# 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知识进行综合，锻炼系统综合能力。

# 二、实验环境

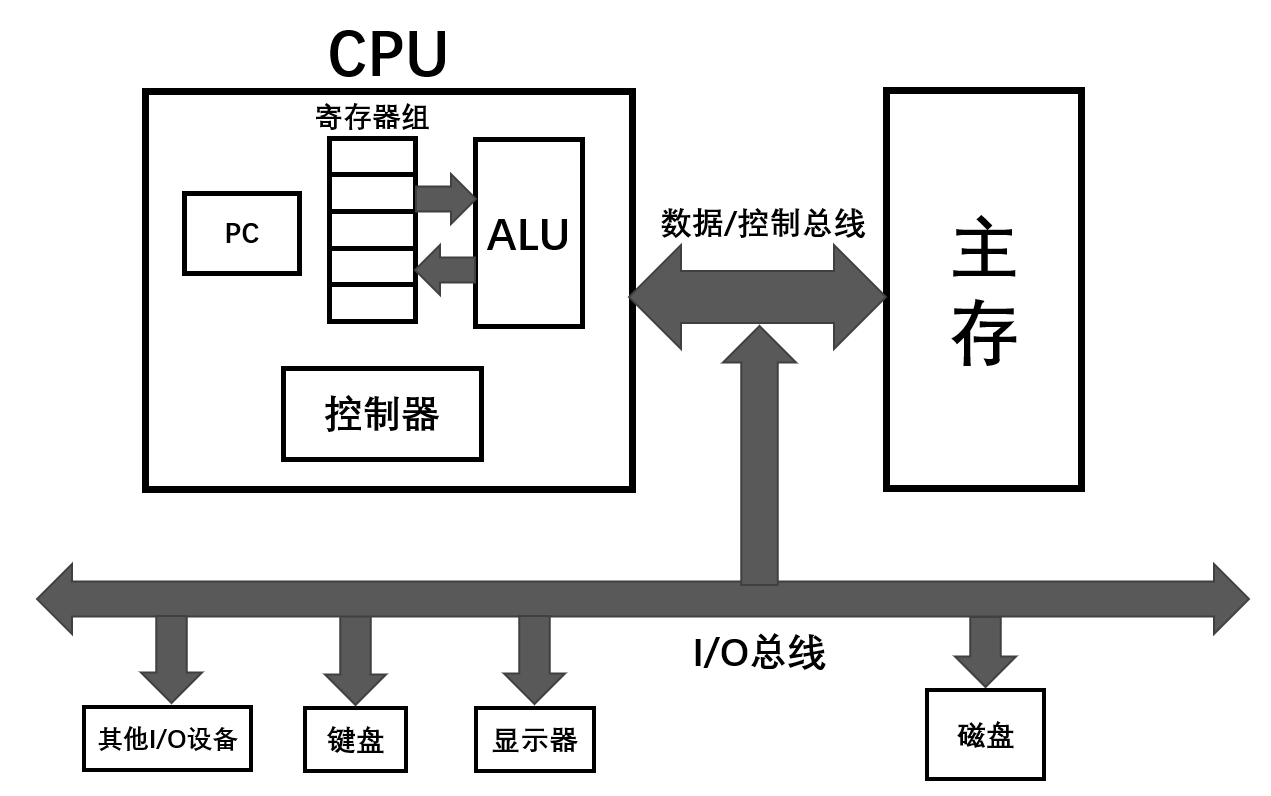
开发语言：C++/Qt

开发环境：windows 10 Qt 5.10.0

开发平台：Qt Creator 4.5.0

# 实验内容

3.1 模拟硬件结构设计



虚拟机硬件总体设计为32位字长，内存大小64KB，按字节寻址，特殊功能寄存器存储无符号整数，而通用寄存器存储补码形式的32位二进制数。

对于虚拟机指令集，采用变长指令设计，各模块组件及模拟数据通路如上图所示，其中键盘外设通过硬件中断方式与内存中的输入缓冲区进行沟通，显示器带有4KB的显示存储空间，能够以ASCII码可打印字符的形式显示存储区的内容。

寄存器组设计如下表：

|  |  |
| --- | --- |
| 寄存器名称 | 功能介绍 |
| 程序计数器PC | 存储内存中接下来要执行指令的地址 |
| 指令寄存器IR | 存储指令二进制代码 |
| 地址寄存器AR | 存储内存地址，与内存沟通 |
| 栈址寄存器SP | 存储内存中堆栈区的栈顶地址 |
| 中断地址寄存器IAR | 记录发生中断时PC指向的地址 |
| 输入寄存器INR | 位于虚拟键盘外设中，存储当前键盘输入对应的ASCII码 |
| 输入队首寄存器IH | 指示内存中输入缓冲队列队首地址 |
| 输入队尾寄存器IT | 指示内存中输入缓冲队列队尾地址 |
| 通用寄存器R0~R7 | 32位通用寄存器 |
| 进位标志寄存器CF | 将操作数视为无符号数，当进行加减操作最高位产生进位/借位时置1，否则置0 |
| 溢出标志寄存器OF | 反映有符号数加减乘运算是否溢出，如果运算结果超过了32位有符号数的表示范围则置1，否则置0 |
| 比较标志寄存器CMF | 与CMP比较指令配合，当Ra < Rb时置1，否则置0 |
| 中断请求寄存器IRR | 当收到中断请求时置1，对中断响应时置0 |
| 中断屏蔽寄存器IM | 当正在响应中断时置1，屏蔽中断请求，其余置0 |

3.2 中断机制与I/O模块设计

1. **虚拟机中断机制**

本次虚拟机设计的中断机制参照了Pentium的中断向量表+中断服务子程序的实现方式[[1]](#endnote-1)[1]­­­，其中中断向量表用于存放对应中断服务子程序的入口地址，而中断服务子程序包括计算机响应中断时所要执行的指令。

中断包括硬件中断和软件中断两部分，其中硬件中断信号是由与CPU相连接的外设所产生的，在硬件中断时，CPU会根据中断源设备的不同，识别中断编号，查找中断向量表，并转入相应的中断服务程序，硬件中断用于对外设请求的响应。

软件中断是执行中断指令产生的，在软件中断时，中断号由指令直接给出，无需中断控制器识别，系统直接获取指令给出的中断号，转入相应的中断程序，软件中断由编写的程序来进行控制，通常用于系统调用，异常处理等。

1. **I/O模块设计**

模拟了一个能够输入ASCII可打印字符的键盘外设来实现虚拟机的输入操作，键盘包含自身的数据寄存器INR，当按下输入键时，键盘首先将对应按键的数字编码（ASCII码）存入自身数据寄存器中，然后向CPU发出中断请求，CPU执行中断服务子程序时，会将键盘数据寄存器中的数据打入内存中的输入缓冲队列，当程序需要从用户处获取输入时，可以通过IN指令读取输入缓冲队列中的内容。

对于输出模块，模拟一个能够显示带扩展的ASCII码字符的显示器。显示器带有显存，保存当前的显示队列，并能够持续以ASCII字符形式显示显存中的数据。为执行输出操作，程序可以通过OUT指令，输出相应的字符至显示队列，或者清空当前显存，以此控制相应的内容输出。

3.3指令集设计

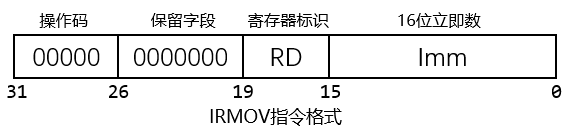
为了避免冗余，提高虚拟机效率，采用变长指令设计，共有32条指令，分为单字节指令，半字长(16位)指令和单字长(32位)指令三种长度的指令。指令根据功能可分为数据传送、运算、程序控制和输入输出四种类别。

### 数据传送类指令

**1 IRMOV 立即数->寄存器传送指令**

IRMOV RD Imm //将16位立即数Imm的值存入寄存器RD中

16位立即数Imm由指令直接给出，从指令寄存器IR获得并写入相应的寄存器R中。框图中X指示ALU的左端口操作数，Y指示ALU右端口操作数，下同。



**2 IRMOVS 立即数->寄存器移位传送指令**

IRMOVS RD Imm //将立即数Imm的值左移16位存入寄存器RD的前16位

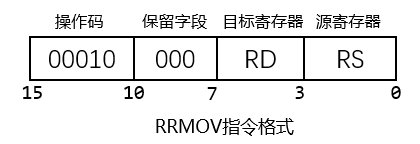
Imm为16位立即数，与IRMOV指令结合使用，实现对32位寄存器的赋值操作。



**3 RRMOV 寄存器->寄存器传送指令**

RRMOV RD RS //寄存器RS->寄存器RD

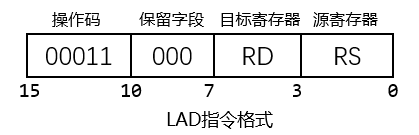
RRMOV与IRMOV本质上都是寄存器中数据的传递，IRMOV的立即数是从指令寄存器IR中获得，而RRMOV是由指令自身指定源操作数所在的寄存器。



