//是否中断
int is_interrupt;
//程序执行条数
int count = 0;
//主频
int M;
```

- cmd：指令操作码。
- X[10]、Y[10]：两个地址码。
- terminated：表示程序是否终止。为 1 时表示停机。
- is_interrupt：表示程序是否中断。为 1 时表示中断。
- count：标志程序总共执行的条数，防止程序陷入死循环。count 最大为 1000000。
- M：主频，执行一条指令需要的时间，单位为毫秒，控制连续执行时快慢。

十、函数调用

内部函数：

```
void Run(); //运行程序
void init();    //初始化
void load();    //加载测试程序
int read();    //读取指令
int operate();  //执行指令
void show_error();  //检错和异常显示
void interrupt();   //中断请求
void recover();   //中断恢复
void show_reg();    //寄存器图形化显示
void show_state();  //运行状态图形化显示
void show_stack();  //堆栈图形化显示
void sleep(int msec);   //延时函数
void setValue(int x);   //修改主频
void protect(); //中断保护现场

unsigned long long int HEX_DEC(char *s, int n);    //16进制转10进制
void DEC_HEX(unsigned long long n, char res[]); //10进制转16进制
void DEC_BIN(unsigned long long n, int res[]);  //10进制转2进制
unsigned long long int BIN_DEC(int res[]);  //2进制转10进制
```

外部函数：(用于交互)

```
void on_pushButton_START_clicked(); //开机

void on_pushButton_STOP_clicked();  //停机

void on_pushButton_LOAD_clicked();  //装载测试程序

void on_pushButton_CONTINUOUS_clicked();    //连续执行

void on_pushButton_STEP_clicked();  //单拍执行

void on_pushButton_INTERRUPT_clicked(); //中断申请

void on_pushButton_RECVER_clicked();    //中断恢复

void on_pushButton_clear_clicked(); //清空输入区

void on_pushButton_work_clicked();  //执行输入区指令

void on_pushButton_quit_clicked();  //退出程序

void on_pushButton_help_clicked();  //帮助
```

函数调用过程：

内部函数：

1. 虚拟机通过 Run()进入，调用 init()、show_reg()、show_stack()、show_state()、show_error()，完成初始化、显示工作。

2. 通过 load()、show()、show_error()，加载测试程序，并显示。

3. 单步执行时：执行 read()、operate()、show_reg()、show_stack()、show_state()、show_error()一遍。

4. 连续执行时：while()循环中，不断执行 read()、operate()、show_reg()、show_stack()、show_state()、show_error()。直到中断信号或者停机，跳出。

5. 当接收到中断信号时，执行 interrupt()，保护现场，进行中断。

6. 当收到恢复中断信号时，执行 recover()，恢复现场，中断返回。

6. 将存储器和寄存器的数据实时通过 show_reg()、show_stack()、show_state()图形化显示。

外部函数：

1. 点击按钮或拖动滑块时，产生一个外部信号。

2. 外部信号连接到槽函数。

3. 槽函数中再调用内部函数，完成外部交互操作。

十一、加载程序

```
void MainWindow::load()
{
    //初始化总行数，程序计数器为0
    lines = PC = 0;
    //括号里的参数分别是：指定父类、标题、默认打开后显示的目录、右下角的文件过滤器。
    file_name = QFileDialog::getOpenFileName(NULL,"选择测试文件",".","*.txt");
    QFile file(file_name);
    if(!file.open(QIODevice::ReadOnly | QIODevice::Text))
    {
        //qDebug()<<"Can't open the file!"<<endl;
        ui->textEdit->append("Can't open the file!");
    }
    lines=0;
    while(!file.atEnd())
    {
        QString line = file.readLine();
        QByteArray ba = line.toLatin1();
        char* ch;
        ch = ba.data();
        int i;
        for(i=0;ch[i]!='\r'&&ch[i]!='\n';i++)
        {
            code[lines][i]=ch[i];
            memory[lines][i]=ch[i];
        }
        ui->textEdit_2->append(QString::number(lines,16)+"    "+QString(QLatin1String(code[lines])));
        lines++;
    }
    state=2;
    ui->statusBar->showMessage("Test program is loaded", 1000);
}
```

测试程序使用汇编指令格式。一条指令占一行，编号从 0 开始。

通过 file_name = QFileDialog::getOpenFileName(NULL,"选择测试文件",".","*.txt")在文件夹中选择 .txt 文件。

通过<QFile>库中的 file.*open*(QIODevice::ReadOnly | QIODevice::Text)打开 .txt 类型的测试程序，加载至 code[105][20]和 memory 中。

十二、读取指令

指令格式：

指令操作码	地址码 1	地址码 2

指令操作码：3 位或 4 位。

地址码：寄存器（iX）或内存地址（0xXXXX）。

指令操作码、地址码 1、地址码 2 之间，通过‘空格’连接。（如‘ADD AX BX’）

