基于高级语言的硬件虚拟机
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、实验目的

通过高级语言以软件模拟仿真方式，实现冯诺依曼计算机系统，模拟计算机系统整机工作原理，直观展现硬件运行过程，将所学的软件基础知识与硬件基础是知识进行综合，锻炼系统综合能力。

设计模型机指令系统，测试程序。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、任务描述

通过高级语言以软件模拟仿真方式，实现冯诺依曼计算机系统

1、设计自己指令集系统（参照计算机组成原理模型机指令集，可以扩展或选择其他指令集）；

2、设计模拟系统数据通路，建立软件－硬件映射关系；

3、给出软件模拟方案设计，包括总体设计、五大模块设计、存储器系统分配方案；

4、选择开发语言，给出具体实现，软件结构要清晰，代码部分要有注释；

5、给出测试方案和用例；

6、总结，完成实验报告。

三、实验环境

虚拟机使用 C++完成底层逻辑和软件框架的实现，使用 QT 完成图形化界面的实现。C++的优势在于更加适合底层的开发。QT 的优势在于 qt 优良的跨平台

特性、面向对象、丰富的 API、支持 2D/3D 图形渲染等等。能很好地完成图形化的任务

具体操作上使用 Visual Studio 2017 和 Qt Creator 5.9.2。

四、虚拟机作用及设计依据

1. 清晰透明的交互界面，实时显示虚拟机各模块的数据，使虚拟机透明化，供使用者进行调试。

2. 简单的交互操作及清晰的运算过程，使初学者也能深入了解虚拟机运行流程。

3. 统计执行测试程序过程中，执行指令条数，以此比较不同算法之间的差异性和优势劣势。

4. 统计测试程序中各指令使用频率。

五、设计目标

1. 虚拟机包含 64 位运算器，64 位存储器，内存为 0x0000——0xFFFF。

2. 虚拟机基于冯诺依曼体系结构。

3. 通过软件实现物理机器中的运算器、存储器、控制器、数据通路、I/O 设备、桥、总线。

4. 通过软件仿照计算机硬件模拟虚拟机的构造，使虚拟机程序运行流程及指令执行过程基本，与物理机器中的微程序控制器作用、逻辑基本相同。

5. 设计指令集，使虚拟机能完成测试程序的执行。

附加功能:

6. 虚拟机包含中断操作，能保护现场、执行中断子程序、恢复现场。

7. 虚拟机支持 I/O，能从外部设备接受鼠标和键盘的信号，进行交互。

8. 虚拟机支持手动输入指令，方便进行逐条指令的测试。

9. 虚拟机包含用户模式和内核模式两种状态。

9. 指令集包含特权指令、中断指令、I/O 指令。

10. 虚拟机提供用户模式和内核模式两种模式的选择与切换。

11. 清晰的图形化界面，充分掌握虚拟机运行状态。

六、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。[1]

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。[3]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

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。[5]

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、特权指令，能完成计算机基本操作。、

其中特权指令，只能在内核模式下执行。用户模式没有特权指令的权限。

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 16 个通用寄存器。用 AX、BX、CX、DX……PX 表示，字母皆为大写。
- **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。

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	申请中断。
中断返回	IRET	中断返回。

I/O 指令：

输入	IN XXXX	从输入区指令。
输出	ECHO AX	直接输出寄存器 AX 中数据到显示区。

特权指令：

取状态字	LPSW AX	读取状态字寄存器 PSW 中的数据，存入寄存器 AX 中。
存程序计数器	SPC AX	将寄存器 AX 中数据，存入程序计数器 PC 中。
读程序计数器	LPC AX	读取程序计数器 PC 中的数据，存入寄存器 AX 中。
读中断地址寄存器	LIAR AX	读取中断地址寄存器 IAR 中的数据，存入寄存器 AX 中。