**4 LAD 内存->寄存器 取数指令**

LAD RD RS //内存M[RS]->寄存器RD

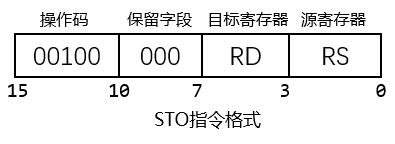
采用寄存器间接寻址的方法，根据寄存器RS的值，指示相应的内存地址，从内存单元中读取一个字长的数据写入寄存器。



**5 STO 寄存器->内存 存数指令**

STO RD RS //寄存器RS->内存M[RD]

采用寄存器间接寻址的方法，根据寄存器RD的值，指示相应的内存地址，从寄存器中读取一个字长的数据写入内存单元。



**6 CLA 清零指令**

CLA R //全0->R

CLA指定一个寄存器R将其清零。



**7 SET 置1指令**

SET R //全1->R

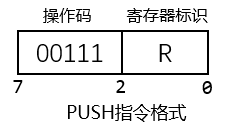
SET指定一个寄存器R置为全1。



**8 PUSH 压栈指令**

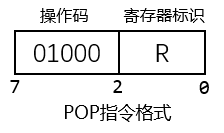
PUSH R //寄存器R->内存M[SP+1 ~ SP+4], SP=SP+4

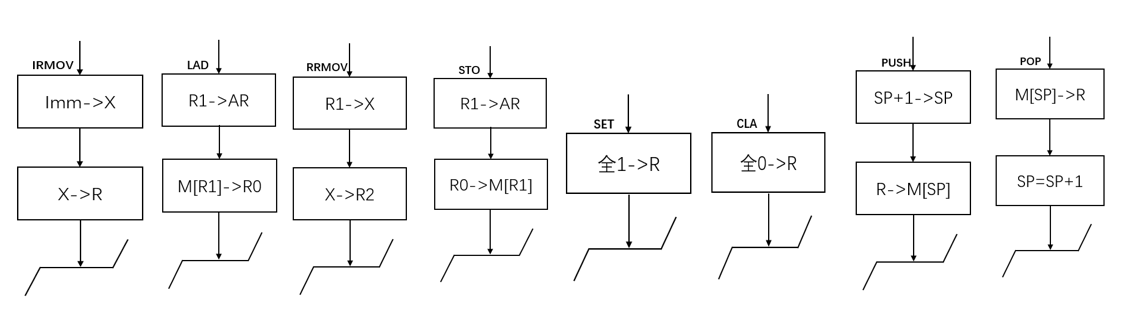
SP为栈址寄存器，记录当前内存中栈区的栈顶地址，通过PUSH指令压栈可以保存寄存器的值。



**9 POP 出栈指令**

POP R //内存M[SP-3 ~ SP]->寄存器R，SP=SP-4





### 运算类指令

**10 ADD 加法指令**

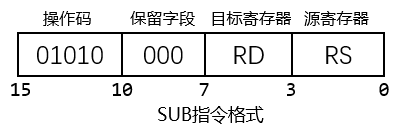
ADD RD RS //RD + RS->RD

RS指示源操作数寄存器, RD指示目标寄存器，两者相加的结果存至RD中。



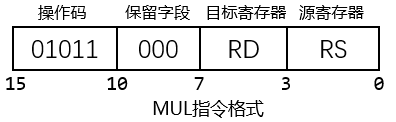
**11 SUB 减法指令**

SUB RD RS //RD - RS->RD



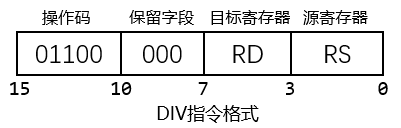
**12 MUL 乘法指令**

MUL RD RS //RD \* RS->RD



**13 DIV 除法指令**

DIV RD RS //RD / RS->RD



**14 NEG 取负指令**

NEG R //-R->R

对相应寄存器内的数据改变符号，由正变负或由负变正，数据采用补码表示。



**15 INC 自增指令**

INC R //R+1->R

使相应寄存器内的操作数加一。



**16 DEC 自减指令**

DEC R //R-1->R

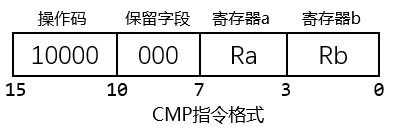
使相应寄存器内的操作数减一。



**17 CMP 比较指令**

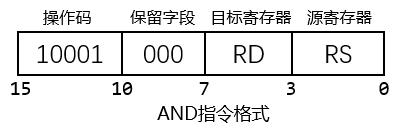
CMP Ra Rb //比较Ra和Rb，将比较结果存入条件码寄存器CMF中。

CMF为存储比较结果的条件码寄存器，当Ra < Rb时置1，否则置0。



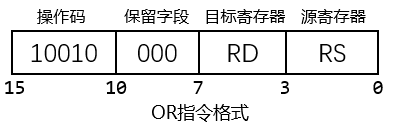
**18 AND 逻辑与指令**

AND RD RS //RD & RS->RD



**19 OR 逻辑或指令**

OR RD RS //RD | RS->RD



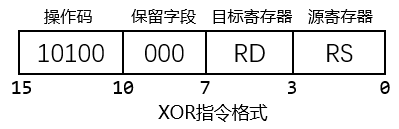
**20 NOT 逻辑非指令**

NOT R //~R->R



**21 XOR 逻辑异或指令**

XOR RD RS //RD ^ RS->RD



**22 SHL 左移指令**

SHL RD RS //(RD << RS)->RD

逻辑/算术左移，丢弃最高位，最右侧补0。

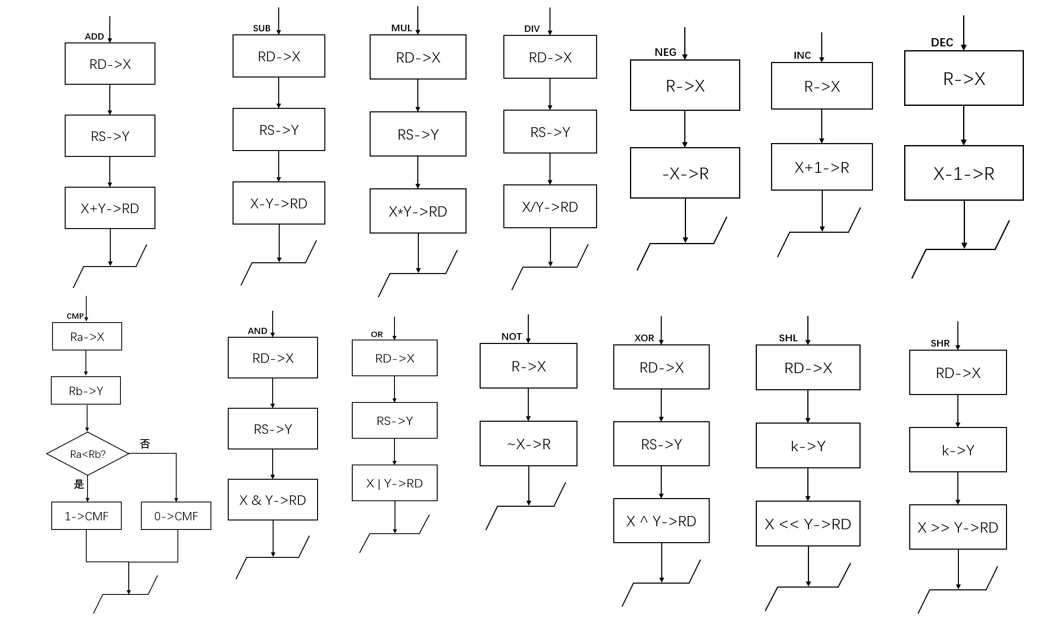


**23 SHR 右移指令**

SHR RD RS //(RD << RS)->RD

算术右移，保持符号位不变，最左侧正数补0，负数补1。





### 程序控制类指令

**24 JMP 无条件跳转指令**

JMP Imm //PC+Imm->PC

JMP指令采用相对跳转的方式，其中Imm是17位原码形式二进制立即数，能够使PC无条件增加或减去指定值，下面的跳转指令相同。



**25 JC C条件跳转指令**

JC Imm //IF CF=1，PC+Imm->PC

JC指令检查条件码寄存器中的进位标志CF，当CF为1时，使PC增加或减去相应的值。



**26 JO Z条件跳转指令**

JO Imm //IF OF=1，PC+Imm->PC

JC指令检查条件码寄存器中的溢出标志OF，当最近一次算术运算操作发生溢出，即OF为1时，使PC增加或减去相应的值。



**27 JCM CM条件跳转指令**

JCM Imm //IF CMF=1，PC+Imm->PC

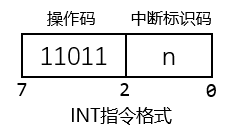
JCM指令检查条件码寄存器中的比较标志CMF确定最近一次比较操作的结果，当CMF为1时，使PC增加或减去相应的值。