```
void MainWindow::read()
{
    strcpy(IR, code[PC]);
    ui->textEdit->append(QString(QLatin1String(IR)));
    int len = strlen(IR);
    int blank = 0;
    int j = 0, k = len;

    for (int i = 0; i < len; i++)
    {
        if (IR[i] == ' ' && blank == 0)
        {
            j = i;
            blank++;
        }
        else if (IR[i] == ' ' && blank == 1)
        {
            k = i;
            blank++;
        }
    }
    for (int i = 0; i < 26; i++)
    {
        if (strstr(IR, CMD[i]) != NULL)
        {
            cmd = i;
            break;
        }
    }
    for (int i = 0; i < k - j - 1; i++)
    {
        X[i] = IR[j + i + 1];
        //printf("******\n");
    }
    for (int i = 0; i < len - k -  1; i++)
    {
        Y[i] = IR[k + i + 1];
    }
    //ui->textEdit->append(QString::number(cmd));
    //ui->textEdit->append(QString(QLatin1String(X)));
    //ui->textEdit->append(QString(QLatin1String(Y)));
}
```

通过 read()取一条汇编指令，通过统一的指令格式，将指令拆分为操作码、地址码 1、地址码 2。将操作码与指令集中指令通过 strstr()进行比较，获得指令编号。

cmd：指令编号。

X[4]：地址码 1。

Y[4]：地址码 2。

十三、执行指令

```
403  //执行指令
404  int MainWindow::operate()
405  {
406      //PC++;
407      int len_a,len_b;
408      int ta, tb; //两个操作数
409      unsigned long long int a,b,c;
410      state=3;
411      int aa[64], bb[64], cc[64];
412      char dd[16];
413      ui->statusBar->showMessage("Instruction Executing", 500);
414      switch (cmd) {...}
880      show_reg();
881      show_state();
882      QApplication::processEvents();
883      //PC++;
884      count++;
885      if(count>1000000)
886      {
887          error=8;
888          return -1;
889      }
890      return 0;
891  }
```

通过 operate()函数，将地址码 1 X[4]和地址码 2 Y[4]，转化成寄存器编号或者内存地址，存入 ta、tb。

a、b、c、aa[64]、bb[64]、cc[64]、dd[16]为执行指令时的中间变量。

通过 switch-case，判断指令编号 cmd，然后执行相应指令，并且输出到日志中。

每条指令执行后，更新显示区各个寄存器、堆栈的值。

十四、中断

中断申请：

```
979  //中断申请
980  void MainWindow::interrupt()
981  {
982      if(is_interrupt==0)
983      {
984          is_interrupt=1;
985          state=4;
986          memset(reg_tmp, 0, sizeof(reg_tmp));
987          IAR=PC;
988          PSW_tmp=PSW;
989
990          for(int i=0;i<16;i++)
991          {
992              reg_tmp[i]=reg[i];
993          }
994          PC=0xE000;
995      }
996      else
997      {
998          error=9;
999      }
1000 }
```

收到中断申请信号时，判断当前是否已经中断：

若已经中断，则输出异常，并且忽略中断信号。

若程序执行中，将中断信号置为 1。开始保护现场：用中断地址寄存器 IAR 记录当前程序计数器 PC 的值。PSW_tmp 记录当前状态字寄存器 PSW 的值。用

reg_tmp[16]记录当前 16 个通用寄存器的值。将 PC 置为内存中中断子程序区的首地址 0xE000。

中断恢复：

```
1002  //中断恢复
1003  void MainWindow::recover()
1004  {
1005      if(is_interrupt==1)
1006      {
1007          PC=IAR;
1008          IAR=0;
1009          for(int i=0;i<16;i++)
1010          {
1011              reg[i]=reg_tmp[i];
1012          }
1013          PSW=PSW_tmp;
1014          ui->textEdit_IN->clear();
1015          ui->textEdit_3->clear();
1016          is_interrupt=0;
1017          state=3;
1018          for(int i=0xE000;i<0xFFFF;i++)
1019          {
1020              memset(memory[i],'\0',sizeof(memory[i]));
1021          }
1022      }
1023      else
1024      {
1025          error=10;
1026      }
1027  }
```

收到中断恢复信号时，判断当前是否处于中断中：

若未中断，则输出异常，并且忽略中断恢复信号。

若处于中断中，将中断信号置为 0。开始恢复现场：程序计数器 PC 用中断地址寄存器 IAR 当前的值代替。状态字寄存器 PSW 用 PSW_tmp 记录的值代替。16 个通用寄存器的值恢复成 reg_tmp[16]记录的中断前的值。将内存中中断子程序区的数据清零。

十五、手动输入

读入中断子程序：