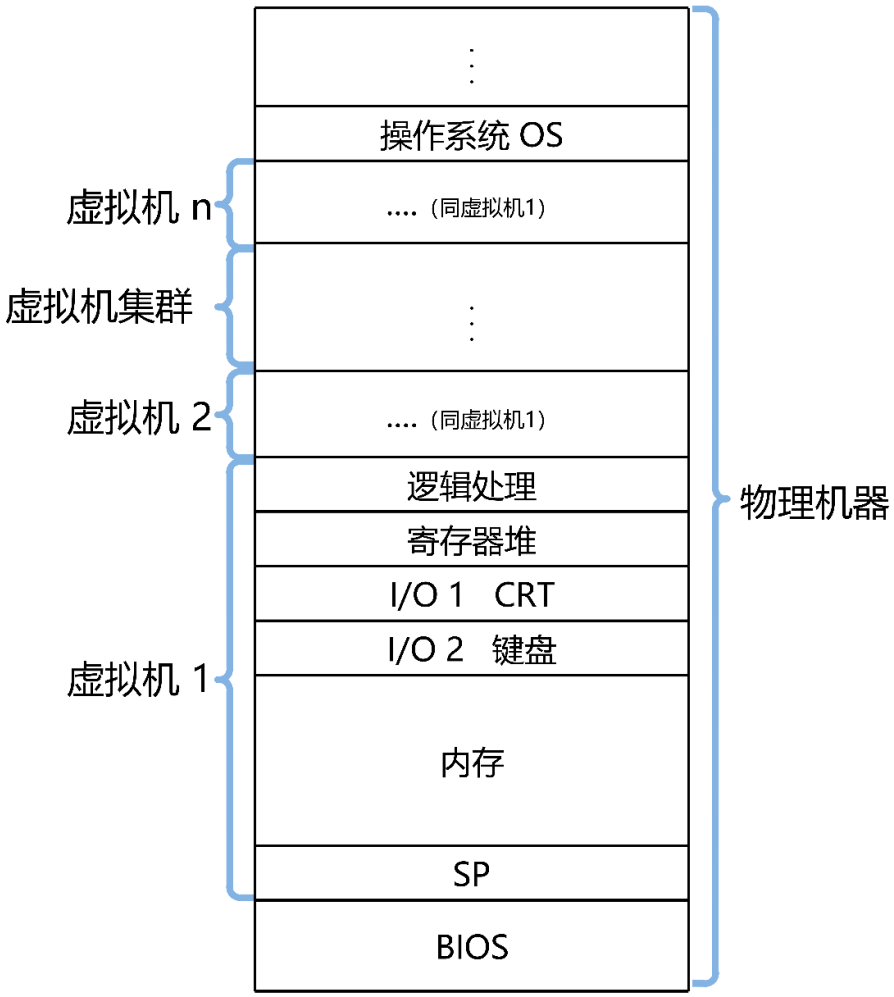
存中断地址寄存器	SIAR AX	将寄存器 AX 中数据，存入中断地址寄存器 IAR 中。

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、虚拟机底层设计

1、整体构架设计

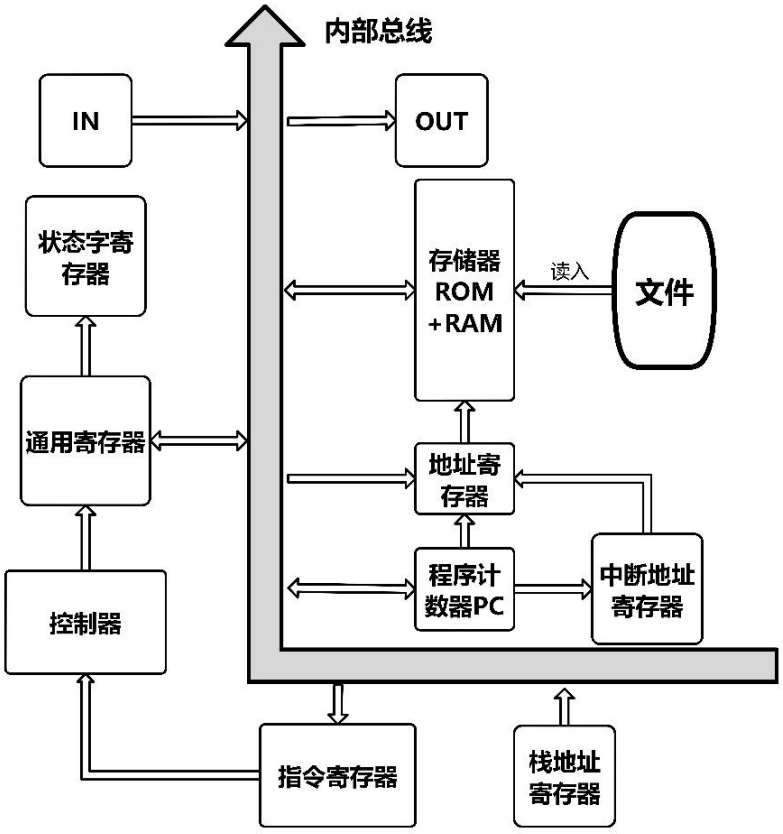


虚拟机指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。虚拟机的软件从电脑资源中分出一部分的 CPU、内存、硬盘存储....等等，然后虚拟机软件把这些资源整合，组成了一台电脑。