**28 INT 软件中断指令**

INT n //保存现场，转入中断标识码n对应的中断服务子程序

INT指令执行过程中，首先通过PUSH操作保存现场，将当前的PC、寄存器组及条件码寄存器的值存入栈区，然后通过查找中断向量表，转入给出的中断标识码n对应的中断服务子程序的入口地址，从而执行相应的指令。



**29 RETI 中断返回指令**

RETI //还原现场，恢复中断前状态

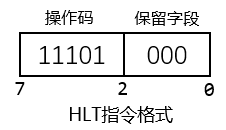
RETI指令通过POP操作还原现场，将栈区中存入的通用寄存器和条件码寄存器的值出栈，恢复相应寄存器的值，并返回中断时PC的地址。

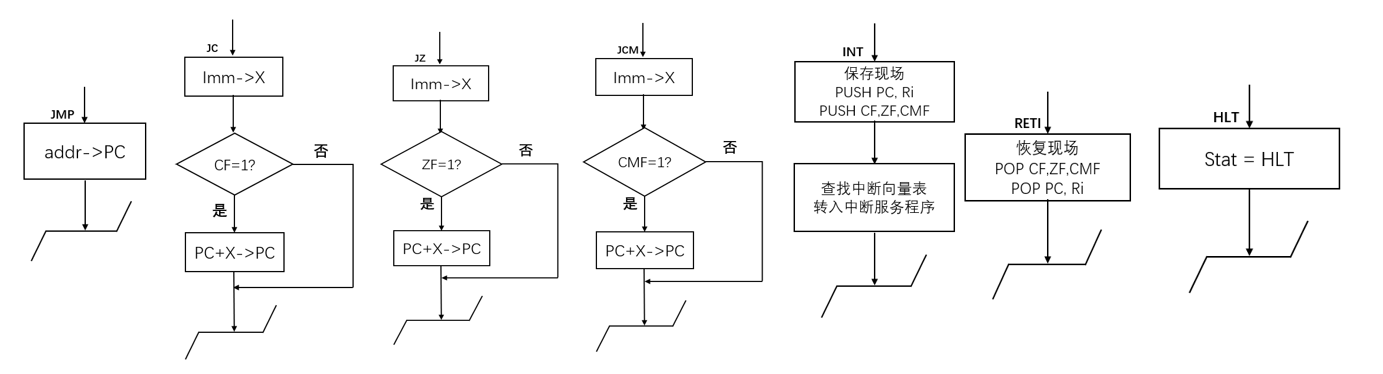


**30 HLT 停机指令**

HLT //停机，中止系统运行

HLT将系统状态标识码Stat变为停机标识HLT，中止系统运转，系统不再读取并执行下一条指令及响应中断。





### 输入输出指令

**31 IN 输入指令**

IN R //从输入缓冲队列队首取出一个字节放入寄存器R

虚拟机内存中有存储输入缓冲队列的区段，并采用两个寄存器IH和IT分别用来标识输入缓冲队列的队首和队尾地址，IN指令会从输入缓冲队首取出一个字节存入寄存器R，并使IH增1，当输入缓冲区为空（即IH = IT）时，系统会发生阻塞等待，直到发生输入事件并转入执行对应的中断服务程序。

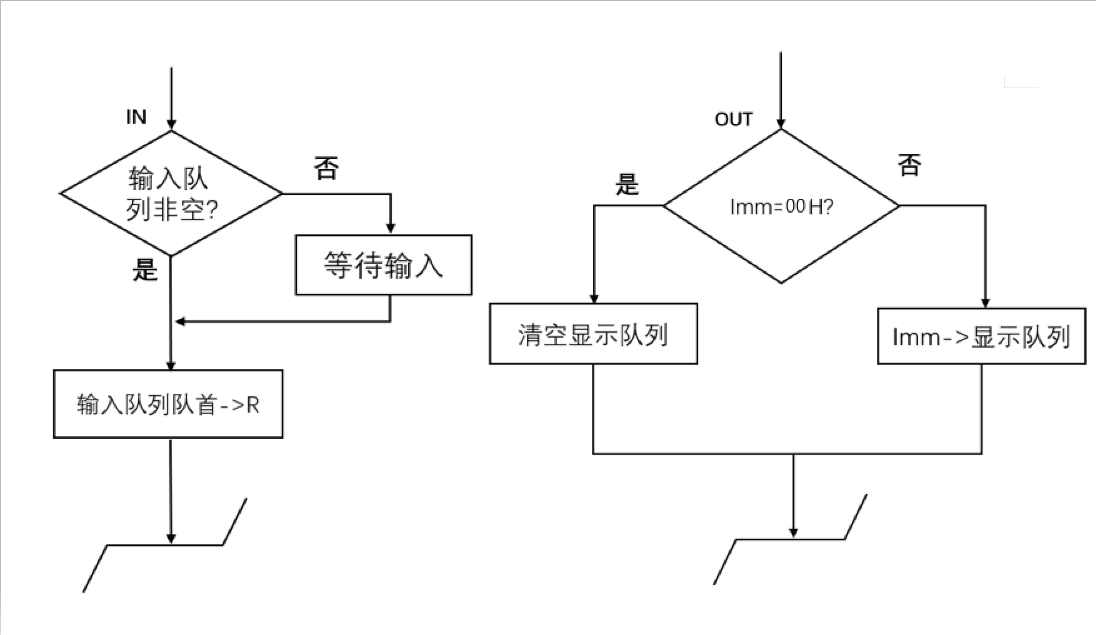


**32 OUT 输入指令**

OUT R //输出寄存器R的后八位至显示器显示队列中

取寄存器R的后八位送至显存，显示器以带扩展的ASCII码形式读取显示队列并呈现至虚拟机显示区，并规定当Imm为00H时，执行特殊操作，将显示队列清空。





3.4虚拟机软件实现

### 总体设计

#### 宏定义及数据结构

**宏定义：**

typedef unsigned char byte;

//define the state of VMulatorA

#define AOK 1 //no exceptions

#define HLT 2 //instruction HLT executed

#define ADR 3 //invalid address

#define INS 4 //invalid instruction

#define DIV 5 //divided by zero

#define STE 6 //pop when stack empty

#define STF 7 //push when stack full

#define WAI 8 //waiting input

#define INT 9 //hardware interrupt

//define the execution mode of VMulator

#define Stepping 0

#define Continuous 1

#define MemorySize 0x10000 //64KB Memory

#define InsStartPos 0x5000

//start position for instruction/data segment in memory

#define InputQueueAddr 0x2000

//start position for storing the input data

#define StackAddr 0x3000 //start position for the stack pointer

**数据结构：**

byte Memory[MemorySize];

std::vector<byte> DisplayMemory;

byte Stat;

//denoting the execution state of VMulator(halted, waiting input etc.)

bool RunningMode; //Stepping or Continuous

unsigned int clockFreq; //This clockFreq variable simply denotes that how many instructions the cpu can executive in a second;

//Registers

bool CF, CMF, OF, IRR, IM; //state bit registers

unsigned int PC, IR, AR, SP, IAR, INR, IH, IT;

//special purpose registers

int R[8]; //general purpose registers

QTimer \*timer = new QTimer; //clock timer using to control the frequency of execution in continuous running mode

QString InputCacheQueue;

std::map<int,int> addrInsMap; //a map from address to the instructin number to specify the position of each instrciton in the instrction list

QStringList intRoutineLists[3]; //the instruction lists of interrupt routine to show in interrupter state

**函数声明：**

bool **load**(QFile &insFile, int &pos);

//load instructions from file to VMulator memory

void **initInterrupt**();

//initialize interrupt vector table and interrupt service routines

void **init**();

void **display**();

void **insFetch**(); //fetch instructions from memory to IR

void **wordTo4Byte**(int word, int addr);