```
//执行输入区指令
void MainWindow::on_pushButton_work_clicked()
{
    if(is_interrupt==1&&!terminated)
    {
        inter_lines=0;
        QString text = ui->textEdit_IN->toPlainText();
        QStringList number_list = text.split("\n");
        memset(inter_code, '\0', sizeof(inter_code));
        for (inter_lines = 0; inter_lines < number_list.size(); inter_lines++)
        {
            char*  ch;
            QByteArray ba = number_list.at(inter_lines).toLatin1(); // must
            ch=ba.data();
            for(int i=0;i<20;i++)
            {
                inter_code[inter_lines][i]=ch[i];
                memory[inter_lines+PC][i]=ch[i];

            }

            ui->textEdit_3->append(QString::number(PC+inter_lines,16)+"    "+QString(QLatin1String(inter_code[inter_lines])));

        }
```

通过函数 toPlainText()将手动输入区的内容读入。通过函数 text.split("\n")将读入的内容以“回车”为标志，分割成一条条指令。

将指令读入 inter_code 和 memory 中的中断子程序区中。

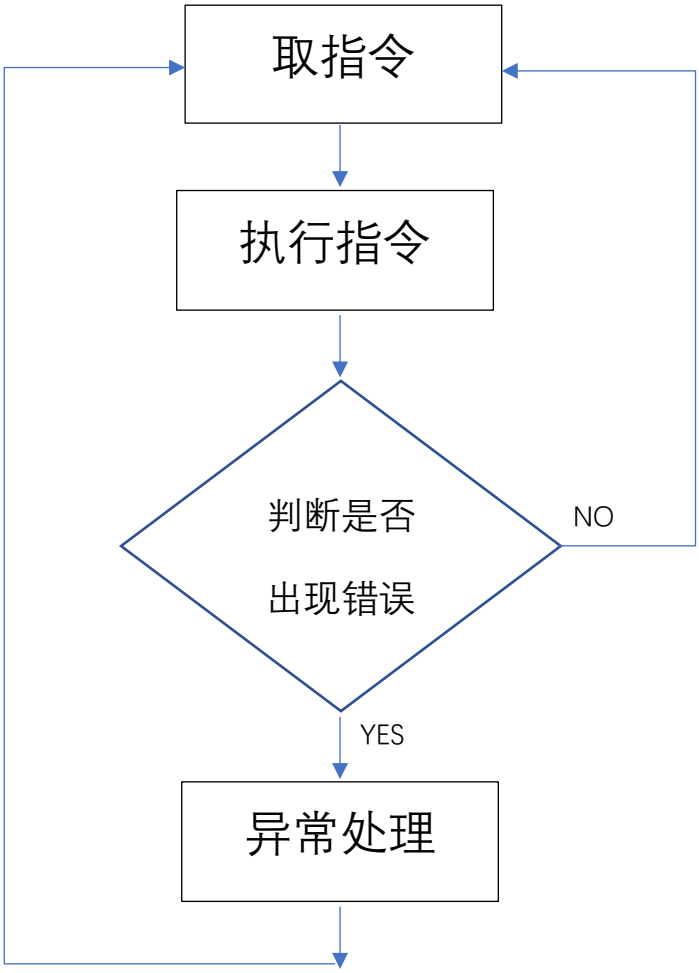
执行中断子程序：（大只致同连续执行的操作）

```
        while(inter_lines--&&memory[PC][0]!='\0')
        {

            if(state==0||state==1)
            {
                error=1;
                break;
            }
            else
            {
                int e1,e2;
                e1=read();
                if(e1!=-1)
                {
                    e2=operate();
                }
                if(e1==-1||e2==-1)
                {
                    ui->textEdit->append("ERROR\n");
                }
                show_state();
                show_reg();
                show_stack();
            }
            PC++;
            show_error();
            //sleep(M);
        }
    }
    else
    {
        error=11;
        show_state();
        show_reg();
        show_stack();
        show_error();
    }
}
```

十六、检错与异常

增加如下的流程，每次执行指令之后检查是否出现异常，若出现异常，则显示异常并且处理异常跳过该指令：



异常包括：

1: 未加载程序时开始执行；

2: 指令包含非法操作数；

3: 寄存器超出规定数量；

4: 条件跳转时，未比较，但尝试跳转；

5: 在存储器中寻址时超出预设范围；

6: 弹栈时，堆栈里无数据；

7: 堆栈区溢出；

8: 陷入循环；

9: 中断中再次申请中断；

10: 未中断时，中断恢复；

11: 未中断时，手动输入指令；

12: 除数为 0；

十七、参考文献

《深入理解计算机系统 第三版》——（美）布赖恩特（Bryant,R.E.）

《8051 虚拟机的设计与实现》——卢彩林、丁刚毅

《x86 汇编指令详解》——lzy's notes