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。

2、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	I/O 操作

3、软件框架[2]

寄存器实现：

通用寄存器：共 16 个通用寄存器。AX、BX、CX、DX……PX。每个通用寄存器其数据为 64 位，用一个无符号 long long int 型整数表示。

程序计数器、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：通过 char 型数组表示。

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。

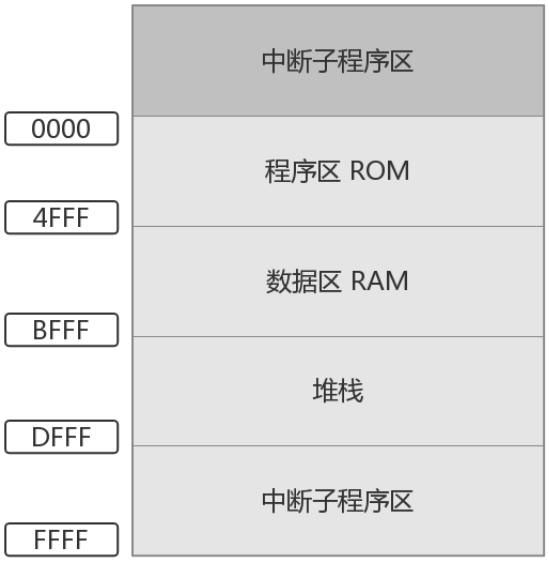
堆栈区：地址为 C000-DFFF。存放堆栈中的数据。

中断子程序区：地址为 E000-FFFF。存放通过手动输入的中断子程序。

输入缓冲区实现：

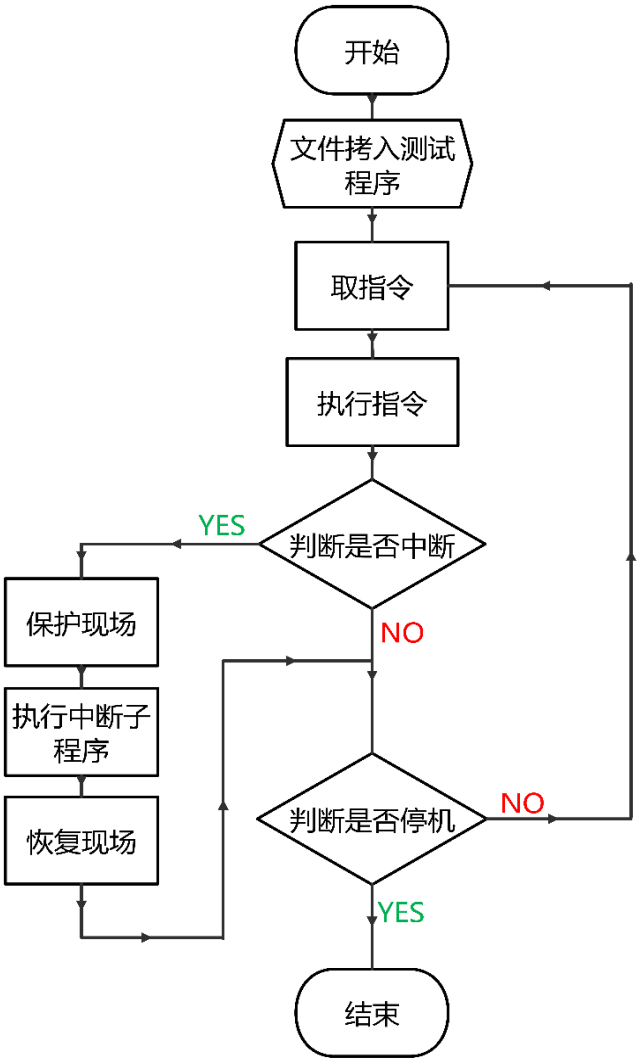
通过数组 int buffer[100]实现队列的形式。buffer_first 记录缓冲区数据首地址，buffer_last 缓冲区数据尾地址。