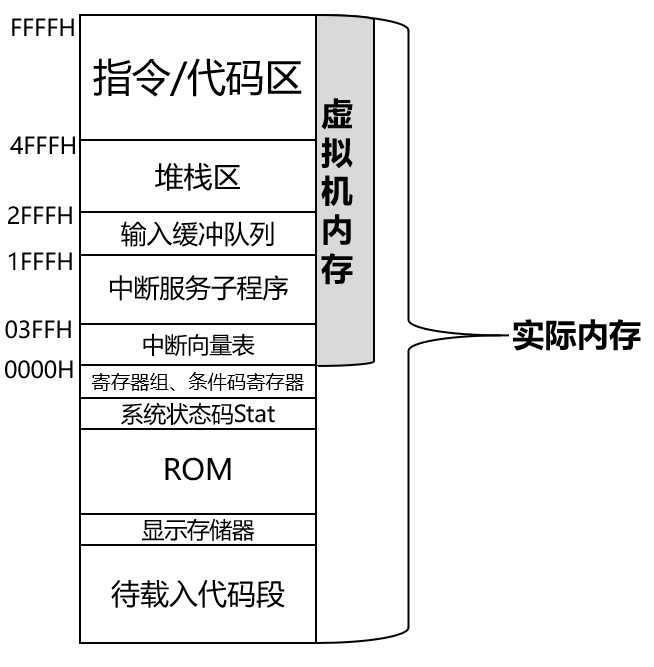
int **byte4ToWord**(int addr);

void **execInput**();

void **execForward**(); //defined as Qt slot

void **execBackward**();

#### 软件内存区段划分



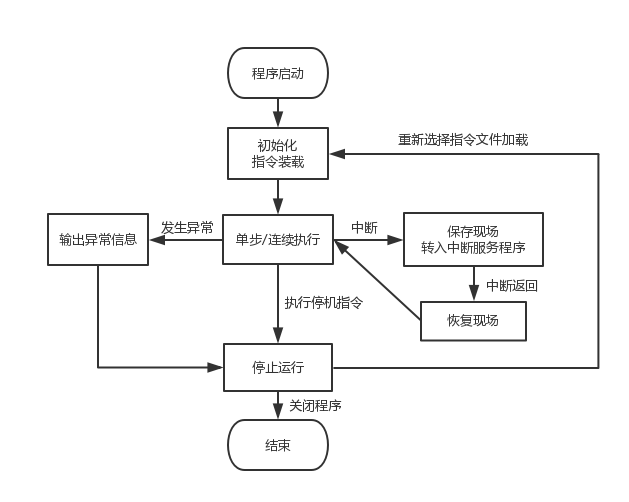
#### 图形界面设计与交互操作



当前指令区利用Qt的QTableWidget组件分条显示指令，对于每一条指令，用浅蓝底色表示即将执行的下一条指令，黄色表示输入指令正等待输入，红色表示发生错误，灰色表示停机，如下图所示：



#### 软件执行流程框图



### 虚拟机初始化

虚拟机在执行指令前先要进行初始化操作，包括初始化虚拟机状态，写入中断向量表及中断服务子程序，清空其余内存区段，给寄存器组赋初值，重置控制程序连续执行的QTimer定时器，以及一些界面显示操作等，其核心代码及注释如下：

void VMulator::**init**()

{ //load instructions, initialize VMulator and display corresponing variables

static bool firstInit = true;

if(firstInit) //interrupt vector table only need to be initialized once

{

initInterrupt();

firstInit = false;

}

QString filename = QFileDialog::getOpenFileName(this,"打开文件","./","VMulator instructions(\*.txt \*.vmi)");

QFile insFile(filename);

if(!insFile.*open*(QFile::ReadOnly | QFile::Text))

{

QMessageBox::warning(this,"警告","文件打开失败");

return;

}

memset(Memory + InputQueueAddr,0,MemorySize - InputQueueAddr); //clear memory except interrupt service routine

ui->InstructionList->clear(); //clear current instruction list

int pos = InsStartPos;

if(!load(insFile, pos)) //load instructions

return;

for(int i = 0;i < MemorySize;i++)

ramMonitor->changeMem(i,Memory[i]); //show memory data in ram monitor

Stat = AOK; //reset VMulator state

DisplayMemory.clear();

InputCacheQueue.clear();

ui->textBrowser\_InputQueue->clear();

ui->InputText->clear();

RunningMode = Stepping; //reset running mode

clockFreq = ui->lcdNumber\_Freq->value(); //10 instructions per second at initial

delete timer;

timer = new QTimer(this); //reset clock timer

timer->setInterval(1000/clockFreq);

connect(timer, SIGNAL(timeout()), this, SLOT(execForward()));

//initialize registers

CF = CMF = OF = IRR = IM = 0;

PC = InsStartPos;

SP = StackAddr;

IAR = AR = IR = INR = 0;

IH = IT = InputQueueAddr;

for(int i = 0;i < 8;i++)

R[i] = 0;

display();

}

### 指令汇编与装载

虚拟机的指令汇编其实是一个字符串匹配问题，采用Qt提供的正则式支持类QRegExp[[2]](#endnote-2)[2]实现。各条指令的正则式模式定义如下：

QRegExp IRMOV("^IRMOV R([0-7]) 0x([0-9A-F]{4,4})$");

QRegExp IRMOVS("^IRMOVS R([0-7]) 0x([0-9A-F]{4,4})$");

QRegExp RRMOV("^RRMOV R([0-7]) R([0-7])$");

QRegExp LAD("^LAD R([0-7]) R([0-7])$");

QRegExp STO("^STO R([0-7]) R([0-7])$");

QRegExp CLA("^CLA R([0-7])$");

QRegExp SET("^SET R([0-7])$");

QRegExp PUSH("^PUSH R([0-7])$");

QRegExp POP("^POP R([0-7])$");

QRegExp ADD("^ADD R([0-7]) R([0-7])$");

QRegExp SUB("^SUB R([0-7]) R([0-7])$");

QRegExp MUL("^MUL R([0-7]) R([0-7])$");

QRegExp DIV("^DIV R([0-7]) R([0-7])$");

QRegExp NEG("^NEG R([0-7])$");

QRegExp INC("^INC R([0-7])$");

QRegExp DEC("^DEC R([0-7])$");

QRegExp CMP("^CMP R([0-7]) R([0-7])$");

QRegExp AND("^AND R([0-7]) R([0-7])$");

QRegExp OR("^OR R([0-7]) R([0-7])$");

QRegExp NOT("^NOT R([0-7])$");

QRegExp XOR("^XOR R([0-7]) R([0-7])$");

QRegExp SHL("^SHL R([0-7]) R([0-7])$");

QRegExp SHR("^SHR R([0-7]) R([0-7])$");

QRegExp JMP("^JMP (-?)0x([0-9A-F]{4,4})$");

QRegExp JC("^JC (-?)0x([0-9A-F]{4,4})$");

QRegExp JO("^JO (-?)0x([0-9A-F]{4,4})$");

QRegExp JCM("^JCM (-?)0x([0-9A-F]{4,4})$");

QRegExp INT("^INT ([1-3])$");

QRegExp RETI("^RETI$");

QRegExp HLT("^HLT$");

QRegExp IN("^IN R([0-7])$");

QRegExp OUT("^OUT R([0-7])$");

在执行汇编操作时，每次从文件中读取一行文本串，与指令正则式进行匹配，得到指令类别，再利用QRegExp类提供的cap()函数获取模式串括号内的特定文本内容(指令中寄存器标识，立即数等)，转换为相应的01串。

对IRMOV指令进行汇编的示例代码如下，其中hexToBin()函数为自定义的功能函数，能够将单个十六进制字符(‘0’~‘F’)转化为对应的四位二进制01字符串。

QString ins; // store assembled instruction

if(IRMOV.indexIn(s) != -1) //IRMOV matched

{

ins += "00000"; //operating code

ins += "0000000"; //reserved bits

ins += hexToBin(IRMOV.cap(1)); //number of RD

ins += hexToBin(IRMOV.cap(2)); //immediate operand

}

装载指令的核心代码如下图，其运行流程可描述为:

打开指令文件->逐行读取文本->汇编->装入内存->显示至指令列表->关闭文件

bool VMulator::**load**(QFile &insFile, int &pos)

{ //load instructions from file to VMulator memory

QTextStream in(&insFile);

QString line,ins;

while(!in.atEnd())

{

line = in.readLine();

ins = assemble(line);

if(ins == "") //no instruction matched

{

ui->InstructionList->clear();

QMessageBox::warning(this,"指令加载失败","检测到异常指令" + line + "\n请重新加载指令");

return false;

}

strToMem(ins, Memory, pos); //write instructions to memory

ui->InstructionList->addItem(line); //show instruction in UI

}

insFile.*close*();

return true;

}

### 取指与执行

取指操作用于将程序计数器PC指向的内存区中的指令装入指令寄存器IR中，实现代码如下，其中insLen为预先设定好的存储各指令字节数的数组，用于兼容变长指令的取指。

void VMulator::**insFetch**() //fetch instructions from memory to IR

{

int op = (Memory[PC] >> 3) & 0x1f; //get the first 5 bits of a byte

for(int i = 0;i < insLen[op];i++) //write instruction to IR

IR = (IR << 8) + Memory[PC++];