寄存器	
PC	程序计数器
IR	指令寄存器
IAR	中断地址寄存器
DR	栈地址寄存器
PSW	状态字寄存器



4、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。

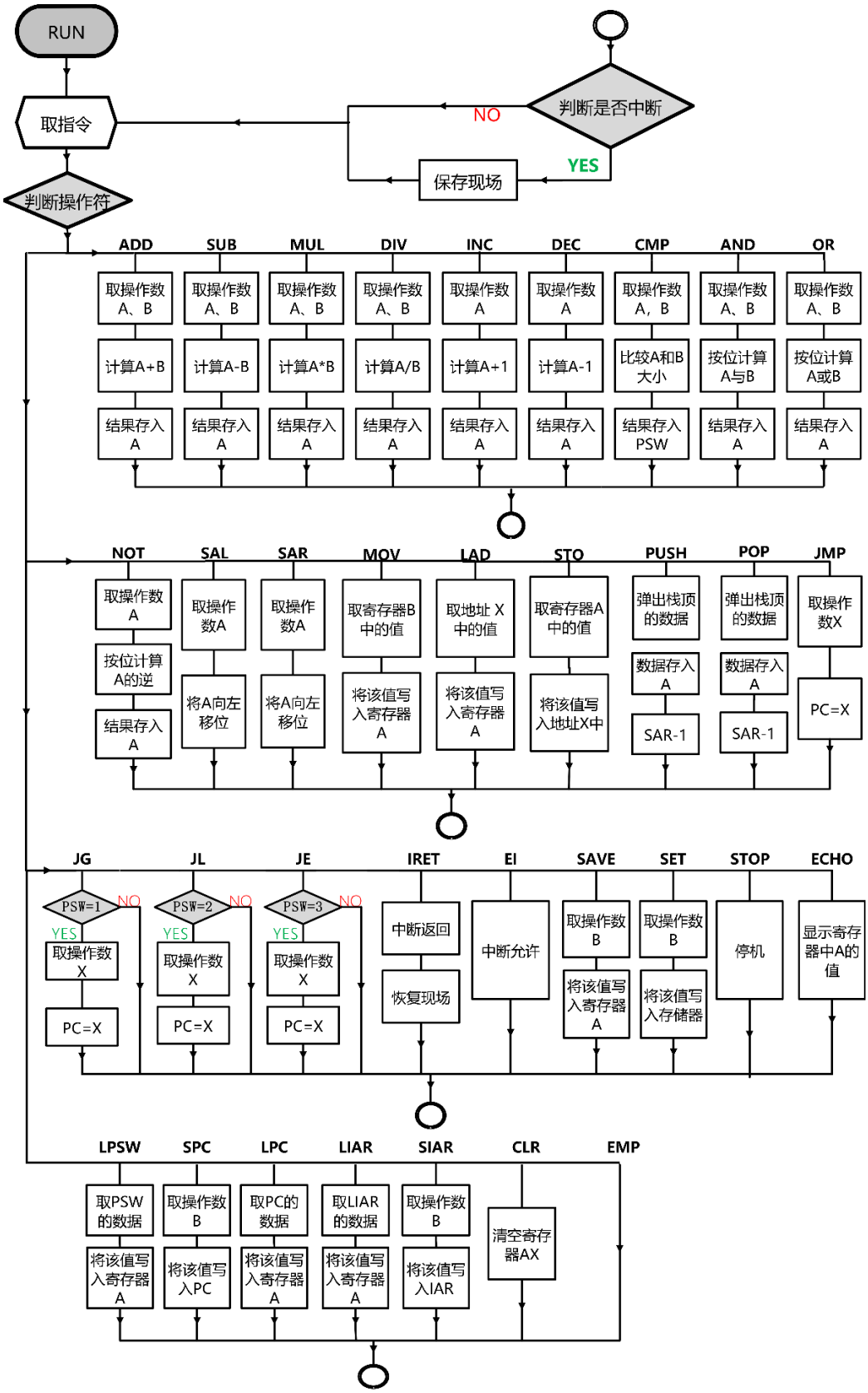
程序执行时在取指令、执行指令两个状态间转移。

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[4]：**

共 37 条指令，指令流程图如下：



九、虚拟机交互界面设计

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。
- 中断请求：发送中断信号，请求中断。
- 中断恢复：发送中断恢复信号，从中断返回主程序。
- 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。
- 输入：手动一个整数到输入缓冲区。
- 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；
    - 执行指令数：记录虚拟机从开机开始，执行的指令条数。
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. 输入区：输入一个整数到输入缓冲区。