IR = IR << ((4-insLen[op]) \* 8); //make IR starts from the leftmost bit

}

对于指令的单步执行操作，除了进行各指令所对应的操作外，还需要执行中断检测、异常处理、可视化等操作，其核心代码如下，其中繁杂的指令执行操作代码用伪代码代替，可查阅源码文件execute.cpp中的exec\_forward()函数。

void VMulator::**execForward**() //execute the next instruction

{

if(IRR == 1) //check hardware interrupt

{

IAR = PC; //record interrupt address

IRR = 0; //clear interrupt request

IM = 1; //mask interrupt

PC = byte4ToWord(0); //read interrupt vector table(the input interrupt's interrupt identification code is 0)

Stat = INT;

display();

return;

}

if(Stat != AOK) //show error information and stop

{

if(Stat == WAI)

QMessageBox::information(this,"等待中","等待输入，无法继续执行");

else if(Stat == HLT)

QMessageBox::information(this,"停机","已停机");

else if(Stat == INT)

execInput();

else

QMessageBox::warning(this,"错误","状态异常，无法继续执行");

return;

}

if(execDone)

{

QMessageBox::information(this,"执行完毕","已执行完毕所有指令");

if(RunningMode == Continuous)

{

RunningMode = Stepping;

ui->Button\_RunOrPause->setIcon(QIcon(":/Icons/Icons/Run.ico"));

timer->stop();

}

return;

}

//clear the background color of current instruction

ui->InstructionList->setCurrentRow(addrInsMap[PC]);

ui->InstructionList->currentItem()->*setBackgroundColor*(Qt::white);

insFetch(); //fetch instruction from memory to IR

int op = (IR >> 27) & 0x1f; //get the first 5 bits of IR as operation code

int D,S,Imm; //D,S = the corresponding index(0~7) of registers Imm = the immediate operand

bool sign; //the sign bit of the Imm in JMP JC JO JCM

long long ovCheck; //check overflow

int n; //number n in instruction INT n

switch(op)

{

………… //execute the corresponding instruction according to the operation code

}

if(addrInsMap.count(PC) == 0) //if reached undefined position

{

QMessageBox::warning(this,"错误","跳转错误，PC指向未装载指令的地址");

Stat = ADR;

}

else if(addrInsMap[PC] == ui->InstructionList->count()) //if reached the ending position of instrucitons

execDone = true;

if(Stat != AOK) //highlight the error instruction

{

if(Stat == WAI)

ui->InstructionList->currentItem()->*setBackgroundColor*(Qt::yellow);

else if(Stat == HLT)

ui->InstructionList->currentItem()->*setBackgroundColor*(Qt::gray);

else

ui->InstructionList->currentItem()->*setBackgroundColor*(Qt::red);

}

else if(!execDone) //if not done, highlight the next instruction to be executed

{

ui->InstructionList->setCurrentRow(addrInsMap[PC]);

ui->InstructionList->currentItem()->*setBackgroundColor*(QColor(175,238,238));

}

display(); //show changes

}

对于连续执行方式，设置一个定时器QTimer，通过图形界面获取用户设定的执行频率，设置定时器timeout事件的发生间隔，并利用Qt的信号与槽机制，将定时器的timeout信号与单步执行功能函数连接起来，当用户点击连续/单步执行按钮时，设置QTimer开始/暂停，以此实现连续执行的效果，核心代码如下：

QTimer \*timer = new QTimer;

//clock timer using to control the frequency of execution in continuous running mode

timer->setInterval(1000/clockFreq);

connect(timer, SIGNAL(timeout()), this, SLOT(execForward()));

void VMulator::**on\_Button\_RunOrPause\_clicked**()

{

if(ui->InstructionList->count() == 0) //no response if instruction list empty

return;

if( Stat != AOK && !(Stat == WAI && IRR == 1) ) //if not normal state, show error

{

execForward();

return;

}

//Change RunningMode

if(RunningMode == Stepping)

{

RunningMode = Continuous;

ui->Button\_RunOrPause->setIcon(QIcon(":/Icons/Icons/Pause.ico"))

timer->start();

}

else

{

RunningMode = Stepping;

ui->Button\_RunOrPause->setIcon(QIcon(":/Icons/Icons/Run.ico"));

timer->stop();

}

}

### 中断及输入操作

对于随机出现的用户输入事件，采用硬件中断方式进行相应，当用户在非中断屏蔽状态下点击输入区Input按钮时，即发出硬件中断请求，虚拟机会将中断请求寄存器IRR置1，并读取输入区首字符对应的ASCII值打入模拟键盘外设中的输入寄存器INR中。在执行流程中，虚拟机在取指前会先检查IRR寄存器，如有中断请求即保存现场，将中断屏蔽寄存器IM置1屏蔽中断，并转入执行中断服务，硬件中断对应的中断标识码为0，其入口地址可通过查阅中断向量表首4位字节得出（即实际地址0000F~0004F）。

本虚拟机的硬件中断服务程序需要将INR寄存器中的数值，借助输入队列头尾地址寄存器IH和IT，写入内存中的输入缓冲区段中，具体指令有三条： STO IT INR 用于将INR寄存器中数据写入内存中输入队列队尾，INC IT指令实现输入队尾地址的移动，RETI指令执行中断返回操作，实现代码如下：

void VMulator::**execInput**()//execute hardware interrupt routine to get input

{

static int cnt = 0; //count the number of executed instructions, which is used as the row number of current instruction

if(cnt == 0) //STO IT INR

{

Memory[IT] = INR; //store input

ramMonitor->changeMem(IT,INR); //show memory change in ram monitor

InputCacheQueue += (unsigned char)INR; //input cache queue enqueue

PC += 2;

ui->textBrowser\_InputQueue->setText(InputCacheQueue);

//display input cache queue

}

else if(cnt == 1) //INC IT

{

IT++;

PC++;

}

else if(cnt == 2) //RETI

{

PC = IAR; //restore PC

IM = 0; //unblock interrupt

Stat = AOK;

//restore the instruction list before

ui->InstructionList->clear();

ui->InstructionList->addItems(insList);

insList.clear();

ui->InstructionList->setCurrentRow(addrInsMap[PC]);

}

cnt = (cnt + 1) % 3;

display();

}

此外，还可以通过执行INT n指令来触发软件中断，其中软件中断所需的中断向量表和对应的中断服务程序在虚拟机初始化阶段已经写入内存，在转入软件中断时，虚拟机需要保存现场(包括保存当前指令列表)，置IM为1屏蔽中断，然后查找中断向量表，获取对应的中断服务程序入口地址并转入。

作为示例，VMulator虚拟机初始化操作时在内存中预置了三条软件中断服务子程序INT 1~3以及写入了对应的中断向量，三个中断服务程序分别为：

INT 1：Hello, world程序，在显示器输出Hello, world字符

INT 2：计算斐波那契数列前100项

INT 3：包含所有指令的指令格式示例程序

具体指令可在虚拟机中执行INT n指令后查看所转入的对应指令列表

实现代码如下：

//INT n

n = (IR >> 24) & 0x7;

IAR = PC; //record interrupt address

PC = byte4ToWord(n\*4); //read interrupt vector table(the input interrupt's interrupt identification code is 0)

IM = 1; //mask interrupt

//push state bit registers to stack segment

Memory[++SP] = CF; ramMonitor->changeMem(SP, CF);

Memory[++SP] = CMF; ramMonitor->changeMem(SP, CMF);

Memory[++SP] = OF; ramMonitor->changeMem(SP, OF);

//push general purpose registers

for(int i = 0; i < 8;i++)

{

wordTo4Byte(R[i],SP+1);

SP += 4;

}

insList.clear();

//record current instruction list

for(int i = 0;i < ui->InstructionList->count();i++)

insList << ui->InstructionList->item(i)->text();

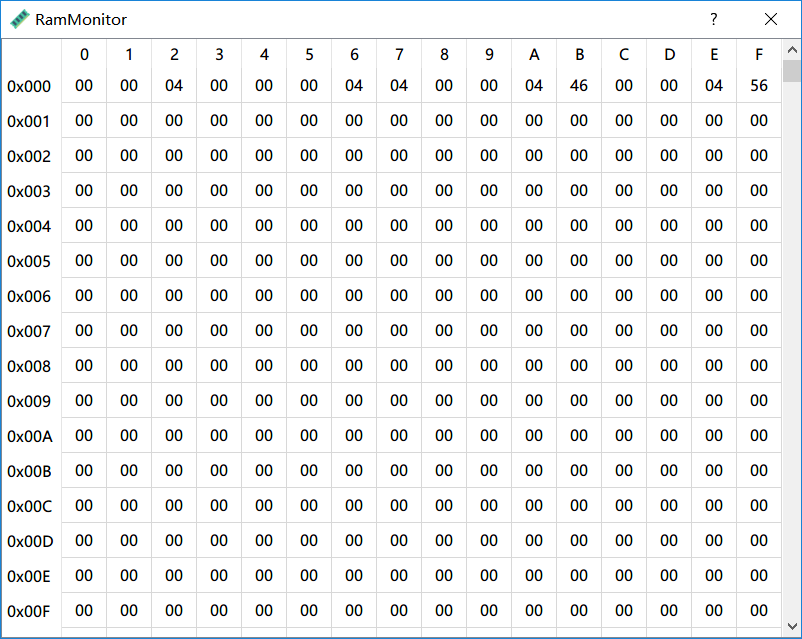
//show instruction list of interrupt routines

ui->InstructionList->clear();

ui->InstructionList->addItems(intRoutineLists[n]);