7. 输出区：将寄存器的值输出到屏幕上。

8. 手动输入区：中断时，通过命令行手动输入中断子程序。

9. 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。

十、高级语言实现原理

1、整体框架

将虚拟机封装成一个 VM 类，操作函数中负责文件读取、图形化界面显示、中断接收。伪代码如下：

```
struct VM
{

    char memory[0xFFFF][16];        //64 位存储器  0000-FFFF
    unsigned long long reg[16];      //16 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 operate()
{
      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;
}
```

2、变量声明

1. 指令集合

```
//指令操作名称
const char *CMD[37] = { "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",
                        "LPSW","SPC","LPC","LIAR","SIAR"
                        };
```

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. 输入缓冲区

```
//输入缓冲区
int buffer[100];
//输入缓冲区数据首地址
int buffer_first;
//输入缓冲区数据尾地址
int buffer_last;
```

8. 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
```

9. 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
```

10. 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。

3、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：(用于交互操作)

```
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
void on_radioButton_user_clicked(); //切换用户模式
void on_radioButton_kernel_clicked();   //切换内核模式
void on_pushButton_clicked();   //输入
```

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。

4、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。

5、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。

6、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。

7、中断

虚拟机包含两种中断模式[3]：

1. 通过单击“中断申请”，向虚拟机发送外部中断信号。

2. 通过指令“EI”，从虚拟机内部产生中断信号。

中断处理流程如右图所示

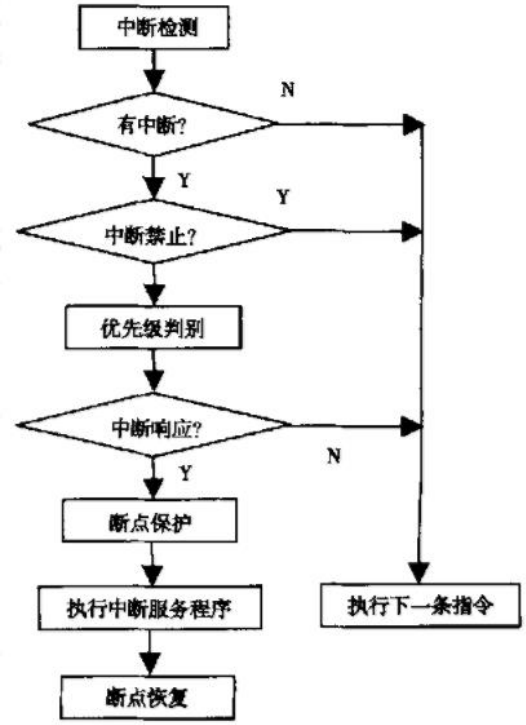


图 4　中断处理流程图

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。

8、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。

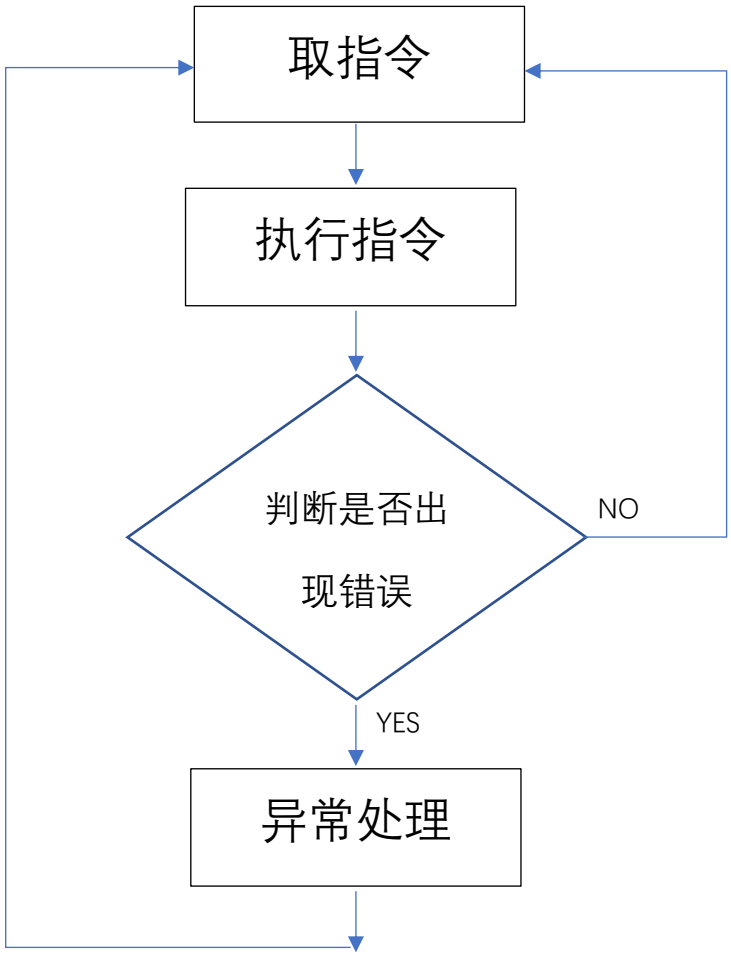
执行中断子程序：（大只致同连续执行的操作）

```
        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();
    }
}
```

9、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；

十一、测试程序与测试步骤

测试程序说明：

1. test.txt：测试程序包含指令集中所有指令。用于测试并验证指令正确性及虚拟机运行正确性。

2. error.txt：测试程序包含的指令均为错误指令。用于测试并验证虚拟机异常与检错的准确性。

3. Fibonacci_Iteration.txt：测试程序为通过迭代的算法，求斐波那契数列前 50 项，并输出到输出区。用于验证跳转、循环等指令及虚拟机运行流程和逻辑的正确性。

4. Fibonacci_Stack.txt：测试程序为通过堆栈的操作，求斐波那契数列前 50 项，将斐波那契数列前 50 项依次压入堆栈区，并输出到输出区。用于验证入栈、出栈等指令及虚拟机维护堆栈区的操作逻辑的正确性。

5. Prime-Continuous.txt：测试程序为通过循环求 50 以内的质数，并输出到输出区。用于验证无条件跳转、条件跳转、比较、输出等指令及虚拟机执行程序中的双层循环操作逻辑的正确性。

6. Prime-Interrupt.txt：测试程序为通过中断子程序的方式求 50 以内的质数，并输出到输出区。为了测试特权指令，本测试程序加入了写中断子程序地址的指令，故请务必在开机之后，选择内核模式。用于验证主程序中断、跳转至中断子程序、中断恢复，函数间通过堆栈传递参数及特权指令逻辑的正确性。**由于选择内核模式，加入了特权指令。请在充分了解该测试程序的基础上，进行测试。否则如错误点击按钮、重复申请中断等操作，会引起不必要的错误与程序异常。**

7. 手动输入区程序：本虚拟机提供手动输入指令并执行的功能。单击中断申请后，主程序进入中断，可在手动输入区输入多条指令，指令格式参考“附件：指令集“。单击执行，将手动输入的程序加载进中断子程序，并执行。

具体测试程序如下：

内存地址	test	error	Fibonacci_Iteration	Fibonacci-Stack	Prime-Continuous	Prime-Interrupt
0	RUN	RUN	RUN	RUN	RUN	RUN
1	SAVE AX AAAAAA	SAVE AX AAAA	SAVE AX 1	SAVE AX 1	SAVE AX 2	SAVE AX 2
2	SAVE BX 1112	ABC AX	SAVE BX 1	SAVE BX 1	SAVE EX 0	SAVE BX 32
3	SAVE CX BBBB	POP BX	SAVE CX 0	SAVE CX 2	SAVE FX 32	SAVE DX 0
4	SAVE DX 3333	SFGHSFG	SAVE DX 2	IN DX	SAVE HX 2	ECHO AX
5	SAVE EX 7777	INC ZX	IN EX	PUSH AX	ECHO AX	SIAR 0012
6	ECHO BX	DIV AX CX	ECHO BX	ECHO AX	INC AX	EI
7	ADD AX BX	JE 0002	ECHO AX	PUSH BX	SAVE BX 0	POP CX
8	SUB CX DX	STOP	MOV CX BX	ECHO BX	SAVE CX 2	CMP CX DX
9	MUL AX BX		MOV BX AX	POP AX	MOV GX AX	JE 000B
A	DIV AX BX		ADD AX CX	POP BX	MOL AX CX	ECHO AX
B	MOL EX DX		ECHO AX	PUSH BX	MOV DX AX	CMP AX BX
C	INC AX		INC DX	PUSH AX	MOV AX GX	JG 000F
D	DEC DX		CMP DX EX	ADD AX BX	CMP DX EX	INC AX
E	CMP AX BX		JE 0010	ECHO AX	JE 001A	JMP 0005
F	AND AX BX		JMP 0008	PUSH AX	INC CX	STOP
10	OR AX BX		STOP	INC CX	DIV GX HX	
11	NOT AX			CMP CX DX	CMP GX CX	中断程序：
12	SAL AX			JE 0014	JL 0014	SAVE EX 2
13	SAR BX			JMP 0009	JMP 0009	SAVE HX 2
14	MOV AX BX			STOP	CMP BX EX	SAVE FX 1
15	STO AAAA AX				JE 001C	MOV IX AX
16	LAD FX AAAA				CMP AX FX	DIV IX HX
17	PUSH AX				JE 001E	MOV GX AX
18	POP BX				INC AX	MOL GX EX
19	STOP				JMP 0007	CMP GX DX
1A					SAVE BX 1	JE 001F
1B					JMP 0014	CMP EX IX
1C					ECHO AX	JG 0021
1D					JMP 0016	INC EX
1E					STOP	JMP 0017
1F						SAVE FX 0
					PUSH FX	
					IRET	