### 内存监视

采用子窗口的形式，利用QTableWidget部件提供内存监视功能，界面如下图



其中，左侧表头表示内存地址的前三位十六进制数，上方表头表示末位数，能够在执行时实时监测内存区的变化，核心代码为RamMonitor类提供的changeMem(int addr, int value)接口函数，在主程序执行过程中更改内存的值时(如执行存数STO、入栈PUSH指令)，调用changeMem()函数，即可更改内存监视器中表格相应位置的值。

void RamMonitor::**changeMem**(int addr, int value)

{ //change the value of Memory[addr]

QString s;

s.sprintf("%02X",value);

QTableWidgetItem \*Memdata = new QTableWidgetItem(s);

Memdata->setTextAlignment(Qt::AlignCenter);

ui->RamTable->setItem(addr/16, addr%16, Memdata);

}

### 回退操作

回退操作是在虚拟机设计过程中，为增强调试功能，使操作流程更加友好直观而引入的特色功能，能够实现在非中断状态下，回退至上一条指令未执行时的状态。

回退操作的实现思路为利用新定义的系统状态堆栈数据结构，在执行指令前将系统的各个状态，如运行状态、寄存器值、显存内容等保存压栈，而在执行回退操作时将对应数据恢复并出栈。

对内存中存储值的恢复是需要特别注意的一点，由于虚拟机内存较大，直接将内存整个记录并恢复会造成虚拟机执行效率过低。为解决这个问题，在每次发生对内存值的修改操作前，会先记录下被修改的内存单元的地址和值，存储在自定义堆栈数据结构changedMemAddr和changedMemWord中，当执行恢复操作并检测到上一条执行的指令修改了内存值(STO，PUSH等)时，通过读取堆栈内容对相应的内存单元进行写回，实现回退。核心代码如下：

//stacks to store the states before execution, using to roll back to the last instruction

std::stack<int> changedMemAddr;

std::stack<int> changedMemWord;

std::stack< std::vector<byte> >DisplayMemoryStack;

std::stack<VM\_State> lastState;

void VMulator::**execBackward**()

{

if(lastState.empty())

{

QMessageBox::warning(this,"回退失败","无可回退的上一条指令");

return;

}

int op = (IR >> 27) & 0x1f;

//get the operation code of the last executed instruction

if((op == 4 || op == 7) && Stat == AOK) //if the last instruction is STO or PUSH, restore the changed memory unit

{

int addr = changedMemAddr.top(), word = changedMemWord.top();

wordTo4Byte(word,addr);

changedMemAddr.pop();

changedMemWord.pop();

}

else if(op == 28 && Stat != INS) //if the last state is executing the interrupt service routine, rollback fails

{

QMessageBox::warning(this,"回退失败","无法回退至中断状态");

return;

}

else if(op == 31) //if the last instruction is OUT, delete the last character in display memory

{

if(DisplayMemory.empty()) //if DisplayMemory cleared

{

DisplayMemory = DisplayMemoryStack.top();

DisplayMemoryStack.pop();

}

else

DisplayMemory.pop\_back();

}

//restore the state before the last instruction execution

VM\_State vm\_state = lastState.top();

Stat = vm\_state.Stat;

CF = vm\_state.CF; CMF = vm\_state.CMF; vm\_state.OF = vm\_state.OF;

PC = vm\_state.PC; IR = vm\_state.IR; AR = vm\_state.AR; SP = vm\_state.SP; IH = vm\_state.IH;

for(int i = 0;i < 8;i++) R[i] = vm\_state.R[i];

lastState.pop();

display();

}

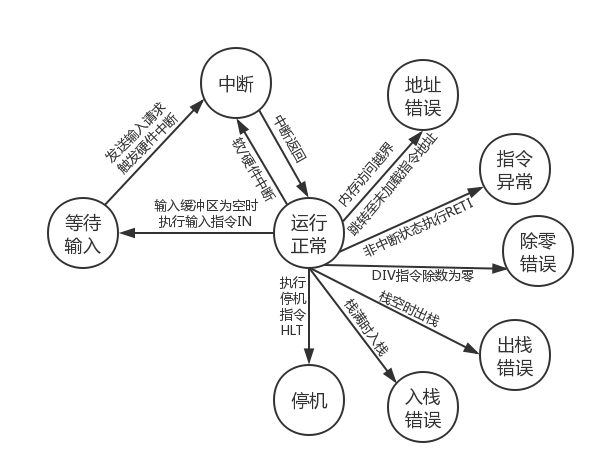
### 异常处理

在虚拟机的开发过程中，力求实现完善的异常处理机制，保证执行结果的正确性，避免程序运行错误和崩溃，并使软件能够反馈充足的异常信息，增强虚拟机的调试能力，提高对虚拟机软件进行操作的直观性和方便性。

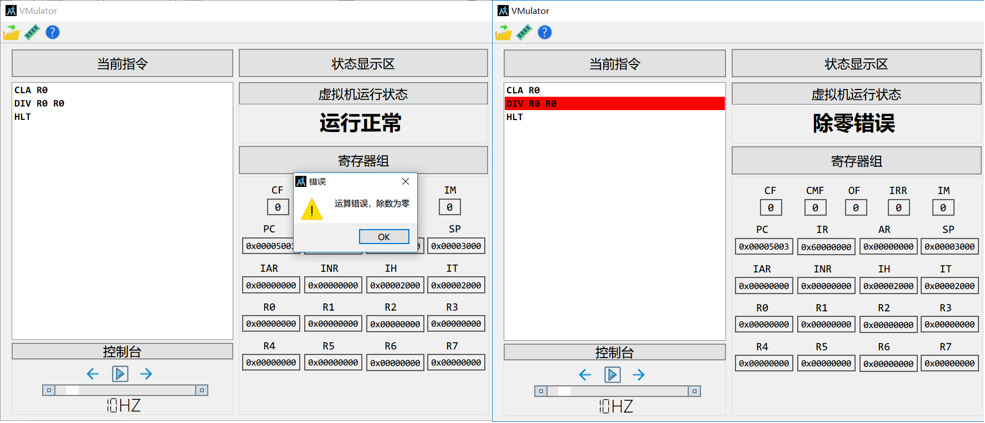
本机的异常处理参考了《深入理解计算机系统》一书中Y86的异常处理机制[[3]](#endnote-3)[3]并进行了改进，为处理指令执行中产生的异常，共定义了9种虚拟机状态，具体解释如下表：

|  |  |  |
| --- | --- | --- |
| 状态名称 | 状态简记符 | 含义 |
| 运行正常 | AOK | 无异常发生 |
| 停机 | HLT | 执行了停机指令HLT |
| 等待输入 | WAI | 输入队列为空时运行输入指令IN，悬置等待 |
| 中断 | INT | 硬件中断执行中 |
| 地址错误 | ADR | 地址访问错误，包括跳转指令跳转至未加载指令的地址和存/取数操作访问内存地址越界，对非指令/代码区的内存进行了访问 |
| 指令异常 | INS | 执行了异常指令，如在非中断状态下执行中断返回指令RETI |
| 除零错误 | DIV | 执行除法运算操作时除数为0 |
| 出栈错误 | STE | 内存栈区为空时执行出栈POP操作 |
| 入栈错误 | STF | 内存栈区已满时执行入栈PUSH操作 |

各状态之间的转换关系如下图所示：



在发生异常时，虚拟机会停止执行，给出相应的异常提示信息，显示状态信息并高亮产生错误的对应指令，如下图所示：



除了处理指令执行中产生的异常外，虚拟机还能够对其他软件运行中产生的错误操作及用户的异常输入做出反应，如文件打开失败，装载时检测到无效指令，在初始指令时进行回退操作，以及中断屏蔽时请求输入等，如下图所示：



注：在异常处理样例文件夹下有展示虚拟机异常处理功能的部分样例程序，可使用VMulator运行相应程序观察测试。

3.5样例测试与分析

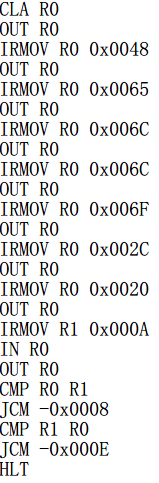
共编写了两组虚拟机程序进行样例测试，程序对应的txt文件存放在VMulator目录下的测试程序文件夹下。