操作步骤：

0. 开始进行测试之前，建议单击**『帮助』**，查看虚拟机操作说明书及测试教程。

1. 单击**『开机』**，启动虚拟机。

2. 单击**『装载』**，选择 test.txt、error.txt、Fibonacci_Iteration.txt、Fibonacci_Stack.txt、Prime-Continuous.txt、Prime-Interrupt.txt 六个测试样例之一，装载进虚拟机。

3. 若加载的测试程序为 test.txt、error.txt、Fibonacci_Iteration.txt、

Fibonacci_Stack.txt、Prime-Continuous.txt 之一，则模式选择选中**『用户模式』**。

若加载的程序为 Prime-Interrupt.txt，则模式选择选中**『内核模式』**。

4. 若加载的测试程序为 Fibonacci_Iteration.txt、Fibonacci_Stack.txt 之一，该两个测试程序存在输入内容，在输入区**输入 n**（n 是在 10 和 100 之间的整数），单击**『输入』**。其余测试程序可忽略此步。

5. i：单击**『单拍』**，执行一条指令。

ii：单击**『连续』**，连续执行测试程序。左右调整滑块，可调整虚拟机主频，改变每一条指令执行速度。

6. test.txt 、 error.txt 、 Fibonacci_Iteration.txt 、 Fibonacci_Stack.txt 、 Prime-Continuous.txt 测试过程中，可单击**『中断申请』**，从外部向虚拟机发送中断信号，中断测试程序。

Prime-Interrupt.txt 测试过程中，由于已经存在中断子程序，并且在内核模式下测试，**请勿轻易点击『中断申请』**，以防造成不必要的错误。

7. 虚拟机处于中断中时，可在手动输入区输入多条指令，指令格式参考“附件：指令集“。

i：单击**『执行』**，将手动输入区的程序加载进中断子程序，并执行。

ii：单击**『清空』**，将手动输入区的内容清空。

8. 虚拟机处于中断中时，可单击**『中断恢复』**，恢复到测试主程序。

9. 测试程序执行完后，可单击**『停机』**，关闭虚拟机，清空所有数据。

10. 单击**『退出』**，退出虚拟机界面，关闭程序。

测试结果：

test.txt：

内存地址	指令	执行结果
0	RUN	START
1	SAVE AX AAAAAA	REGISTER AX : aaaaaa
2	SAVE BX 1112	REGISTER BX : 1112
3	SAVE CX BBBB	REGISTER CX : bbbb
4	SAVE DX 3333	REGISTER DX : 3333
5	SAVE EX 7777	REGISTER EX : 7777
6	ECHO BX	BX : 1112
7	ADD AX BX	aaaaaa + 1112 = aabbbc
8	SUB CX DX	bbbb - 3333 = 8888
9	MUL AX BX	aabbbc * 1112 = b6278af38
A	DIV AX BX	b6278af38 / 1112 = aabbbc
B	MOL EX DX	7777 MOL 3333 = 1111
C	INC AX	aabbbc + 1 = aabbbd
D	DEC DX	3333 - 1 = 3332
E	CMP AX BX	aabbbd > 1112
F	AND AX BX	101010101011101110111101 AND 1000100010010 = 1000100010000
10	OR AX BX	1000100010000 OR 1000100010010 = 1000100010010
11	NOT AX	NOT 1000100010010 = 11111111111111111111 11111111111111111111111111101110111011101101
12	SAL AX	11111111111111111111111111111111 11111111111111111111101110111011101101 SAL = 11111111111111111111111111111111 11111111111111111011101110111010
13	SAR BX	1000100010010 SAR = 100010001001
14	MOV AX BX	REGISTER AX : 889
15	STO AAAA AX	REGISTER->MEMORY AAAA : 889
16	LAD FX AAAA	MEMORY->REGISTER FX : 889
17	PUSH AX	STACK c000 : 889
18	POP BX	REGISTER BX : 889
19	STOP	SHUT DOWN

error.txt：

内存地址	指令	执行结果
0	RUN	START
1	SAVE AX AAAA	REGISTER AX : aaaaa
2	ABC AX	操作数不合法
3	POP BX	弹栈时，堆栈里无数据
4	SFGHSFG	不合法指令
5	INC ZX	寄存器超出规定数量
6	DIV AX CX	除数为 0
7	JE 0002	条件跳转时，未比较，但尝试跳转
8	STOP	SHUT DOWN

Fibonacci_Iteration.txt：

输出区输出："1、1、2、3、5、8、13、21、34 ……" 斐波那契前 n 项。

Fibonacci_Stack.txt：

输出区输出："1、1、2、3、5、8、13、21、34 ……" 斐波那契前 n 项。

堆栈区 0xC000-0xC031 为"1、1、2、3、5、8、13、21、34 ……" 斐波那契前 n 项。

Prime-Continuous.txt、Prime-Interrupt.txt：

输出区输出："2、3、5、7、11 …… 43、47" 50 以内的质数。

十二、实验总结

虚拟机特点：

1. 有清晰的软件框架，模拟系统数据通路，建立了软件－硬件映射关系。

2. 指令集基本包含开机指令、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。额外包括中断指令、I/O 指令、特权指令。

3. 虚拟机包含总体设计、五大模块设计、存储器系统分配方案。

4. 虚拟包含中断，手动输入程序的功能。

5. 虚拟机包含异常与检错的功能。

6. 虚拟机包含清晰的界面设计，使虚拟机内部运行流程一目了然，方便测试虚拟机运行正确性。

7. 虚拟机可通过点击按钮和拖动滑块，与外界进行交互控制。

8. 设计了六个虚拟机测试程序，分别完成基础指令功能的测试、检错与异常的测试、跳转逻辑的测试、堆栈维护的测试、中断子程序的测试。囊括了虚拟机运行状态的各个分支。

收获与体会：

通过这次的使用高级语言以软件模拟仿真方式，实现冯诺依曼计算机系统的实验，无论是从知识上还是从能力上我都获益良多，平时我们能见到的都是计算机的外部结构，在计算机组成原理的学习中，逐步对计算机的内部结构有了一些了解，但始终都停留在理论阶段。

上学期，在 Proteus 的实验中，通过硬件实现了计算机内部结构。而在这次课程设计中，让我们使用高级语言，仿真计算机组成。让我体会到软件仿真相比硬件的不同和优势。例如寄存器、存储器只需要通过数组进行模拟。而硬件系统中的数据通路在软件中，变成了逻辑上的数据转移。例如中断、输入输出等复杂

操作，在硬件上需要许多部件实现，而软件仿真中，可以直接通过一个变量记录是否中断，用一个数组模拟输入缓冲区。软件仿真相比硬件实现更加强调计算机内部逻辑，而减少了硬件操作的复杂程度。

这次高级语言模拟仿真虚拟机的实验，让我对运算器，存储器，控制器以及输入输出各个系统的内部结构都有了更深的了解。同时对计算机中断、I/O 等机制有了更深的理解，并且对计算机组成原理也有了更深层次的理解。

展望：

1. 优化程序，提高虚拟机运行速率。

2. 给虚拟机增加联网功能，实现多台虚拟机的交互与数据传输，实现虚拟机集群。

3. 实现基于当前虚拟机的操作系统及 BIOS。

十三、参考文献

[1]《深入理解计算机系统 第三版》——（美）布赖恩特（Bryant,R.E.）

[2]《虚拟机的设计与实现》——（美） Bill Blunden

[3]《8051 虚拟机的设计与实现》——卢彩林、丁刚毅

[4]《PDSS 虚拟机的设计》——欧阳星明、刘迎午、朱尚文

[5]《x86 汇编指令详解》——lzy's notes