### 样例测试1——Hello, user程序

样例测试1是为了测试虚拟机的输入输出、程序控制及硬件中断功能而设计的Hello,user程序，程序功能为从输入区读取以回车换行符为结尾的一串字符作为用户名称，并最终在显示区输出Hello, user(即输入的用户名)字样，指令的执行逻辑流程可描述为:

1. 输出 “Hello, ” 字符串
2. IN指令读取一个字符并输出
3. 如果字符为’\n’(ASCII码值为10)，转到第4步，否则回到第2步
4. 停机

首先，运行虚拟机，单击左上角第一个按钮，从测试程序目录下打开Hello, user.txt文件，程序指令如下：

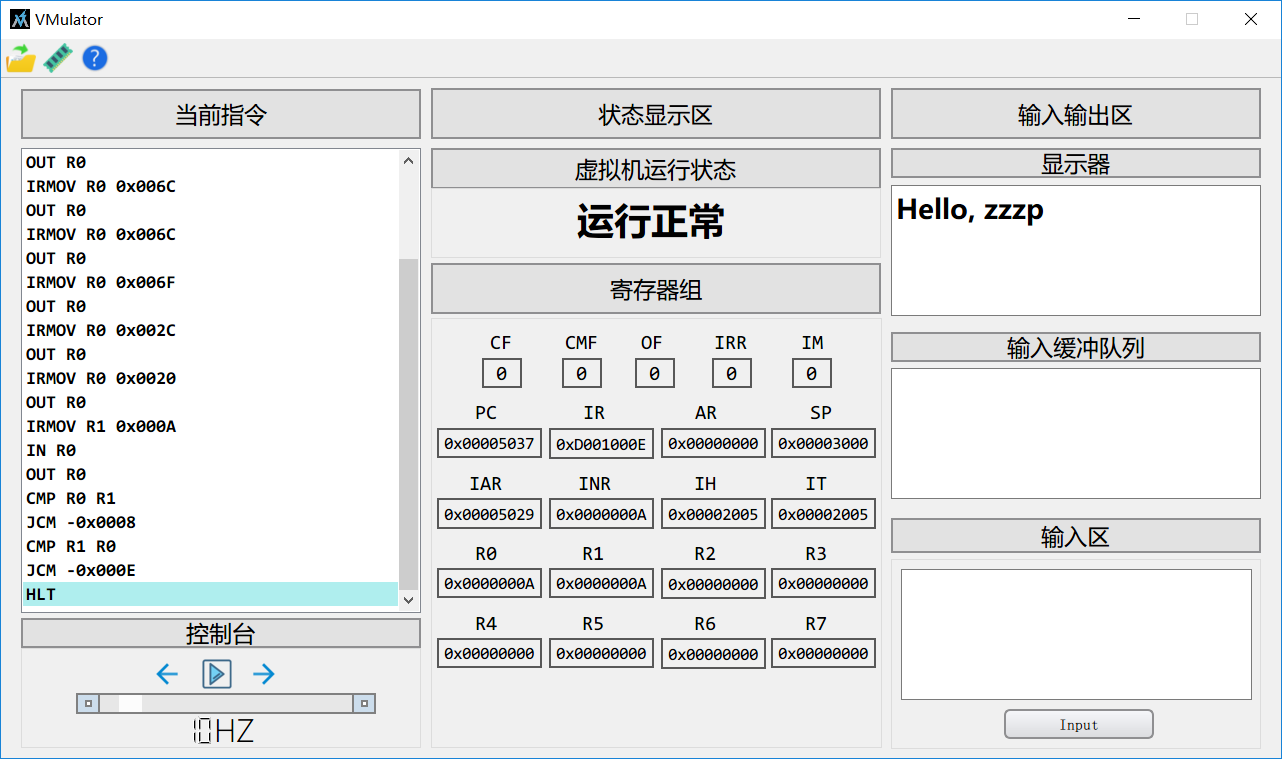


值得说明的是，对于本程序的条件控制，使用了两次CMP指令进行比较，这是由于本机的CMP Ra Rb指令定义为，如果Ra < Rb，则CMF寄存器置1，故为了实现条件控制转移，在R1里预置了回车字符的ASCII码值，如果CMP R0 R1和CMP R1 R0均未使CMF置1，即R0 ≮ R1且R1 ≮ R0, 则R0 = R1, 跳转指令无效，虚拟机执行停机指令HLT转为停机状态。

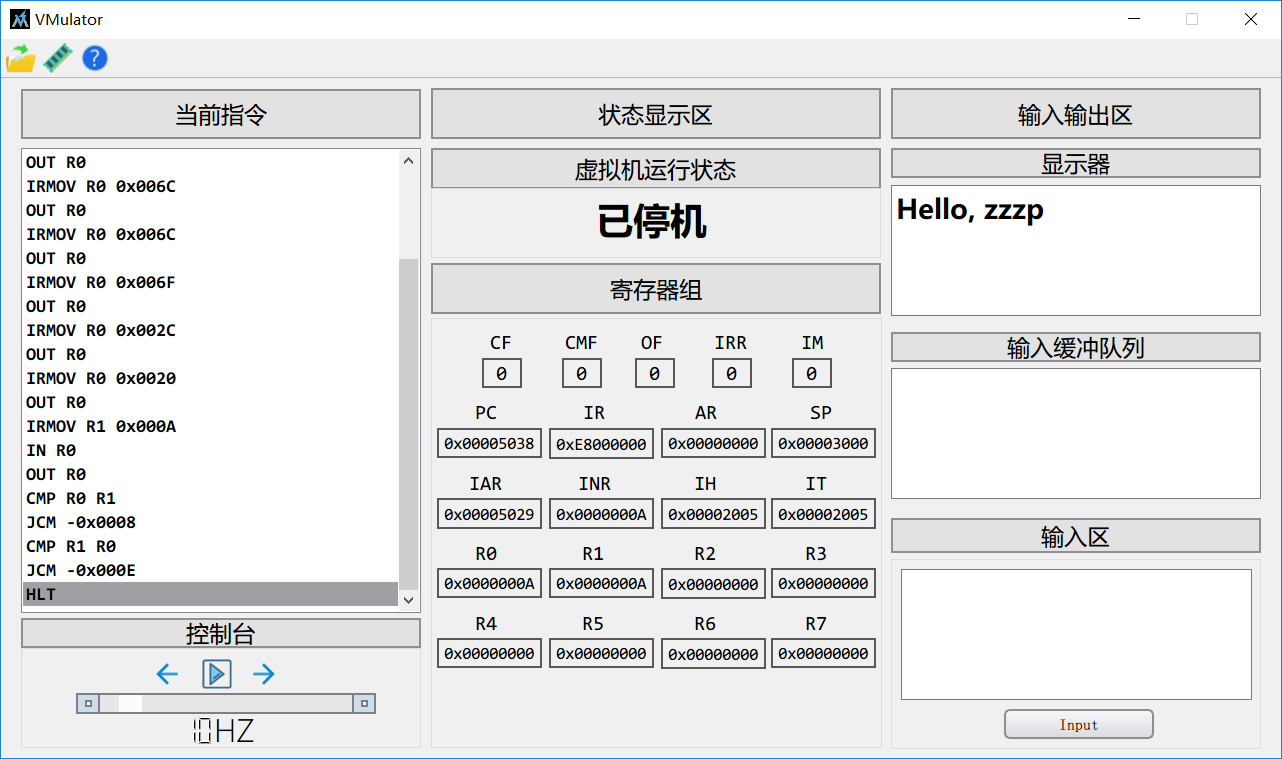
然后，利用控制台中的连续/单步模式切换按钮或单步执行按钮执行指令，在指令执行的任意时刻，在输入区输入以回车结尾的字符串(以“zzzp\n“为例)，并单击若干次Input按钮，虚拟机会转入中断服务程序读入输入并返回，如下图所示：



反复若干次之后，虚拟机读入回车字符，指令JMP跳转无效，指令执行至HLT指令位置:



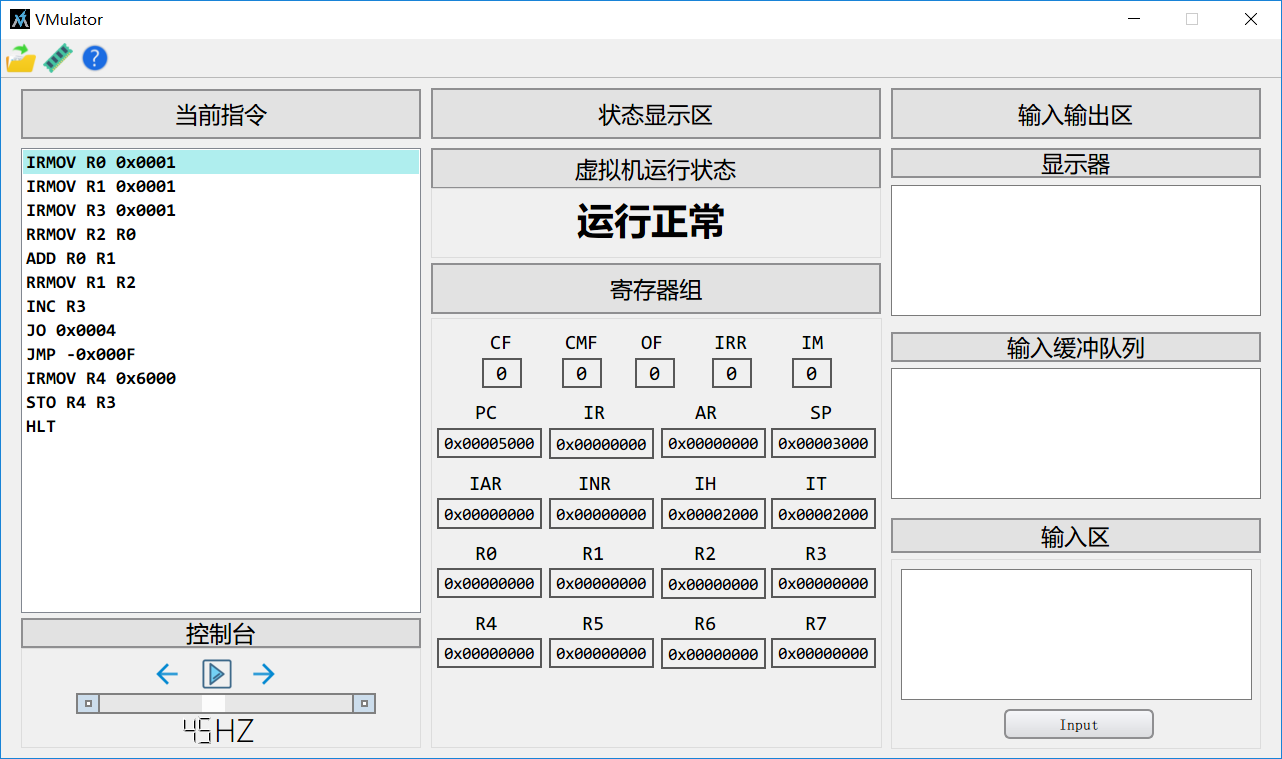
最后，执行HLT指令，虚拟机停机，显示区输出Hello, zzzp字样。



### 样例测试2——斐波那契数列计算

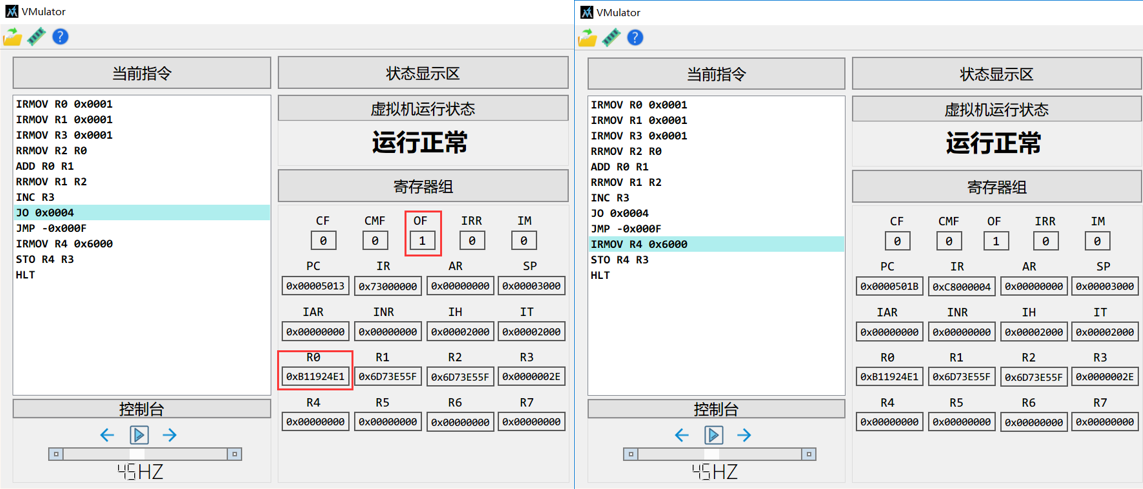
样例测试2是为了测试虚拟机的数据传送功能、运算功能、溢出判断以及内存监视模块而设计的斐波那契数列计算程序，程序功能为计算斐波那契数列至运算溢出(32位补码情况下)，并将当前项的计数值存入内存0x6000地址。

首先，从测试程序目录下打开Fibonacci.txt文件：

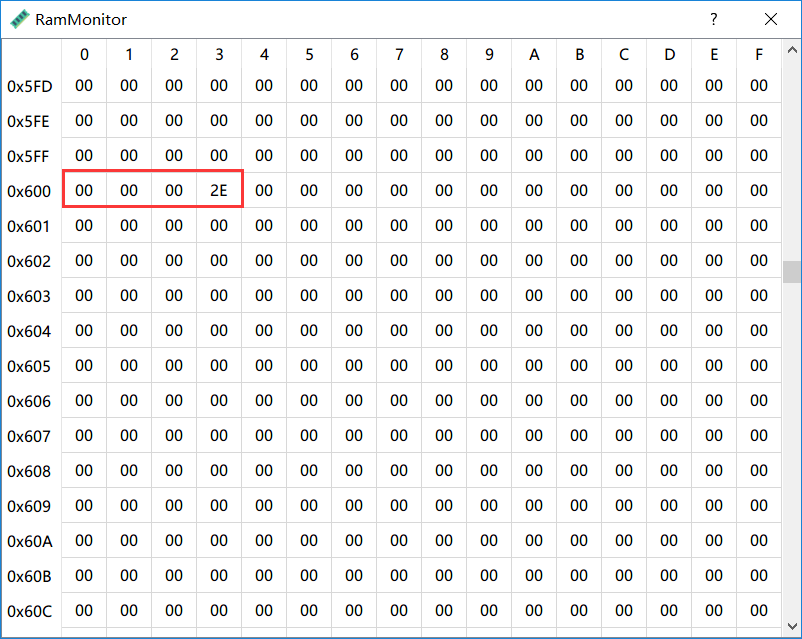


前几条立即数->寄存器指令IRMOV用于程序的初始化，在运行过程中，使寄存器R0存储当前最新项的数值，R1存储倒数第二项的数值，初值均为1，R2用于数据中转，R3用于计数当前最新项是斐波那契数列的第几项。

然后，直接点击控制台区域中央的连续/单步模式切换按钮开始执行指令，在开始的若干步运算中，R0和R1的值较小，不会发生溢出，OF状态位为0，JO跳转指令无效，程序执行JMP指令跳转回RRMOV R2 R0指令再次执行运算。

 在执行若干次后，可观察到R0变为0xB11924E1，即R0变成了补码数负数，发生了溢出，溢出标志位寄存器OF置1。此时，执行JO指令，会跳过下一条JMP指令，并继续执行指令将绝对地址0x0600装入寄存器R4，并利用寄存器间接寻址方法进行存数操作将R3值存入内存，最终停机，如下图所示：

最后，点击左上角第二个图标打开内存监视窗口，转至0x6000内存区段，可观察到所存数据如下图：



可见，内存区段0x6000~0x6003的四个字节存储了数值为0x0000002E的一个字长的数据，与寄存器R3的值符合，测试成功。

# 实验总结

收获与心得

在本次虚拟机的设计实现过程中，通过大量地查阅相关资料以及构思相关功能，我对计算机底层原理的认识更加深入。包括冯诺依曼结构组织的高效及合理性，如何搭建简洁明了并能够满足计算需求的通用指令集，以及如何通过中断机制协调外设与内部模块的关系，实现计算机与用户的有效交互等。

思考与展望

值得思考的是，相较上学期所做的proteus模型机，本次最终完成的软件虚拟机不单单具有模拟特定指令运行的功能，而是加入了额外的状态监视、异常处理、回退等功能，相当于是在模拟硬件层次上构建了简易的操作系统模块，这也使得我对计算机各大知识体系之间如何相互作用，合同发挥计算机功能有了更加深刻的理解。

本次完成的虚拟机具有较好调试和演示性能，并能够在已有功能基础上做进一步扩充，相信在以后的编译原理、计算机体系结构等课程中，也能够发挥帮助作用，增强对相关知识的理解。

# 五、参考文献

1. [1] 白中英，戴志涛等.计算机组成原理[M].第五版.北京：科学出版社，2013.249－251. [↑](#endnote-ref-1)
2. [2] 霍亚飞. Qt Creator快速入门[M].第三版.北京：北京航空航天大学出版社，2017. 165－173. [↑](#endnote-ref-2)
3. [3] Randal E.Bryant, David O'Hallaron.深入理解计算机系统[M].第三版.龚奕利，贺莲译.北京：机械工业出版社,2016.250-251. [↑](#endnote-ref-3)